Tuesday March 3
ENSF 409, 202

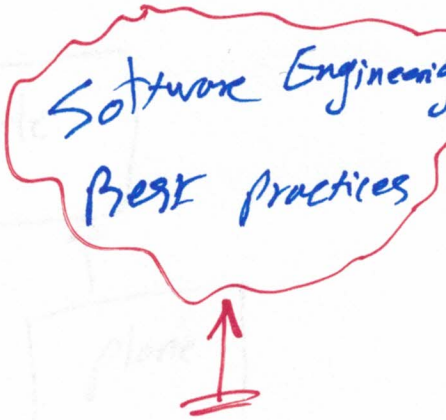
Single Responsibility principle

Open-closed principle.

Liskovs substitution Principle

Interface Segregation

Dependency Inversion




## Single Responsibility

→ A class should only have one Responsibility

↓ unit → or a method

That is
→ A unit should only have one reason to change.

How does "S" help?
→ Increasing cohesion
→ Decrease coupling
} less functionality which reduces dependency.

→ Testing → a class with one responsibility will have fewer test cases.

→ Better organization

## Open-Closed principle

Classes should be open for extension, but closed to modification.

↳ By following this principle (OC) we stop modifying existing classes, and don't cause potential Bugs! in an application that was working fine before we tried to change it!

eg

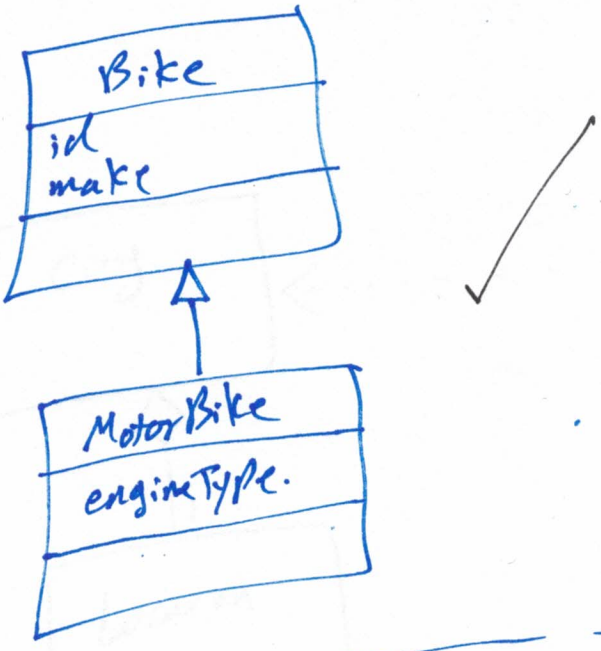
```
class Bike {
    int id;
    string make;
    string engineType;
}
```

X ← this is against the OC principle

Now I want my app to also support motorbikes.

Instead of adding engine type, I can extend Bike.

