

设计模式 (上)

软件工程中，设计模式是对软件设计中普遍存在 (反复出现) 的各种问题，所提出的解决方案
目的是让程序具有更好的

1. 代码重用性 (相同功能的代码，不用多次编写)
2. 可读性 (编程规范，便于阅读和理解)
3. 可扩展性 (需求增加时，方便扩展)
4. 可靠性 (增加新功能后，对原来的功能没有影响)
5. 低耦合，高内聚

七大设计原则

设计模式的原则，其实就是 程序猿在编程时，应当遵守的原则，也是各种设计模式的基础

- 单一职责原则
- 接口隔离原则
- 依赖倒转原则
- 里氏替换原则
- 开闭原则
- 迪米特法则
- 合成复用原则

单一职责原则

对类来说，一个类应该只负责一项职责。

eg: 类 A 负责两个不同的职责：职责 1，职责 2。当职责 1 需求改变时而更改 A，可能会造成 职责 2 的错误。因此需要将类 A 分解为两个类，负责不同的职责。

满足单一职责原则，来保证某一功能的变更不会影响其它的功能 (可靠性)

看一下下面这段简单的代码和它的运行结果

```
public class SingleResponsibilityOne {  
  
    public static void main(String[] args) {  
  
        vehicle vehicle = new vehicle();  
        vehicle.run("摩托车");  
        vehicle.run("巴士");  
        vehicle.run("直升飞机");  
    }  
}
```

```

/**
 * 交通工具类
 */
class Vehicle{

    public void run(String vehicle){
        System.out.println(vehicle + "在公路上奔跑");
    }

}

```

```

摩托车在公路上奔跑
巴士在公路上奔跑
直升飞机在公路上奔跑

```

很显然，第三条输出违反了单一职责原则。从 Vehicle 类中的 run 方法看来，这个类的职责应该是只管公路上跑的。因此可以将 Vehicle 类进行拆分 (如拆分为：LoadVehicle，SkyVehicle，WaterVehicle 等)，并将 Vehicle 抽象出来成为父类

例如下面代码

```

public class SingleResponsibilityThree {
    public static void main(String[] args) {

        BaseVehicle landVehicle = new LandVehicle2();
        BaseVehicle waterVehicle = new WaterVehicle2();
        BaseVehicle skyVehicle = new SkyVehicle2();

        landVehicle.run("巴士");
        waterVehicle.run("邮轮");
        skyVehicle.run("小鸟");
    }
}

class BaseVehicle{

    void run(String vehicle){};
}

class LandVehicle2 extends BaseVehicle{

    @Override
    void run(String vehicle) {
        System.out.println(vehicle + "在地上跑");
    }
}

class WaterVehicle2 extends BaseVehicle{

    @Override
    void run(String vehicle) {
        System.out.println(vehicle + "在水里游");
    }
}

```

```
class skyVehicle2 extends BaseVehicle{

    @Override
    void run(String vehicle) {
        System.out.println(vehicle + "在天上飞");
    }
}
```

当然，对于这种不 "复杂" 的类，我们可以在方法级别保持单一职责原则
例如下面的代码

```
public class SingleResponsibilityTwo {

    public static void main(String[] args) {
        vehicleAll vehicleAll = new VehicleAll();

        vehicleAll.runLoad("汽车");
        vehicleAll.runSky("灰机");
        vehicleAll.runWater("潜艇");
    }
}

/**
 *
 * 遵守单一职责原则，保持方法级别的单一职责原则
 *
 */
class VehicleAll{

    public void runLoad(String vehicle) {
        System.out.println(vehicle + "在地上跑");
    }

    public void runWater(String vehicle){
        System.out.println(vehicle + "在水里游");
    }

    public void runSky(String vehicle){
        System.out.println(vehicle + "在天上飞");
    }
}
```

