三.设计模式七大原则

设计模式原则，其实就是程序员在编程时，应当遵守的原则，也是各种设计模式的基础(即：设计模式为什么这样设计的依据)

1) 单一职责原则
2) 接口隔离原则
3) 依赖倒转(倒置)原则
4) 里氏替换原则
5) 开闭原则
6) 迪米特法则
7) 合成复用原则

设计的核心思想

1) 找出应用中可能需要变化之处，把它们独立出来，不要和那些不需要变化的代码混在一起。
2) 针对接口编程，而不是针对实现编程。
3) 为了交互对象之间的松耦合设计而努力

1.单一职责原则

1.基本介绍:

对类来说的，即一个类应该只负责一项职责。如类 A 负责两个不同职责：职责 1，职责 2。当职责 1 需求变更而改变 A 时，可能造成职责 2 执行错误，所以需要将类 A 的粒度分解为 A1，A2

2.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，才可以在代码级违反单一职责原则；只有类中方法数量足够少，可以在方法级别保持单一职责原则

下代码为类级别保持单一职责原则

```
//方案2的分析
//1.遵守单一职责原则
//2.但是这样做的改动很大，即将类分解，同时修改客户端
//3.改进:直接修改Vehicle类，改动的代码会比较少=>方案三
class RoadVehicle{
    public void run(String vehicle){
        System.out.println(vehicle+"公路运行");
    }
}
class AirVehicle{
    public void run(String vehicle){
        System.out.println(vehicle+"天空运行");
    }
}
class WaterVehicle{
    public void run(String vehicle){
        System.out.println(vehicle+"水中运行");
    }
}
```

下代码为方法级别保持单一职责原则

```
//方式3的分析
//1.这种修改方法没有对原来的类做大的修改，只是增加方法
//2.这里虽然没有在类这个级别上遵守单一职责原则,但是在方法级别上，仍然是遵守单一职责
class Vehicle2{
    public void run(String vehicle){
        System.out.println(vehicle+"在公路上运行");
    }

    public void runAir(String vehicle){
        System.out.println(vehicle+"在天空运行");
    }

    public void runWater(String vehicle){
        System.out.println(vehicle+"在水中运行");
    }
}
```

