Agenda.

SOLID Principles.

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

Design Principles

↳ Rules/Guidelines

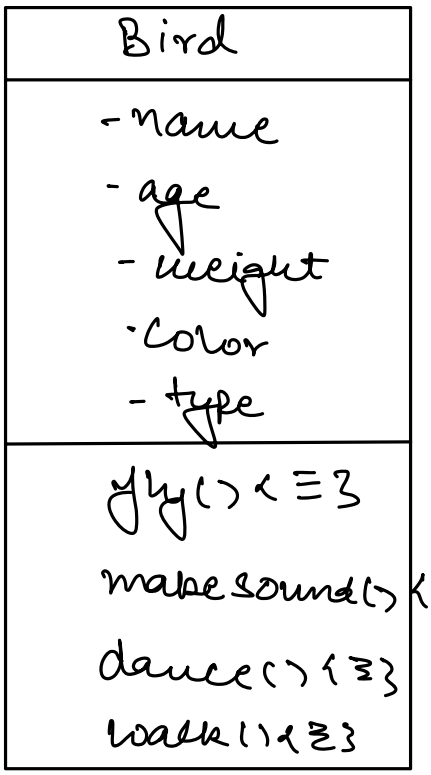
⇒ Set of rules/guideline that we should follow in order to design better Software Systems.

Properties of a good S/w System.

1) Extensible
2) Maintainable
3) Readable
4) Modular
5) Easily testable

Design a Bird

Design a Software System where we need to store all the Species of Bird in a Object Oriented manner.




```
Bird b1 = new Bird();
b1.setName(-);
b1.setType(-)
=
b1.fly();
```

```
Bird b2 = new Bird();
b2.setName(-);
b2.setType(-)
=
b2.fly();
```

⇒ Every bird fly in a different.

```
void fly(type){
    if(type == "Crow"){
        ==
        ==
        ==
    }
    else if(type == "Sparrow"){
        ==
        ==
        ==
    }
    else if(type == "Eagle"){
        ==
        ==
        ==
    }
    .
    .
    .
}
```

⇒ Too many if-else conditions

Issues

1) Readability
2) Testing
3) Code Duplication. ⇒ DRY (Don't Repeat Yourself).
4) less code reusability.
5) Violates SRP.

(Single Responsibility Principle)

⇒ fly() : responsible for multiple birds to fly.

Single Responsibility Principle

Every code unit (Class | Interface | Method) in our codebase should have exactly 1 responsibility.

There should only be single reason to change.

⇒ Bird class is responsible to hold attrs and behaviours of all the type of Birds. Ideally Bird class should only contain the general attrs | behaviours of Birds & specific details should go to the respective classes.

How to identify the violation of SRP.:

1) Lot of if-else conditions.

```
CheckPrimeNo(—) {
    if (divisors - - -) {
        ===
    }
    else {

    }
}
```

2) Monster Method.

Method which does more that what its name suggests.

(I)

```
void saveToDB(User user) {
    String q = "_____";
    Database db = new Database();
    db.setURL(_);
    db.connect();
    db.executeQuery(q);
}
```

⇐ Monster Method.

(II)

```
void saveToDB(User user) {
    String q = createQuery(User);
    Database db = createDBInstance();
    db.execute(q);
}
```

Note: SRP violation can be noticed wherever there's code duplication.

3) Common | Utils.

Utils |
- StringUtil.java ⇐ } String methods
- DateUtil.java ⇐ } Date methods.
- DBUtil.java ⇐

Summary of SRP-

Every code unit should have a single reason to change (or) Single responsibility.

→ too many if-else
→ Monster method
→ Common | Util

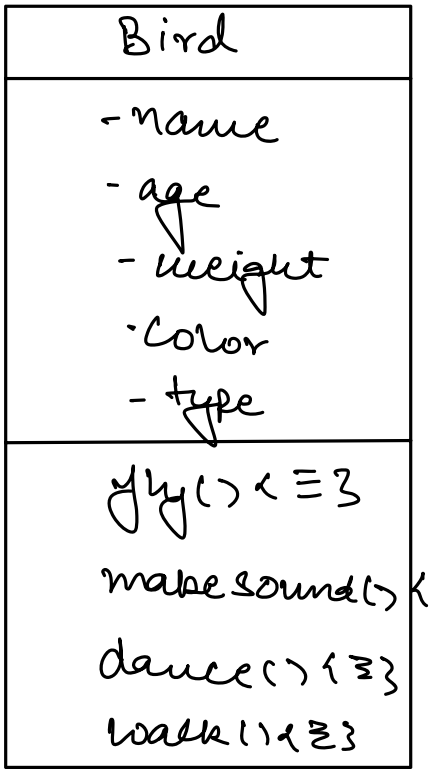
OCP (Open Close Principle)

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

⇒ Our codebase should be easily extensible but to add new functionalities it should'nt require modifications in existing codebase.