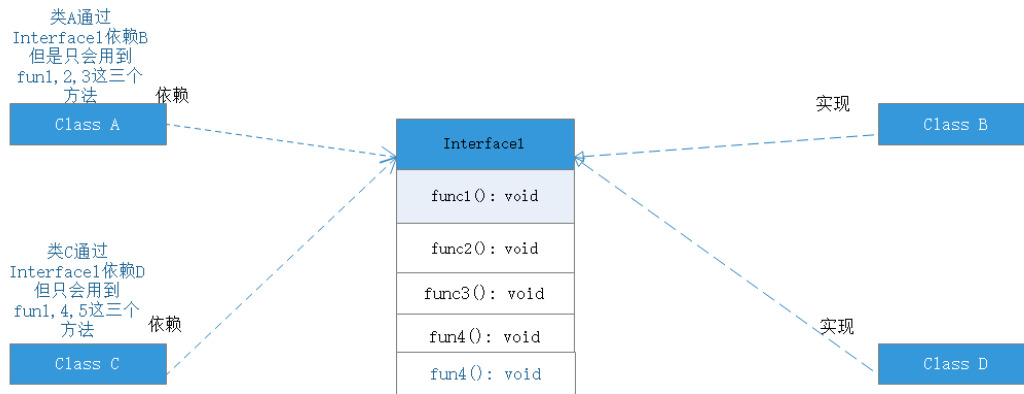
单一职责原则注意事项和细节

1. 降低类的复杂度，一个类只负责一项职责
2. 提高类的可读性，可维护性
3. 降低变更需求引起的风险
4. 通常情况下，我们应当遵守单一职责原则，只有逻辑足够简单，才可以在代码上违反单一职责原则；只有类中的方法足够少，才可以在方法级别保持单一原则

接口隔离原则

1. 客户端不应该依赖它不需要的接口，即一个类对另一个类的依赖应该建立在最小的接口上

看下下面这张图



转换为代码后

```
public class SegregationOne {
    public static void main(String[] args) {

        A a = new A();
        B b = new B();

        C c = new C();
        D d = new D();

        a.depend1(b); //A类通过接口依赖(使用)B类
        a.depend2(b);
        a.depend3(b);

        c.depend1(d);
        c.depend4(d);
        c.depend5(d);
    }
}

/**
 * 接口
 */
interface Interface1{
    void func1();
    void func2();
    void func3();
    void func4();
    void func5();
}

/**
 * 实现类 B
 */
class B implements Interface1{

    @Override
    public void func1() {
        System.out.println("B实现方法1");
    }
}
```

```

    }

    @Override
    public void func2() {
        System.out.println("B实现方法2");
    }

    @Override
    public void func3() {
        System.out.println("B实现方法3");
    }

    @Override
    public void func4() {
        System.out.println("B实现方法4");
    }

    @Override
    public void func5() {
        System.out.println("B实现方法5");
    }
}

/**
 * 实现类 D
 */
class D implements Interface1{

    @Override
    public void func1() {
        System.out.println("D实现方法1");
    }

    @Override
    public void func2() {
        System.out.println("D实现方法2");
    }

    @Override
    public void func3() {
        System.out.println("D实现方法3");
    }

    @Override
    public void func4() {
        System.out.println("D实现方法4");
    }

    @Override
    public void func5() {
        System.out.println("D实现方法5");
    }
}

/**
 * 依赖类 A
 */
class A{

```

```

    public void depend1(Interface1 i){
        i.func1();
    }
    public void depend2(Interface1 i){
        i.func2();
    }
    public void depend3(Interface1 i){
        i.func3();
    }
}

/**
 * 依赖类 C
 */
class C{

    public void depend1(Interface1 i){
        i.func1();
    }
    public void depend4(Interface1 i){
        i.func4();
    }
    public void depend5(Interface1 i){
        i.func5();
    }
}

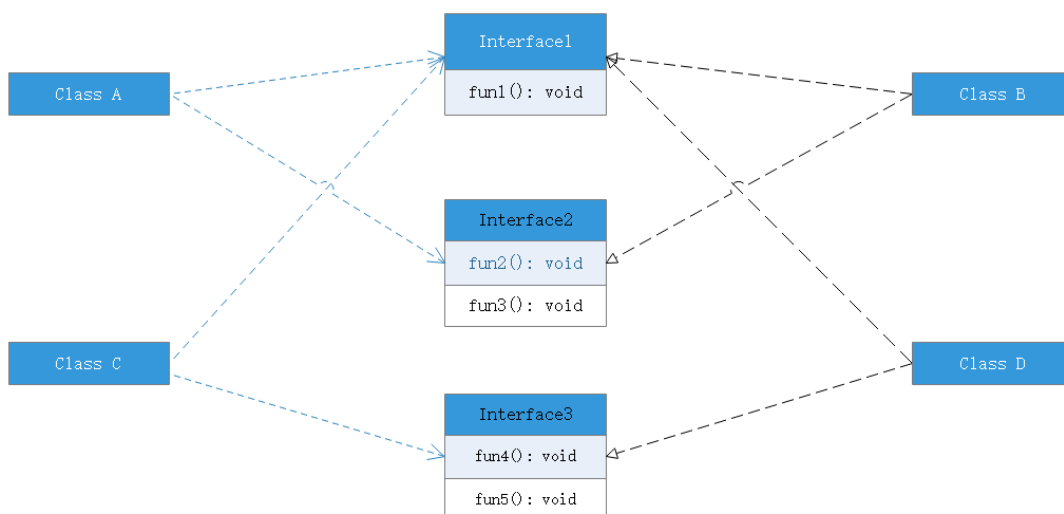
```

