System Design Principles

1. DRY
2. KISS
3. SOLID
4. CUPID

DRY → Dont Repeat Yourself

KISS → Keep it simple Stupid

SOLID

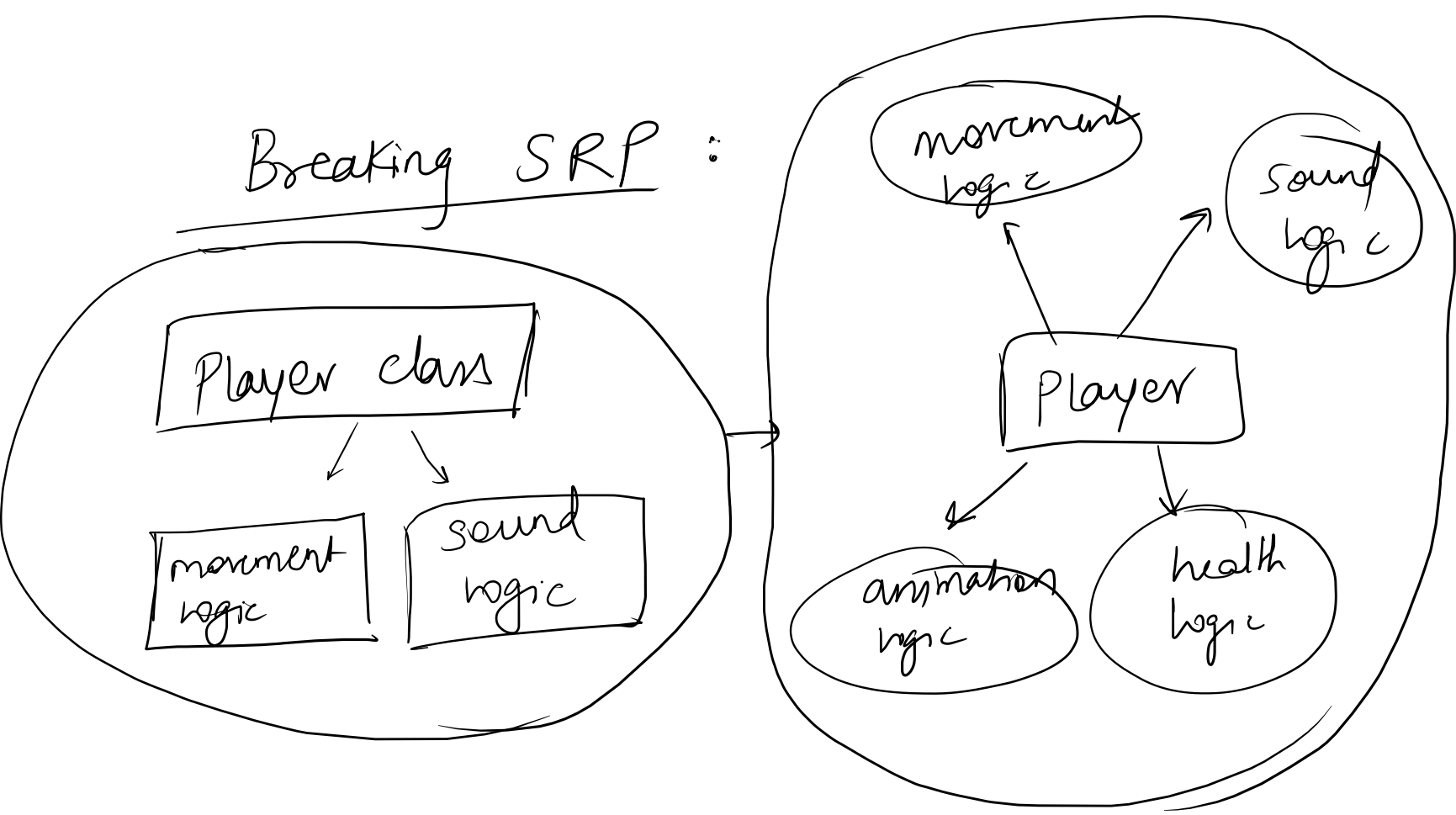
↳ helps to plan and execute a project

* more understandable
* more maintable

1. S - Single Responsibility Principle
2. O - Open / Closed Principle
3. L - Liskov Substitution principle
4. I - Interface Segregation principle
5. D - Dependency Inversion principle

