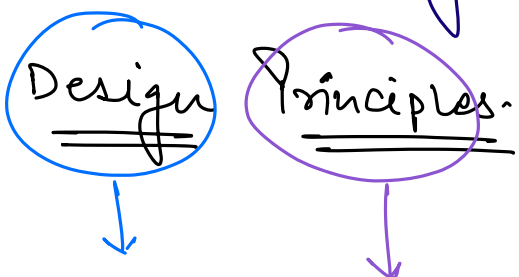
Agenda

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

SOLID Design Principles.



⇒ Set of rules/guidelines defined in order to design software system that will have following properties.

1. Extensible
2. Maintainable
3. Understandable/Readable
4. Reusable
5. Modular.

DRY
⇒ Don't Repeat Yourself

Disclaimer: LLD is Subjective.

→ There's NO single right answer.

Design a BIRD.

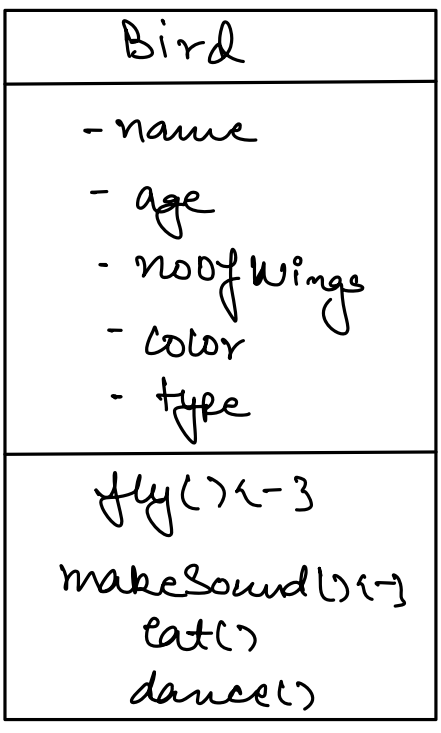
↳ Amazon Interviews.

⇒ Problem Statement

Build a Software System where we can store all the type of Birds.

⇒ Diversity of Birds.

V1



```
Bird b1 = new Bird();
b1.setName(—);
b1.setAge(—)
b1.setType("Crow");
```

⇒ b1.makeSound()

```
Bird b2 = new Bird();
b2.setName(—);
b2.setAge(—)
b2.setType("Pigeon");
```

⇒ b2.makeSound()

```
Void makeSound(type) {
    if(type == "Pigeon"){
        ——
        ——
        ——
    }
    else if(type == "Crow"){
        ——
        ——
        ——
        ——
        ——
        ——
    }
}
```

Problems with TOO MANY if-else conditions.

1. Understandability.
2. Difficult to test
3. Code duplicacy
4. No code reusability
5. Violates (S) of SOLID.

Single Responsibility Principle.

Every code unit (Class | Method | interface) in our codebase should have a single responsibility.

↓

There should be a single reason to change.

⇒ makeSound(—) is responsible for every bird to make a sound.

How to identify violation of SRP.

1. Method with too many if-else conditions.

⇒ Not always true.

⇒ Algorithm / Business logic.

```
CheckIfLeapYear(—){
    if(—){
        ——
    }
    else if(—){
        ——
    }
    ——
}
```

2. Monster Method.

↓

When a method does lot more things than what its name suggests.

```
saveToDatabase(User user){
    String query = "insert into users
                    ____
                    ____
                    ____";                    } (1)

    Database db = new DB()
    db.setURL(—);
    db.createConnect();                       } (2)

    db.execute(query)                         ] (3)
}
```

⇒

```
saveToDatabase(User user){
    String query = createQuery(—);
    CreateDBConnection()

    // Execute
}
```

Note: SRP ensures code reusability.

③ Commons | Utils.

→ Discouraged.

/utils
  | StringUtils.java
  | DateUtils.java
  | StudentUtils.java

SRP Summary

↓

Single Responsibility Principle.

① Too many if-else
② Monster method
③ Common | Utils.

