## 单例模式

单例模式的要素：私有的构造方法；指向自己实例的私有静态引用；以自己实例为返回值的静态的公有的方法。

### 饿汉式单例

public class Singleton {

//在类被加载时就实例化一个对象

private static Singleton singleton = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return singleton;

}

}

### 懒汉式单例

public class Singleton {

private static Singleton singleton;

private Singleton(){}

public static synchronized Singleton getInstance(){

if(singleton==null){

singleton = new Singleton();

}

return singleton;

}

}

## 工厂模式

创建对象时不会对客户端暴露创建逻辑，并通过一个共同的接口来指向新创建的对象。工厂模式适用于构建同产品类型（同一个接口基类）的不同对象时，这些对象**new很复杂**，需要很多的参数，而这些参数中大部分都是固定的，即可用工厂模式封装。

### (简单)静态工厂

Spring中的BeanFactory使用了此模式

public interface Shape {//产品类接口

void draw();

}

public class Rectangle implements Shape {//产品的实现类

public void draw() {

System.out.println("Inside Rectangle::draw() method.");

}

}

public class Square implements Shape {

public void draw() {

System.out.println("Inside Square::draw() method.");

}

}

public class ShapeFactory {//创建产品的工厂类

//使用静态getShape 方法获取形状类型的对象

Public static Shape getShape(String shapeType){

if(shapeType == null){

return null;

}

if(shapeType.equalsIgnoreCase("CIRCLE")){

return new Circle();

} else if(shapeType.equalsIgnoreCase("RECTANGLE")){

return new Rectangle();

} else if(shapeType.equalsIgnoreCase("SQUARE")){

return new Square();

}

return null;

}

}

public class FactoryPatternDemo {//使用工厂类创建对象

public static void main(String[] args) {

//获取 Rectangle 的对象，并调用它的 draw 方法

Shape shape = ShapeFactory.getShape("RECTANGLE");

//调用 Rectangle 的 draw 方法

shape.draw();

}

### 使用反射的简单工厂

使用反射机制可以解决每次增加一个产品是都要增加一个对象实现工厂的缺点。

public class ShapeFactory {

public static Object getClass(Class<?extends Shape> clazz) {

Object obj = null;

try {

obj = Class.forName(clazz.getName()).newInstance();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (InstantiationException e) {

e.printStackTrace();

} catch (IllegalAccessException e) {

e.printStackTrace();

}

return obj;

}

}

Rectangle rect = (Rectangle) ShapeFactory.getClass(Rectangle.class);

rect.draw();//调用工厂类时传入class对象

### 多方法工厂

为不同的产品提供不同的生产方法

public class ShapeFactory {

public static Shape getsq(){

return new Square();

}

public static Rectangle getrt(){

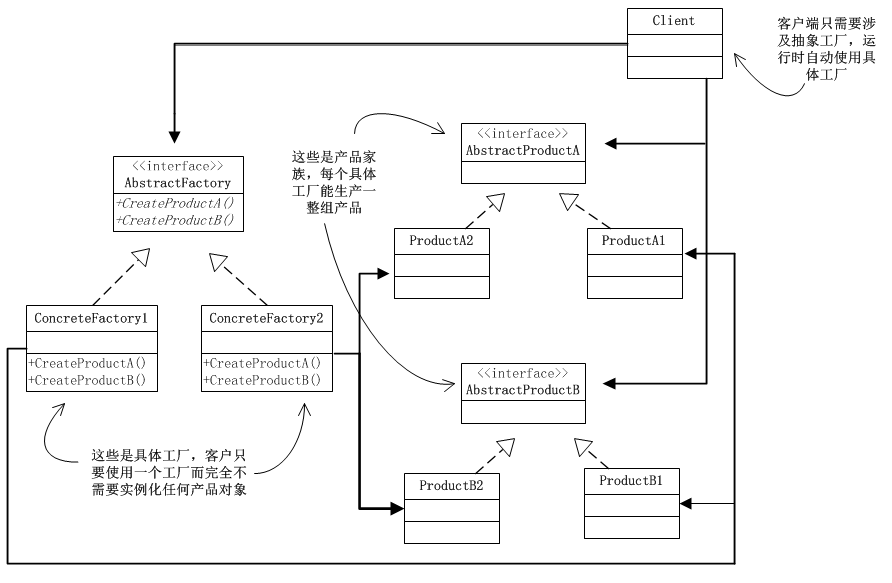
return new Rectangle();

