Agenda

Start @ 9:05

SOLID

{ Solid Liquid Gas }

S : Single Responsibility Principle ✓
O : Open Close Principle ✓
L : Liskov's Substitution Principle ✓
I : Interface Segregation Principle ✓
D : Dependency Inversion Principle ✓

Design Principle
↓ ↳ set of rules / guidelines
S/w System

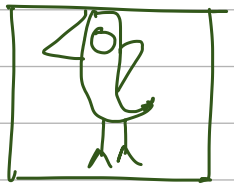
## Properties of Software System

1. Extensible
2. Maintainable
3. Easy to test
4. Readable / Understand
5. Reusable (DRY)
   ↳ Don't Repeat Yourself

# Design a Bird (OOP)

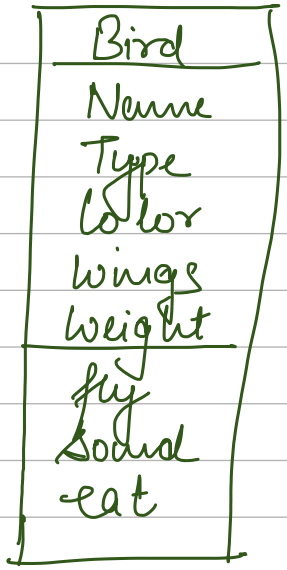
↓

S/w System



⇒ Assume that we have to build a system that store info. about all birds on earth

V1




```
Bird b1 = new bird();
    b1.name = "xyz";
    b1.type = "Piegon";

    b1.sound();
```

```
Bird b2 = new bird()
    b2.name = "abc"
    b2.type = "sparrow"

    b2.sound();
```

```
Void sound ( String type)
    if (type == "Piegon")
        sout ("PeigonSound");
    else if (type == "Sparrow")
        sout ("Sparrow Sound");
    else
        sout ("Default Sound");
}
```

Too Many If/else

Issues with too many If/else

1) Readability / Understandability
2) Difficult in testing
3) Code duplicacy → Less Reusability
4) Voilation of (S) of SOLID

Single Responsibility

# Single Responsibility Principle (SRP)

⇒ Every code unit (class | Package | interface | method) should have exactly 1 Responsibility

There should be only one reason to change the code unit.

⇒ Sound() method is responsible for all the sounds or sound for each bird, so lot of reason to change the code

⇒ How to check SRP is voilated?

① Too many if | else

```
Void doSomething (input i)
{
    if (input == 1)
    {
        ...
    }  Play Music

    else if (input == 2)
    {
        ...
    }  Shutdown

    else if
    {
        ...
    }  Start browser
}
```

Independent

restart

else if (input = 3) { start ide

2) Not always true

2) If the if/else condition is part of business logic or any algo then SRP is not violated

```
if (num % 2 ==0)
    even ≡
else
    odd ≡
```

SRP violation
NO

```
if (num % 3 ≥ 1
                2
                3
```

```
doSomething (input i)
{
    if (i == 1)
        playmusic ()
    elseif (i=2)
        shutdown()
    else
        startbrowser()
}
```

i) for input decision
ii) change
SRP

## ② Monster Method

2) Method that has a code to do lot more than it name suggests

eg

```
saveToDB(User user){
    Query q = "Insert ---
              ---
              ---";                    // (1) creating query
    Database db = new database();      // (2) creating DB
    db.url("url");                     // (3) creating DB
    db.execute(q);                     // (4) saving
}
```

Can be reused

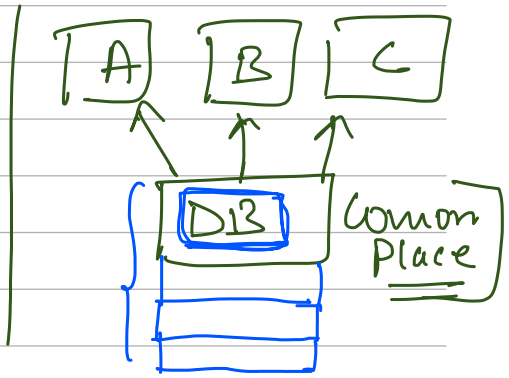
instead

```
saveToDB(User user)
{
    Query q = createQuery(user);
    Database db = getDatabase(-->);
    db.execute();
}
```

(3) Common | Utils

→ Discouraged

→ Common package | class becomes a garbage place for code where developer doesn't want to give a thought on Right Place