8:30 Am.

Open Close Principle. (OCP)

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

extension → extensibility

⇒ Our codebase should be easily extensible but to add new feature we shouldn't require to change the existing code files.

adding new feature.

⇒ Rather than modifying the existing code we should try to add new code units.

⇒ Adding a new feature in our codebase should require very less changes in the existing codebase.

codebase → Project.

Bird
- name - age - noOfWings - color - type
fly() {-3 makeSound() {-} eat() dance()

⇒ Till now.

Sparrow, Crow, Pigeon

⇒ Add a new type of Bird

Peacock.

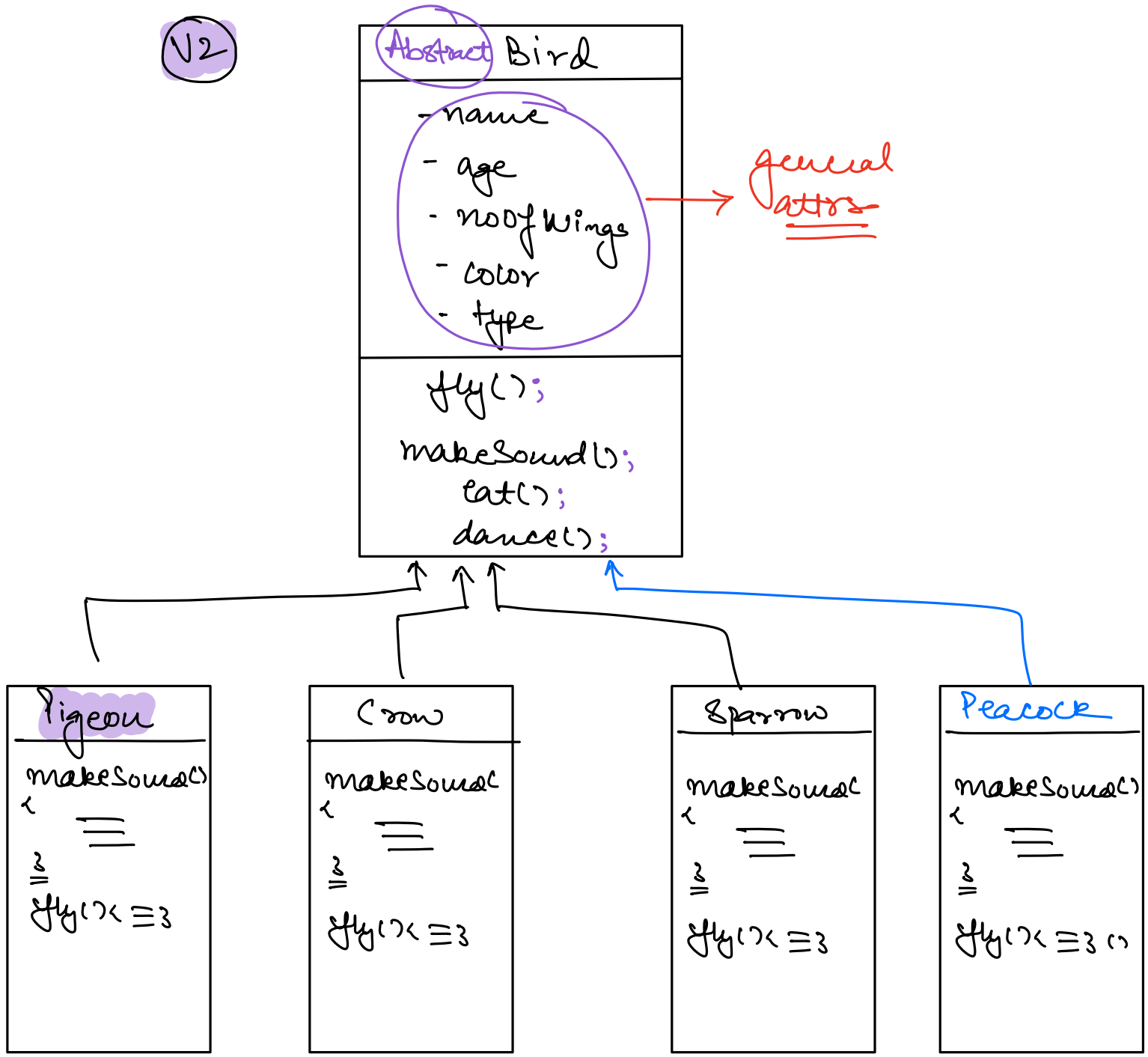
Why OCP ?

