8__24-07-2022

Sunday, 24 July 2022 8:03 PM

L - Liskov's Substitution Principle

Inheritance

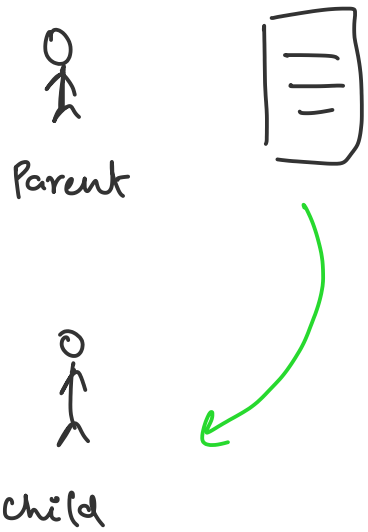
↳ is-A test.

C_1 [is-A] C_2

↳ True

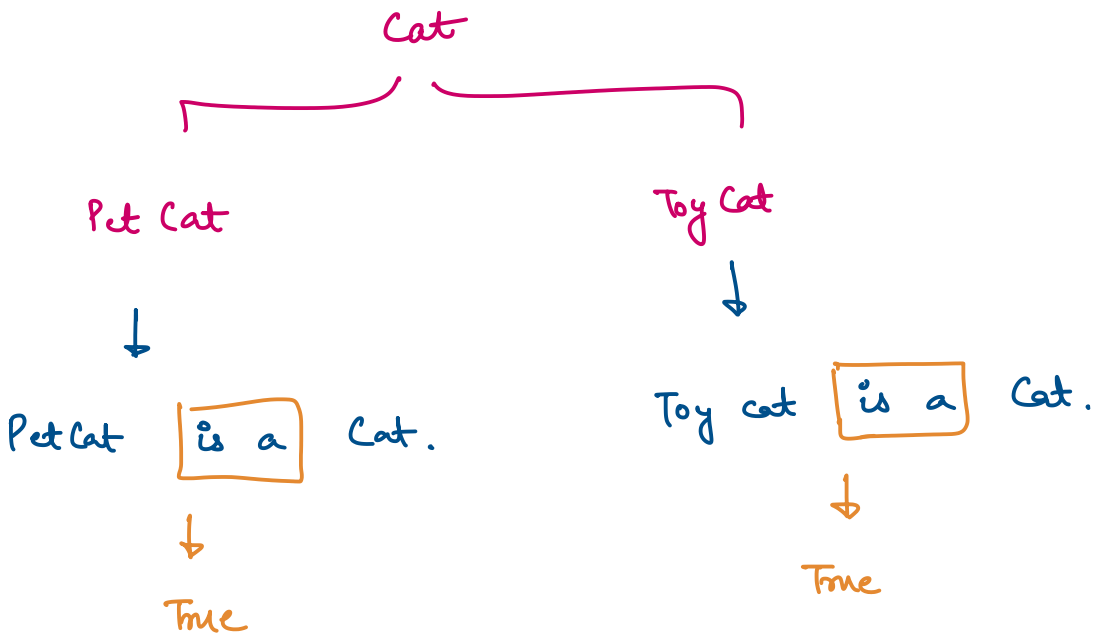
↓

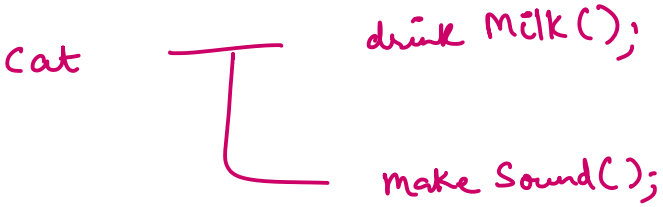
C_1 extends C_2 } Concept of inheritance



