SOLID Principles [Class-2]

Special class

Love Babbar • Mar 15, 2024


Intro to SOLID Principles
Special class
Love Babbar • Mar 13, 2024

SOLID Principles

- Lakshay

Good coder :: Quality code

1. Reusable code
2. Extensible
3. Flexible
4. Stable → Exception Handling
5. Reliability
6. Modularity
7. Security
8. Correctness

functional

= ↓

Tree

=

fun()

↓

fun2()

↓

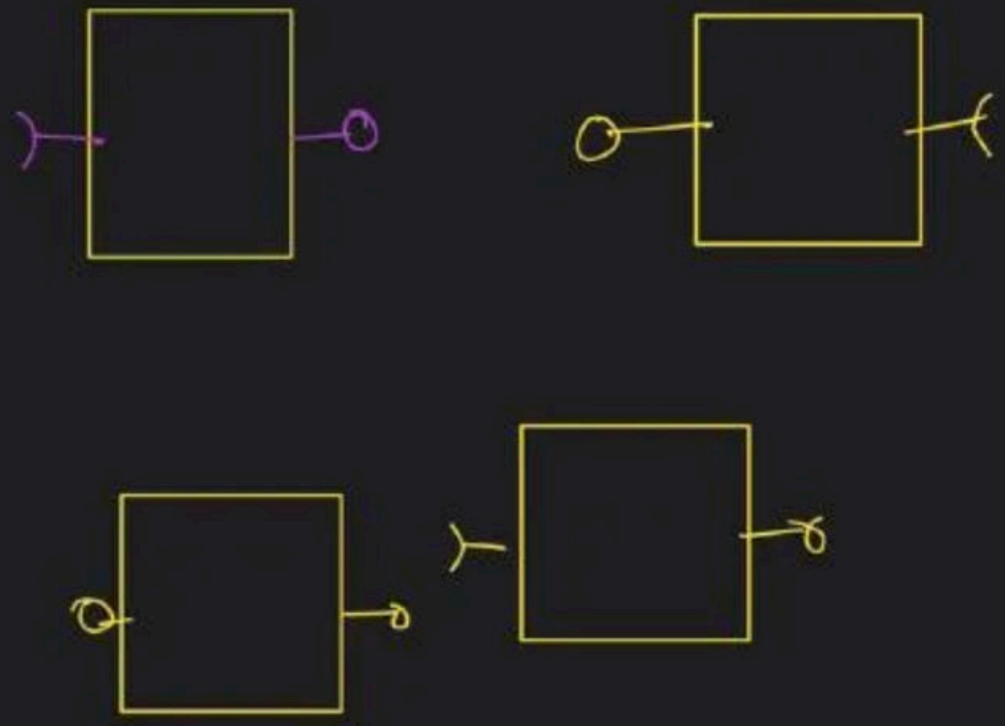
3()

↘ 4()

⋮

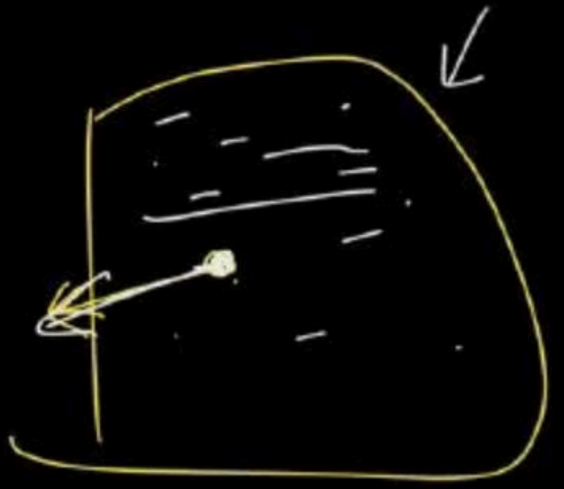
OO (Lego)

=



Purpose

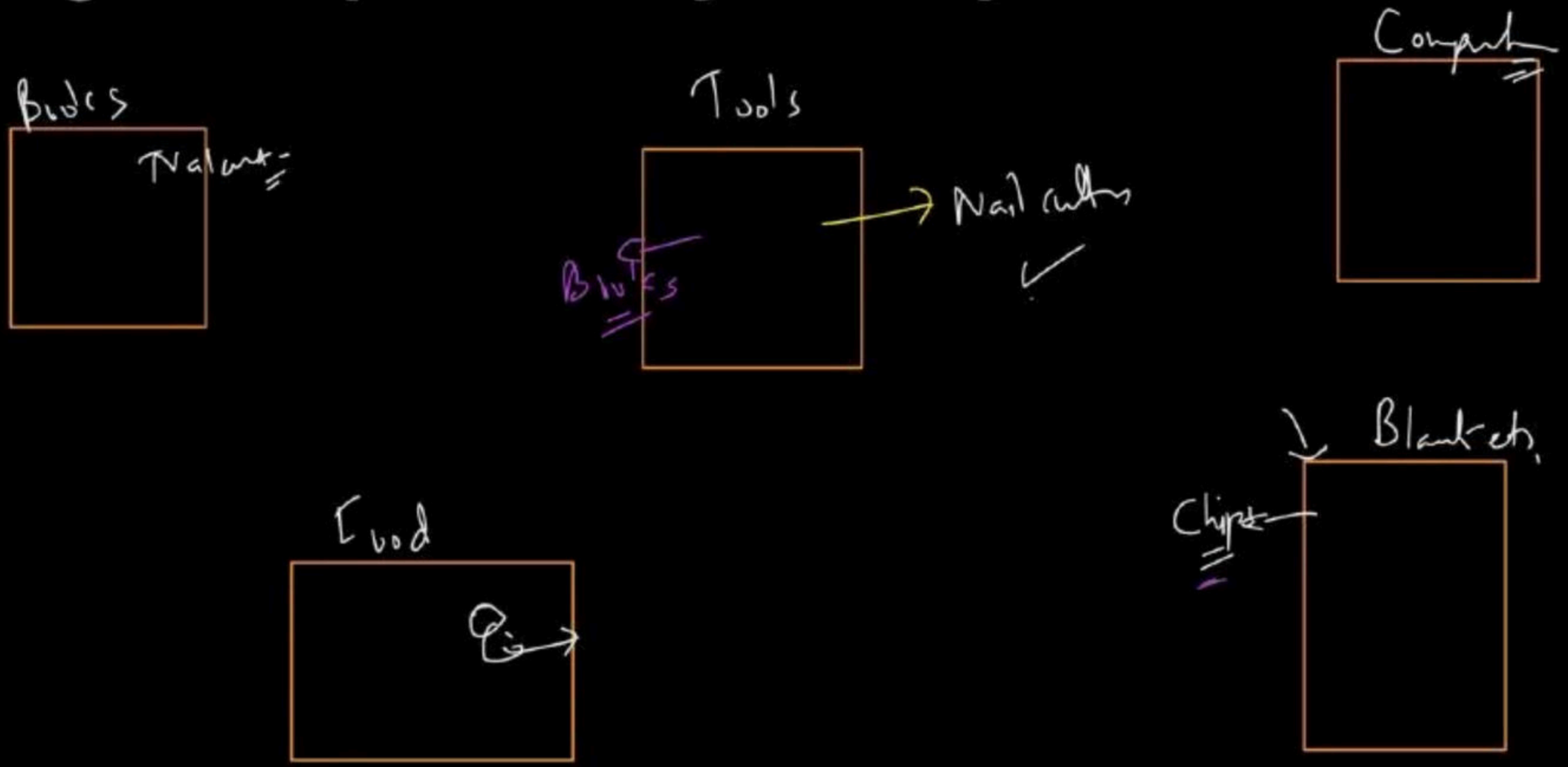
1. Introduced by Robert Martin (Uncle Bob), named by Michael Feathers.
2. To make code more maintainable, easy to reuse.
3. To make it easier to quickly extend the system with new functionality without breaking the existing ones.
4. To make the code easier to read and understand, thus spend less time figuring out what it does and more time actually developing the solution. (Time Saving)