∴ According to OC principle, I proceed as follows



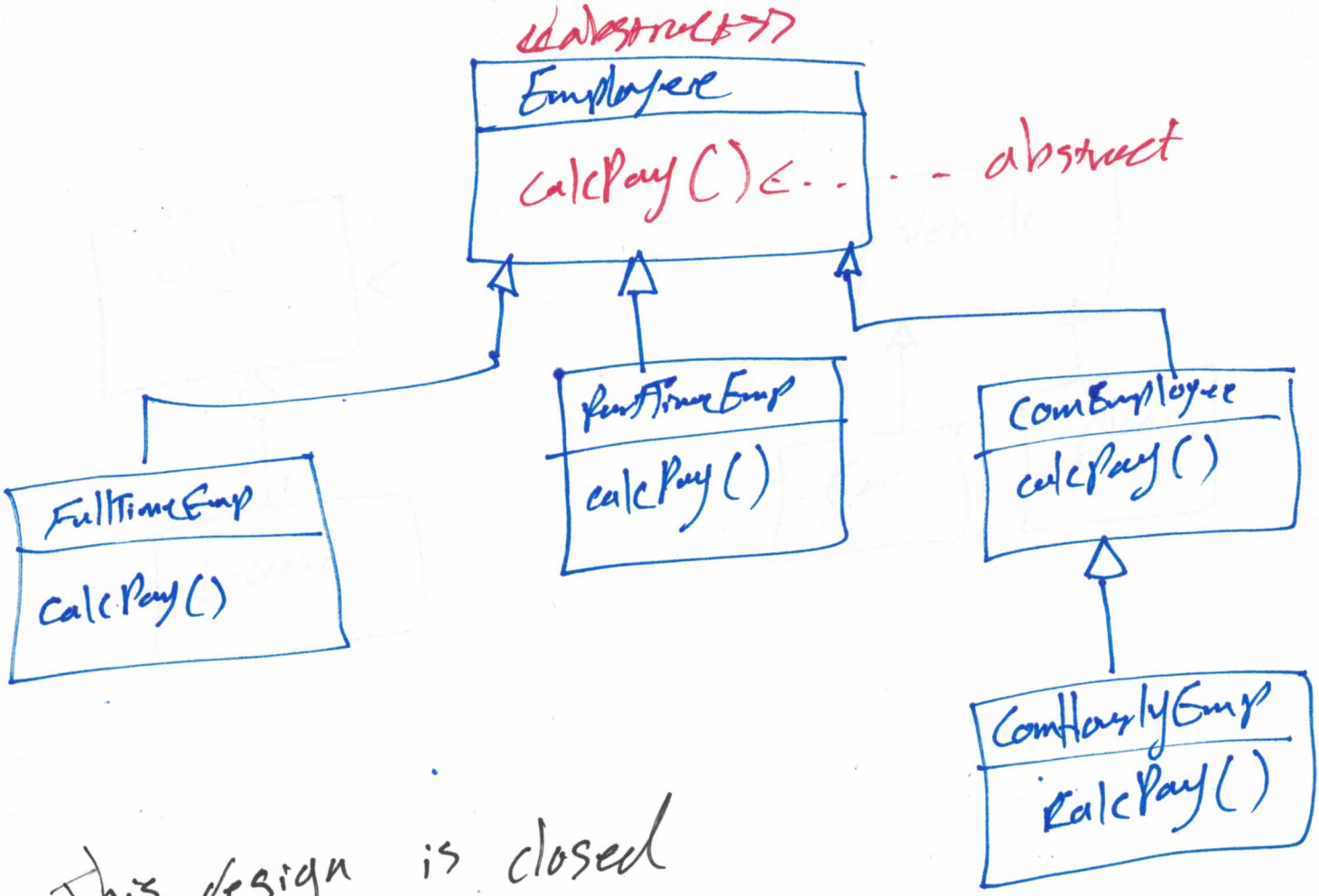

eg 2

```
class Employee {
    ~~private int employeeType;~~
    double salary;

    void calcSalary (int type) {
        if (type == 1) {
            calcFullTime()
        } else (type == 2){
            calcPartTime()
        }
        :
    }
}
```

1 - Full time
2 - part time
3 - Com. salary.