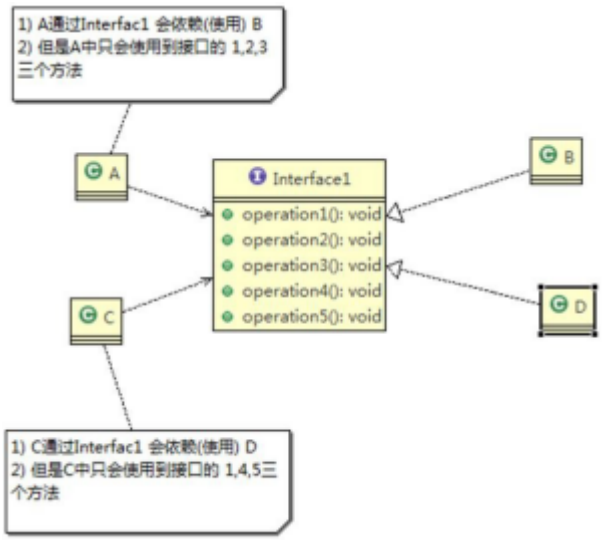
2.接口隔离原则

1. 基本介绍

1.1　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

2.不使用接口隔离原则



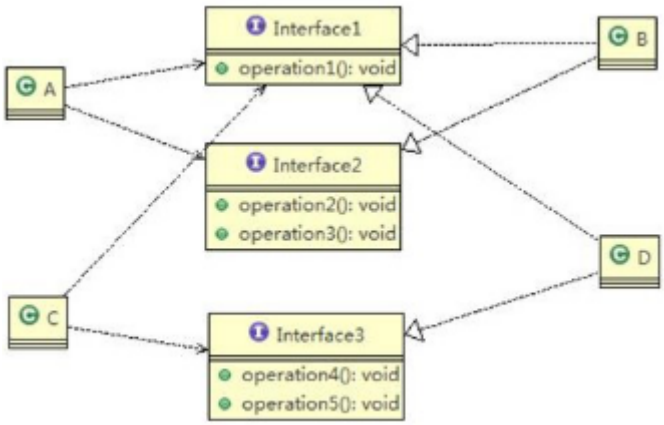
1)　类 A　通过接口 Interface1　依赖类 B，类 C　通过接口 Interface1　依赖类 D，如果接口 Interface1　对于类 A　和类 C来说不是最小接口，那么类 B 和类 D 必须去实现他们不需要的方法。

2)　按隔离原则应当这样处理：

将接口 Interface1 拆分为独立的几个接口(这里我们拆分成 3 个接口)，类 A 和类 C 分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则

```
interface Interface1{
    void operation1();
    void operation2();
    void operation3();
    void operation4();
    void operation5();
}
```

3应传统方法的问题和使用接口隔离原则改进



1） 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 必须去实现他们不需要的方法

2） 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。也就是采用接口隔离原则

3） 接口 Interface1 中出现的方法，根据实际情况拆分为三个接口

```
// 接 口 1
interface Interface1 {
    void operation1();
}

// 接 口 2
interface Interface2 {
    void operation2();
    void operation3();
}

// 接 口 3
interface Interface3 {
    void operation4();
    void operation5();
}
```

3 依赖倒转原则

1.基本介绍

依赖倒转原则(Dependence Inversion Principle)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(倒置)的中心思想是面向接口编程

2.依赖倒转原则是基于这样的设计理念：

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

3 使用接口或抽象类的目的是制定好规范，而不涉及任何具体的操作，把展现细节的任务交给他们的实现类去完成

4.注意细节

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，这样我们的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

```
//定义接口
interface IReceiver {
    public String getInfo();
}

class Email implements IReceiver {
    public String getInfo() {
        return "电子邮件信息: hello,world";
    }
}

//增加微信
class WeiXin implements IReceiver {
    public String getInfo() {
        return "微信信息: hello,ok";
    }
}

class Person {
    //这里我们是对接口的依赖
```

```
    public void receive(IReceiver receiver ) {
        System.out.println(receiver.getInfo());
    }
}
```

4.里氏替换原则

1.OO 中的继承性的思考和说明

1） 继承包含这样一层含义：父类中凡是已经实现好的方法，实际上是在设定规范 和契约，虽然它不强制要求所有的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） 继承在给程序设计带来便利的同时，也带来了弊端。比如使用继承会给程序带来侵入性，程序的可移植性降低， 增加对象间的耦合性，如果一个类被其他的类所继承，则当这个类需要修改时，必须考虑到所有的子类，并且父类修改后，所有涉及到子类的功能都有可能产生故障

3） 问题提出：在编程中，如何正确的使用继承? => 里氏替换原则

2 基本介绍

1） 里氏替换原则(Liskov Substitution Principle)在 1988 年，由麻省理工学院的以为姓里的女士提出的。

2） 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 都代换成 o2 时，程序 P 的行为没有发生变化，那么类型 T2 是类型 T1 的子类型。换句话说，所有引用基类的地方必须能透明地使用其子类的对象。

3） 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的方法

4） 里氏替换原则告诉我们，继承实际上让两个类耦合性增强了，在适当的情况下，可以通过聚合，组合，依赖 来解决问题。.