Single Responsibility Principle

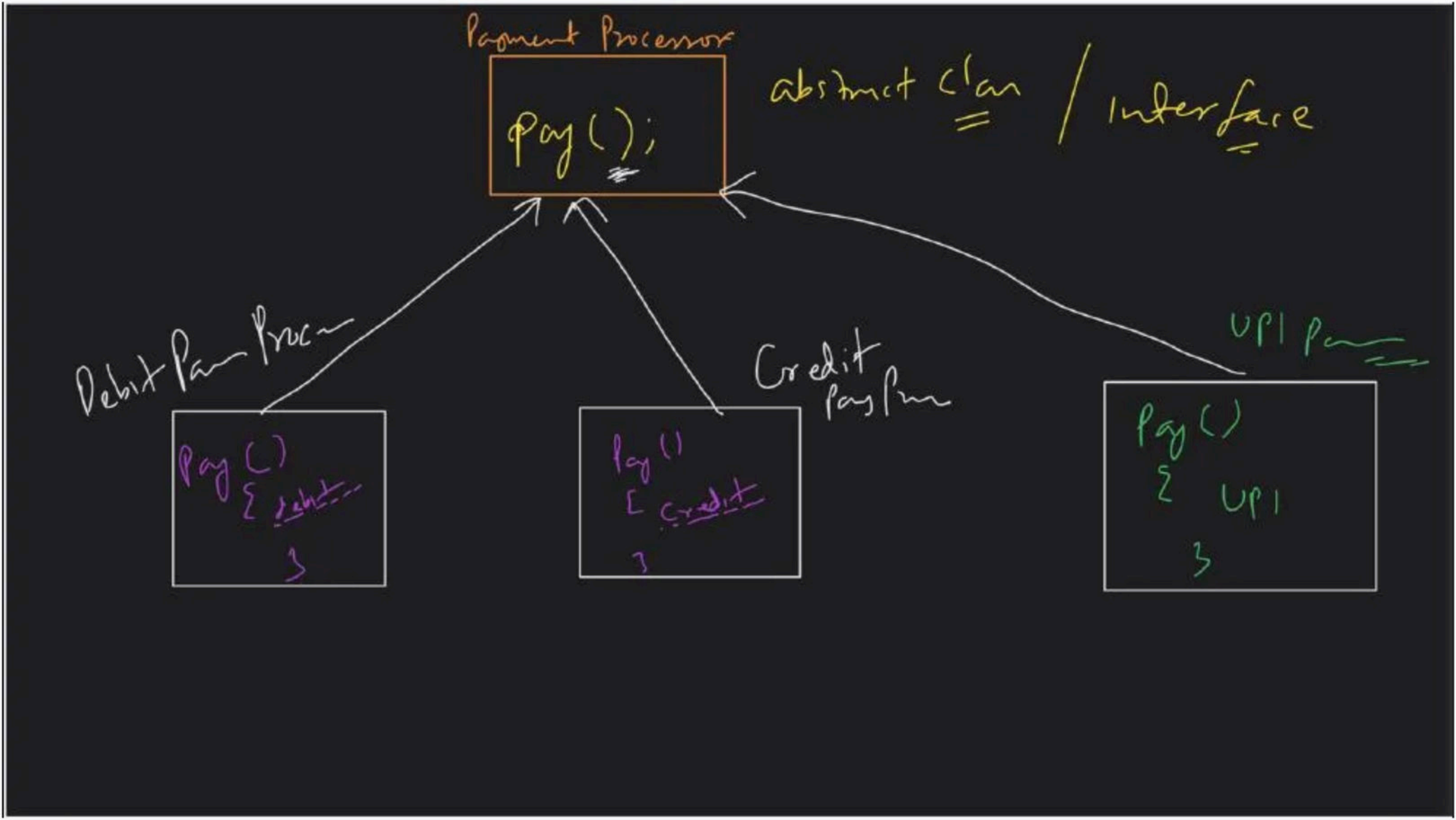
1. A class should have one, and only one reason to change. This means that a class should only have one job or responsibility.
2. A class should only be responsible for one thing.
3. There's a place for everything and everything in its place.
4. Find one reason to change and take everything else out of the class.
5. **Importance**: Following SRP makes your code more modular, easier to understand, maintain, and extend. It helps in isolating functionalities, making debugging and testing more straightforward.

Single Responsibility Principle

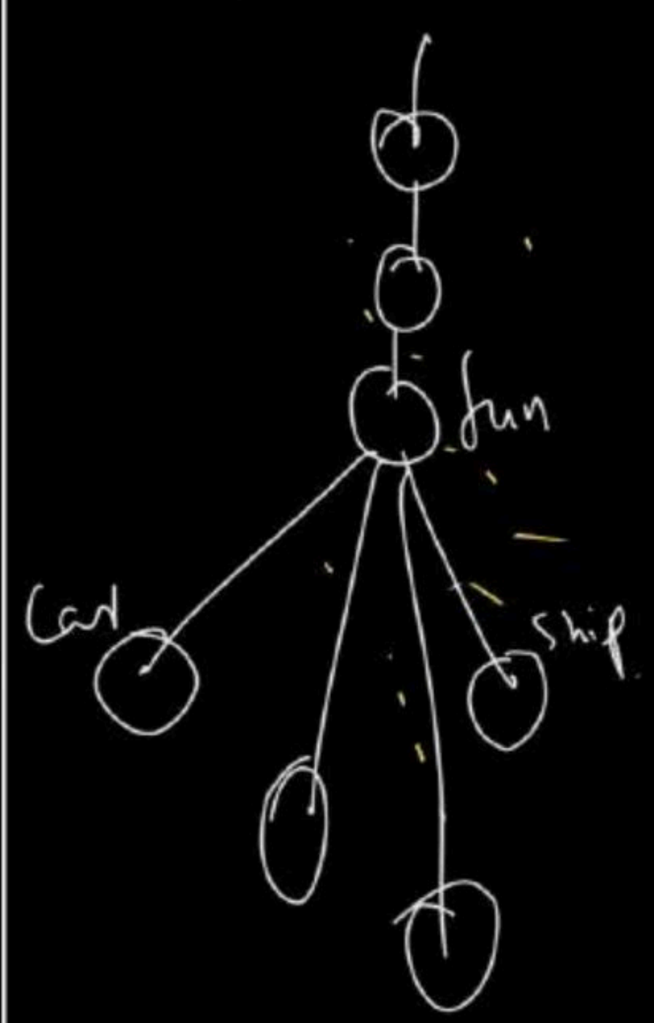


Open-Closed Principle

1. An entity should be open for extension but closed for modification. This means you should be able to add new functionality without changing the existing code.
2. Extend functionality by adding new code instead of changing existing code.
3. **Goal**: Get to a point where you can never break the core of your system.
4. **Importance**: OCP encourages a more stable and resilient codebase. It promotes the use of interfaces and abstract classes to allow for behaviors to be extended without modifying existing code.
5. Writing code structure in such a way new functionality can be added by adding new code not by modifying existing code.


Payment Processor
pay();
abstract class / interface
Debit Pay Proc
Credit Pay Proc
UPI Pay
pay()
{ debit
}
pay()
{ credit
}
pay()
{ UPI
}

Open-Closed Principle




```
for(Vehicle vehicle : vehicles) {
    switch(vehicle.getType()) {
        case CAR:
            vehicle.lock();
            vehicle.go();
            break;
        case SHIP:
            vehicle.balance();
            vehicle.swim();
            break;
        case AIRPLANE:
            vehicle.go();
            vehicle.fly();
            break;
        case TANK:
            vehicle.move();
            vehicle.stop();
            vehicle.fire();
            break;
    }
    vehicle.stop();
}
```

Bad

1. if, switch
2. Violate OCP
3. Cyclometric Complexity
4. Downcasting
5. Typechecking
   anti-ABS

Open-Closed Principle

```
1   do(Car v){
2       vehicle.lock();
3       vehicle.go();
4   }
5   do(Ship v){
6       vehicle.balance();
7       vehicle.swim();
8   }
9   do(Airplane v){
10      vehicle.go();
11      vehicle.fly();
12  }
13  do(Tank v){
14      vehicle.move();
15      vehicle.stop();
16      vehicle.fire();
17  }
18
19  execute(List<Vehicle> vehicles){
20      for(Vehicle vehicle : vehicles) {
21          do(vehicle);
22          vehicle.stop();
23      }
24  }
```

Compile X
↳ Time Error
X

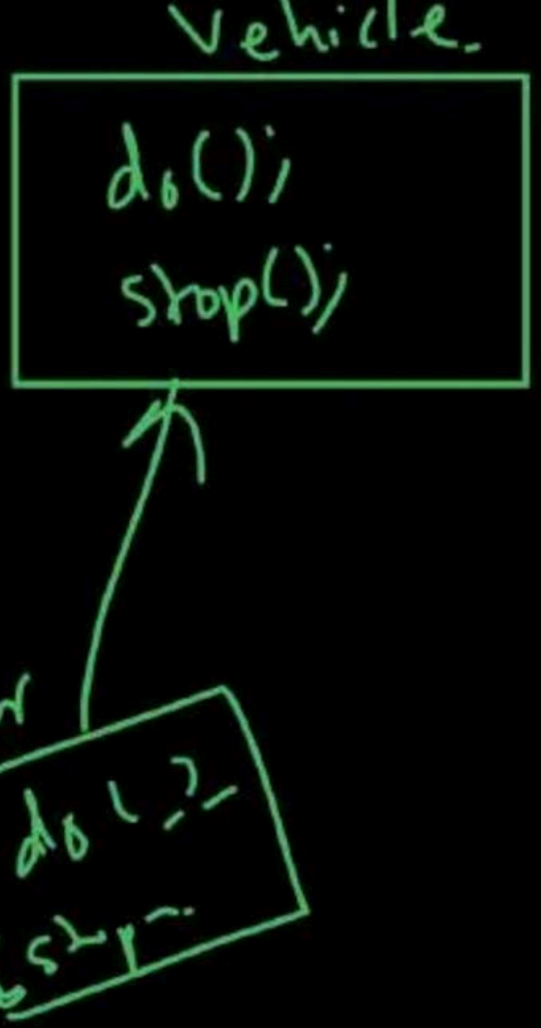
→ Down casting

Open-Closed Principle

```
1   do(Car vehicle){
2       vehicle.lock();
3       vehicle.go();
4   }
5   do(Ship vehicle){
6       vehicle.balance();
7       vehicle.swim();
8   }
9   do(Airplane vehicle){
10      vehicle.go();
11      vehicle.fly();
12  }
13  do(Tank vehicle){
14      vehicle.move();
15      vehicle.stop();
16      vehicle.fire();
17  }
18
19  execute(List<Vehicle> vehicles){
20      for(Vehicle vehicle : vehicles) {
21          if(vehicle.instanceof(Car))
22              do((Car) vehicle)
23          if(vehicle.instanceof(Tank))
24              do((Tank) vehicle)
25
26          ....
27      }
28  }
```

