SOLID Principles

[S - Single Responsibility Principle
O - Open closed principle
L - Liskov Substitution principle
I - Interface Segregation principle
D - Dependency inversion principle

SOLID Design Principles ⇐ guidelines / fundamental approach

Design ↓ S/w design

→ A set of guidelines that help a SWE design better software systems, and helps to achieve the following:-

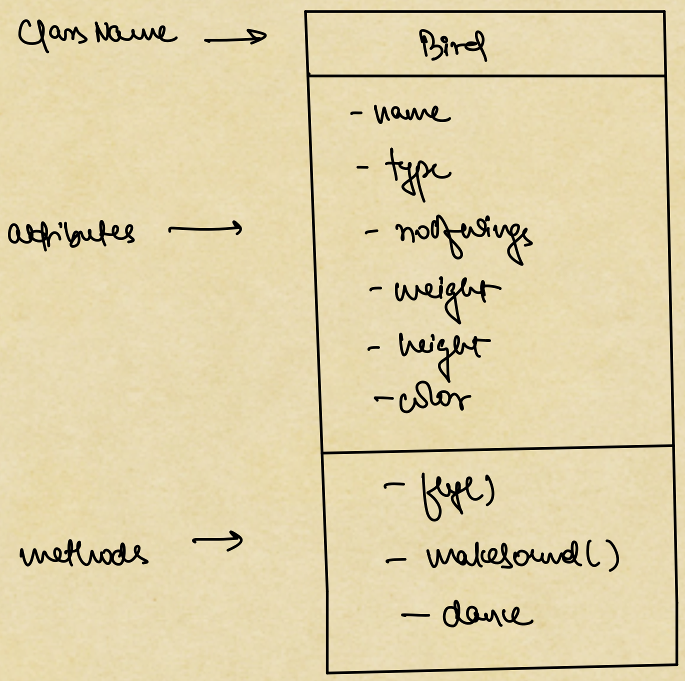
i) Extensible
ii) Maintainable
iii) Reusable
iv) Easily testable
v) Modular
vi) Understandable

⇒ Design a bird :- { Amazon Interview Q" }

Req

Assume you have to build a S/W system where we have to store information about birds. Maintain diversity of birds.

V1



```
Bird sparrow = new Bird();
    sparrow.name = "sparrow";
    sparrow.type = smallBird;
    sparrow.height = ____
    sparrow.weight = ____
        :
```

```
Bird crow = new Bird();
    crow.name = " crow ";
    crow.type = mediumBird;
    crow.height = ____
    crow.weight = ____
        {
```

sparrow.makesound(); —— dont make same sound —— crow.makesound();

```
makesound() {
        if( name == crow)
                sout( kaw kaw)
        else if( name == sparrow)
                sout( chi chi)
        else if( name = pigeon)
                sout( gutargu)
        .
        |
}
```

too many if-else

⇒ Problem with too many if-else

i) Readability & understandibility goes down

ii) Difficult testing

iii) Multiple developers working on it will lead to merge conflict

iv) code duplication

v) less code reuse

vi) error prone

vii) violated $\boxed{S}$ of SOLID

⇒ Single Responsibility Principle (SRP)

→ Every code unit (class | method | package) in our codebase should have exactly 1 responsibility
↓
There should be exactly 1 reason to change code

⇒ makeSound() → should be responsible for how every bird will make sound.

⇒ fly() → should be responsible for how every bird fly

⇒ class Bird → hold attributes and methods for all kind of birds.

⇒ How to identify violation of SRP:-

a) Method with multiple if | else :-

: It might not be always true (business case)

: If, we are trying to achieve same functionality with unnecessary if/else ⇒ violation.

b) <u>MONSTER METHODS:</u>

→ method which has a lot of code, that does a lot more than it should do.