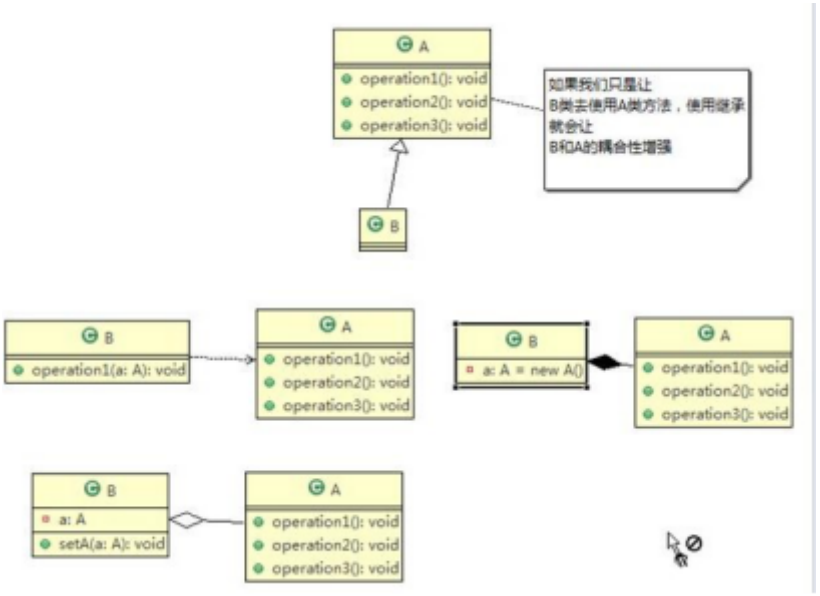
3.使用场景

如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中发生重写或者重载，则建议断开父子继承关系，采用依赖、聚集、组合等关系代替继

承。如果非要继承，那么当重载的时候就一定要使子类方法的参数一定要大于或等于父类的参数。此时父类称为基类，子类拥有父类的全部属性和方法。并且:

有子类出现的地方父类未必就可以出现

父类出现的地方子类就可以出现



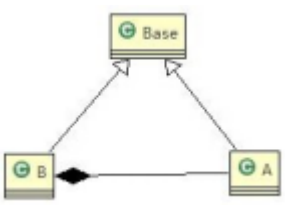
（左上是依赖，左下是聚合，右上是组合）

3.代码实例

1） 我们发现原来运行正常的相减功能发生了错误。原因就是类 B 无意中重写了父类的方法，造成原有功能出现错误。在实际编程中，我们常常会通过重写父类的方法完成新的功能，这样写起来虽然简单，但整个继承体系的复用性会比较差。特别是运行多态比较频繁的时候

2） 通用的做法是：原来的父类和子类都继承一个更通俗的基类，原有的继承关系去掉，采用依赖，聚合，组合等关系代替.

3） 改进方案




```
// A 类
class A extends Base {
```

```
    // 返回两个数的差
    public int func1(int num1, int num2) {
        return num1 - num2;
    }
}

// B 类继承了 A(原先是继承A类，现在进行改进)

// 增加了一个新功能：完成两个数相加,然后和 9 求和
class B extends Base {
    //如果 B 需要使用 A 类的方法,使用组合关系
    private A a = new A();

    //这里，重写了 A 类的方法，可能是无意识
    public int func1(int a, int b) {
        return a + b;
    }

    public int func2(int a, int b) {
        return func1(a, b) + 9;
    }

    //我们仍然想使用 A 的方法
    public int func3(int a, int b) {
        return this.a.func1(a, b);
    }
}
```

5 .开闭原则(具体实现与依赖倒转原则相似)

1.基本介绍

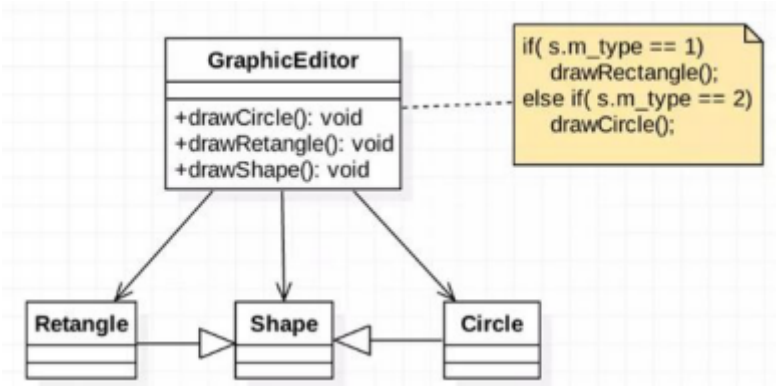
1） 开闭原则（Open Closed Principle）是编程中最基础、最重要的设计原则