Downcasting

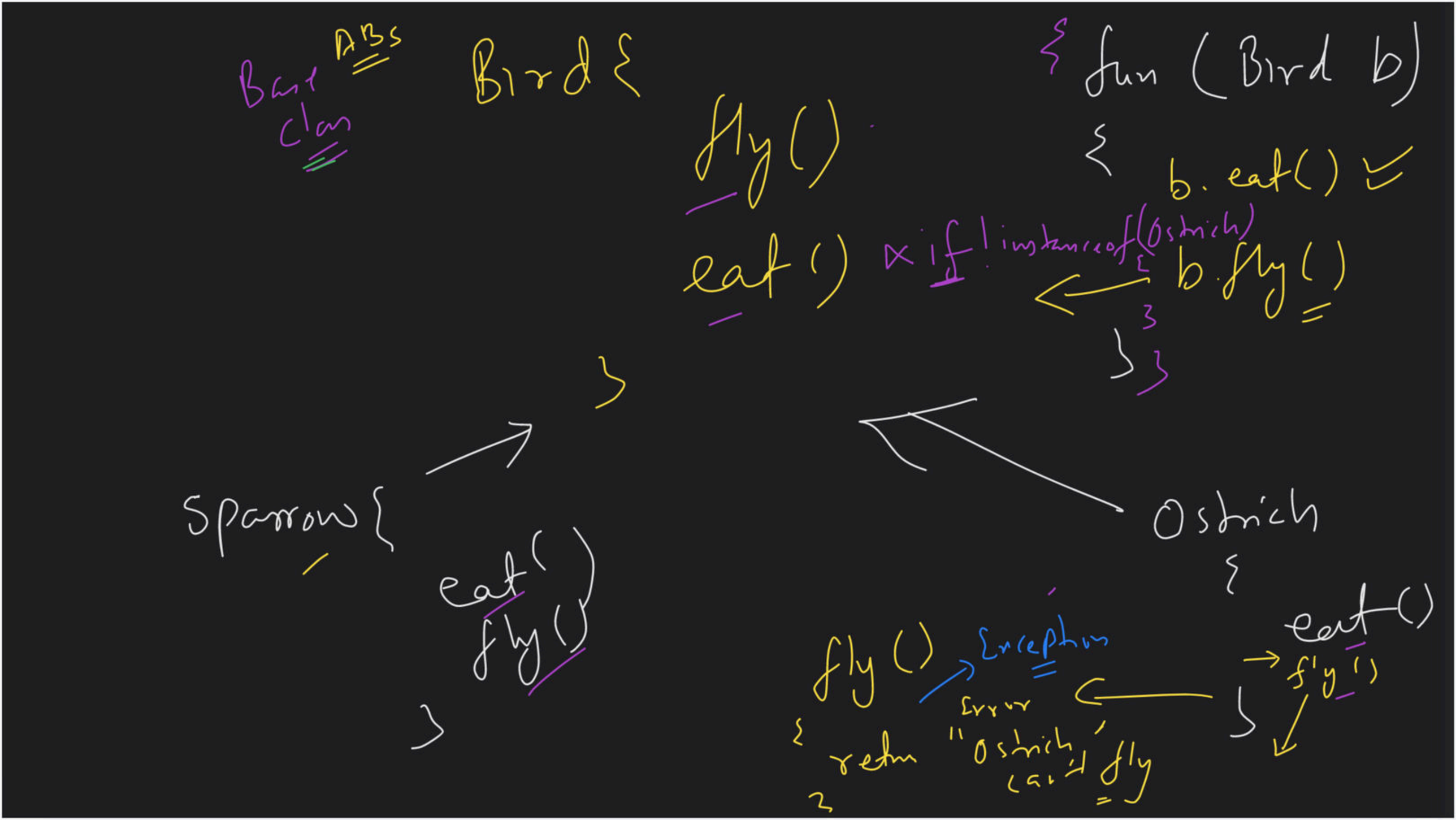
Open-Closed Principle




```
interface Vehicle{
  do();
  stop();
}
class Car implements Vehicle{
  do(){
    lock();
    go();
  }
  ...
}
class Ship implements Vehicle{
  do(){
    balance();
    swim();
  }
  ...
}
class Airplane implements Vehicle{
  do(){
    go();
    fly();
  }
  ...
}
class Tank implements Vehicle{
  do(){
    move();
    stop();
    fire();
  }
  ...
}

...(List<Vehicle> vehicles){
  for(Vehicle vehicle : vehicles) {
    vehicle.do();
    vehicle.stop();
```

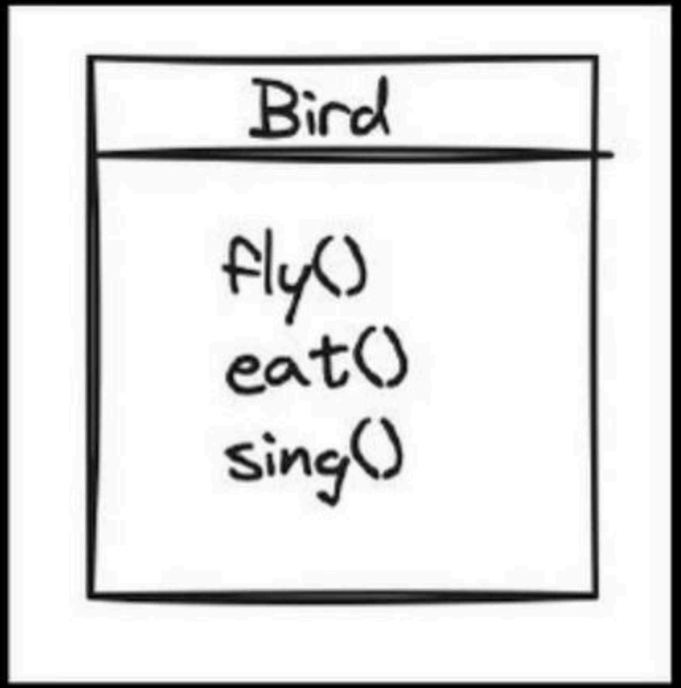
Liskov Substitution Principle

1. Any derived class should be able to substitute its parent class without the consumer knowing it.
2. Every part of the code should get the expected result no matter what instance of a class you send to it, given it implements the same interface.
3. If a function takes a Base class as parameter then, this code should work for all the derived classes.
4. LSP insures that the good application i.e., built using abstraction does not break.
5. It states that the objects of a subclass should behave the same way as the objects of the superclass, such that they are replaceable.
6. Child class should be able to do what a parent class can.
7. **Goal**: The goal of LSP is to ensure that a subclass can stand in for its superclass. This principle helps in maintaining the correctness of the program when objects of a superclass are replaced with objects of a subclass.


Bird clan
ABS
Bird {
fly()
eat()
}
Sparrow {
eat()
fly()
}
{ fun (Bird b)
{
b.eat()
if (!instanceof(Ostrich)
{
b.fly()
}
}
}
Ostrich
{
fly()
Exception
Error
{
return "Ostrich can't fly"
}
eat()
fly()
}

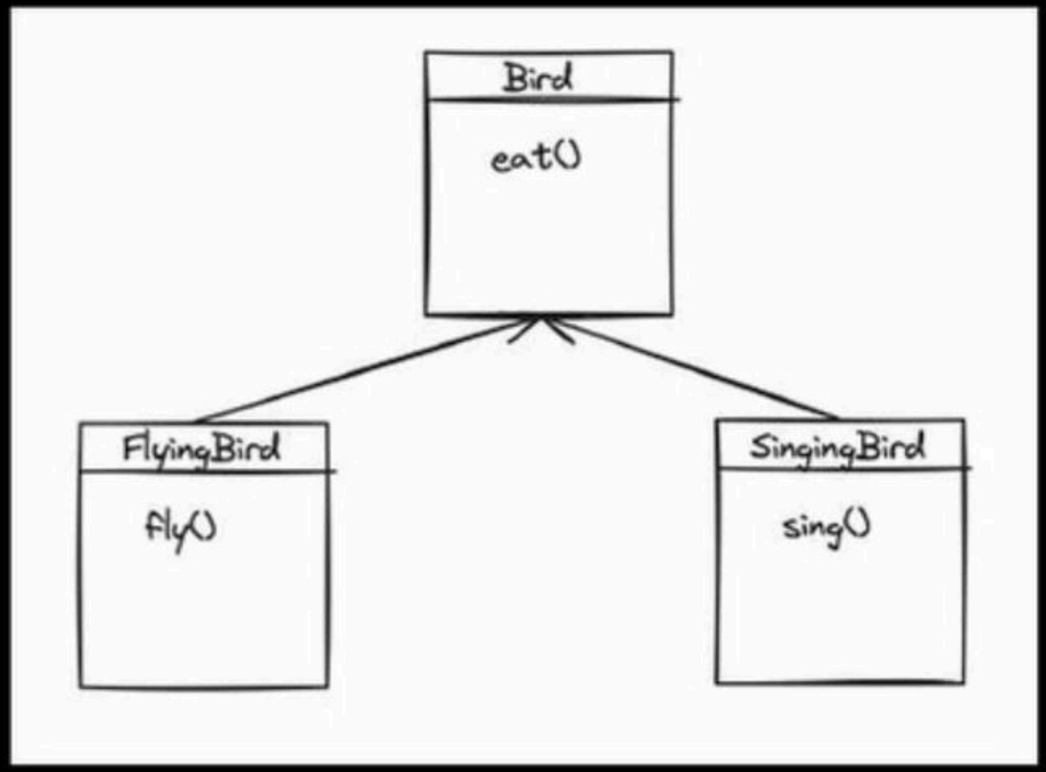
Liskov Substitution Principle

1. Bird Example



Liskov Substitution Principle

1. Bird Example



Bird {

eat();