→ LLD | HLD ⇒ subjective
→ there is no one correct answer
→ dont do over engineering

eg ⇒

```
saveToDatabase( User user, Database dB) {

    String query = " INSERT INTO -- ____          ] query creation
                               - -   -
                               . - - - -
                               . - - ___ -   "

    Database dB = new Database();             ] dB connection & setup
        dB.setURL(" ____________ ");
        dB.setUserName(" ____________ ")
        dB.setPassword(" ____________ ").
      dB.connect();

    dB.execute(q)                             save → ]
}
```

↓ Optimised

```
saveToDatabase( User user ) {
  String q = createQuery(user);
  Database db = getDBConnection();
   db.execute(q);
}
```

C) <u>Common / Utils</u>

→ largely discourage

→ common / utils become a garbage place for all methods that an engineer doesnt want to think about where to put

~~utils/~~

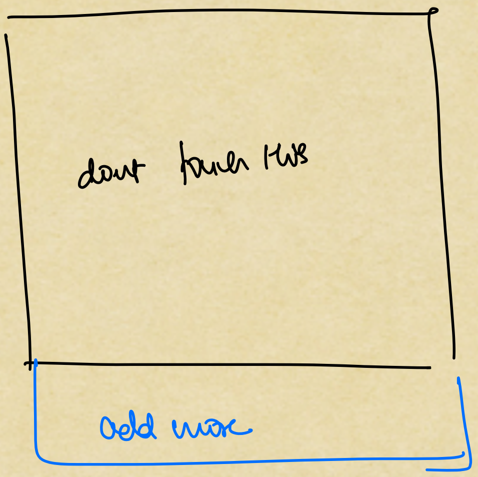
utils/
- DateTimeUtils
- CalenderUtils
- StringUtils

⇒ <u>Open closed Principle</u> :- (OCP)

⇒ Code should be open for extention and closed for modification.

⇒ makes your code more extensible

⇒ Adding new features, means adding more code not modifying current code.



⇒ Adding new features, should have as minm or 0 code modification as possible.

⇒ Bird design

→ need to add support for peacock

```
makeSound() {
        if(    )
        else if (    )
        else if (    )

fly() {
        if( ——— )
        else if( ——— )
        else if( ——— )
        else if (peacock) {
            ═══
        elseif(peacock){                                  =
                  =
                  =                             }
        }                                   }
   }
```