类 A 通过接口 Interface1 依赖类 B，类 C 通过接口 Interface1 依赖类 D，如果接口 Interface1 对于类 A 和类 C 来说不是最小接口，那么类 B 和类 D 就必须去实现他们不需要的方法

按照接口隔离原则，应当这样处理

将接口 Interface1 拆分为独立的几个接口，类 A 和类 C 分别与他们需要的接口建立依赖关系，就达成了接口隔离原则

因此，我们可以对接口做如下拆分



实现代码如下

```

public class SegregationTwo {

```

```

    public static void main(String[] args) {

        A1 a = new A1();
        B1 b = new B1();

        C1 c = new C1();
        D1 d = new D1();

        a.dependeFun1(b);
        a.dependeFun2(b);
        a.dependeFun3(b);

        c.dependeFun1(d);
        c.dependeFun4(d);
        c.dependeFun5(d);
    }
}

interface Interface1{

    void fun1();

}

interface Interface2{

    void fun2();
    void fun3();

}

interface Interface3{

    void fun4();
    void fun5();

}

/**
 * 实现类 B
 * */
class B1 implements Interface1, Interface2{

    @Override
    public void fun1() {
        System.out.println("B实现接口1的方法1");
    }

    @Override
    public void fun2() {
        System.out.println("B实现接口2的方法2");
    }

    @Override
    public void fun3() {
        System.out.println("B实现接口2的方法3");
    }

}

```

```

/**
 * 实现类 D
 */
class D1 implements Interface11, Interface3{

    @Override
    public void fun1() {
        System.out.println("D实现接口1的方法1");
    }

    @Override
    public void fun4() {
        System.out.println("D实现接口3的方法4");
    }

    @Override
    public void fun5() {
        System.out.println("D实现接口3的方法5");
    }
}

/**
 * 依赖类 A
 */
class A1{

    public void dependeFun1(Interface11 i){
        i.fun1();
    }

    public void dependeFun2(Interface2 i){
        i.fun2();
    }

    public void dependeFun3(Interface2 i){
        i.fun3();
    }
}

class C1{

    public void dependeFun1(Interface11 i){
        i.fun1();
    }

    public void dependeFun4(Interface3 i){
        i.fun4();
    }

    public void dependeFun5(Interface3 i){
        i.fun5();
    }
}

```


依赖倒转原则

依赖倒转原则是指

1. 高层模块不应该依赖低层模块，二者都应该依赖其抽象
2. 抽象不应该依赖细节，**细节应该依赖抽象**
3. 依赖倒转的中心思想是**面向接口编程**
4. 相对于细节的多变性，抽象的东西要稳定的多。以**抽象为基础**搭建的架构比以**细节为基础**的架构**要稳定的多**。在 Java 中，抽象指的是接口或抽象类，细节就是具体的实现类
5. 使用**接口或抽象类**的目的是**制定规范**，而不设计任何具体操作，把**展现细节**的任务交给他们的**实现类**去完成

看看下面这个例子

```
public class DependencyInversion {
    public static void main(String[] args) {

        Person person = new Person();
        person.receive(new Email());
    }
}

class Person{

    /**
     * 接收消息
     * @param email: 电子邮件类
     */
    public void receive(Email email){
        System.out.println(email.getInfo());
    }
}

class Email{

    public String getInfo(){
        return "邮件信息: hello";
    }
}
```

可以看见，Person 类中的方法依赖了 Email 类，很大的弊端就是 **Email 这个类的抽象层次太低，而 receive() 这个方法的抽象层次太高**，如果传来的是来自微信，QQ，短信等不同类的消息，那么使用 Email 接收显然就是错误的了。

因此可以**抽象出接口**，然后 receive() 依赖于这个接口，可以接收来自于不同类的消息

```
public class DependencyInversionBetter {
    public static void main(String[] args) {
```

```

        Email1 email1 = new Email1();
        Weixin weixin = new Weixin();
        Person1 person1 = new Person1();

        person 1.recevie(email1);
        person1.recevie(weixin);
    }
}

/**
 * 抽象出的接口
 */
interface GetInfo{
    String getInfo();
}

class Email1 implements GetInfo{

    @Override
    public String getInfo() {
        return "电子邮件信息: hello ";
    }
}

class Weixin implements GetInfo{

    @Override
    public String getInfo() {
        return "微信信息: hello ";
    }
}

class Person1{

    public void recevie(GetInfo getInfo){
        System.out.println(getInfo.getInfo());
    }
}