}

IFlyingBird {

fly()

}

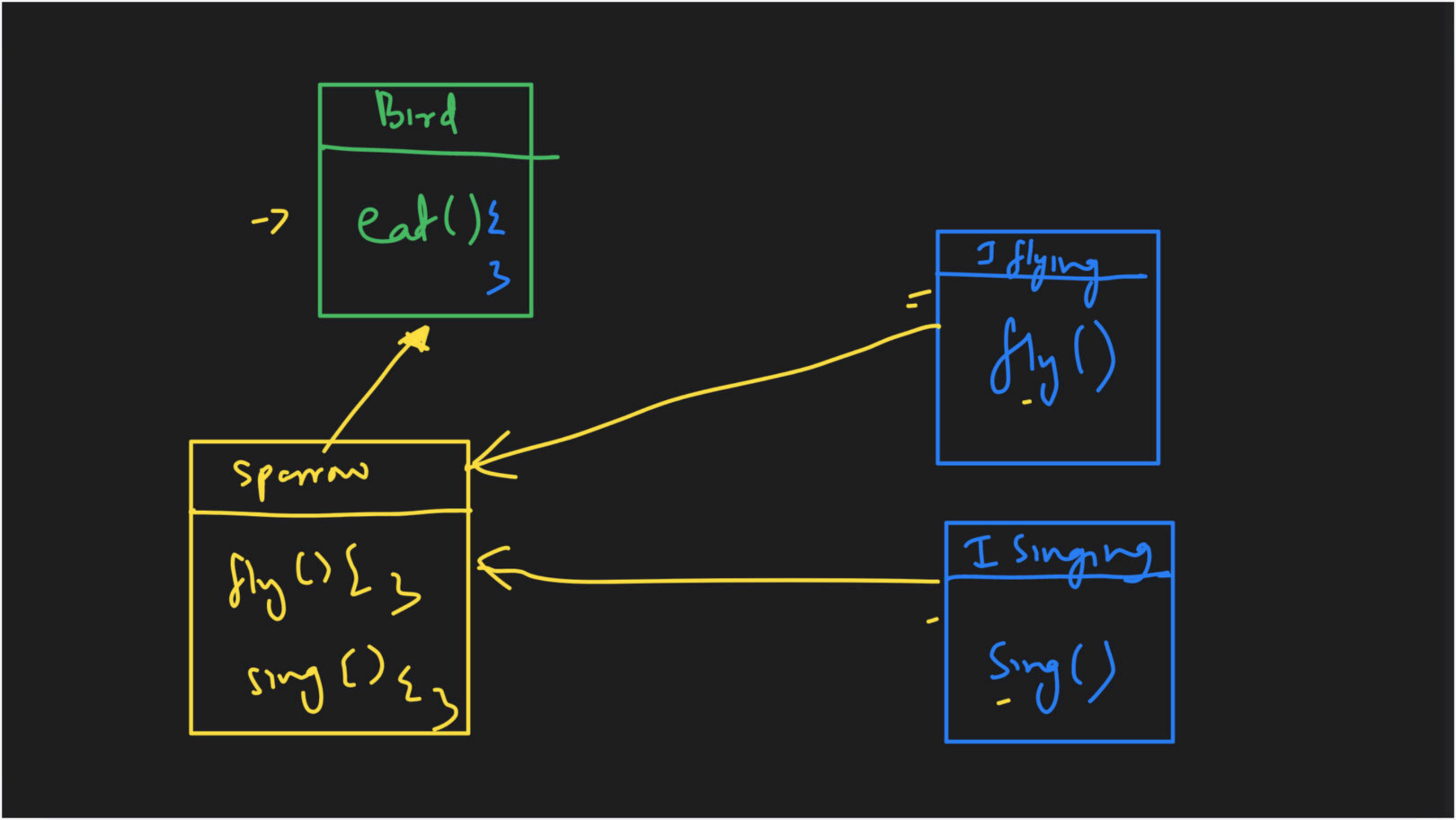
Sparrow extend Bird implement IFlyBird, ISingBird {

}

ISingBird {

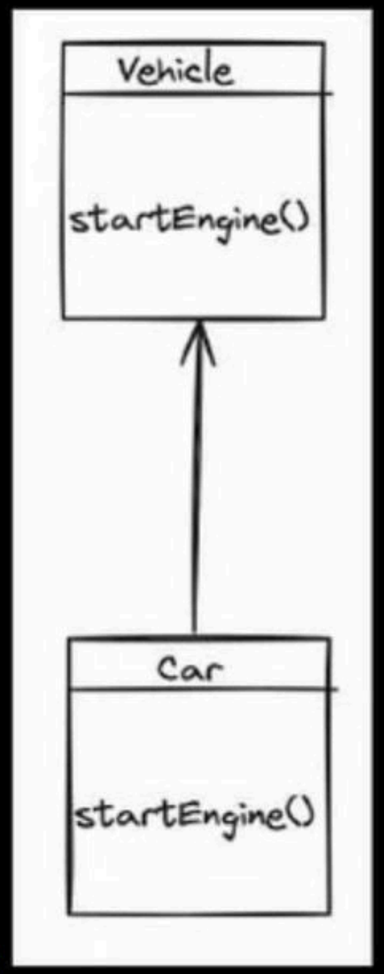
sing();

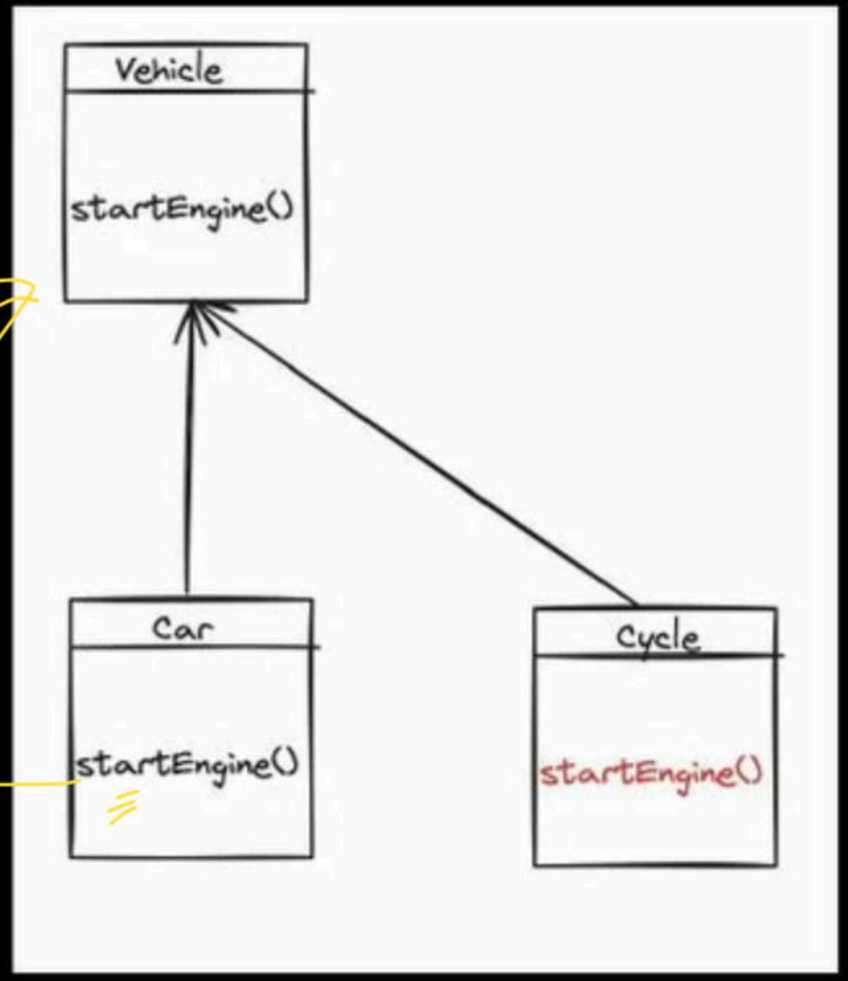
}


Bird
eat(){
}
IFlying
fly()
Sparrow
fly(){}
sing(){}
ISinging
Sing()