⇒

1. Existing code might start breaking.
2. Testing
   - Regression. (QA).

Sol4

Let the Bird class be only responsible for storing the general attributes/methods for Bird.



⇒ Too many if-else conditions. ✗

⇒ Req: Add a new type of Bird Peacock.

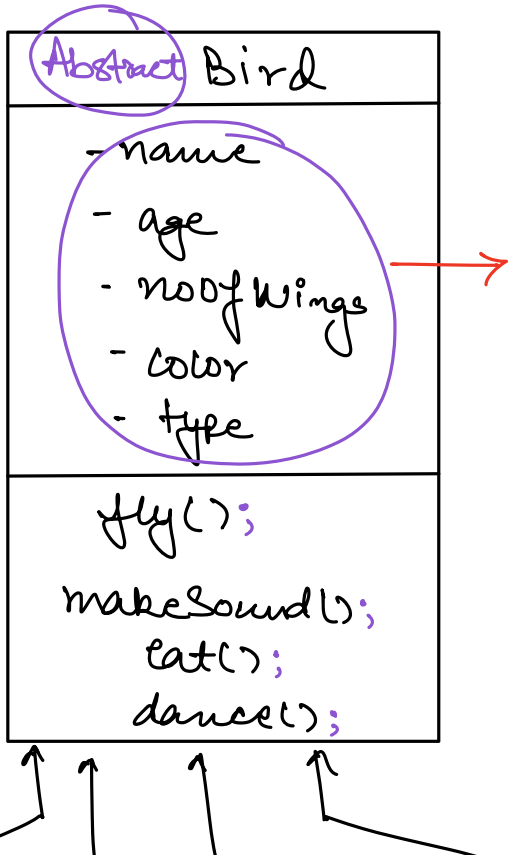
① To add a new Bird, we just have to create a new without making code changes in the existing codebases.