① Single Responsibility Principle

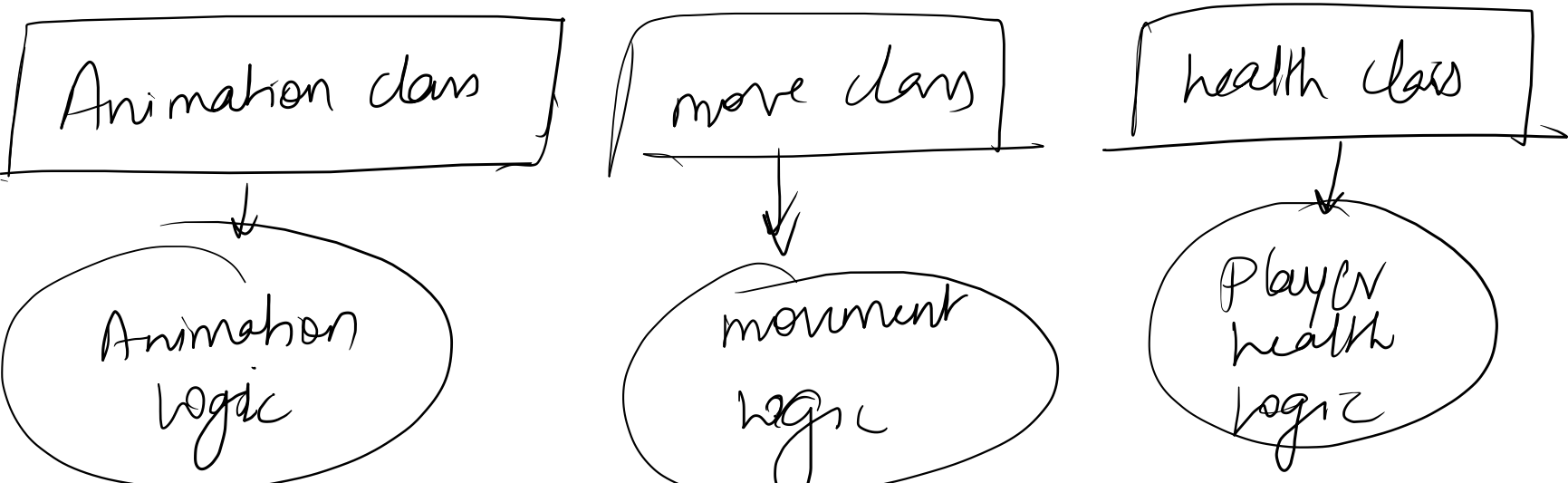
↳ A class should have only one responsibility.


Breaking SRP :
Player class
movement logic
sound logic
movement logic
sound logic
Player
animation logic
health logic

fix big classes

→ Refactoring

→ Break 1 big class into multiple smaller classes.


Animation class
move class
health class
Animation logic
movment logic
Player health logic


Player class
movement class
Animation class
Health class

Pros:

- Breaking code into multiple classes
  - Less code to read for your class.
  - easier to maintain the class

* Reusability
  each class has 1 responsibility.
  other classes can use when required.

Cons :

→ No of classes & files will increase
→ affects folder structure
→ breaking up classes unnecessarily can lead to issues.

② open closed Principle

↳ classes / modules / functions should be open for extension, but closed for modification.

Inheritance

→ allows us to change functions and properties without affecting the base class.

* **method overriding**

we can override in the subclass.
(superclass not affected)

Pros :

(1) Allows reusability

(2) classes can use 'extend functionality of superclass without modifying it.