Liskov Substitution Principle

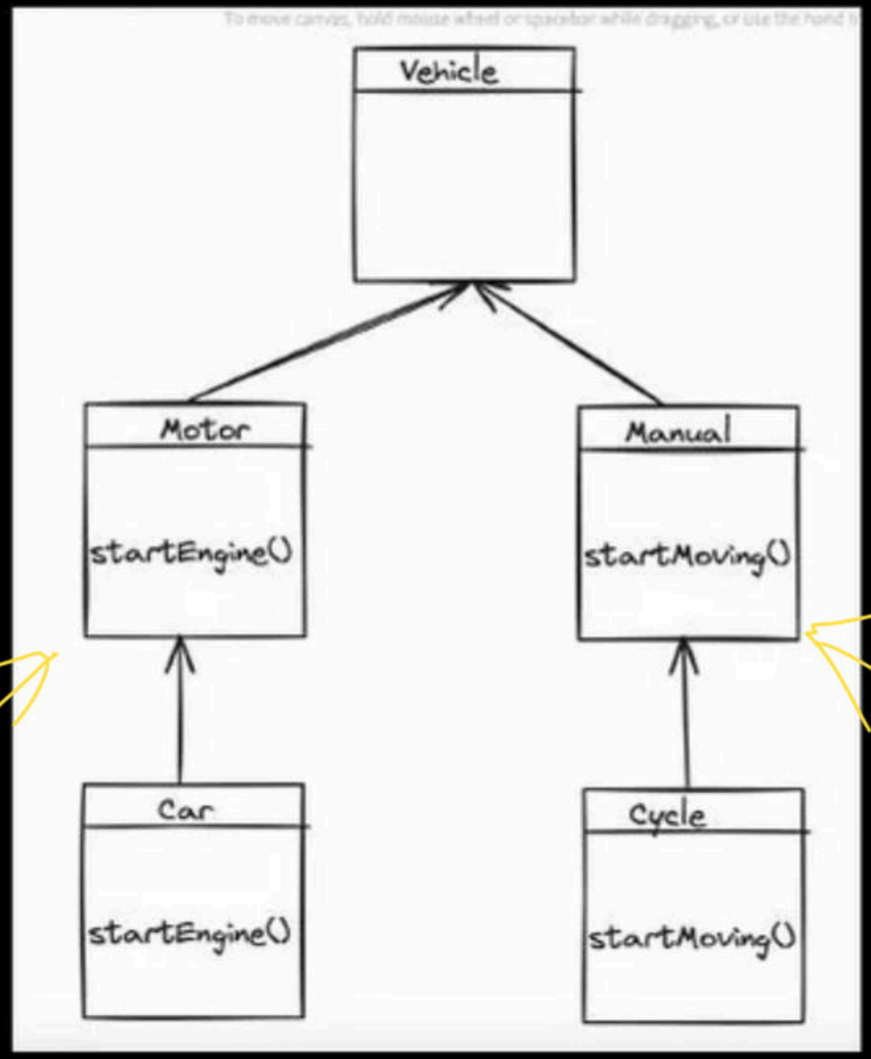
1. Vehicle Example





Liskov Substitution Principle

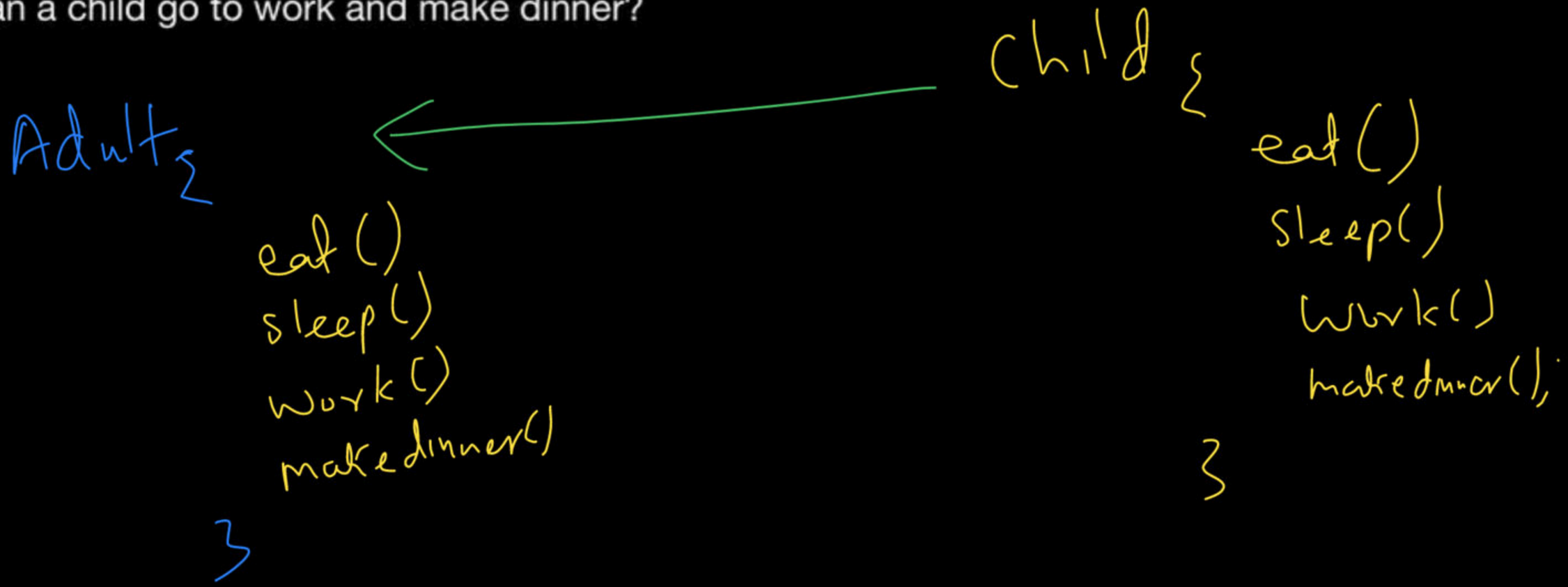
1. Vehicle Example

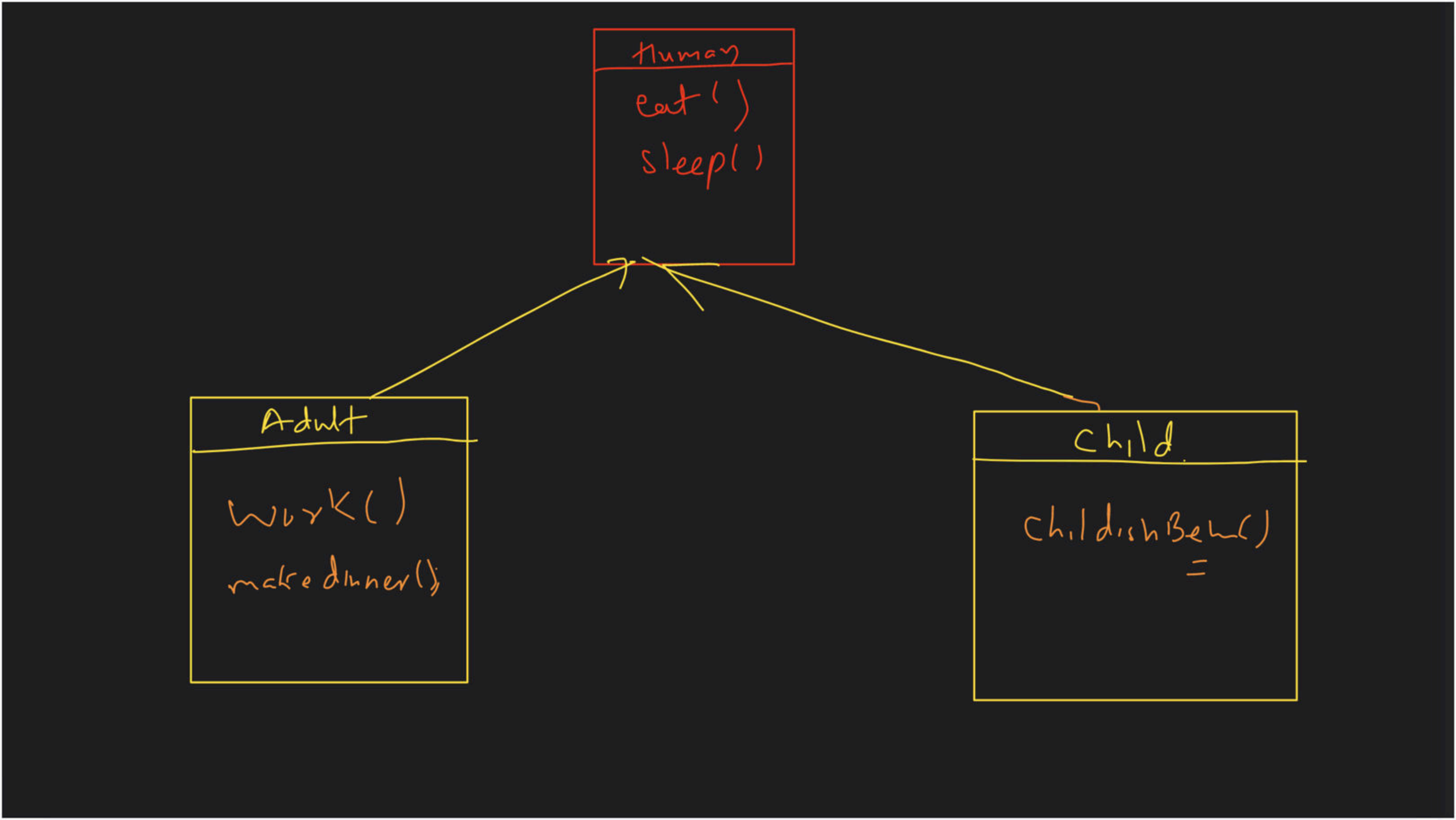


- Car is substitutable with its superclass, Motor, and Cycle is substitutable with its superclass, Manual, without breaking the functionality.
- Their methods can also override the methods of the superclass.

