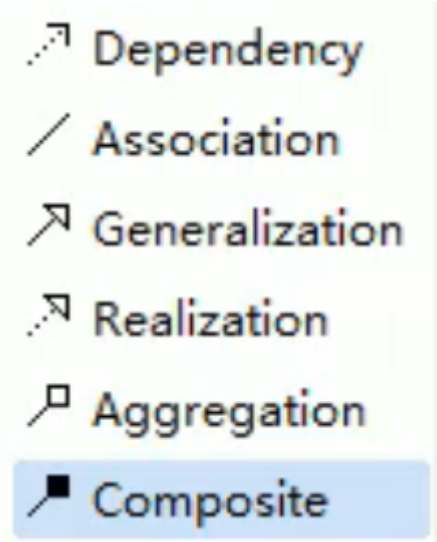
设计模式七大原则

单一职责原则

UML类图



单例模式

工厂模式

原型模式

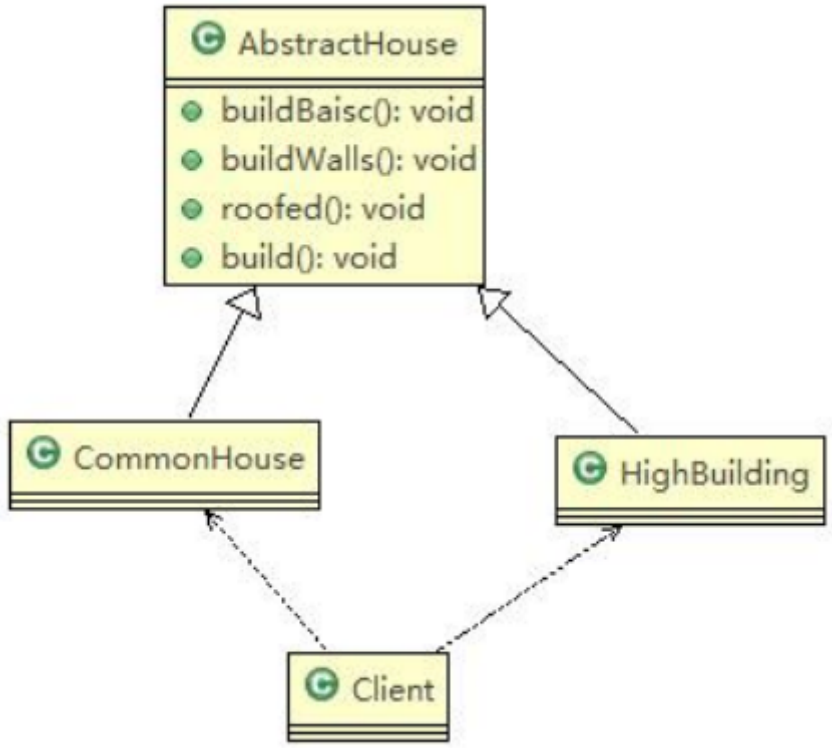
建造者模式

需求

1. 需要建房子：这一过程为打桩、砌墙、封顶
2. 房子有各种各样的，比如普通房，高楼，别墅，各种房子的过程虽然一样，但是要求不要相同的.
3. 请编写程序，完成需求.

传统方式

类图



代码

AbstractHouse.java

```
public abstract class AbstractHouse {
    public abstract void buildBasic();
    public abstract void buildWall();
    public abstract void roofed();
    public void build(){
        buildBasic();
        buildwall();
        roofed();
    }
}
```

CommonHouse.java

```
public class CommonHouse extends AbstractHouse{
    @Override
    public void buildBasic() {
        System.out.println("给普通房子打地基");
    }
    @Override
    public void buildwall() {
```

```
        System.out.println("给普通房子砌墙");
    }
    @Override
    public void roofed() {
        System.out.println("给普通房子封顶");
    }
}
```

Client.java

```
public class Client {
    public static void main(String[] args) {
        CommonHouse commonHouse = new CommonHouse();
        commonHouse.build();
    }
}
```

优缺点

1. 优点是比较好理解，简单易操作。
2. 设计的程序结构，过于简单，没有设计缓存层对象，程序的扩展和维护不好. 也就是说，这种设计方案，把产品(即：房子) 和 创建产品的过程(即：建房子流程) 封装在一起，耦合性增强了。
3. 解决方案：将产品和产品建造过程解耦 => 建造者模式.