}

}

### 抽象工厂方法

在工厂方法模式中，具体创建者每次使用都只能创建一个同类型的对象，假如需要的是多个不同类型的对象，工厂方法就满足不了需求了。这时我们可以把多个工厂方法组合到一个类，这就是抽象工厂模式，它就是专门用来创建多个产品，也就是创建产品家族的。

类图：

具体的工厂会创建几种特定的产品的组合

## 建造者模式

将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。建造者返回给客户一个完整的的产品对象，而客户端无须关心该对象所包含的额属性和组建方式。

建造者模式主要包含四个角色：

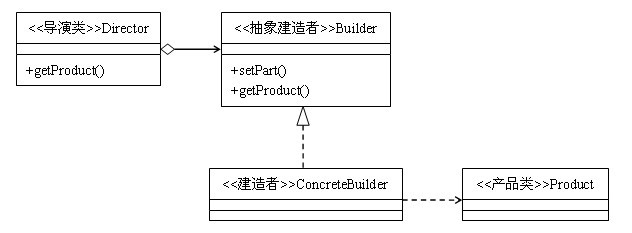
Builder：抽象建造者，它声明为创建一个Product对象的各个部件指定的抽象接口。

ConcreteBuilder：具体建造者，实现抽象接口，构建和装配各个部件。

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：隔离客户与对象的生产过程，负责控制产品对象的生产过程。

Product：产品角色，一个具体的产品对象。

类图：



实例：

class Product {//产品类

//定义部件，部件可以是任意类型，包括值类型和引用类型

private String partA;

private String partB;

private String partC;

//Getter方法和Setter方法省略

}

abstract class Builder {//抽象建造者类

//创建产品对象

protected Product product=new Product();//产品类为建造者的成员变量

public abstract void buildPartA();

public abstract void buildPartB();

public abstract void buildPartC();

//返回产品对象

public Product getResult() {

return product;

}

}

在抽象类Builder中声明了一系列抽象的buildPartX()方法用于创建复杂产品的各个部件，具体建造过程在ConcreteBuilder中实现，此外还提供了工厂方法getResult()，用于返回一个建造好的完整产品。在ConcreteBuilder中实现了buildPartX()方法，通过调用Product的setPartX()方法可以给产品对象的成员属性设值。不同的具体建造者在实现buildPartX()方法时将有所区别。

class Director {//指挥类

private Builder builder; //具体的建造者为指挥类的成员变量

public Director(Builder builder) {

this.builder=builder;

}

public void setBuilder(Builder builder) {

this.builder=builer;

}

//产品构建与组装方法

public Product construct() {

builder.buildPartA();

builder.buildPartB();

builder.buildPartC();

return builder.getResult();

}

}

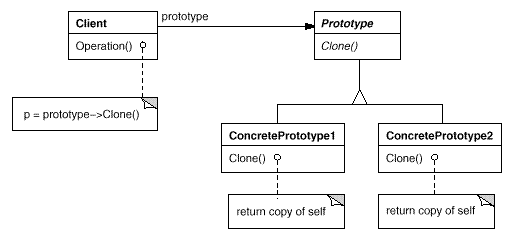
建造者模式与抽象工厂模式有点相似，但是建造者模式返回一个完整的复杂产品，而抽象工厂模式返回一系列相关的产品；在抽象工厂模式中，客户端通过选择具体工厂来生成所需对象，而在建造者模式中，客户端通过指定具体建造者类型并指导Director类如何去生成对象，侧重于一步步构造一个复杂对象，然后将结果返回。如果将抽象工厂模式看成一个汽车配件生产厂，生成不同类型的汽车配件，那么建造者模式就是一个汽车组装厂，通过对配件进行组装返回一辆完整的汽车。

**原型模式**

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。原型模式的主要思想是基于现有的对象克隆一个新的对象出来，一般是有对象的内部提供克隆的方法，通过该方法返回一个对象的副本；使用原型模式复制对象不会调用类的构造方法