⇒ Rather than modifying the existing codebase we should be adding new code units.

⇒ Adding new feature in a code should require very less to zero changes in the existing codebase.



Crow

Sparrow

Owl.

→ New Bird

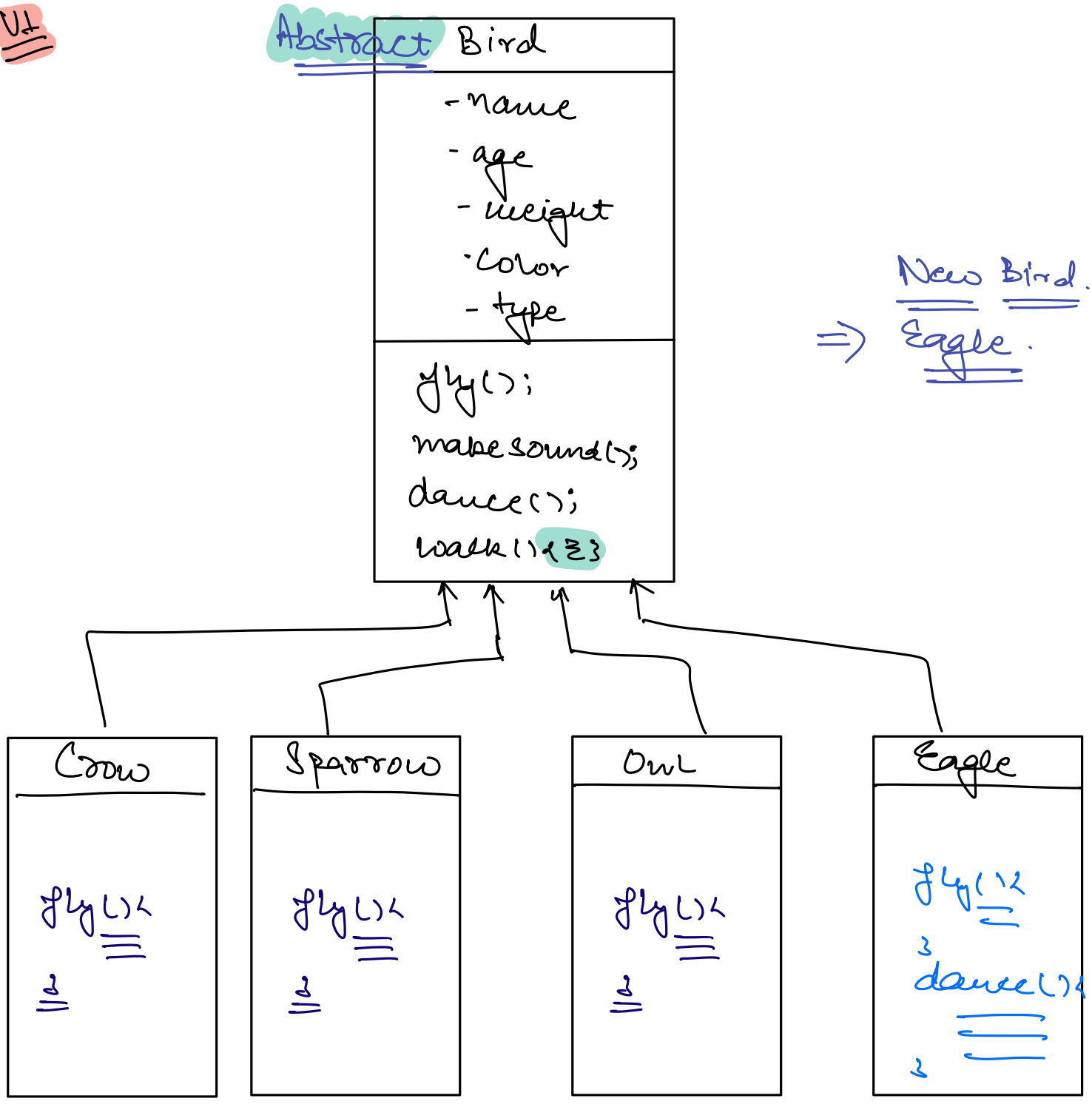
Eagle

⇒ OCP X.