建造者模式

基本介绍

1) 建造者模式（Builder Pattern）又叫生成器模式，是一种对象构建模式。它可以将复杂对象的建造过程抽象出来（抽象类别），使这个抽象过程的不同实现方

法可以构造出不同表现（属性）的对象。

2) 建造者模式 是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。

四个角色

建造者模式的四个角色

1) Product（产品角色）：一个具体的产品对象。

2) Builder（抽象建造者）：*创建一个Product对象的各个部件指定的 接口/抽象类。

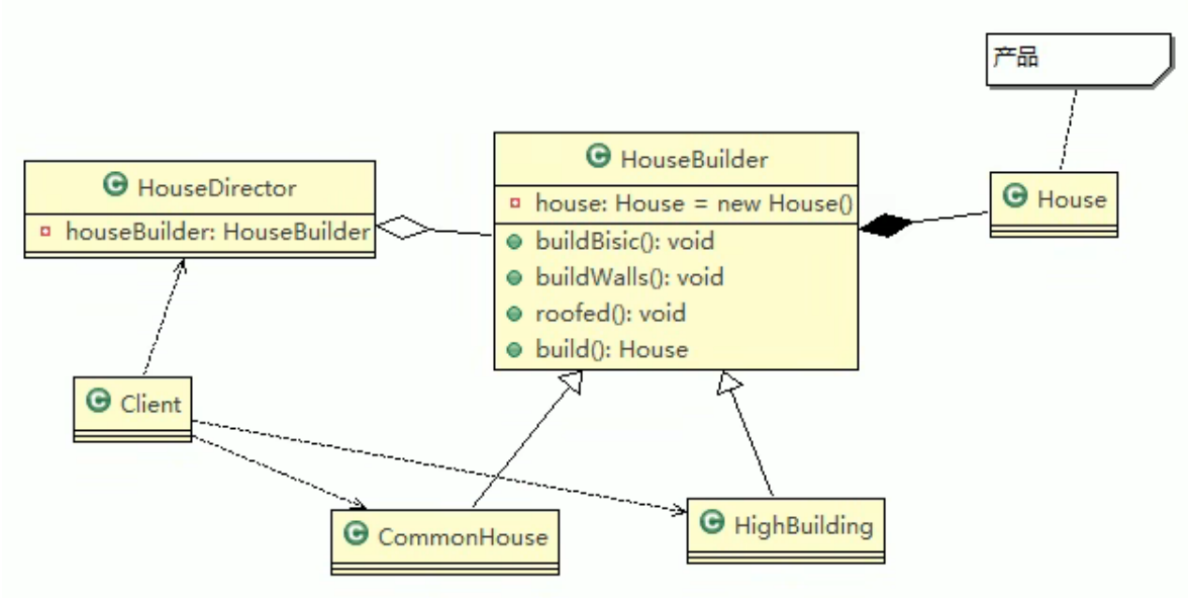
3) ConcreteBuilder（具体建造者）： 实现接口，构建和装配各个部件。

4) Director（指挥者）： 构建一个使用Builder接口的对象。它主要是用于创建一个复杂的对象。它主要有两个作用，

一是：隔离了客户与对象的生产过程

二是：负责控制产品对象的生产过程

类图



代码

House.java 产品

```
public class House {
    private String basic;//地基
    private String wall;
    private String roof;
    public String getBasic() {
        return basic;
    }
    public void setBasic(String basic) {
        this.basic = basic;
    }
    public String getWall() {
        return wall;
    }
    public void setWall(String wall) {
```

```
        this.wall = wall;
    }
    public String getRoof() {
        return roof;
    }
    public void setRoof(String roof) {
        this.roof = roof;
    }

    @Override
    public String toString() {
        return "House{" +
                "basic='" + basic + '\'' +
                ", wall='" + wall + '\'' +
                ", roof='" + roof + '\'' +
                '}';
    }
}
```