Pros :

(1) Allows reusability

(2) classes can use extend functionality of superclass without modifying it.

(3) Liskov Substitution Principle

→ objects of the superclass should be replacable with the objects of its subclasses without breaking the application

→ make sure that child class functions are 100% compatible with parent class.

parent

```
{
    int addition(int a, int b)
    {
        return a+b;
    }
}
```

child extends parent

```
{
    String addition(int a, int b)
    {
        int sum = a+b;
        return sum.toString();
    }
}
```

```
int ans = p.addition(10, 20);
```

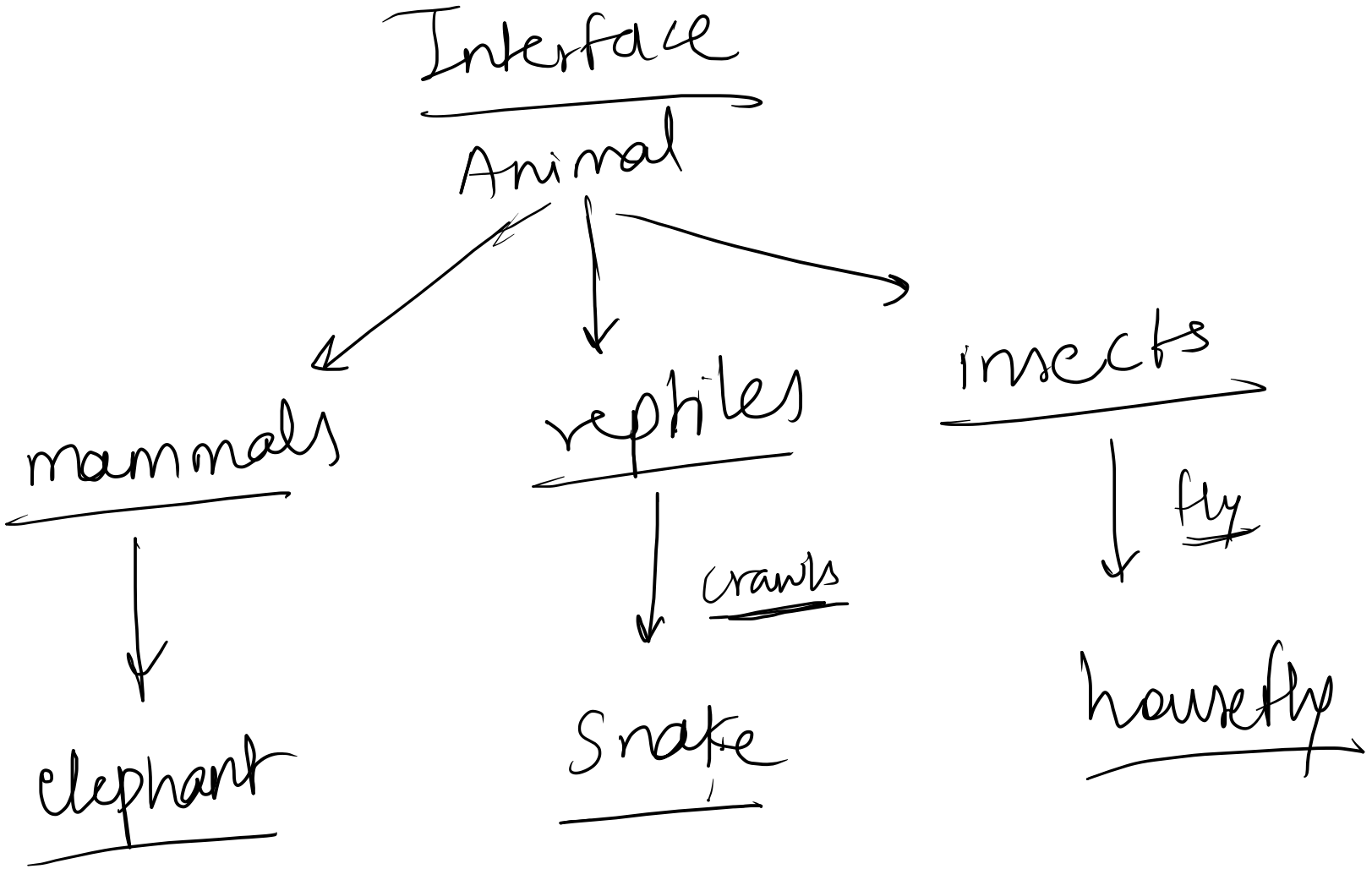
④ Interface Segregation Principle

↳ programmers should not be forced to depend upon interfaces that they do not use.