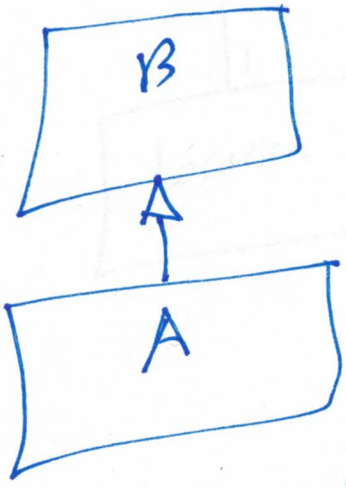
✗ bcz against OC




This design is closed to modification, and open to extention

## Liskov's substitution ← used to evaluate inheritance in an application

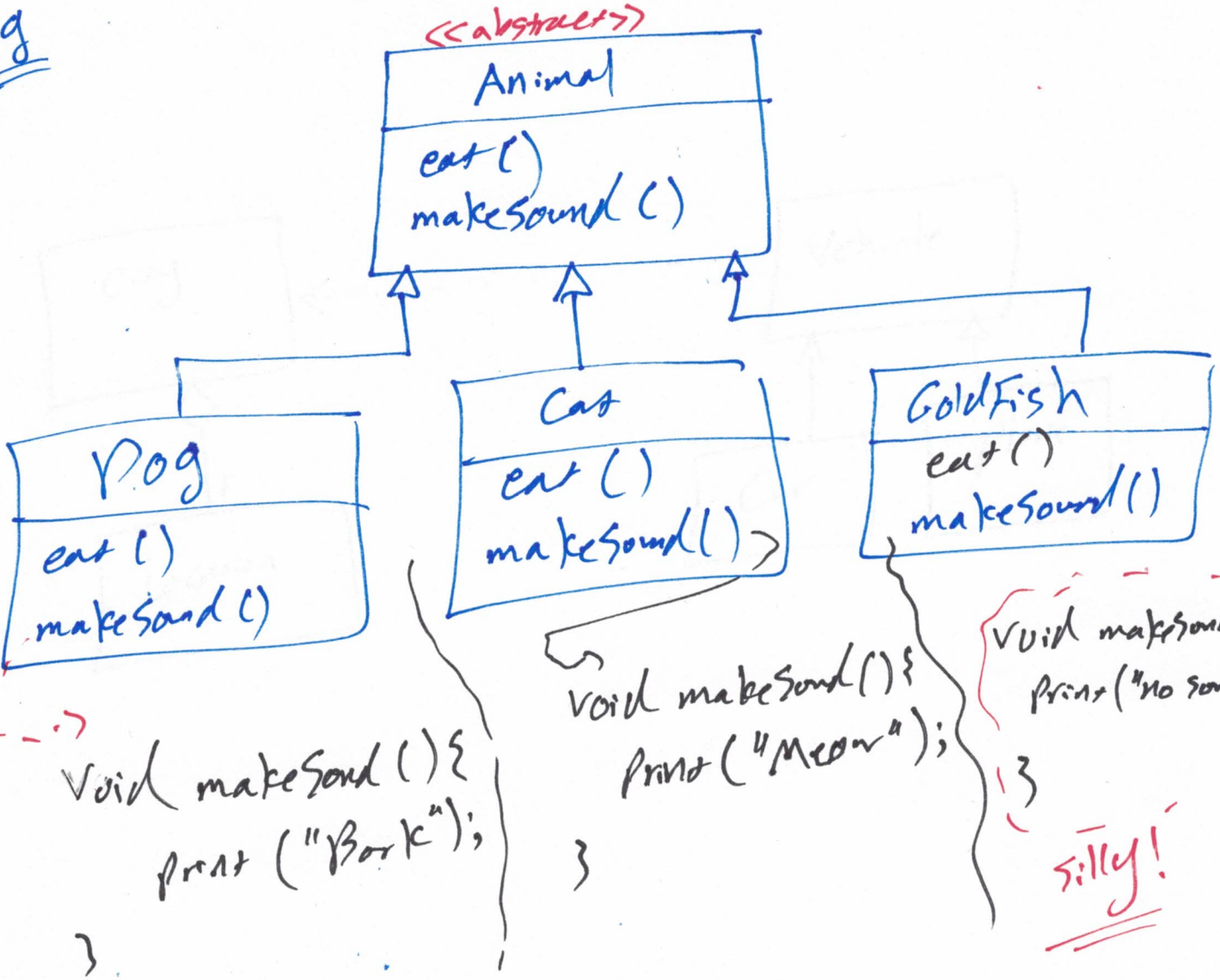
If class A is a ~~subtle~~ subclass of class B, then we should be able to replace B with A without disrupting the behavior of our program.




Inheritance is problematic if the parent is NOT general enough for its children.

In this case, the children may end up with methods/variables that they do NOT need!