HouseBuilder.java 抽象建造者

```
public abstract class HouseBuilder {
    House house = new House();
    public abstract void buildBasic();
    public abstract void buildWall();
    public abstract void roof();
    public House build(){
        return house;
    }
}
```

CommonHouse.java 具体建造者（取个CommonHouseBuilder更好些）

```
public class CommonHouse extends HouseBuilder{
    @Override
    public void buildBasic() {
        this.house.setBasic("普通地基");
    }
    @Override
    public void buildWall() {
        this.house.setWall("普通砌墙");
    }
    @Override
    public void roof() {
        this.house.setRoof("普通封顶");
    }
}
```

HouseDirector.java 指挥者，组合了建造者，传进来什么类型的建造者就建造什么样的房子

```
public class HouseDirector {
    HouseBuilder houseBuilder;
    public HouseDirector(HouseBuilder houseBuilder) {
        this.houseBuilder = houseBuilder;
    }
    public House constructHouse(){
        houseBuilder.buildBasic();
        houseBuilder.buildWall();
        houseBuilder.roof();
        return houseBuilder.house;
    }
}
```

Client.java

```
public class Client {
    public static void main(String[] args) {
        HouseDirector houseDirector = new HouseDirector(new CommonHouse());
        House house = houseDirector.constructHouse();
        System.out.println(house);
    }
}
```

适配器模式

基本介绍

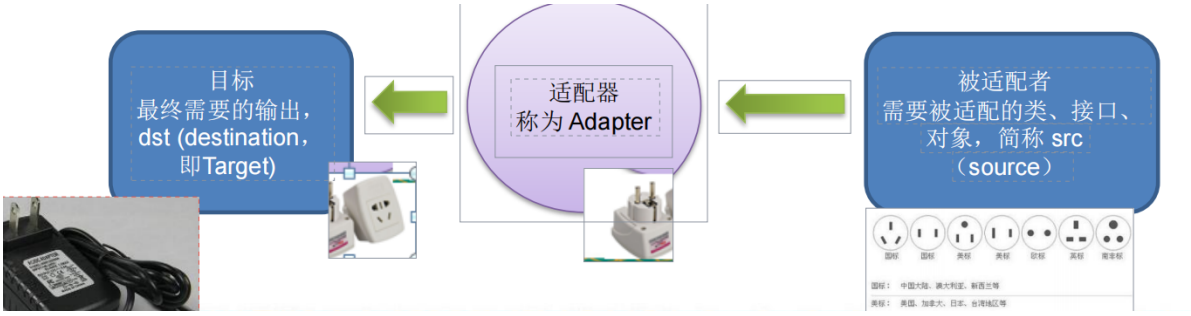
1. 适配器模式(Adapter Pattern)将某个类的接口转换成客户端期望的另一个接口表示，主的目的是兼容性，让原本因接口不匹配不能一起工作的两个类可以协同

工作。其别名为包装器(Wrapper)

2. 适配器模式属于结构型模式
3. 主要分为三类：**类适配器模式、对象适配器模式、接口适配器模式**

工作原理

1. 适配器模式：将一个类的接口转换成另一种接口.让原本接口不兼容的类可以兼容
2. 从用户的角度看不到被适配者，是解耦的
3. 用户调用适配器转化出来的目标接口方法，适配器再调用被适配者的相关接方法
4. 用户收到反馈结果，感觉只是和目标接口交互，如图