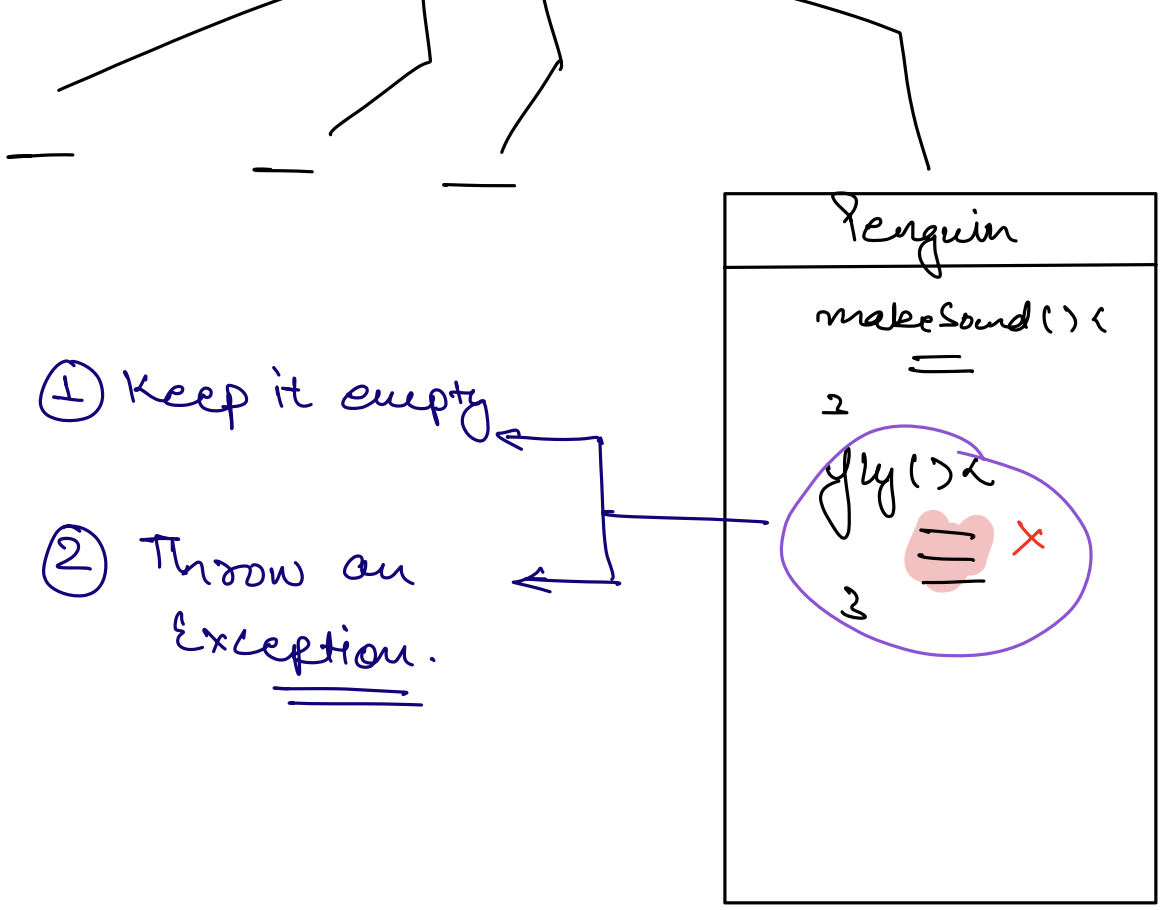
② Bird class is only responsible for general bird behaviours.

Requirement -

⇒ Add a new bird Penguin to the System.

Penguin → Can't fly





Client {

Bird b = new Penguin()

b.fly() ⇒ Unexpected behaviour.