Liskov Substitution Principle

1. Can a child go to work and make dinner?



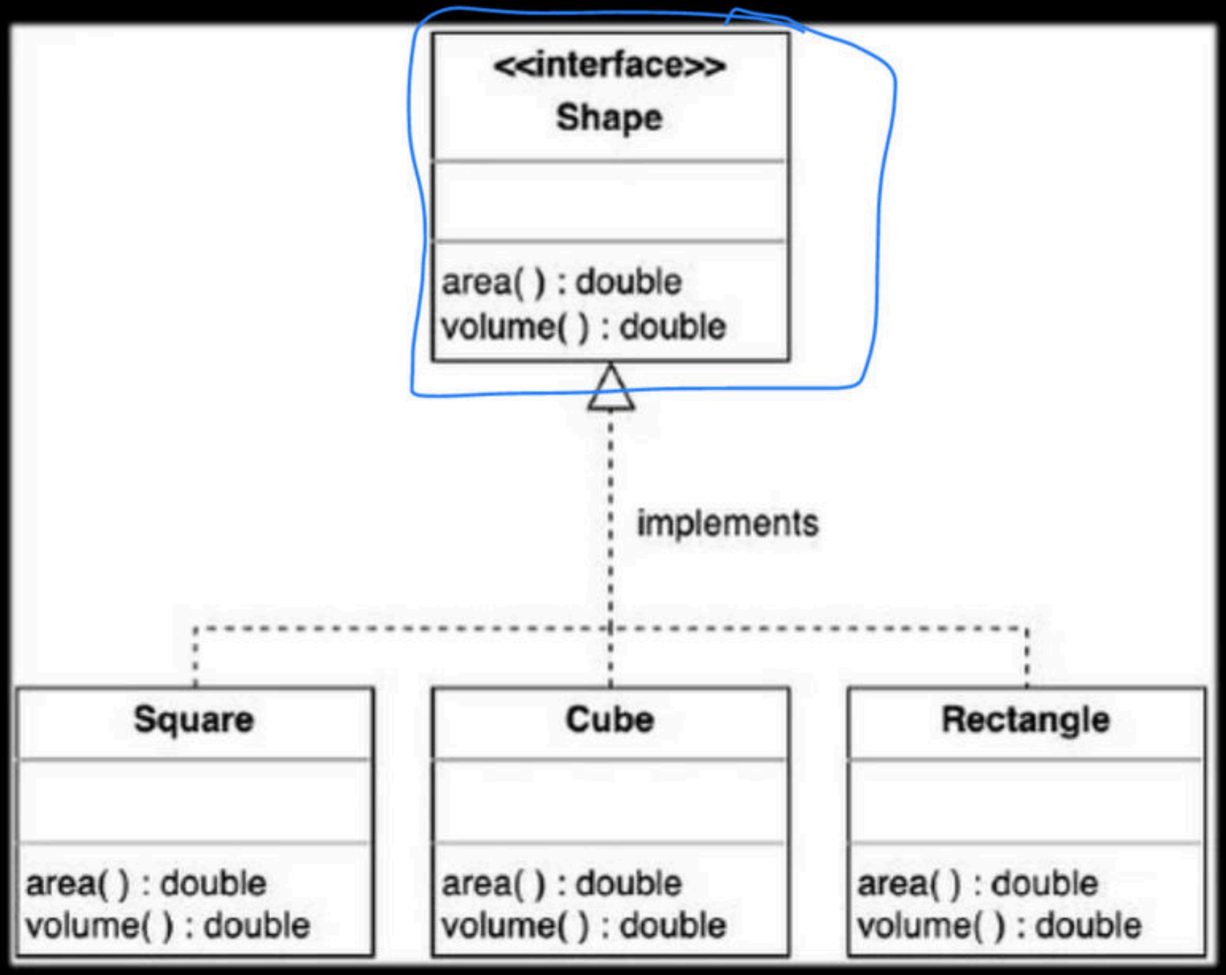

Human
eat()
sleep()
Adult
work()
makeDinner();
Child
childishBehaviour()


fun (Adult a)
- { a.eat()
a.work()
fun(c)
Child ref
3

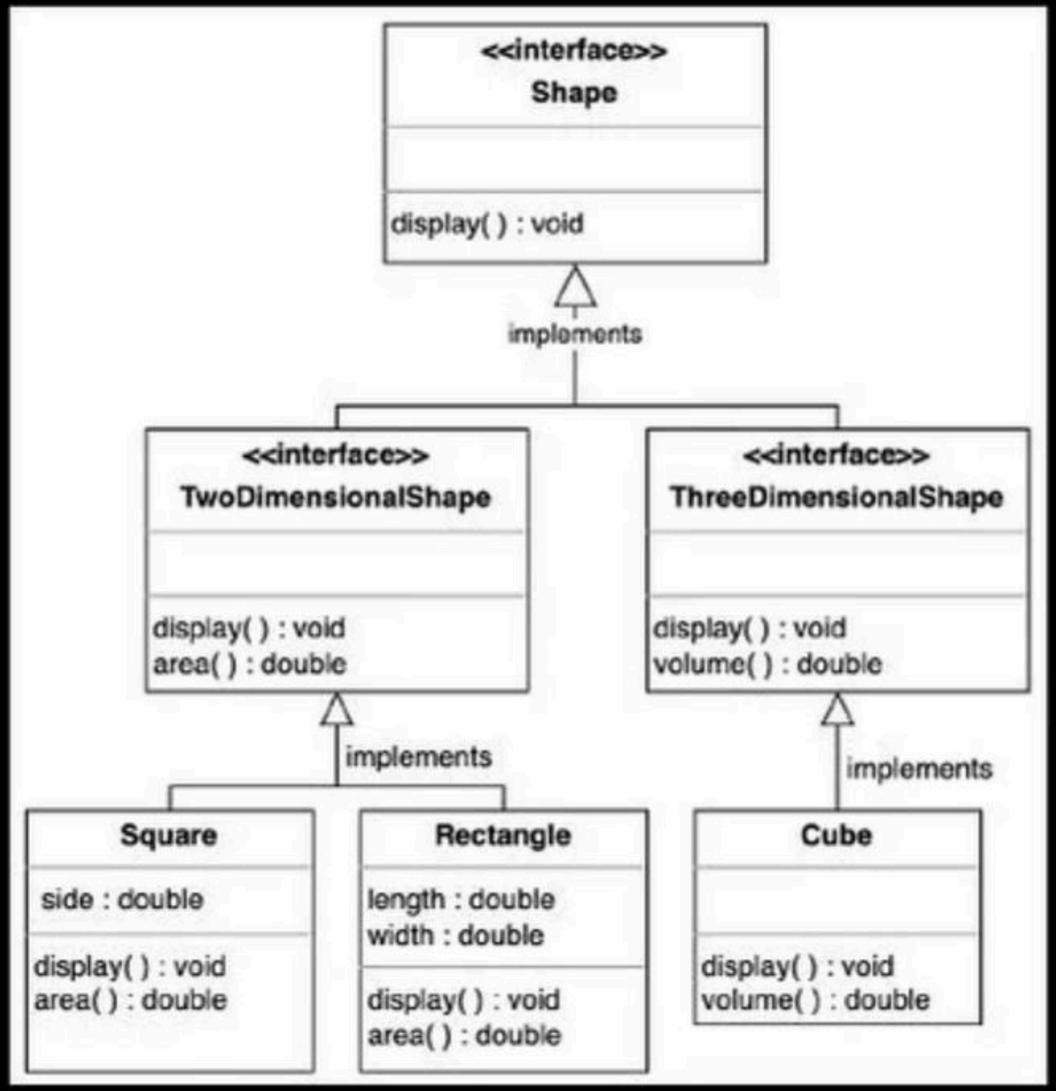
Interface Segregation Principle

1. The Interface Segregation Principle (ISP) is a design principle that does not recommend having methods that an interface would not use and require.
2. Therefore, it goes against having fat interfaces in classes and prefers having small interfaces with a group of methods, each serving a particular purpose.
3. To comply with the Interface Segregation Principle (ISP), it's important to design interfaces that are tailored to specific client needs instead of creating broad, all-purpose interfaces.
4. Do not build one pet interface make smaller and specific ones.