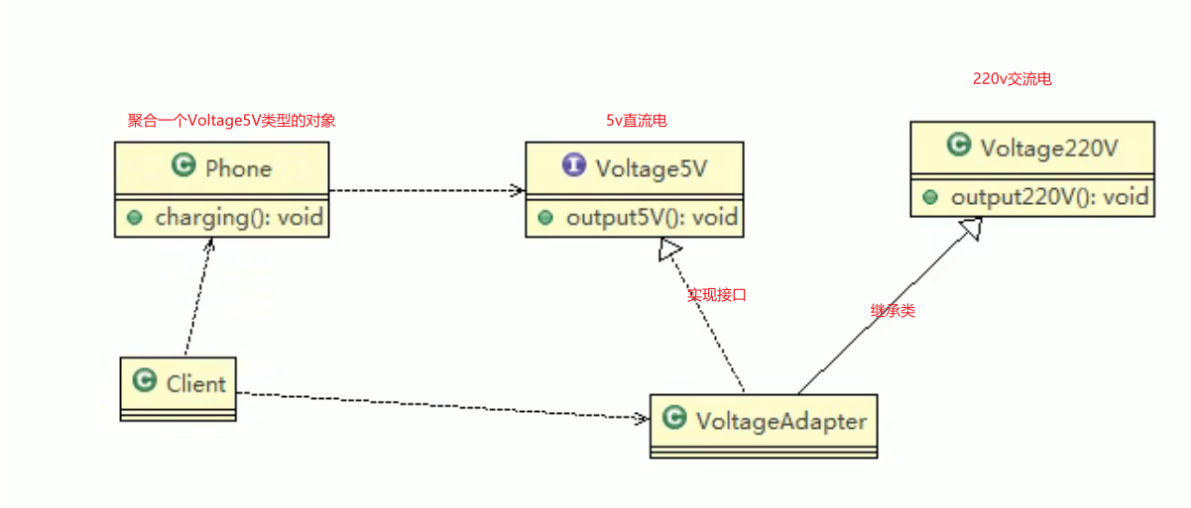
类适配器模式

基本介绍：Adapter类，通过继承src类，实现dst 类接口，完成src->dst的适配。

举例

以生活中充电器的例子来讲解适配器，充电器本身相当于Adapter，220V交流电。相当于src (即被适配者)，我们的目dst(即目标)是5V直流电

类图



代码

Voltage220v.java 被适配类

```
public class Voltage220v {
    public int output220v(){
        System.out.println("这是220v电压");
        return 220;
    }
}
```

Voltage5v 目标类

```
public interface Voltage5v {
    public int output5v();
}
```

VoltageAdapter 适配器类

```
public class VoltageAdapter extends Voltage220v implements Voltage5v{
    @Override
    public int output5v() {
        int src = output220v();
        int dst = src/44;//变压器变压
        System.out.println("转成5v电压...");
        return dst;
    }
}
```

使用者

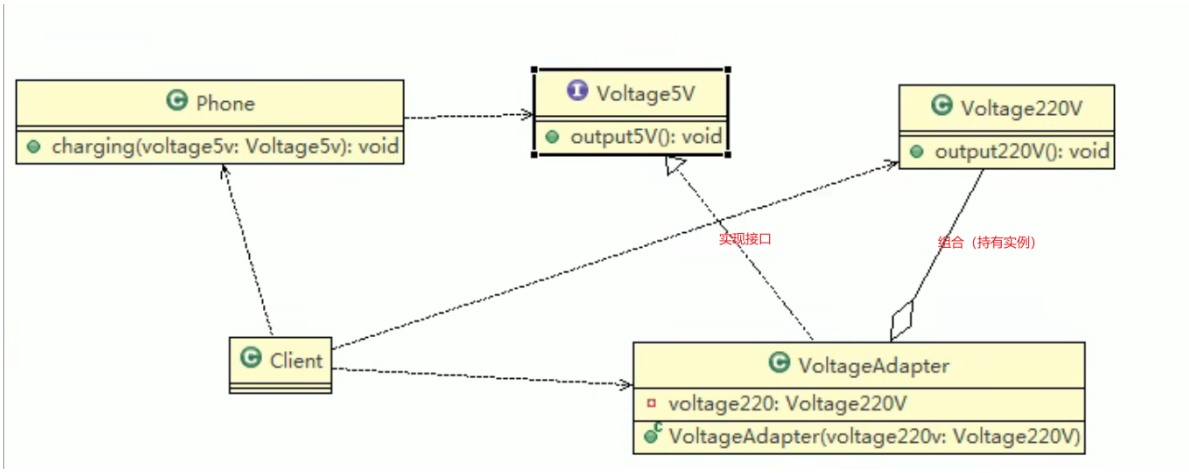
```
public class Phone {
    public void charge(Voltage5v voltage5v){
        if(voltage5v.output5v() == 5){
            System.out.println("可以充电....");
        }
    }
}
```