For inheritance

↳ If parent can replace child at all the places.






```
public interface ICat {
   public void drinkMilk();
   public void makeSound();
```

```
}



public class PetCat implements ICat {
   @Override
   public void drinkMilk() {
      System.out.println("Drinking more milk.");
   }

   @Override
   public void makeSound() {
      System.out.println("Makes meow sound.");
   }
}




public class ToyCat implements ICat {
   @Override
   public void drinkMilk() {
      throw new RuntimeException("I don't drink milk");
   }

   @Override
   public void makeSound() {
      System.out.println("Makes mechanical meow sound.");
   }
}




public class Program {
   public static void main(String[] args) {
      //this works perfectly fine
      ICat cat = new PetCat();
      cat.drinkMilk();
      cat.makeSound();

      ICat toyCat = new ToyCat();
      toyCat.makeSound();
      toyCat.drinkMilk();
   }
}
```

Output:

```
Drinking more milk.
Makes meow sound.
Makes mechanical meow sound.
Exception in thread "main" java.lang.RuntimeException: I don't drink milk
    at ToyCat.drinkMilk(ToyCat.java:4)
    at Program.main(Program.java:10)
```

Toy cat class is not an obedient child because it is not obeying to drinkMilk() method.

↓

Thus violating Liskov's principle.

Liskov's Substitution Principle (LSP) states that "you should be able to use any derived class instead of a parent class and have it behaved in the same manner without modification".

It ensures that a derived class does not affect the behaviour of the parent class, in other words, that a derived class must be substitutable for its base class.

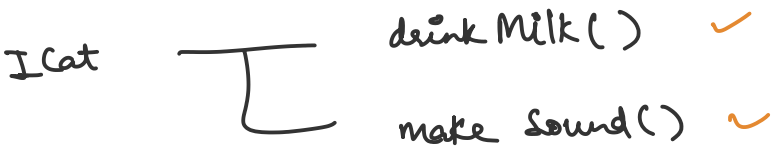
This principle is just an extension of open/close principle and it means that we must ensure that new derived classes extend the base classes without changing the behaviour.

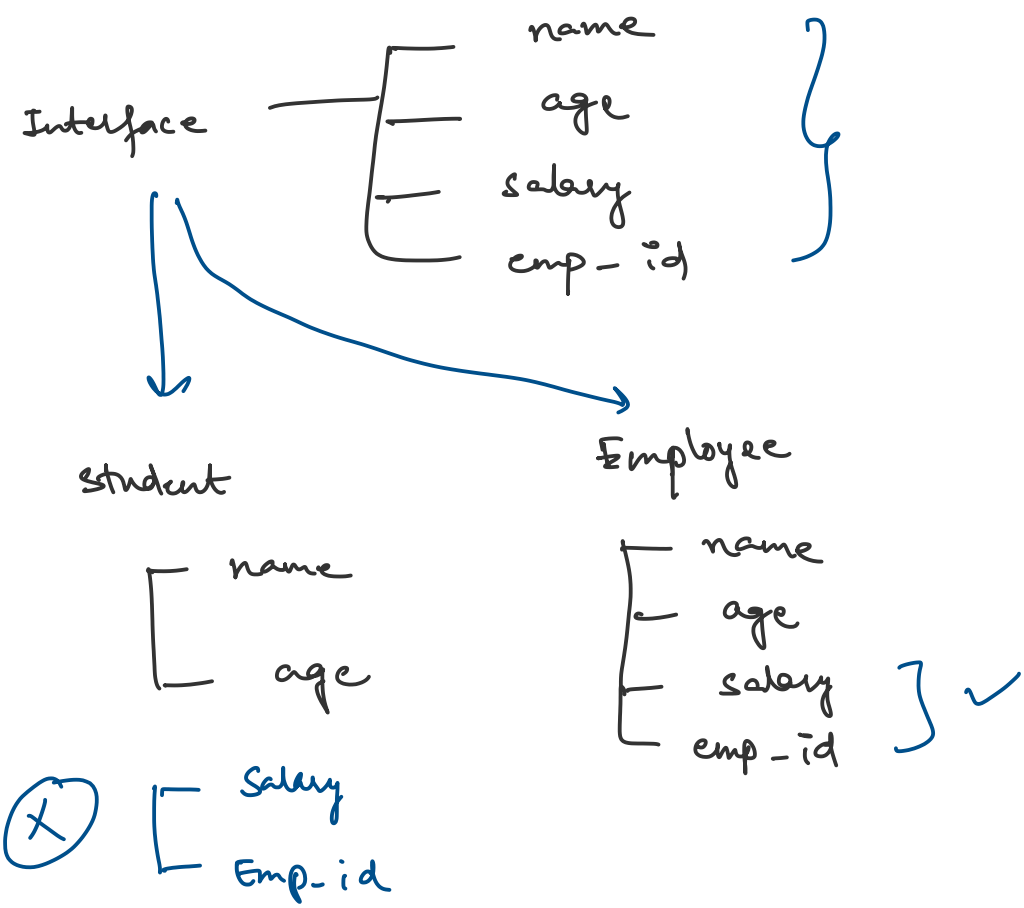
How to sort out this problem?