```

依赖关系传递的三种方式

- 接口传递
- 构造函数传递
- setter 方法传递

依赖倒转原则的注意事项和细节

1. 低层模块尽量都要有抽象类或接口，或者两者都有，程序稳定性更好
2. **变量的声明类型尽量是抽象类或接口**，这样变量引用和实际对象间就存在一个缓冲层，利于程序扩展和优化
3. 继承时遵循里氏替换原则

里氏替换原则

里氏替换原则通俗来讲就是：**子类可以扩展父类的功能，但不能改变父类原有的功能**

继承带来的弊端

1. 父类中凡是已经实现了的方法，实际上是在设定规范，虽然其并不要求子类必须遵循这些契约，但是如果子类对这些已经实现的方法任意修改，就会对整个继承体系造成破坏
2. 继承会给程序带来侵入性，程序的可移植性降低，增加了对象间的耦合，如果一个类被其他类所继承，则当这个类需要修改时，必须考虑到所有的子类。父类修改后，所有涉及到父类的功能都可能出现故障

里氏替换原则其实是告诉我们继承需要注意的问题和需要遵守的规则

里氏替换原则的基本内容

- 如果对每个类型为 T1 的对象 o1，都有类型为 T2 的对象 o2，使得以 T1 定义的所有程序 P 在所有的对象 o1 都替换成 o2 时，程序 P 的功能行为没有发生变化，那么类型 T2 就是类型 T1 的子类型。

即：所有引用基类的地方必须能透明地使用其子类对象

- 在使用继承时，遵循里氏替换原则，在子类中尽量不要重写父类方法
- 在适当的情况下，继承造成的耦合性问题，可以通过聚合，组合，依赖等关系来解决

来看看下面这段代码

```
public class LiExchangePrinciple {
    public static void main(String[] args) {

        A a = new A();
        System.out.println("20-7="+a.fuc1(20, 7));
        System.out.println("1-10="+a.fuc1(1, 10));

        B b = new B();
        System.out.println("20-7="+b.fuc1(20, 7));
        System.out.println("1-10="+b.fuc1(1, 10));
        System.out.println("11+3+9="+b.fuc2(11, 3));
    }
}

/**
 * 类 A
 */
class A{
    /**
     *
     * @param num1
     * @param num2
     * @return: 两个数的差
     */
    public int fuc1(int num1, int num2){
        return num1-num2;
    }
}

/**
 * 类 B，继承类 A
 */
```

```

class B extends A{

    @Override
    public int fuc1(int a, int b){
        return a+b;
    }

    /**
     *
     * @param a
     * @param b
     * @return: 两数相加后再与9相加，然后返回
     */
    public int fuc2(int a, int b){
        return fuc1(a, b) + 9;
    }
}

```

```

20-7=13
1-10=-9
20-7=27
1-10=11
11+3+9=23

```

很明显，由于 B 类修改了从 A 类继承下来的方法，导致结果不是我们想要的

通常的改进方法是：原来的父类和子类都继承一个更加通用的基类，去掉原有的继承关系，采用依赖，聚合，组合等关系代替

因此，可以更改为如下代码

```

public class LiExchangePrincipleBetter {
    public static void main(String[] args) {

        A1 a = new A1();
        System.out.println("20-7="+a.fuc1(20, 7));
        System.out.println("1-10="+a.fuc1(1, 10));

        B1 b = new B1();
        System.out.println("20-7="+b.fuc1(20, 7));
        System.out.println("1-10="+b.fuc1(1, 10));
        System.out.println("11+3+9="+b.fuc2(11, 3));
    }
}

class Base{

    public int fuc1(int num1, int num2){
        return num1-num2;
    }
}

class A1 extends Base{

```

```

}

class B1 extends Base{

    public int fuc2(int a, int b){
        return fuc1(a, b)+9;
    }
}

```

B1 在父类 Base 的基础上，只做增加不做修改

开闭原则

- 开闭原则是编程中最基础，最重要的设计原则
- 一个软件实体中的类，模块，函数等应该 面向扩展开放(对提供方)，面向修改关闭(对使用方)，即扩展了提供方的代码后，使用方的代码不用做修改
- 用抽象构建框架，用实现扩展细节
- 当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化
- 在编程中遵循其它原则，以及使用设计模式的目的就是遵循开闭原则

看下下面这段代码

```

public class OpenAndClose {
    public static void main(String[] args) {

        GraphEditorUtils graphEditor = new GraphEditorUtils();
        graphEditor.drawShape(new Cricle());
        graphEditor.drawShape(new ZhengFangXing());
    }
}

/**
 * 绘图工具类
 */
class GraphEditorUtils{

    public void drawShape(Shape shape){
        if(shape.type == 1){
            drawCricle();
        }else if(shape.type == 2){
            drawZhengFang();
        }
    }

    private void drawCricle(){
        System.out.println("画一个圆形");
    }

    private void drawZhengFang(){
        System.out.println("画一个正方形");
    }
}