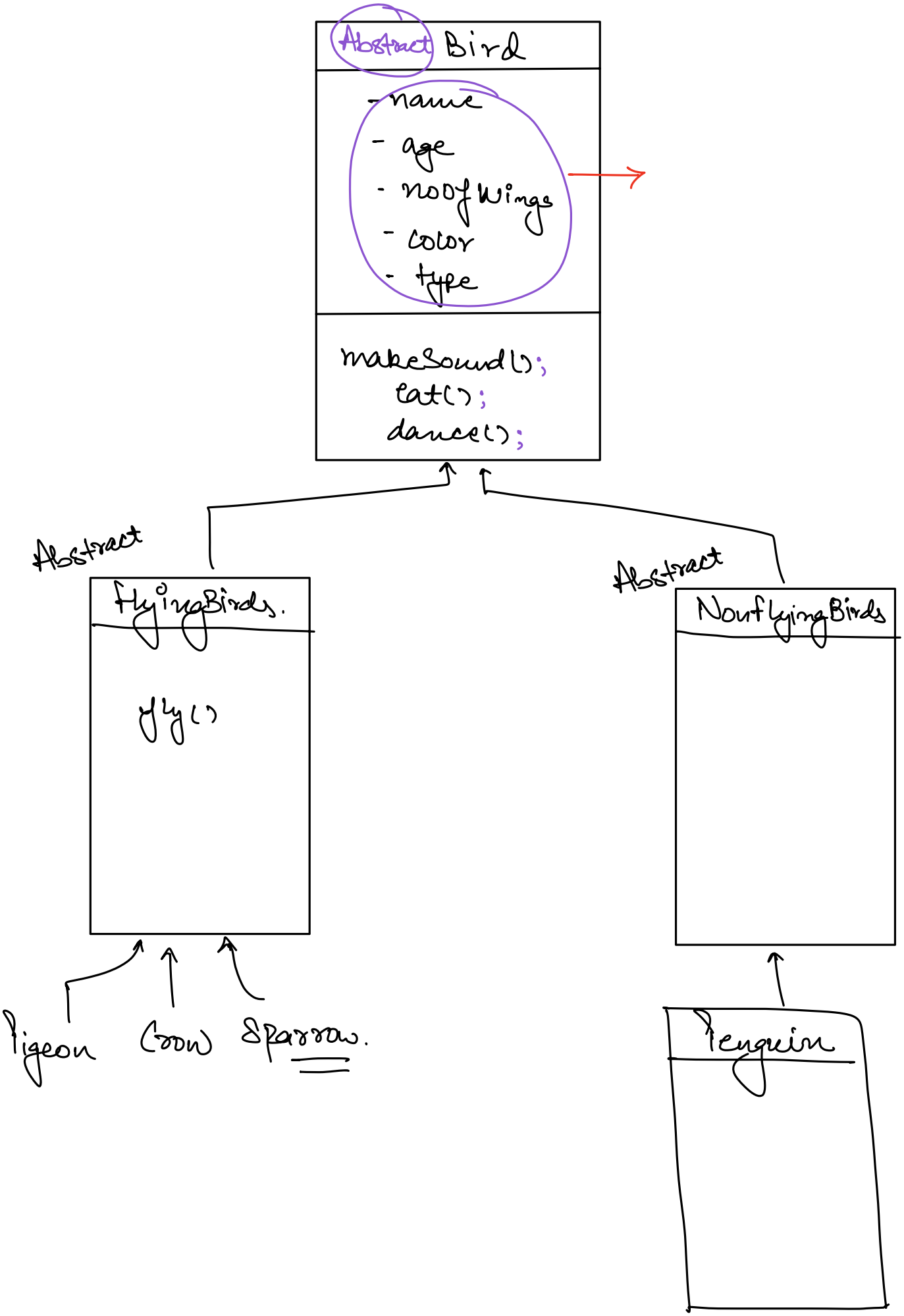
}

⇒ Client will get surprised.

⇒ Never give surprises to Client.

Ideal Soln

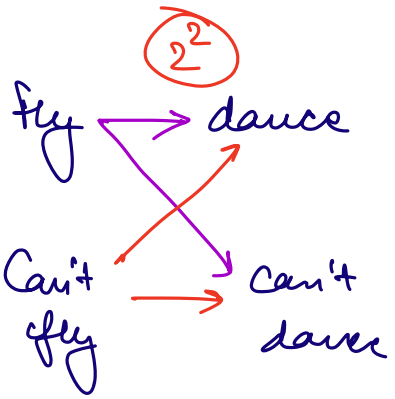
⇒ If a behavior is NOT supported by a particular type of object then that behavior shouldn't be available.