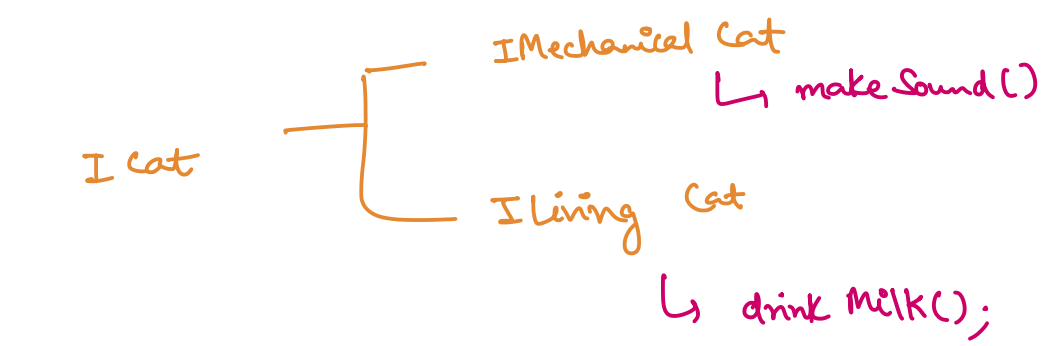
↓

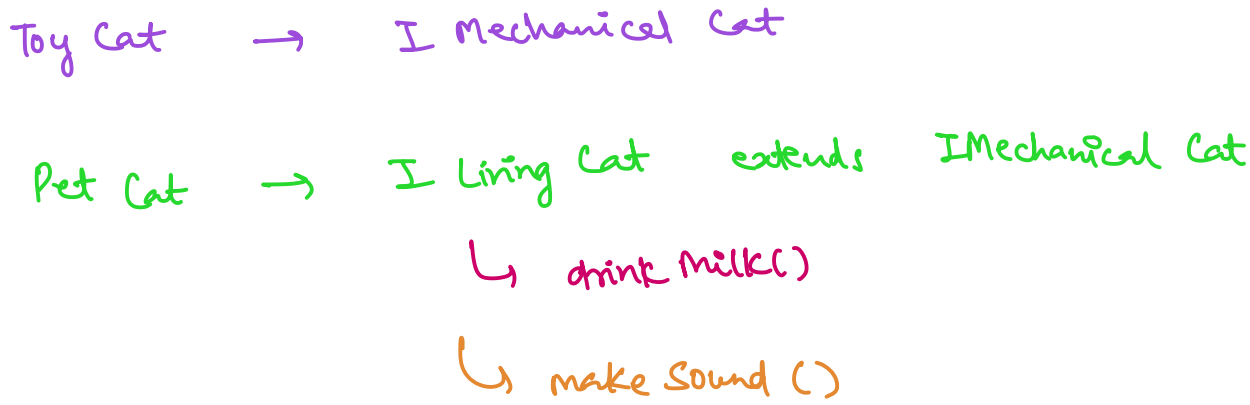
I from SOLID.