```

```

}

class Shape{
    //图形类型
    int type;
}

class Cricle extends Shape{

    public Cricle(){
        super.type = 1;
    }
}

class ZhengFangXing extends Shape{

    public ZhengFangXing(){
        super.type = 2;
    }
}

```

这种方式其实并不满足开闭原则，当我们需要新增加一个图形（如三角形时），需要做如下修改

```

public class OpenAndClose {
    public static void main(String[] args) {

        GraphEditorUtils graphEditor = new GraphEditorUtils();
        graphEditor.drawShape(new Cricle());
        graphEditor.drawShape(new ZhengFangXing());

        //增加绘制三角
        graphEditor.drawShape(new SanJiao());
    }
}

/**
 * 绘图工具类，使用方
 */
class GraphEditorUtils{

    public void drawShape(Shape shape){
        if(shape.type == 1){
            drawCricle();
        }else if(shape.type == 2){
            drawZhengFang();
        }else if(shape.type == 3){
            drawSanJiao();
        }
    }

    private void drawCricle(){
        System.out.println("画一个圆形");
    }

    private void drawZhengFang(){
        System.out.println("画一个正方形");
    }
}

```

```

    /**
     * 由于新增了一个图形，因此需要在使用方新增一个方法
     */
    private void drawSanJiao(){

    }

}

class Shape{
    //图形类型
    int type;
}

class Cricle extends Shape{

    public Cricle(){
        super.type = 1;
    }
}

class ZhengFangXing extends Shape{

    public ZhengFangXing(){
        super.type = 2;
    }
}

/**
    新增三角形类
    */
class SanJiao extends Shape{

    public SanJiao(){
        super.type = 3;
    }
}

```

很明显，我们在使用的方修改了代码，这并不满足开闭原则

因此，我们可以改进，比如将 Shape 做成抽象类或接口，然后增加一个 draw() 方法。这样就只需要让新的图形类继承或者实现 Shape，并实现 draw() 方法即可

```

public class OpenAndCloseBetter {
    public static void main(String[] args) {

        GraphEditorUtils1 graphEditor = new GraphEditorUtils1();
        graphEditor.drawShape(new Cricle1());
        graphEditor.drawShape(new ZhengFangXing1());

        //增加绘制三角
        graphEditor.drawShape(new SanJiao1());
    }
}

class GraphEditorUtils1{

```

```

        public void drawShape(Shape1 shape){
            shape.draw();
        }
    }

    abstract class Shape1{
        //图形类型
        int type;

        abstract public void draw();
    }

    class Cricle1 extends Shape1{

        @Override
        public void draw() {
            System.out.println("画圆");
        }
    }

    class ZhengFangXing1 extends Shape1{

        @Override
        public void draw() {
            System.out.println("画正方形");
        }
    }

    class SanJiao1 extends Shape1{

        @Override
        public void draw() {
            System.out.println("画三角");
        }
    }
}

```

在这种情况下，新增一个新的图形，并不需要修改使用方的代码，满足 ocp 原则

迪米特法则

迪米特法则又叫做 **最少知道原则**，即一个类对自己依赖的类知道的越少越好。对于被依赖的类不管其多么复杂，都尽量把逻辑封装在类的内部，对外除了提供 public 的方法，不再泄露其它任何信息

1. 一个对象应该对其他对象保持最少的了解
2. 类与类的关系越密切，耦合度越高

迪米特法则有一个抽象的说法：**只与直接的朋友通信**

直接的朋友：每个对象都会与其它对象产生耦合关系，只要两个对象之间有耦合关系，就称两个对象之间是朋友关系。其中，称出现在成员变量，方法参数，方法返回值中的类为直接的朋友，而出现在局部变量中的类不是直接的朋友。也就是说，最好不要将其它对象以局部变量的形式放在类的中

```

/**
 * 有一个学校，下属各个学院和总部
 * 现打印出学校总部员工id和学院员工的id