/utils/
- StringUtil.java
- UserUtil.java
- DateUtil.java

/src/main/com/scaler/user/
- userutils.java
- userconst.java

## SRP

↳ Every code unit should have 1 Responsibility

↓

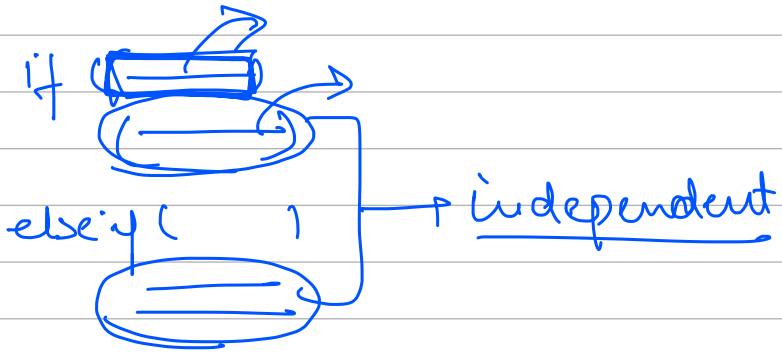
Only one Reason to Change

1) Too Many if/Else
2) Monster Method
3) Common/utils

eg

time = 1hour
time = 2hour

System




Stack [push pop - - - - -]

# Open Close Principle (OCP)

Our codebase should be open for extension but should be closed for modification

⇒ Codebase should be extensible (Add new feature)

but adding new feature should not require any code changes to existing codebase

⇒ Rather than changing the existing code units write new code units

Benefits ⇒ Testing

⇒ Regression ⇒ Existing flow will keep on working.

Note: Adding a new feature should ideally require no code change but in practical if this not possible then minimal code changes

Ex

| Bird |
| --- |
| - Name |
| - Color |
| + fly() |
| + sound() |

Till now

Pigeon, Sparrow

NewBird

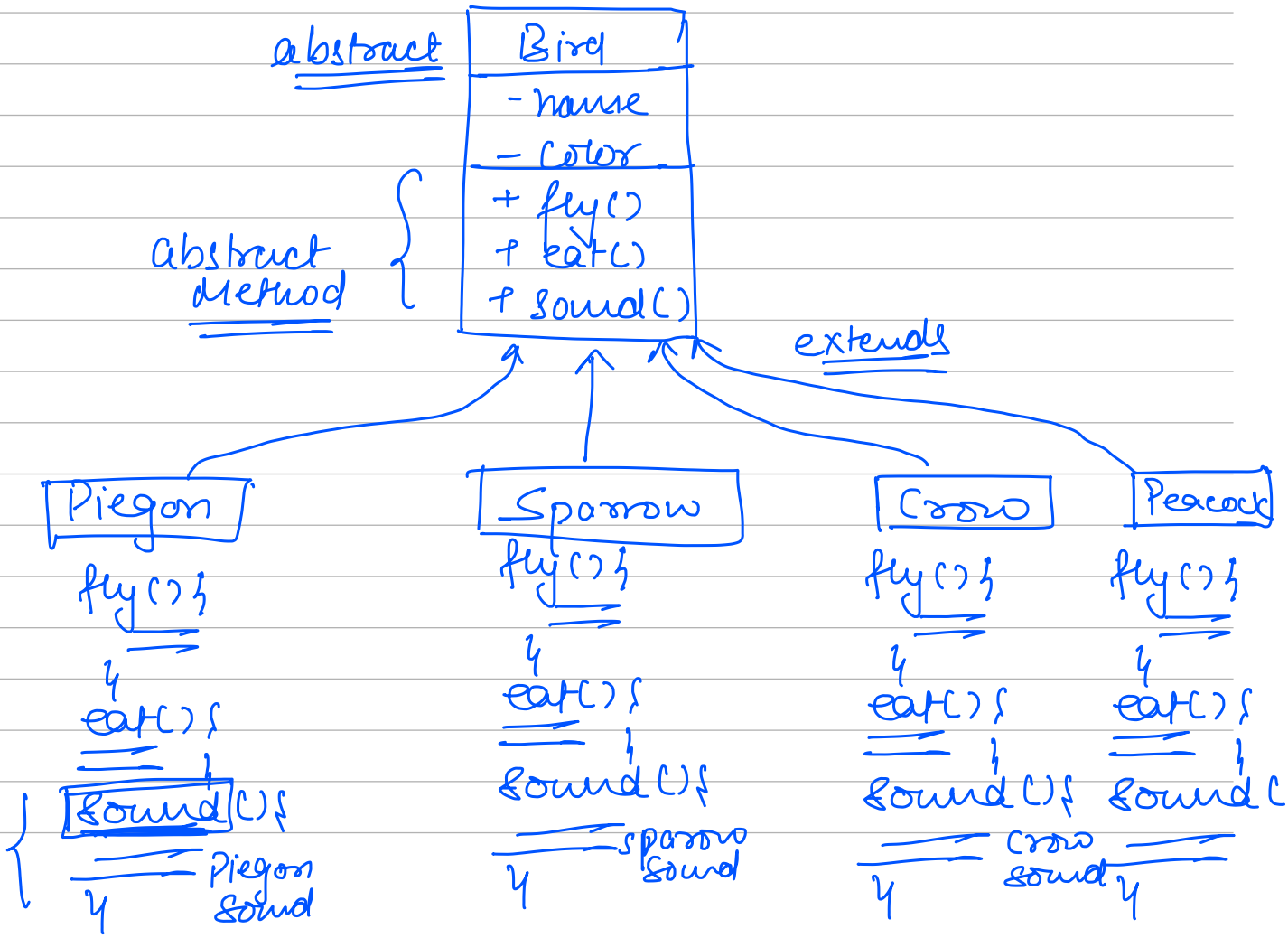
Crow

```
void sound (String type)
{
    if (type == Pigeon)
        sout ("Pigeon Sound")

    else if (type == Sparrow)
        sout ("sparrow sound")
    else if (type == Crow)
        sout ("Crow Sound");
    else
        sout ("Default Sound");
}
```