I – Interface segregation principle.










```
public interface IMechanicalCat {
   public void makeSound();
}
```

```
public interface ILivingCat extends IMechanicalCat {
   public void drinkMilk();
}
```

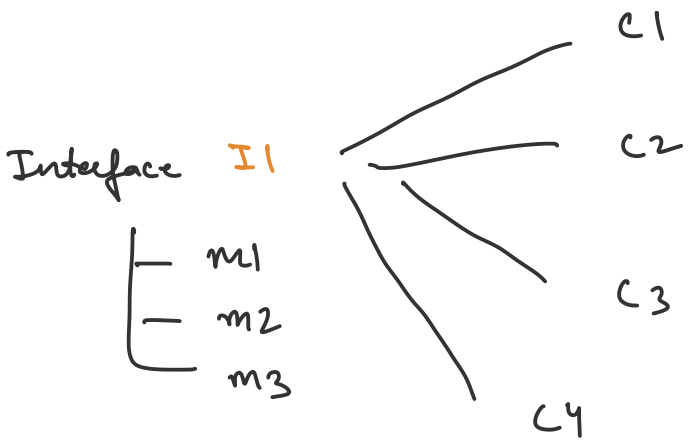
```
public class ToyCat implements IMechanicalCat {
   @Override
   public void makeSound() {
      System.out.println("Makes mechanical meow sound.");
   }
}
```

```
public class PetCat implements ILivingCat {
   @Override
   public void drinkMilk() {
      System.out.println("Drinking more milk.");
   }

   @Override
   public void makeSound() {
      System.out.println("Makes meow sound.");
   }
}
```

```
public class Program {
   public static void main(String[] args) {
      System.out.println("For pet cat:");
      ILivingCat cat = new PetCat();
      cat.drinkMilk();
      cat.makeSound();
      System.out.println();

      System.out.println("For toy cat:");
      IMechanicalCat toyCat = new ToyCat();
      toyCat.makeSound();
   }
}
```

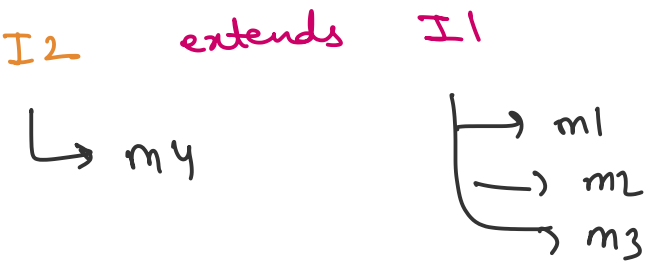
Output:

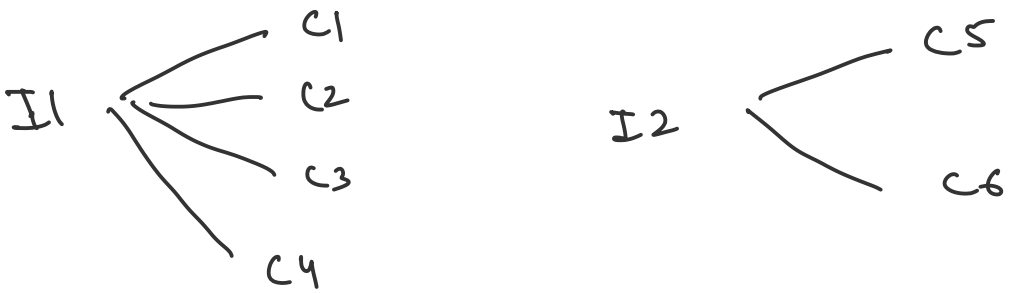
```
For pet cat:
Drinking more milk.
Makes meow sound.

For toy cat:
Makes mechanical meow sound.
```



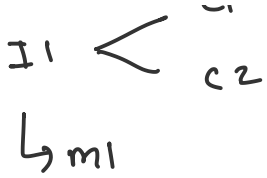
Interface is having methods- m1, m2 and m3. This interface is getting used in classes- C1, C2, C3, and C4.