```



```

    */
    public class Dimite {
        public static void main(String[] args) {

            SchoolManager schoolManager = new SchoolManager();
            schoolManager.printAllEmployee(new CollegeManager());
        }
    }

    /**
     * 学校总部员工类
     */
    class Employee {
        private String id;

        public void setId(String id) {
            this.id = id;
        }

        public String getId() {
            return id;
        }
    }

    /**
     * 学院员工类
     */
    class CollegeEmployee {
        private String id;

        public void setId(String id) {
            this.id = id;
        }

        public String getId() {
            return id;
        }
    }

    /**
     * 学院管理类
     */
    class CollegeManager {

        /**
         *
         * @return: 学院所有员工
         */
        public List<CollegeEmployee> getAllEmployee() {

            List<CollegeEmployee> list = new ArrayList<CollegeEmployee>();
            for (int i = 0; i < 10; i++) {
                CollegeEmployee emp = new CollegeEmployee();
                emp.setId("学院员工id= " + i);
                list.add(emp);
            }
            return list;
        }
    }

```

```

}

/**
 * 学校总部管理类
 */
class SchoolManager {

    public List<Employee> getAllEmployee() {

        List<Employee> list = new ArrayList<Employee>();
        for (int i = 0; i < 5; i++) {
            Employee emp = new Employee();
            emp.setId("学校总部员工id= " + i);
            list.add(emp);
        }
        return list;
    }

    /**
     *
     * @param sub: 学院管理类
     */
    void printAllEmployee(CollegeManager sub) {

        //获取所有的学院员工
        List<CollegeEmployee> list1 = sub.getAllEmployee();
        System.out.println("-----学院员工-----");
        for (CollegeEmployee e : list1) {
            System.out.println(e.getId());
        }
        //获取所有的学校总部员工
        List<Employee> list2 = this.getAllEmployee();
        System.out.println("-----学校总部员工-----");
        for (Employee e : list2) {
            System.out.println(e.getId());
        }
    }
}

```

分析一下这段代码中的 SchoolManager 类中，有哪些直接朋友和陌生人

方法参数，方法返回值，成员变量上的对象称为直接朋友，所以 SchoolManager 的直接朋友有：Employee，CollegeManager

陌生人有：CollegeEmployee，以局部变量的方式出现在了 SchoolManager 中

因此违背了迪米特法则

对于这段代码，通过分析可得知，主要是因为

```

//获取所有的学院员工
List<CollegeEmployee> list1 = sub.getAllEmployee();
System.out.println("-----学院员工-----");
for (CollegeEmployee e : list1) {
    System.out.println(e.getId());
}

```

这里导致了迪米特法则的不满足，因此我们可以将这段代码拆分到 CollegeManager 类中成为一个 public 的方法，再通过调用来满足迪米特法则

```
public class DiMiteBetter {
    public static void main(String[] args) {

        SchoolManager1 schoolManager1 = new SchoolManager1();
        schoolManager1.printAllEmployee(new CollegeManager1());
    }
}

/**
 * 学校总部员工类
 */
class Employee1 {
    private String id;

    public void setId(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}

/**
 * 学院员工类
 */
class CollegeEmployee1 {
    private String id;

    public void setId(String id) {
        this.id = id;
    }

    public String getId() {
        return id;
    }
}

/**
 * 学院管理类
 */
class CollegeManager1 {

    /**
     *
     * @return: 学院所有员工
     */
    public List<CollegeEmployee1> getAllEmployee() {

        List<CollegeEmployee1> list = new ArrayList<>();
        for (int i = 0; i < 10; i++) {
            CollegeEmployee1 emp = new CollegeEmployee1();
            emp.setId("学院员工id= " + i);
        }
    }
}
```

```

        list.add(emp);
    }
    return list;
}

/**
 * 获取所有学院员工
 */
public void printAllEmployee() {

    //获取所有的学院员工
    List<CollegeEmployee1> list1 = this.getAllEmployee();
    System.out.println("-----学院员工-----");
    for (CollegeEmployee1 e : list1) {
        System.out.println(e.getId());
    }
}

/**
 * 学校总部管理类
 */
class SchoolManager1 {

    public List<Employee1> getAllEmployee() {

        List<Employee1> list = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            Employee1 emp = new Employee1();
            emp.setId("学校总部员工id= " + i);
            list.add(emp);
        }
        return list;
    }

    /**
     * 获取所有员工
     */
    void printAllEmployee(CollegeManager1 sub) {

        //通过直接朋友调用其中的方法
        sub.printAllEmployee();

        //获取所有的学校总部员工
        List<Employee1> list2 = this.getAllEmployee();
        System.out.println("-----学校总部员工-----");
        for (Employee1 e : list2) {
            System.out.println(e.getId());
        }
    }
}