Interface Animal

run() ✓
eat() ✓
swim() -
fly() ✗
bark() ✗

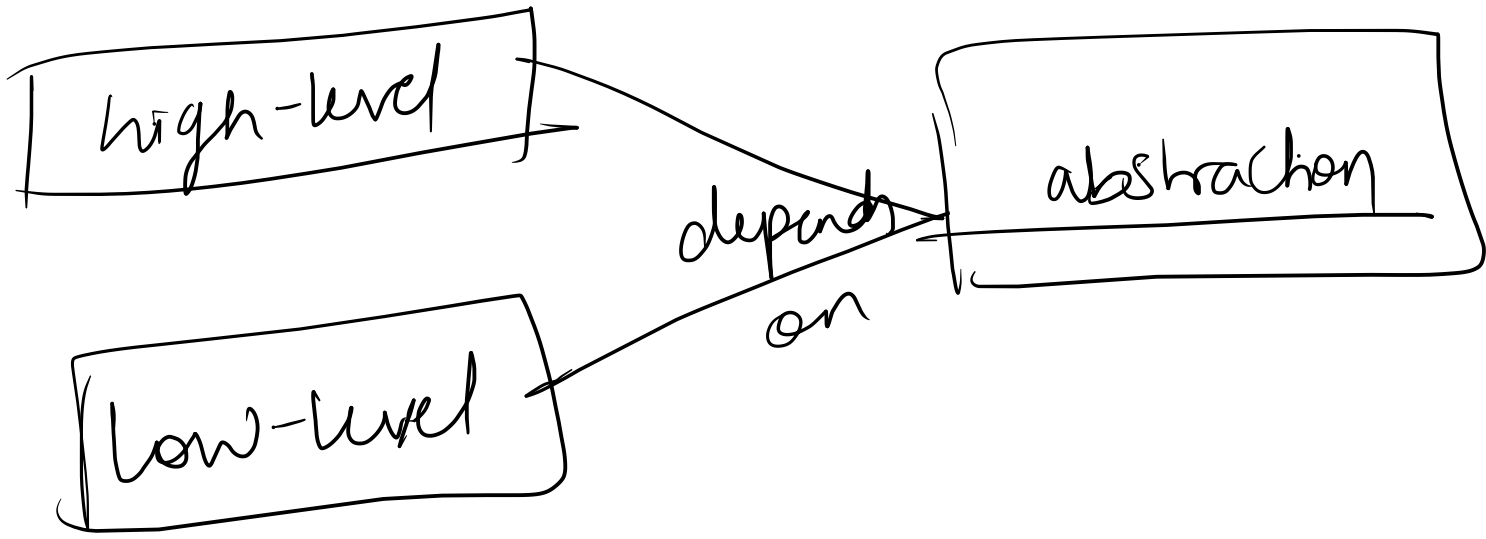
→ Cat implements animal

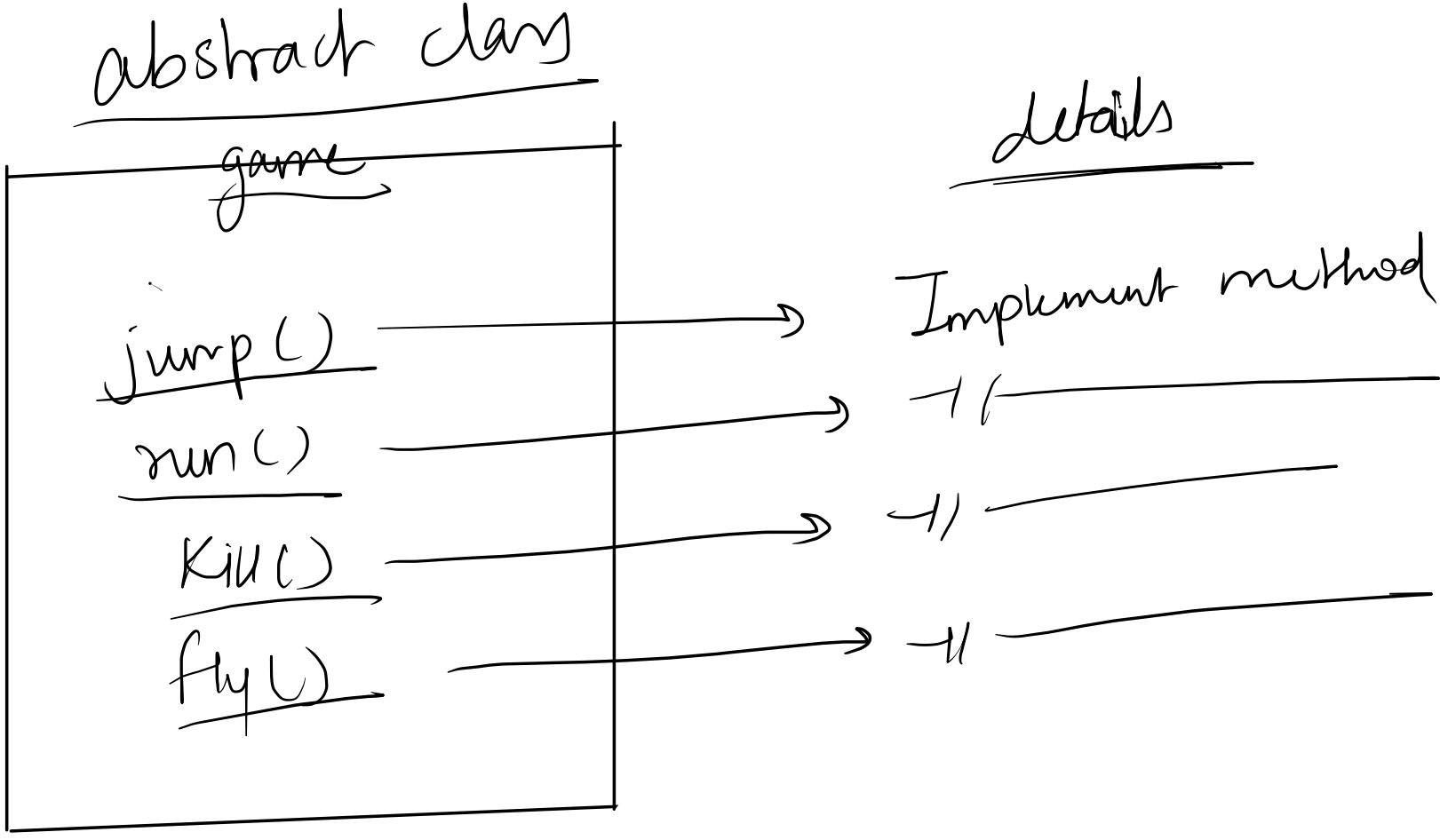

Interface
Animal
mammals
reptiles
insects
fly
crawls
elephant
Snake
housefly

(5) Dependency Inversion Principle

1.) High level classes should not depend on low level classes.
both should depend on abstractions.

2.) Abstractions should not depend on details,
details should depend on abstractions


high-level
low-level
depends on
abstraction


abstract class
game
jump()
run()
Kill()
fly()
details
Implement method
//
//
//