Why OCP. ?

⇒ Testing & Regression.

UI



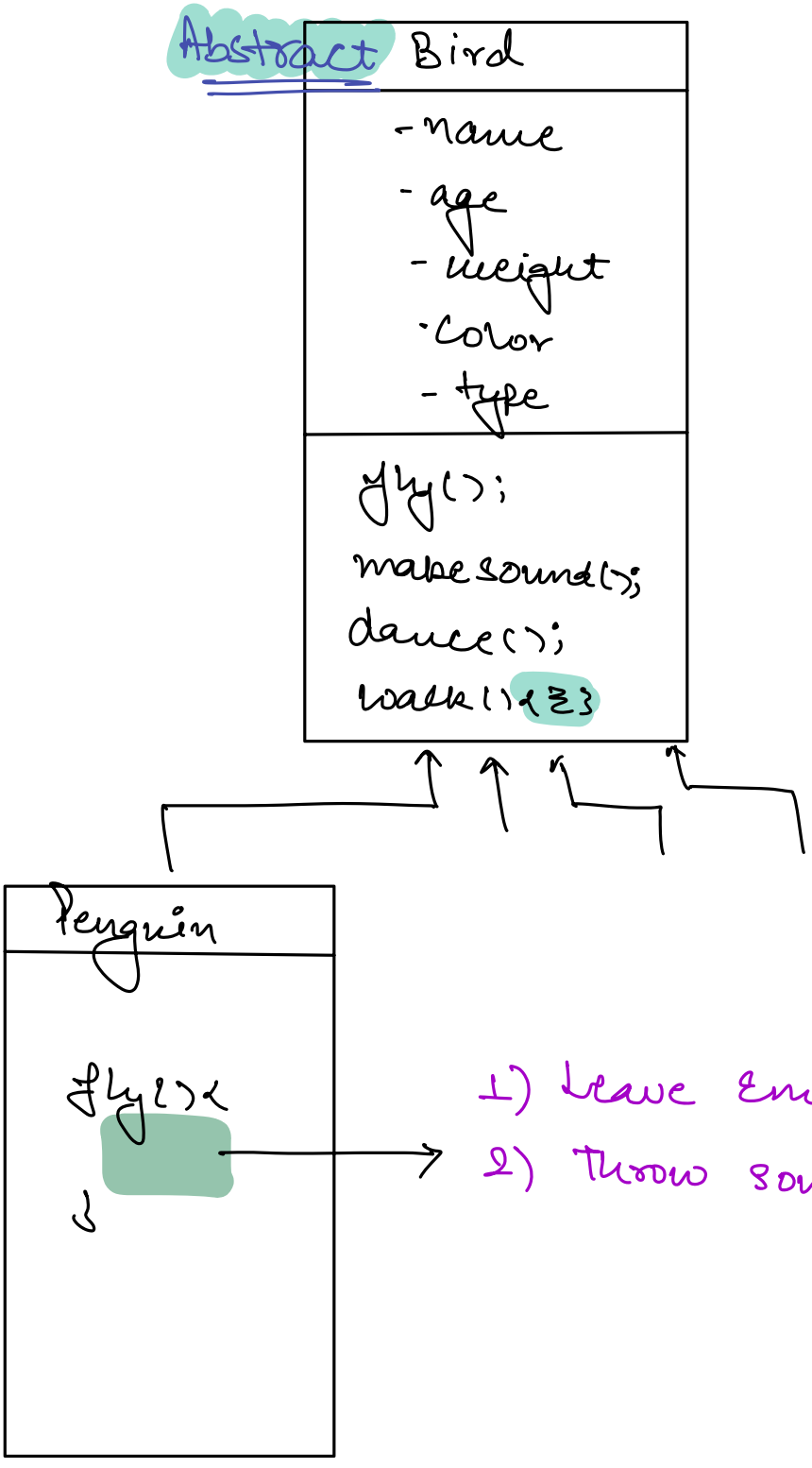
New Bird.
⇒ Eagle.

1) To add a new Bird, we just need to create a new child class.
2) Every Bird class is having single responsibility
3) Bird Class has very less reasons to change.