eg



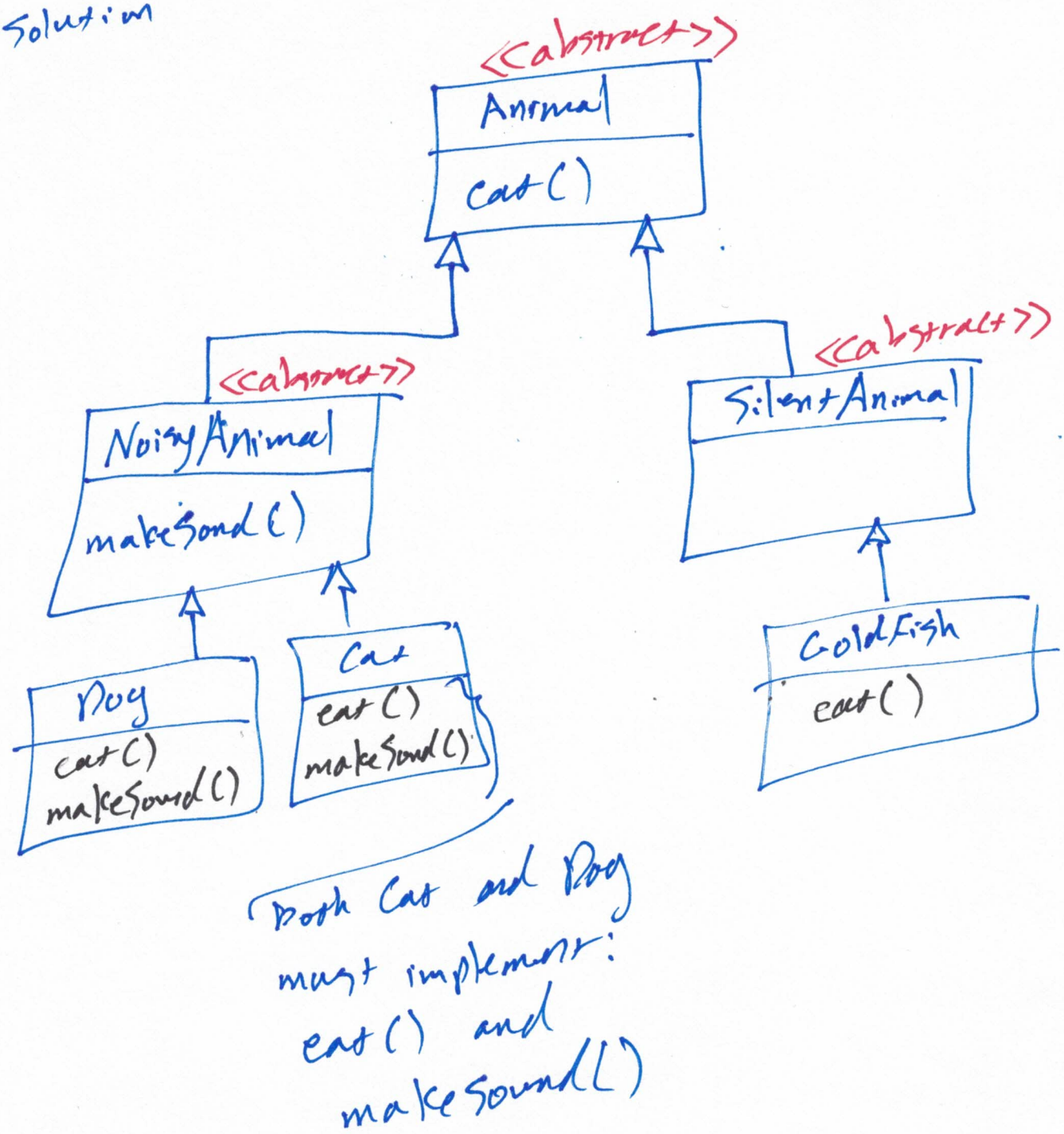

The problem in the above example is that Animal is NOT General enough.

What is the solution?

→ Change the inheritance tree!

Animal should be more general to accont for animals that don't make sounds.