创建新对象成本较大（如初始化需要占用较长的时间，占用太多的CPU资源或网络资源），新的对象可以通过原型模式对已有对象进行复制来获得。

原型模式中的拷贝分为"浅拷贝"和"深拷贝":浅拷贝对值类型的成员变量进行值的复制,对引用类型的成员变量只复制引用,不复制引用的对象.深拷贝对值类型的成员变量进行值的复制,对引用类型的成员变量也进行引用对象的复制.(利用串行化重建对象实现深拷贝)



public class Prototype implements Cloneable {

private String name;

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

return null;

}

}

}

public class TestMain {

public static void main(String[] args) {

testPrototype();

}

private static void testPrototype(){

Prototype pro = new Prototype();

pro.setName("original object");

Prototype pro1 = (Prototype)pro.clone();

pro.setName("changed object1");

}

}

## 策略模式

在策略模式中，一个类的行为或其算法可以在运行时更改。实现某一个功能有多种算法或者策略，我们可以根据环境或者条件的不同选择不同的算法或者策略来完成该功能。

public interface Strategy {//策略接口

public int doOperation(int num1, int num2);

}public class OperationAdd implements Strategy{//具体策略类

public int doOperation(int num1, int num2) {

return num1 + num2;

}

}

public class OperationSubstract implements Strategy{

public int doOperation(int num1, int num2) {

return num1 - num2;

}

}public class Context {//使用策略的类

private Strategy strategy;//策略类为成员变量

public Context(Strategy strategy){

this.strategy = strategy;

}

public int executeStrategy(int num1, int num2){

return strategy.doOperation(num1, num2);//使用策略类算法执行操作

}

}public class StrategyPatternDemo {

public static void main(String[] args) {

Context context = new Context(new OperationAdd());

System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

context = new Context(new OperationSubstract());

System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

}

}

## 装饰者模式

装饰器模式（Decorator Pattern）允许向一个现有的对象添加新的功能，同时又不改变其结构。可以动态地给一个对象添加一些额外的职责，就增加功能来说，装饰器模式相比生成子类更为灵活。

装饰者与被装饰者有共同的超类,装饰者抽象类(Decorator)中持有超类接口，方法委托给该接口调用目的是交给该接口的实现即子类(被装饰者)调用.

Decorator抽象类的子类即具体装饰者里面都持有被装饰者对象，被装饰者被装饰的方法被调用时首先调用super(Decoratoted);即首先完成被装饰者的方法本身的行为，再执行装饰者本身添加的行为。 对扩展开放 对修改关闭。

//定义被装饰者接口,具体被装饰者及装饰者抽象类都实现了这个接口

public interface Human {

public void wearClothes();

}

//定义被装饰者，被装饰者初始状态有些自己的装饰

public class Person implements Human {

public void wearClothes() {//被装饰方法wearClothes()

System.out.println("穿什么呢。。");

}

}

//定义装饰者抽象类

public abstract class Decorator implements Human {

private Human human;

public Decorator(Human human) {

this.human = human;

}

public void wearClothes() {

human.wearClothes();

}

}

//下面定义三种装饰，这是第一个，第二个第三个功能依次细化，即装饰者的功能越来越多

public class Decorator\_zero extends Decorator {

public Decorator\_zero(Human human) {

super(human);//将持有的被装饰者成员变量实例化为构造方法传入的超类对象

}

public void goHome() {

System.out.println("进房子。。");

}

@Override

public void wearClothes() {

//super方法为首先调用被装饰者实例(成员变量human)的被装饰方法

super.wearClothes();

goHome();//执行装饰者添加的装饰动作

}

}

public class Decorator\_first extends Decorator {

public Decorator\_first(Human human) {

super(human);

}

public void goClothespress() {

System.out.println("去衣柜找找看。。");

}

public void wearClothes() {

super.wearClothes();

goClothespress();

}

}

public class Decorator\_two extends Decorator {

public Decorator\_two(Human human) {

super(human);

}

public void findClothes() {

System.out.println("找到一件D&G。。");

}

@Override

public void wearClothes() {

super.wearClothes();

findClothes();

}

}

public class Test {

public static void main(String[] args) {

Human person = new Person();