Interface Segregation Principle



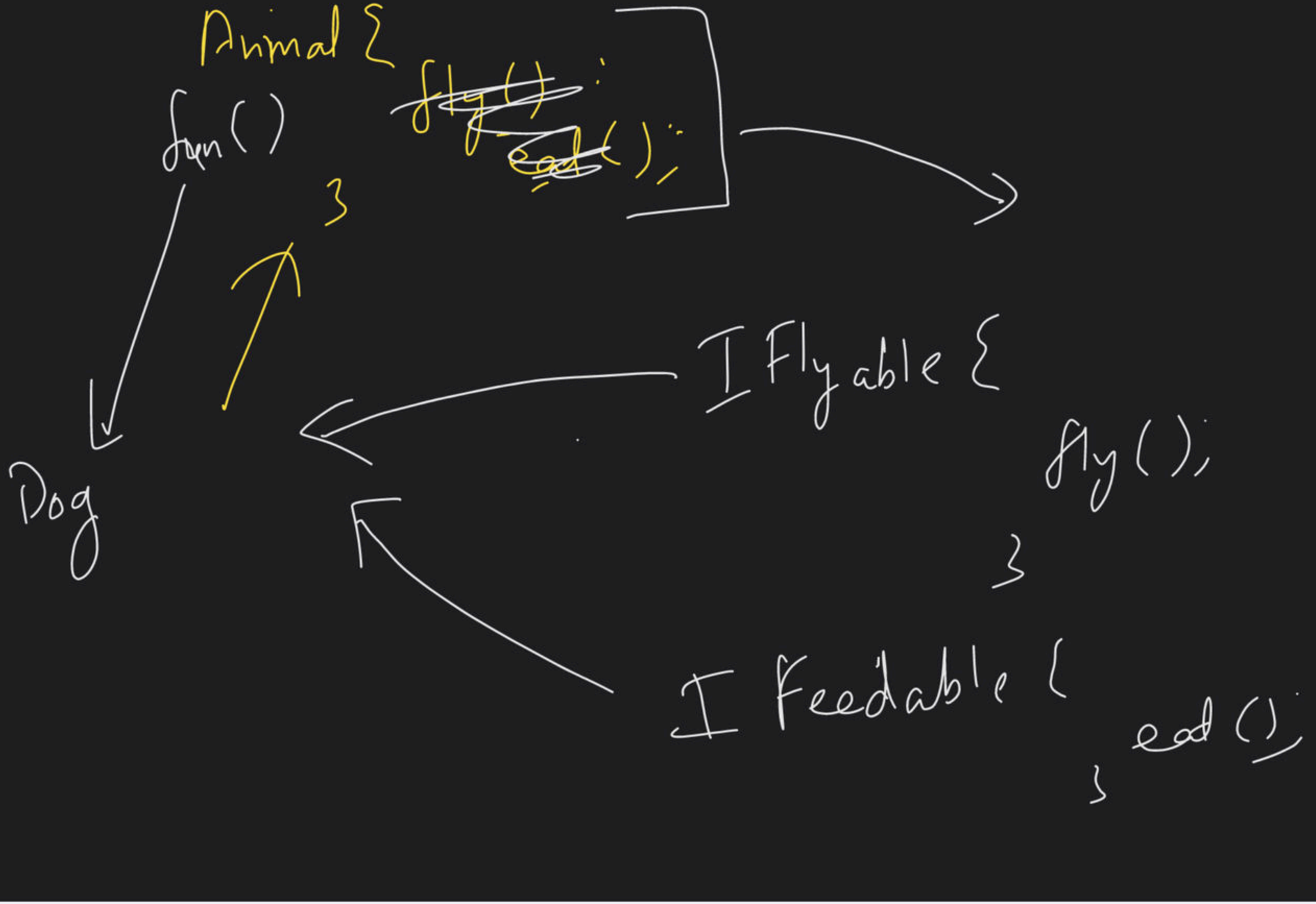
Interface Segregation Principle



LSP => asks you to make sure all child classes have same behaviour as parent class.

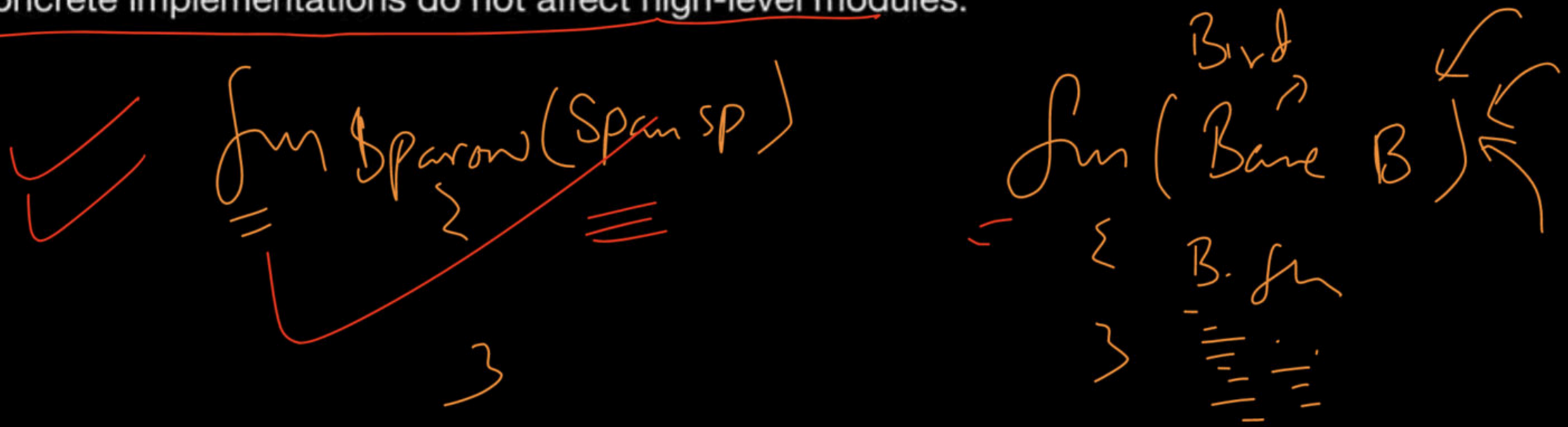
ISP => Create a diff interface for diff responsibility, don't group unrelated Behaviour in one interface.

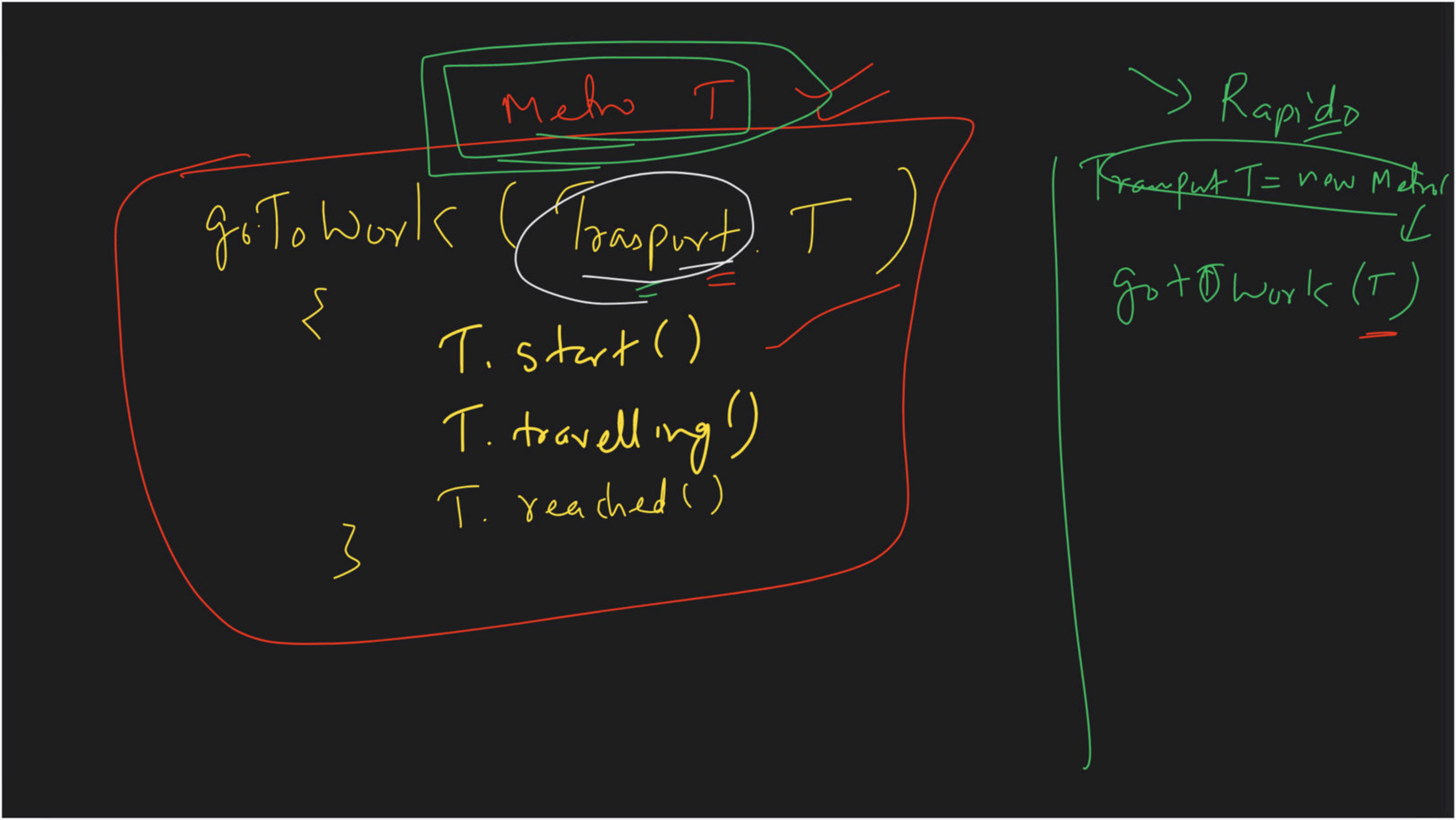
```
class Device {
    void callBaba() {
        // another call.
    }
}

class Calc : Device {
    void callBaba() {
        //exception, // can't do this
    }
}
```