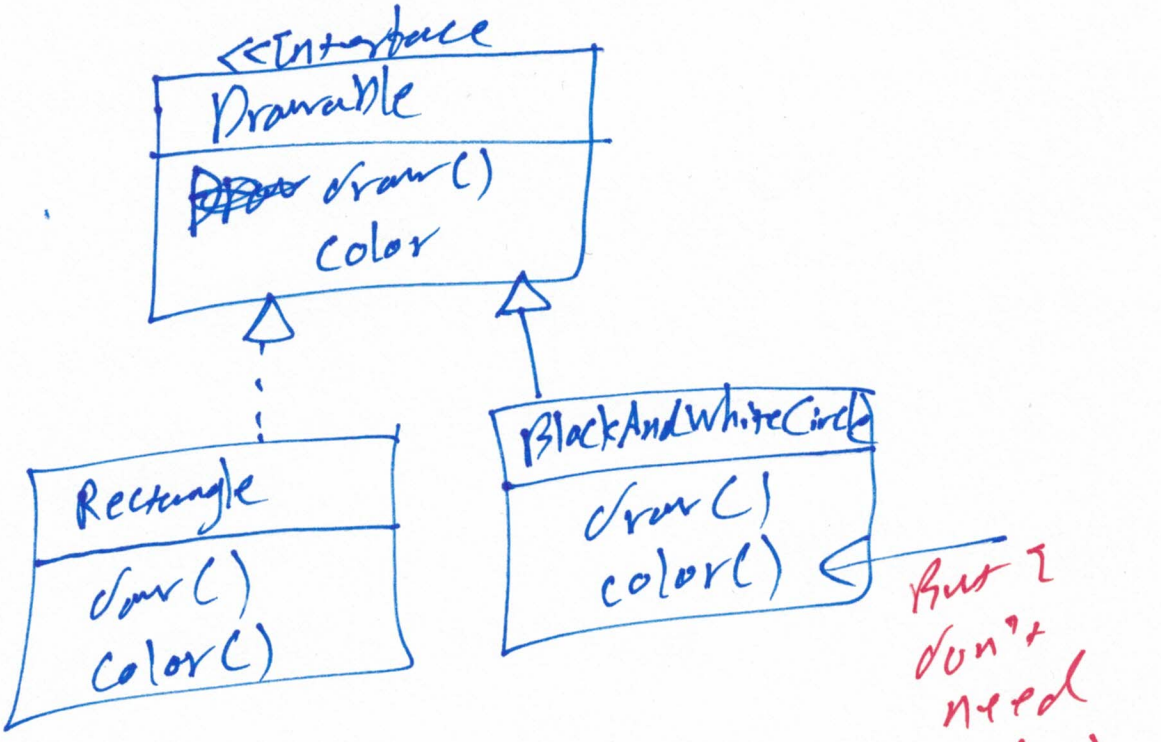
Solution



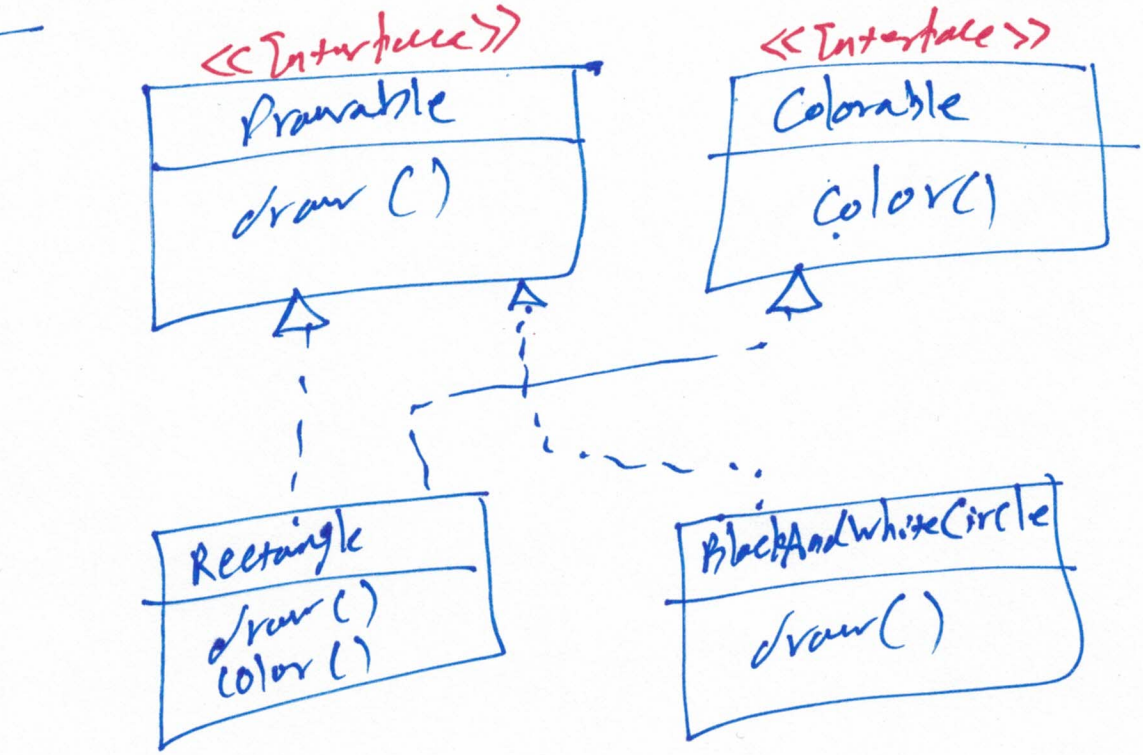

Both Cat and Dog must implement: eat() and makeSound()

# Interface Segregation

eg




Solution



Notes

Larger Interfaces should be split into smaller ones. This way, we can ensure the implementing classes only need to be concerned about the methods they need.

eg

```
public Interface Beerkeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
public Interface BearCleaner(){
    void washTheBear();
}
public Interface BearFeeder(){
    void feedTheBear();
}
public Interface. BearPetter(){
    Void petTheBear();
}
```

Too bulky!