⇒ Not good design

12 Let the Bird class have only general attrs and methods. All specific behaviors, Let the specific classes handle them

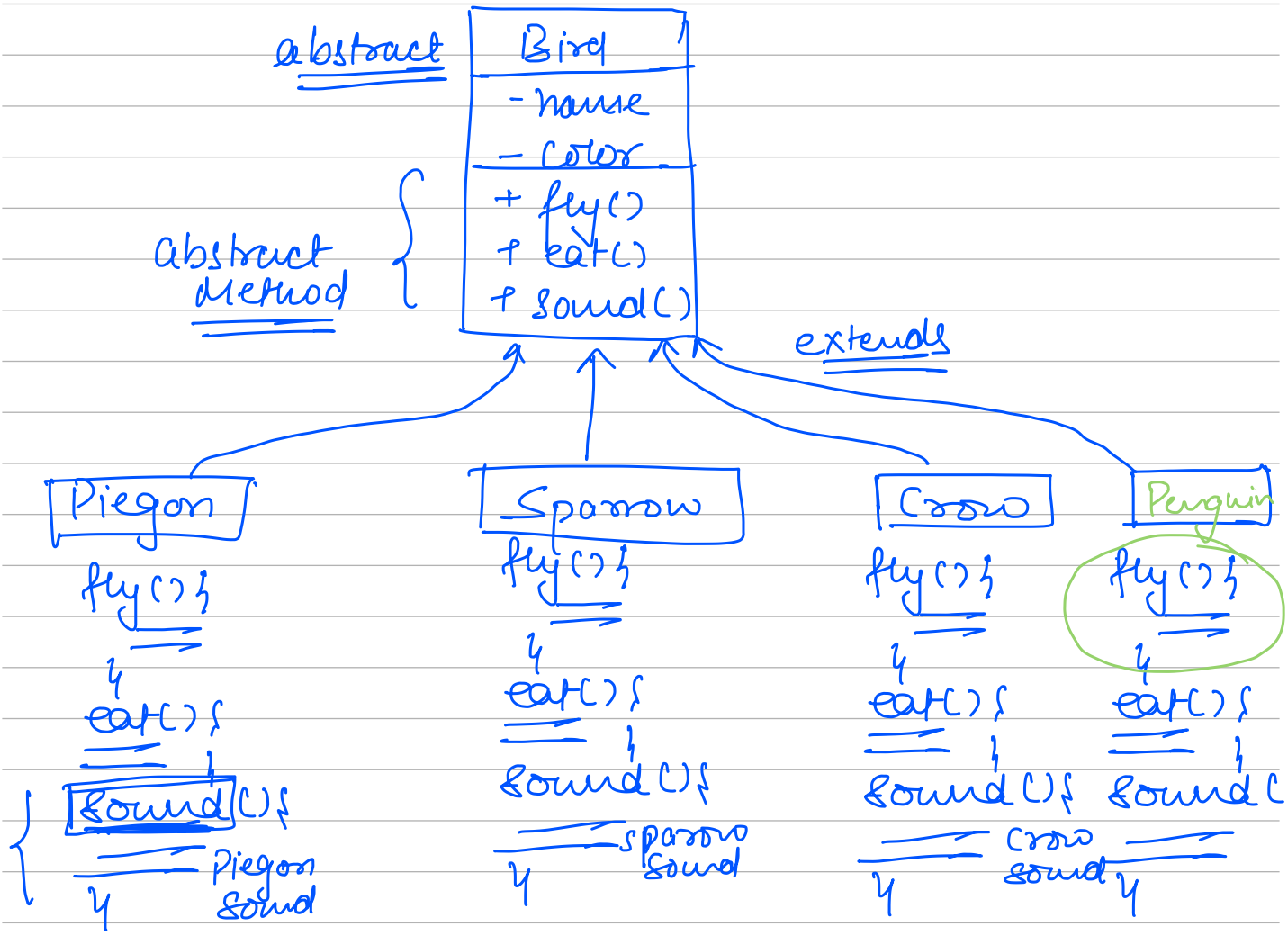


2) To add a new bird, we just need to add a new Child class

3) Every birdclass has a single reason to change (SRP)

List<Bird> = new Peigon()
new Peacock()