```

迪米特法则的注意事项和细节

- 迪米特法则的核心是降低类之间的耦合
- 但是，由于每个类都减少了不必要的依赖，因此迪米特法则只是要求降低类间（对象间）耦合关系，并不是要求完全没有耦合关系

合成复用原则

尽量使用合成 / 聚合等耦合度较低的关系，而不是使用继承

总结

设计原则的核心思想

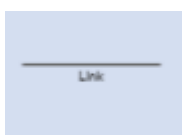

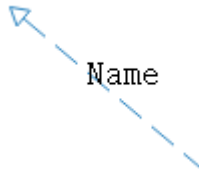


- 找出应用中可能需要变化的地方，把他们独立出来，不要和那些不需要变化的代码混在一起
- 面向接口编程，而不是面向实现编程
- 为了交互对象之间的松耦合设计而努力

UML 类图

UML 统一建模语言，是一种用于软件系统分析和设计的语言工具

UML 本身是一套符号的规定，这些符号用于描述软件模型中的各个元素和他们之间的关系。如：类，接口，实现，泛化，依赖，组合，聚合等

UML 类图的关系

- 依赖 (使用) 
- 关联 (1:n, n:1 等关系) 
- 泛化 (继承) 
- 实现 
- 聚合  是关联关系的特例
- 组合  是关联关系的特例

UML 的分类

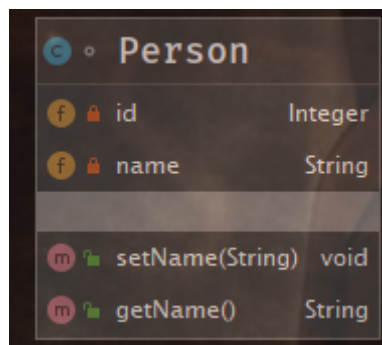
- 用例图

- 静态结构图：类图，对象图，包图，组件图，部署图
- 动态行为图：交互图（时序图和协作图），状态图，活动图

类图是 UML 中最核心的

将代码转化为类图

```
class Person{  
  
    private Integer id;  
    private String name;  
  
    public void setName(String name){  
        this.name = name;  
    }  
  
    public String getName(){  
        return this.name;  
    }  
}
```

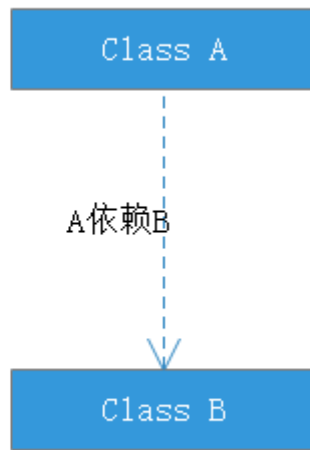


idea 自带类转类图的功能，右键即可

依赖，泛化，实现

依赖

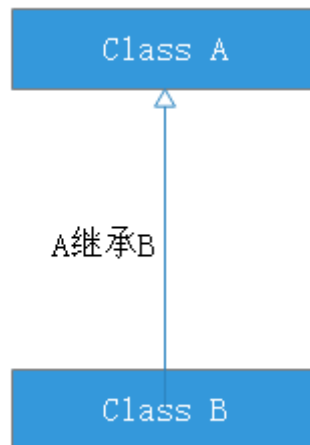
只要**在类中用到了对方**，那么两者之间就存在依赖关系



箭头指向被依赖的类

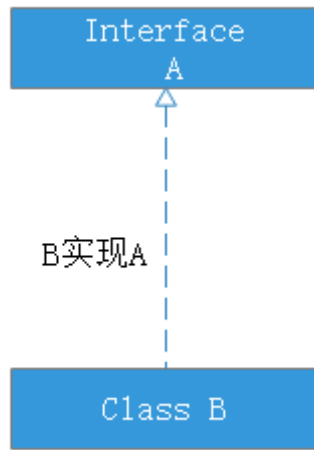
泛化

泛化关系其实就是**继承关系**，是依赖关系的特例



实现

实现关系就是一个类实现另一个类 (接口)，也是依赖关系的特例



关联，聚合，组合

关联

关联关系就是类与类之间的联系，也是依赖关系的特例

关联具有导航性，即双向关系和单向关系

同也有一对一，多对一，一对多，多对多的多重性

如

```
//单向一对一关系
public class Person{
    private IDCard card;
}
public class IDCard{ }

//双向一对一关系
public class Person{
    private IDCard card;
}
public class IDCard{
    private Person person;
}
```