Now I want to another method m4 so that we can use it in class C5 and C6. How will we be doing this?





C1



class C1 implements I1

{

}

class C2 implements I1

{

}

interface I2 extends I1

↳ m1();

{

public void m2();

}

class C3 implements I2

{

==

}

class C4 implements I2

{

==

}

```
public interface Interface1 {
   public void method1();
}
```

```
public class Class1 implements Interface1 {
   @Override
   public void method1() {
      System.out.println("Class 1 : method 1");
   }
}
```

```
public class Class2 implements Interface1 {
   @Override
   public void method1() {
```

```
        System.out.println("Class 2 : method 1");
    }
}
```

```
public interface Interface2 extends Interface1 {
    public void method2();
    public void method3();
}
```

```
//here i want to use all the three methods
public class Class3 implements Interface2 {
    @Override
    public void method1() {
        System.out.println("Class 3 : method 1");
    }

    @Override
    public void method2() {
        System.out.println("Class 3 : method 2");
    }

    @Override
    public void method3() {
        System.out.println("Class 3 : method 3");
    }
}
```

```
public class Program {
    public static void main(String[] args) {
        Class1 obj1 = new Class1();
        obj1.method1();
        System.out.println();

        Class2 obj2 = new Class2();
        obj2.method1();
        System.out.println();

        Class3 obj3 = new Class3();
        obj3.method1();
        obj3.method2();
        obj3.method3();
```

```
    }
}
```

Output:
Class 1 : method 1

Class 2 : method 1

Class 3 : method 1
Class 3 : method 2
Class 3 : method 3

Interface Segregation Principle (ISP) states that "clients should not be forced to implement interfaces they don't use".

Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.

An interface should be more closely related to the code that uses it than the code that implements it.

So the methods on the interface are defined by which methods the client code needs rather than which methods the class implements.

So, clients should not be forced to depend upon interfaces that they don't use.

Like classes, each interface should have a specific purpose / responsibility. You should not be forced to implement an

client does not

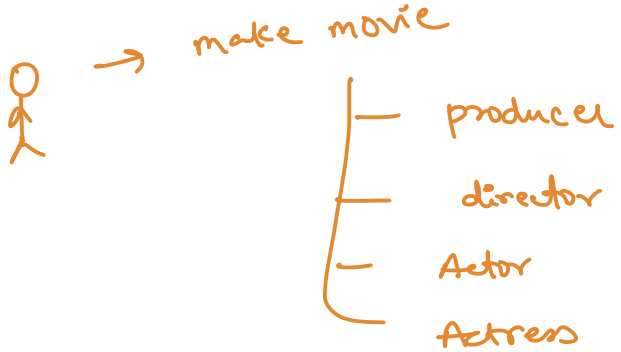
interface when your ...

share that purpose.

The larger the interface, the more likely it includes methods that not all implementers can do.

D – Dependency Inversion Principle (DIP)

It says that higher classes should not directly depend on the low level classes.



class Director

{

}

class Producer

{

}

class Actor

{

}

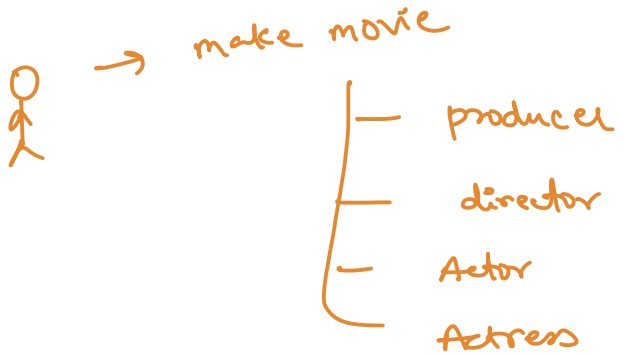
class Actress

{

}

```
class MovieMaker
{
    Director director;
    Producer producer;
    Actor actor;
    Actress actress;

    public void hireDirector()
    {
        ——
    }

    public void hireProducer()
    {
        ——
    }

    public void hireActor()
    {
        ——
    }
```