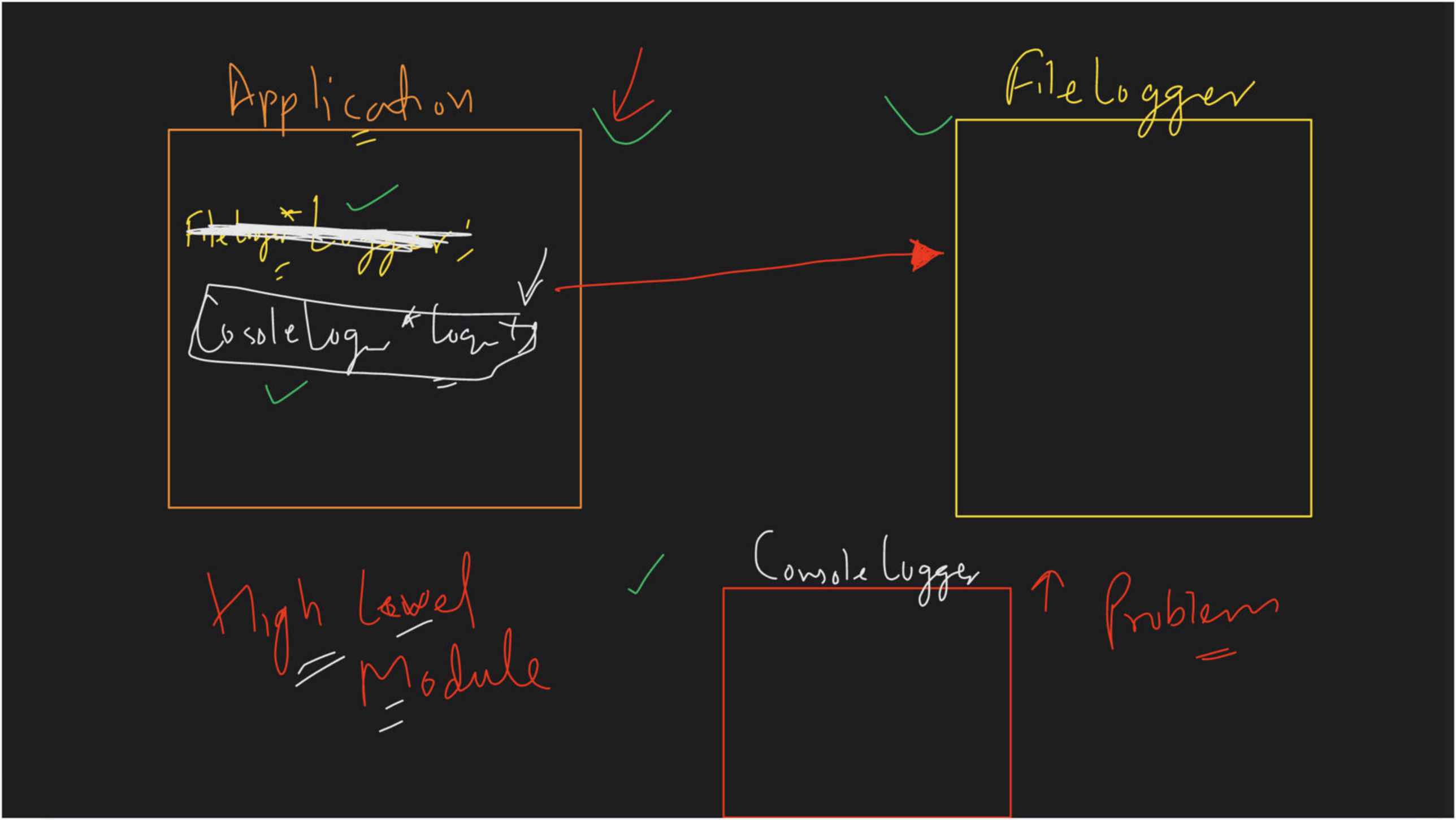

Animal {
fun()
fly();
eat();
}
Dog
IFlyable {
fly();
}
IFeedable {
eat();
}

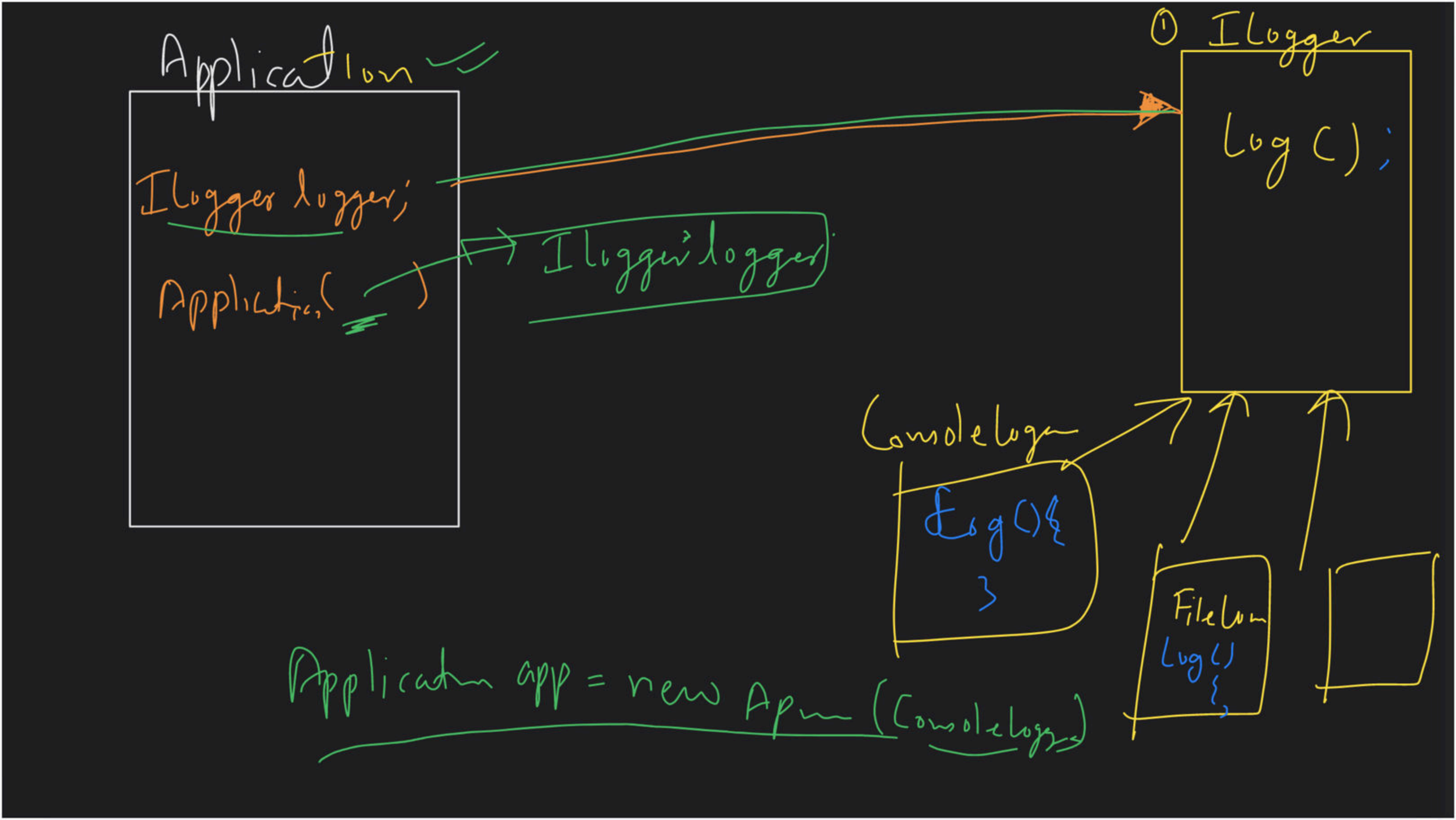
Dependency Inversion Principle

1. Never depend on everything concrete, only depend on Abstraction.
2. High level module should not depend on low level module. They should depend on Abstraction.
3. Able to change an implementation easily without altering the high level code.
4. By adhering to DIP, you can create systems that are resilient to change, as modifications to concrete implementations do not affect high-level modules.




Metro T
goToWork (Transport T)
{
T.start()
T.travelling()
T.reached()
}
Rapido
Transport T = new Metro
goToWork(T)


Application
FileLogger
ConsoleLogger
High Level Module
ConsoleLogger
Problem


Application
ILogger logger;
Application(ILogger logger)
ILogger
Log();
ConsoleLogger
Log(){
}
FileLog
Log()
{
}
Application app = new App(ConsoleLogger)