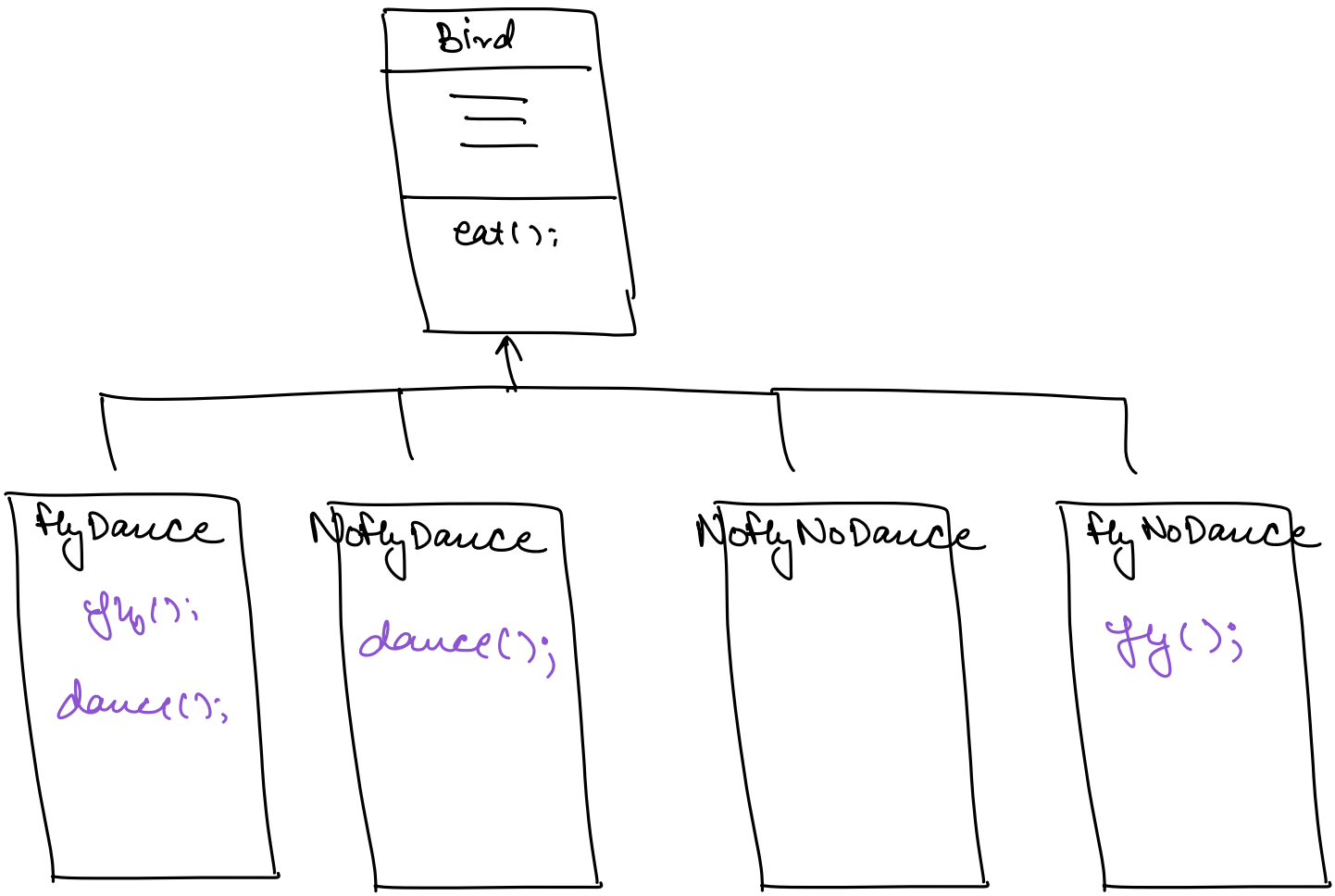

Abstract Bird
- name
- age
- noOfWings
- color
- type
makeSound();
eat();
dance();
Abstract
FlyingBirds.
fly()
Abstract
NonflyingBirds
Pigeon
Crow
Sparrow.
Penguin

printNames(List<Bird> birds)

⇒ fly | Can't fly

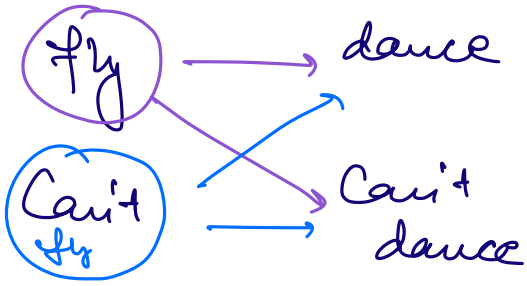
Dance | Can't Dance.

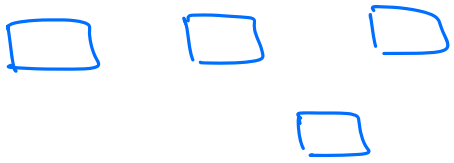




Problems.

⇒ Class Explosion : Too many classes.





1000 Birds

1000 class

⇒ Pigeon (P) = new Pigeon

P.fly();

Bird b = (——) new Pigeon;

b.fly(); ×