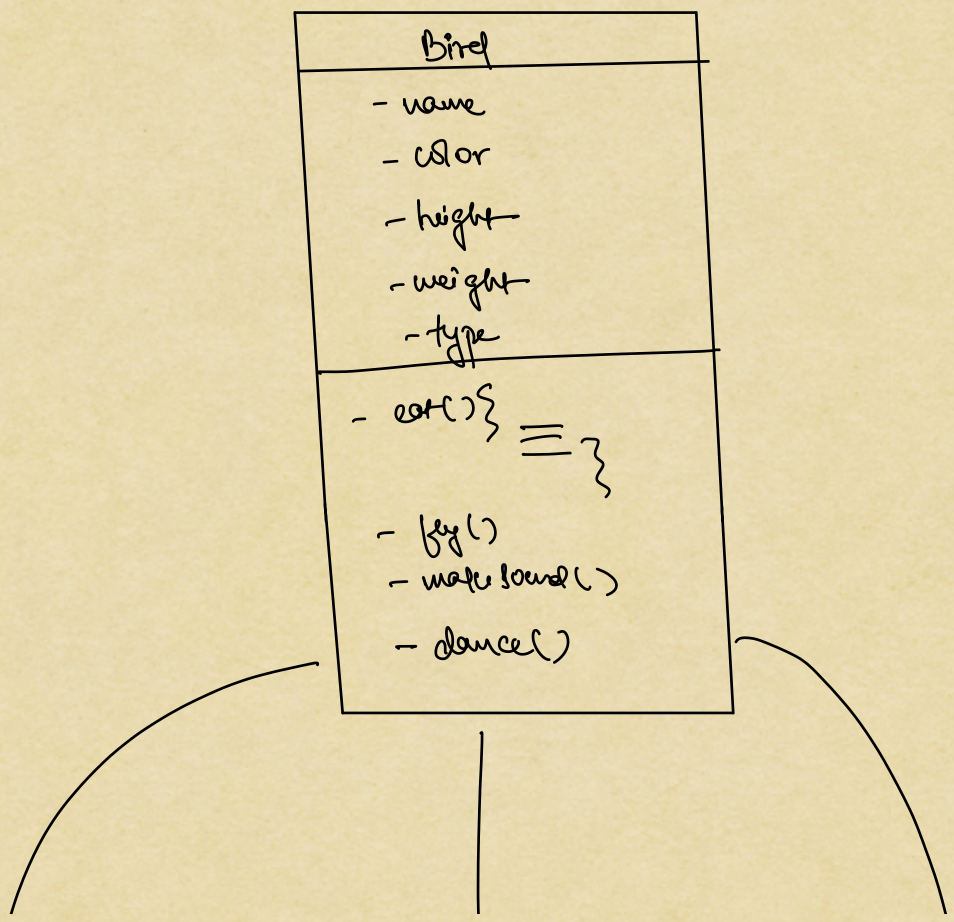
violation of OCP

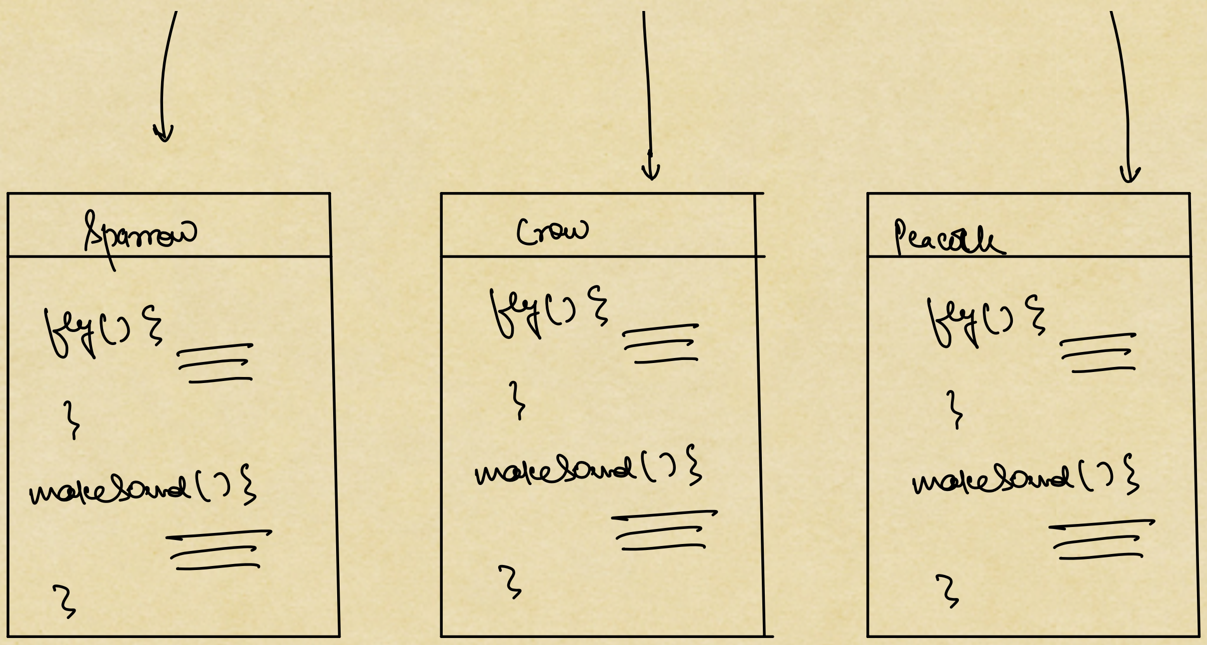
V2

→ every bird should have these actions
→ every bird should have their own implementation.