聚合

聚合关系表示的是整体和部分的关系，整体和部分可以分开。聚合关系是关联关系的特例，所以聚合关系具有关联关系的导航性和多重性

如：一台电脑由键盘，显示器，鼠标等组成。而组成电脑的这些配件是可以从电脑上分离出来的，这种情况下认为键盘，显示器，鼠标等配件和电脑的关系是聚合关系


```

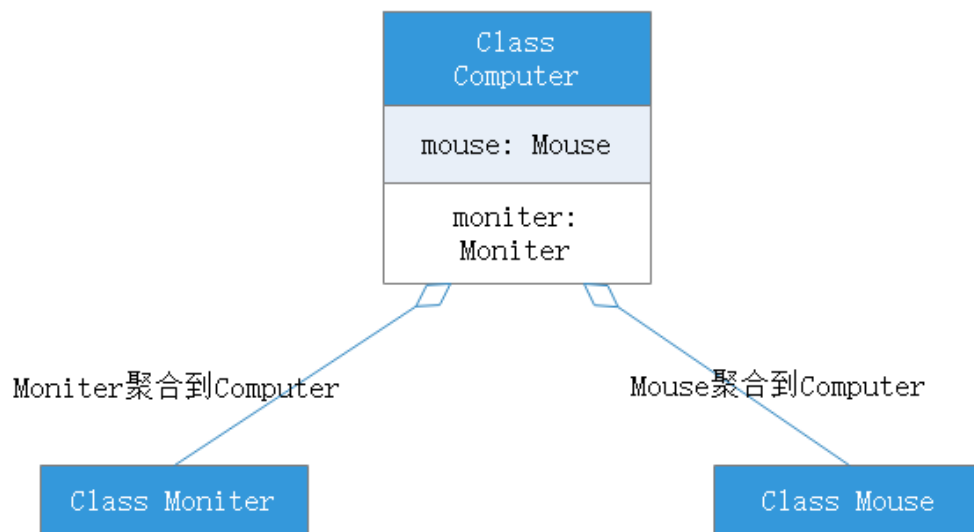
public class Computer{
    private Mouse mouse;
    private Moniter moniter;

    public void setMouse(Mouse mouse){
        this.mouse = mouse;
    }

    public void setMoniter(Moniter moniter){
        this.moniter = moniter;
    }
}

```

Computer 中的 Mouse, Moniter 在 Computer 创建时是不存在的, 只有在需要的时候才会 set (聚合) 进去



组合

组合关系, 也是整体与部分的关系, 但是整体与部分是不可分的

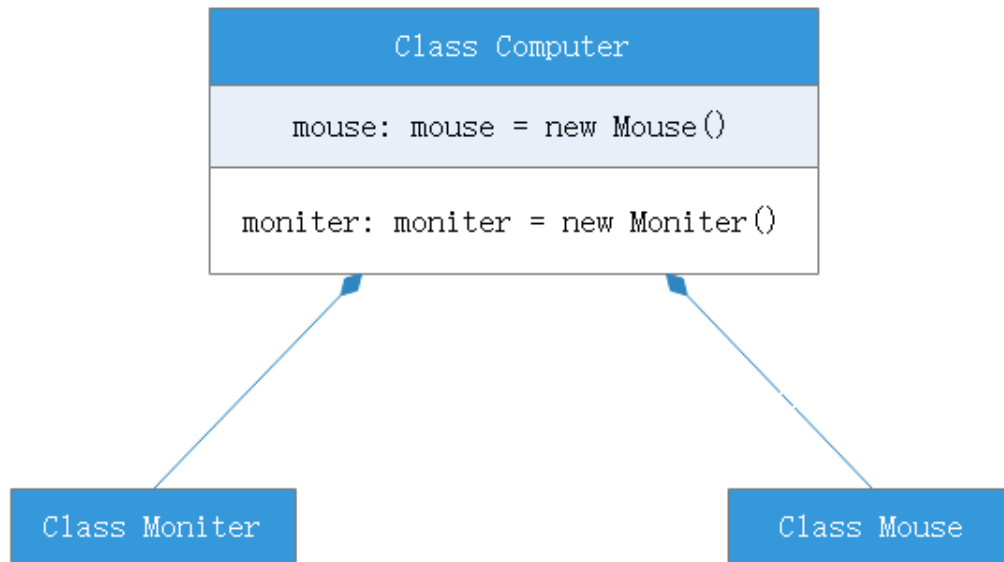
借用聚合关系的例子, 如果我们认为 Mouse, Moniter 和 Computer 是不可分离的, 则升级为组合关系

```

public class Computer{
    private Mouse mouse = new Mouse();
    private Moniter moniter = new Moniter();
}

```

在这个例子下, 如果我们创建了一个 Computer 对象, 那 Mouse 和 Moniter 也会被融入其中, 成为 Computer 不可分离的一部分



比如：Person, IDCard, Head。在这三个实体中，Person 和 Head 就是组合关系，Person 和 IDCard 就是聚合关系

但是，如果在程序中，Person 实体类定义了对 IDCard 的**级联删除**，即删除 Person 时连同 IDCard 一起删除，那么 IDCard 和 Person 就**是组合**了