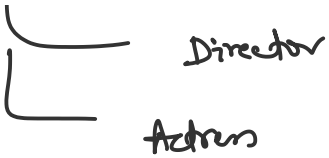
```
public void hireActress()
{
    ==
    ==
}
}
```

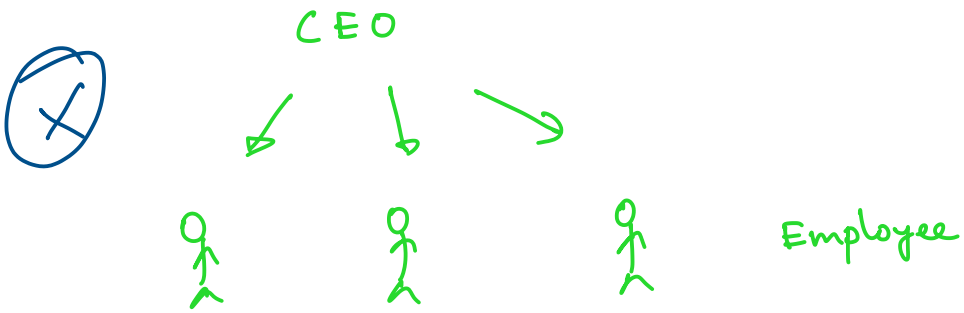


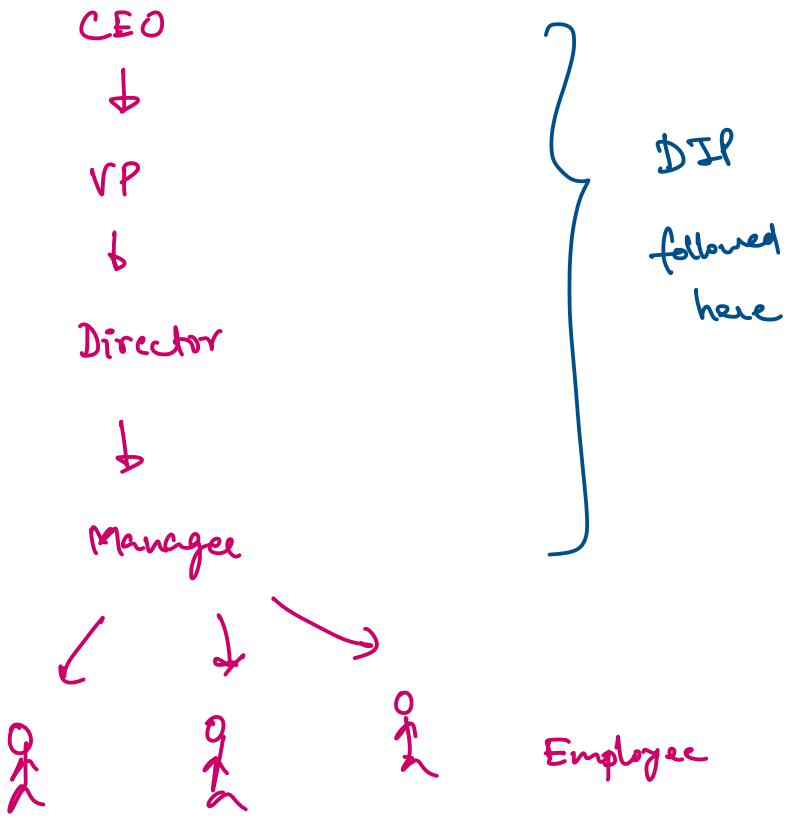
violates open closed principle

Higher level class - movie maker

lower level class - producer, Actor


Director
Actors


CEO
Employee


CEO
VP
Director
Manager
Employee
DIP
followed
here