缺点：继承了src 被适配器类，不好

对象适配器模式

使用关联关系，替代继承

类图



代码

只需要改适配器类

```
public class VoltageAdapter implements Voltage5v{
    private Voltage220v voltage220v;
    public VoltageAdapter(Voltage220v voltage220v) {
        this.voltage220v = voltage220v;
    }
    @Override
    public int output5v() {
        int src = voltage220v.output220v();
        int dst = src/44;//变压器变压
        System.out.println("转成5v电压...");
        return dst;
    }
}
```

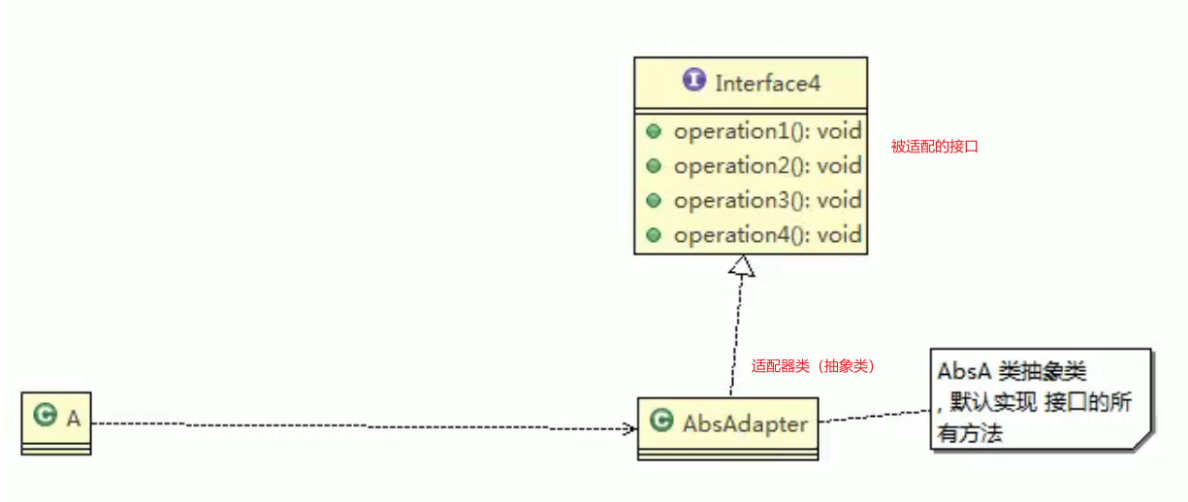
接口适配器模式

原理理解

类适配器模式中，适配器是一个类，被适配的是类，适配器继承了被适配的类

对象适配器中，适配器也是一个类，被适配的是对象，适配器聚合了被适配类型的对象
接口适配器中，适配器是抽象类，被适配的是接口，适配器类继承了被适配的这个接口。但是因为是接口，并不是所有的方法都需要适配怎么办？那就是适配器类是一个抽象类，使用时候可以通过匿名内部类方式。

类图



代码

被适配的接口

```
public interface BeAdaptedInterface {
    public void m1();
    public void m2();
    public void m3();
    public void m4();
}
```

适配器类，抽象类，默认实现所有的方法。没有方法内容

```
public class AbstractAdapter implements BeAdaptedInterface{
    @Override
    public void m1() {}
    @Override
    public void m2() {}
    @Override
    public void m3() {}
    @Override
    public void m4() {}
}
```

目标其实就是这个匿名类创建的对象，只想适配m1就只重写m1方法，

```
public class Client {
    public static void main(String[] args) {
        AbstractAdapter adapter = new AbstractAdapter() {
            @Override
            public void m1() {
                System.out.println("适配方法...");
            }
        };
        adapter.m1();
    }
}
```

桥接模式

装饰着模式

组合模式

外观模式

享元模式

代理模式

模板模式

命令模式

访问者模式

迭代器模式

观察者模式

中介者模式

备忘录模式

解释器模式

状态模式

策略模式

职责链模式