↓

Abstract class(attributes)





→ Now to add a new Bird, we will not need to make changes in an existing method / class but we will add an entire new class.

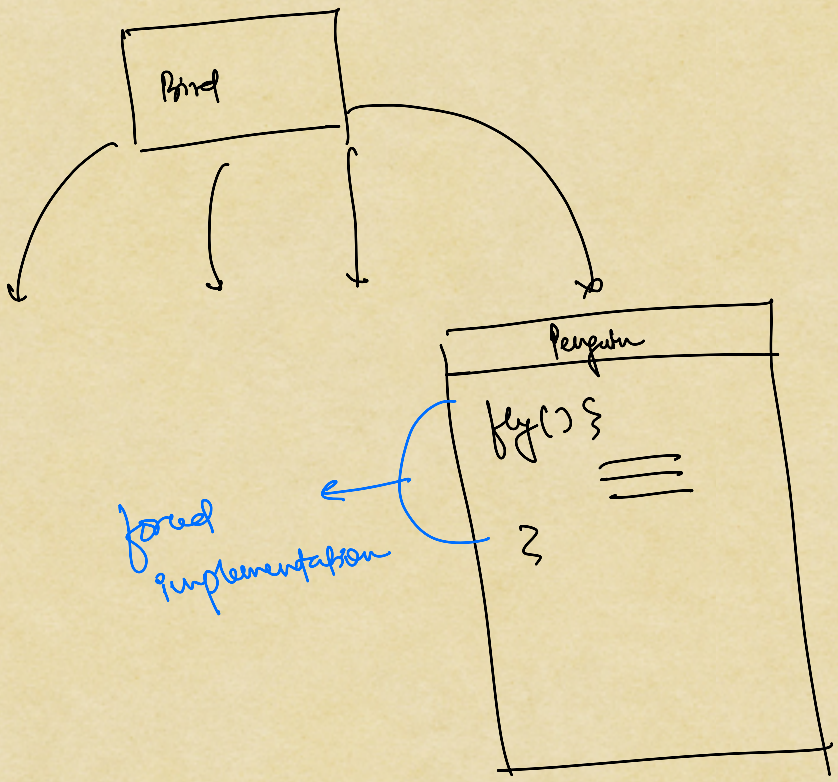
↓

follows OCP

and, SRP

→ add a new Bird ⇒ Penguin

Penguin cant fly

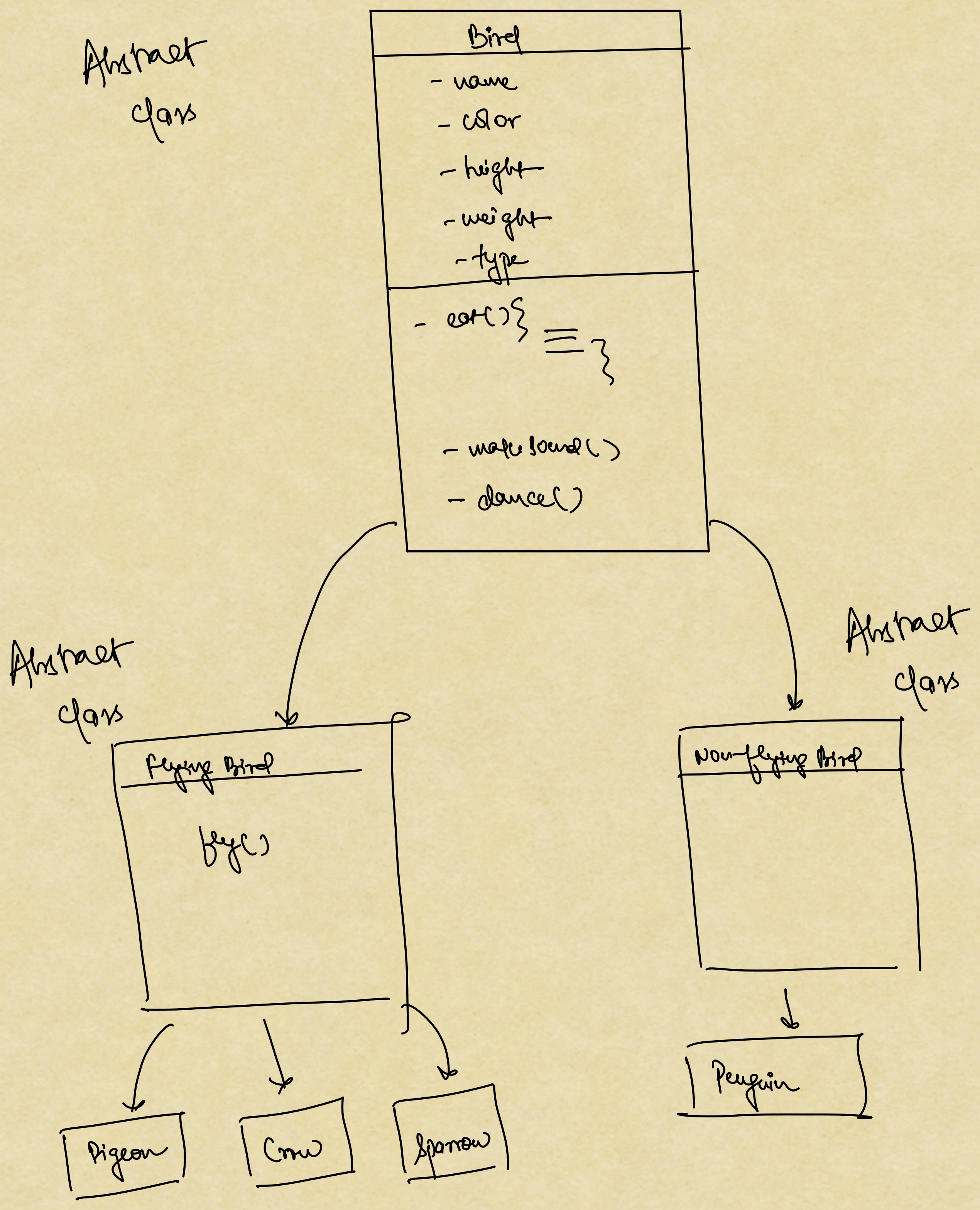


```
Class Penguin extends Bird {
    void fly() {
        ___________        keep it empty
        ___________