Requirement

Add a new Bird Penguin

↳ Can't fly



Client {

psvm() {

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

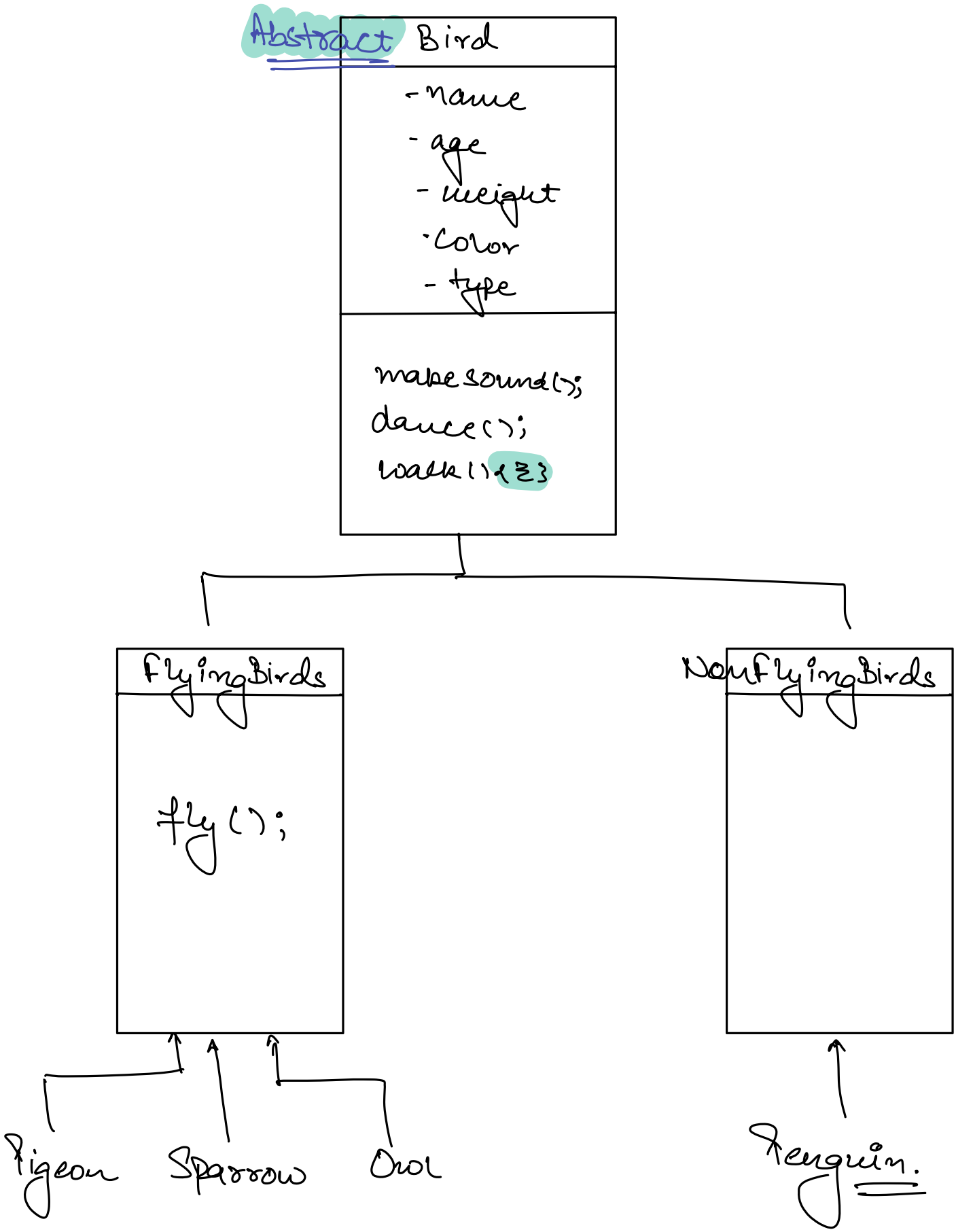
↳ Unexpected behaviour.

}

⇒ Try to reduce surprises for Client.

⇒ Ideally if a behaviour is not supported by a specific type of bird then that bird should not contain that behaviour.