new Crow()

Req New Bird : Penguin
↳ Can't fly



Penguin

```
fly() {

}
```

① Empty
② Throw an Exception

Client

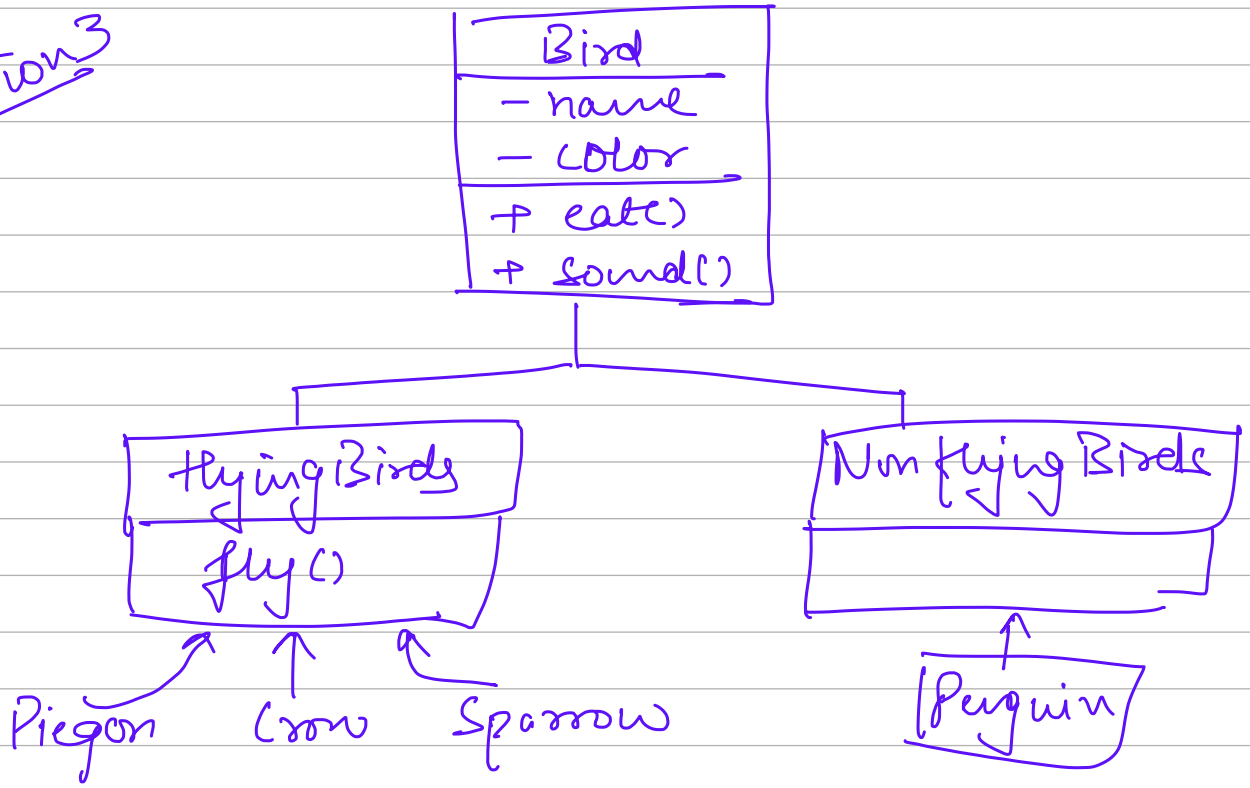
```
List<Bird> b = getBirds()

for {
    b.fly();
}
```

→ Surprise for Client (Don't give surprise to client)

⇒ We should not expose a functionality to client if it is not supported.