        ___________        throw an exception
    }
```

Keep it empty => no info [black box]

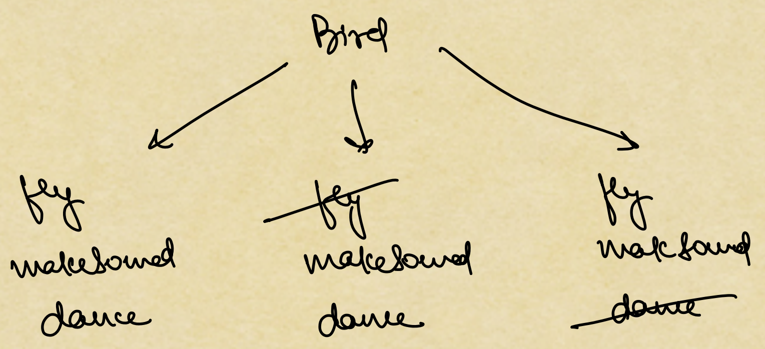
throw an exception => handle it

not a good idea



↓

Creates problem for multiple combinations



2 properties ⇒ A ~~A~~ A ~~A~~
(A B) B B ~~B~~ ~~B~~ ⇒ 4 combn's

3 properties ⇒ 8 combn's

N properties ⇒ 2^N combn's

10 properties ⇒ 1024 combn's

→ Class explosion

→ Liskov Substition Principle