Bird < flying Birds
Non flying Birds



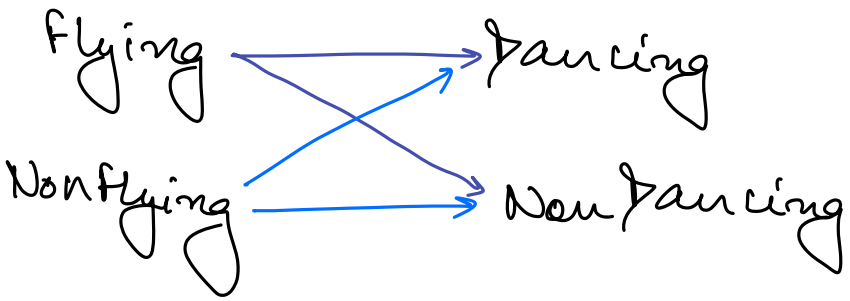
Bird b = new Penguin();

List<Bird>

Bird b = new Owl();

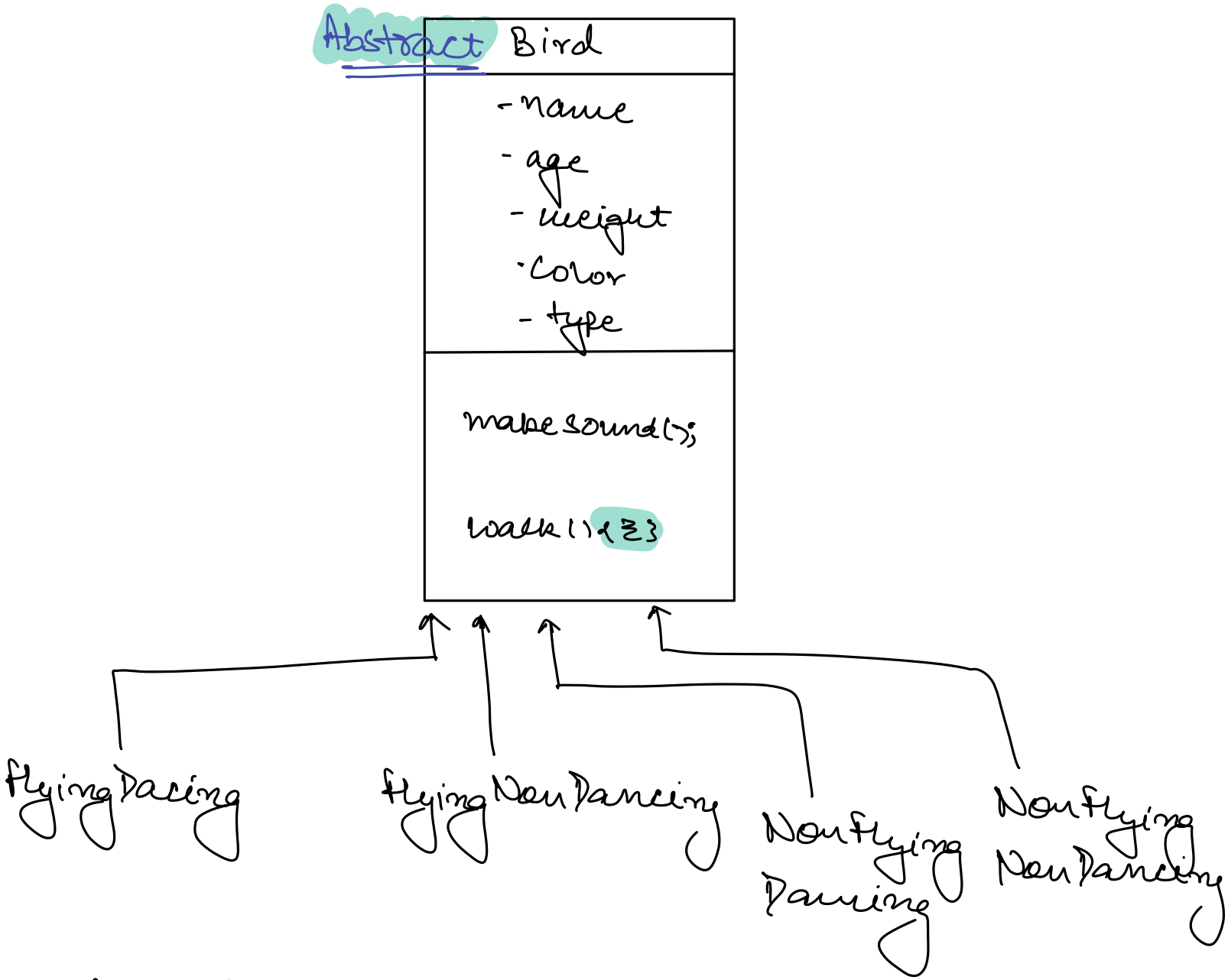
(Owl) b.fly(); ✓

⇒ Some Birds can dance, some birds can't dance.



4 categories of Birds.

N behaviours ⇒ 2^N type of birds.



⇒ Class Explosions.

↳ Too many classes.

Sol^{n} : Interfaces.

———— * ————