2） 一个软件实体如类，模块和函数应该对扩展开放(对提供方)，对修改关闭(对使用方)。用抽象构建框架，用实现扩展细节。

3） 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。

4） 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。

2.不使用开闭原则(具体代码看代码文件夹)



1） 优点是比较好理解，简单易操作。

2） 缺点是违反了设计模式的 ocp 原则，即对扩展开放(提供方)，对修改关闭(使用方)。即当我们给类增加新功能的时候，尽量不修改代码，或者尽可能少修改代码.

3） 比如我们这时要新增加一个图形种类 三角形，我们需要做如下修改，修改的地方较多

4） 代码演示

3.使用ocp原则进行修改

思路：把创建 Shape 类做成抽象类，并提供一个抽象的 draw 方法，让子类去实现即可，这样我们有新的图形种类时，只需要让新的图形类继承 Shape，并实现 draw 方法即可，使用方的代码就不需要修 -> 满足了开闭原则

```
//这是一个用于绘图的类 [使用方] 新增图形时不用此处修改
class GraphicEditor {
  //接收Shape对象，调用draw方法
  public void drawShape(Shape s) {
    s.draw();
  }
}

//Shape类，基类
abstract class Shape {
  int m_type;
```

```
    public abstract void draw();//抽象方法
}

//提供方,新增图形只要继承抽象类或者基类就可以了
class Rectangle extends Shape {
    Rectangle() {
        super.m_type = 1;
    }

    @Override
    public void draw() {
        // TODO Auto-generated method stub
        System.out.println(" 绘制矩形 ");
    }
}
```

6.迪米特法则

1. 基本介绍

1） 一个对象应该对其他对象保持最少的了解

2） 类与类关系越密切，耦合度越大

3） 迪米特法则(Demeter Principle)又叫最少知道原则，即一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类不管多么复杂，都尽量将逻辑封装在类的内部。对外除了提供的 public 方法，不对外泄露任何信息

4） 迪米特法则还有个更简单的定义：只与直接的朋友通信

5） 直接的朋友：每个对象都会与其他对象有耦合关系，只要两个对象之间有耦合关系，我们就说这两个对象之间是朋友关系。耦合的方式很多，依赖，关联，组合，聚合等。其中，我们称出现成员变量，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，陌生的类

最好不要以局部变量的形式出现在类的内部。

2. 迪米特法则注意事项和细节

1） 迪米特法则的核心是降低类之间的耦合

2） 但是注意：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依赖关系

3.代码分析

```
class class SchoolManager{
//该方法完成输出学校总部和学院员工信息(id)
void printAllEmployee(CollegeManager sub) {

  //分析问题
  //1. 这里的 CollegeEmployee 不是  SchoolManager的直接朋友
  //2. CollegeEmployee 是以局部变量方式出现在 SchoolManager
  //3. 违反了 迪米特法则

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

1） 前面设计的问题在于 SchoolManager 中，CollegeEmployee 类并不是 SchoolManager 类的直接朋友（分析）

2） 按照迪米特法则，应该避免类中出现这样非直接朋友关系的耦合

3） 对代码按照迪米特法则 进行改进.,将涉及CollegeEmployee的代码封装到CollegeManager类中,因为CollegeManager类中其他方法出现了CollegeEmployee,是直接朋友。

```
class CollegeManager {
//输出学院员工的信息
public void printEmployee() {
  //获取到学院员工
  List<CollegeEmployee> list1 = getAllEmployee();
  System.out.println("------------学院员工------------");
  for (CollegeEmployee e : list1) {
```

```
      System.out.println(e.getId());
    }
}
}

class SchoolManager{
    void printAllEmployee(CollegeManager sub) {
        //1. 将输出学院的员工方法，封装到CollegeManager
        sub.printEmployee();
        ..............
    }
}
```

7.合成复用原则

基本介绍

原则是尽量使用合成/聚合的方式，而不是使用继承