break it into smaller interfaces

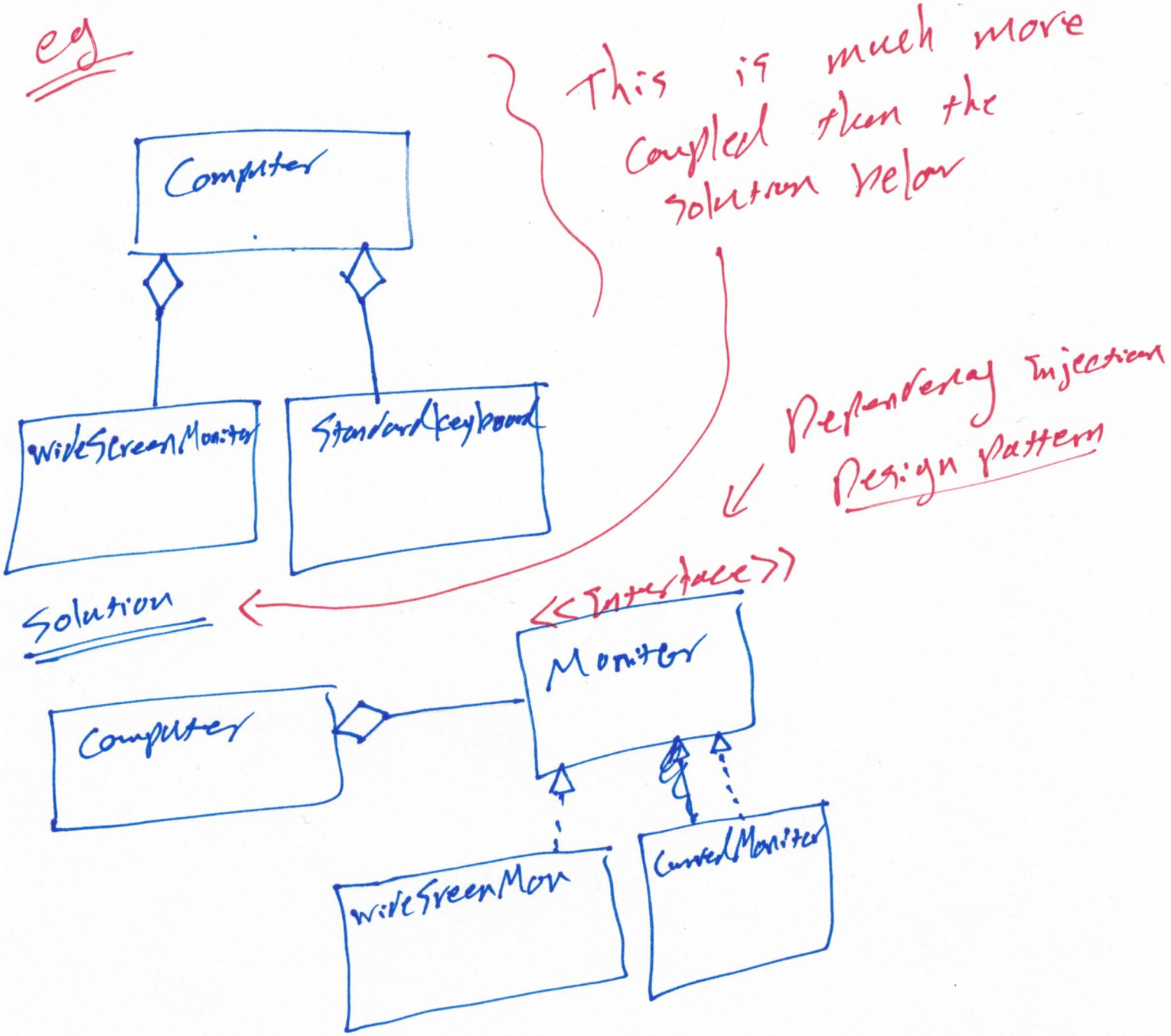
```
class BearCarer implements BearCleaner, BearFeeder {
    // implement Both methods
    void washTheBear() {
        - - ———— -
    }
    void feedTheBear() {
        - - - - - -
    }
}
class CrazyPerson implements BearPetter {
    void petTheBear() {
        - - - —
    }
}
```

This way class BearCarer can implement exactly what it needs.

# Dependency Inversion

Refers to decoupling of software modules.
Instead of high-level modules depending on low-level ones, Both depend on abstraction.

eg



Note

In the solution above, class Computer is now depending on abstraction. you can change the type of the monitor that a computer object uses dynamically at runtime, without changing the class Computer.

By making a class depend on abstraction you ensure that class is closed for modifications, but open to extension