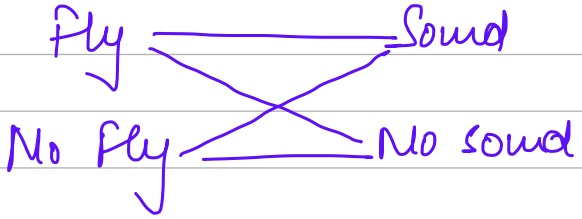
Version 3

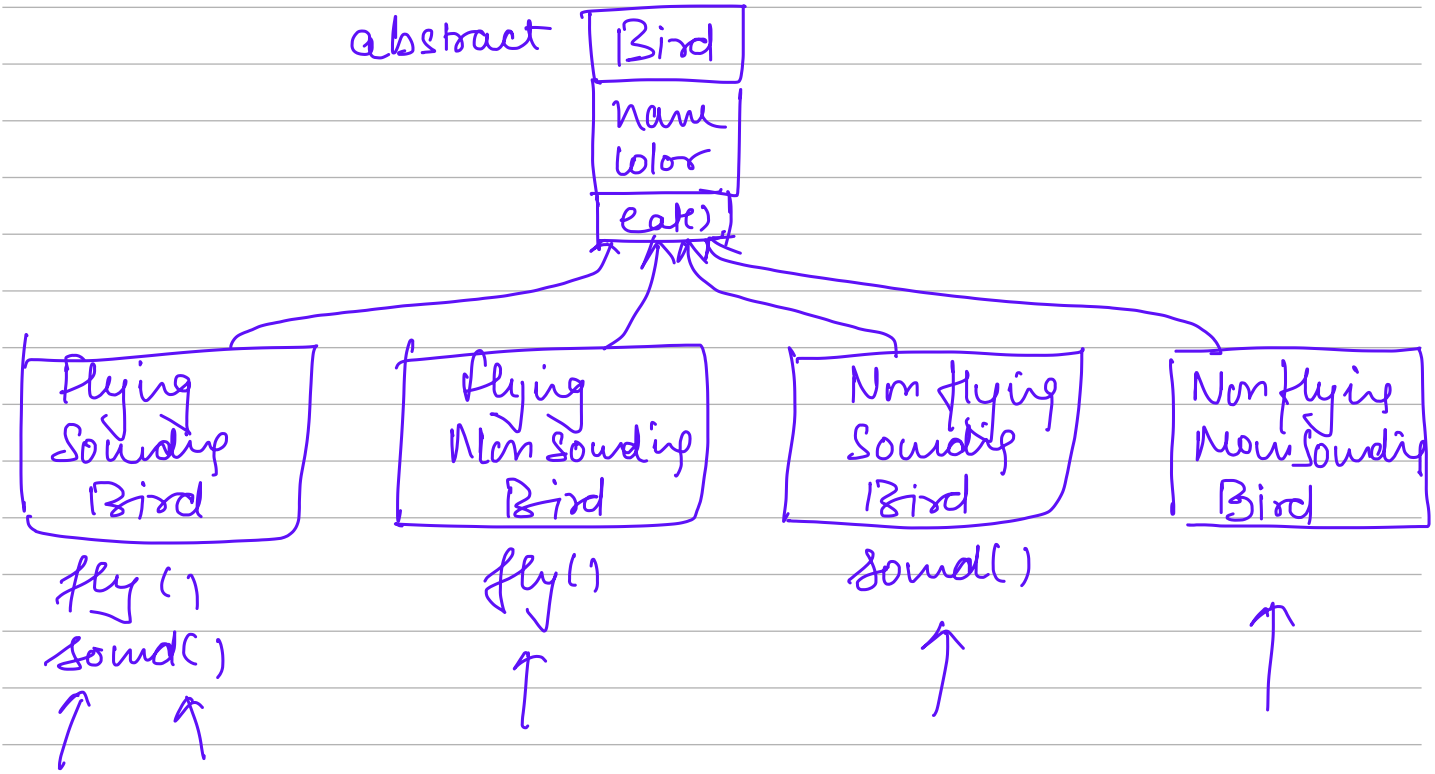


List < Flying Birds >

List < Non flying Birds >

List < Bird > ⇐ All type of Birds




Problems

① Class explosion ( Too many classes)

$2^N$ classes

② List of all flying Birds ✗

Next Class SOLID, LID