Decorator decorator = new Decorator\_two(new Decorator\_first(

new Decorator\_zero(person)));

decorator.wearClothes();//输出：穿什么呢 进房子 找衣柜 找到一件D&G

}

}

## 适配器模式

适配器模式(Adapter Pattern)是作为两个不兼容接口之间的桥梁

分为类适配器模式(使用继承实现)和对象适配器模式。适配器模式的使用情景:使用系统需要使用现有的类，而此类的接口不符合系统的需要；想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有很复杂的接口；（对对象的适配器模式而言）在设计里，需要改变多个已有子类的接口，如果使用类的适配器模式，就要针对每一个子类做一个适配器类，而这不太实际

public interface Target { //目标接口

//这是源类也有的方法 sampleOpertion1

void sampleOperation1();

//这是源类没有的方法sampleOperation2

void sampleOperation2();

}

public class Adaptee {//被适配类

//源类有方法sampleOperation1

public void sampleOperation1(){}

}

public class Adapter implements Target {//适配器类

private Adaptee adaptee;//持有被适配类对象

public Adapter(Adaptee adaptee){

this.adaptee=adaptee;

}

/\*

\* 源类有方法sampleOperation1因此适配器类直接委派即可

\* @see Adapter.Target#sampleOperation1()

\*/

public void sampleOperation1() {

adaptee.sampleOperation1();

}

/\*

\* 源类没有方法sampleOperation2

\* 因此由适配器类需要补充此方法

\* @see Adapter.Target#sampleOperation2()

\*/

public void sampleOperation2() {

//do something;

}

}

## 代理模式

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能，意图是为其他对象提供一种代理以控制对这个对象的访问。主要解决：在直接访问对象时带来的问题，比如要访问的对象在远程的机器上；在面向对象系统中有些对象由于某些原因（比如对象创建开销很大，或者某些操作需要安全控制，或者需要进程外的访问），直接访问会给使用者或者系统结构带来很多麻烦，我们可以在访问此对象时加上一个对此对象的访问层。即想在访问一个类时做一些控制。

### 静态代理

public interface Star { //被代理接口

void signContract();//签合同

void sing();//唱歌

}

代理角色实现类：代理角色中代理了真实角色所需要的操作（唱歌）

public class ProxyStar implements Star{//代理角色（明星经纪人）

private Star star;//真实对象的引用（明星）

@Override

public void signContract() {

System.out.println("ProxyStar.signContract()");

}

@Override

public void sing() {

star.sing();//真实对象的操作（明星唱歌）

}

public ProxyStar(Star star) {//通过构造器给真实角色赋值

this.star = star;

}

}

真实角色实现类：这里的真实角色中其实只做了一个唱歌的操作，这是真实角色真正的业务逻辑部分

public class RealStar implements Star{//真实角色（明星艺人）

@Override

public void signContract() {

System.out.println("RealStar.signContract()");

}

@Override

public void sing() {

System.out.println("张学友.sing()");//真实角色的操作：真正的业务逻辑

}

}

测试代理类：

public static void main(String[] args) {

Star real = new RealStar();

Star proxy = new ProxyStar(real);

proxy.signContract();

proxy.sing();//真实对象的操作（明星唱歌）

}

### 动态代理

import java.lang.reflect.InvocationHandler;

import java.lang.reflect.Method;

public class StarHandler implements InvocationHandler{//动态代理类

    private Star realStar;//真实角色

    /\*\*

     \* 所有的流程控制都在invoke方法中

     \* proxy：代理类

     \* method：正在调用的方法

     \* args：方法的参数

     \*/

    @Override

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

        Object object = null;

        if (method.getName().equals("sing")) {

//可以在目标方法不受影响情况下在主题方法前后添加新方法或增强主题方法；(执行前置通知)从侧面切入从而达到扩展编程的效果。面向切面的思想

}

        object = method.invoke(realStar, args);//激活调用的方法

        return object;

    }

    //通过构造器来初始化真实角色

    public StarHandler(Star realStar) {

        super();

        this.realStar = realStar;

    }

}

public static void main(String[] args) {

    Star realStar = new RealStar();//真实角色

    //处理器

    StarHandler handler = new StarHandler(realStar);

    //代理类

Star proxy = (Star) Proxy.newProxyInstance(ClassLoader.getSystem

ClassLoader(), realStar.getClass().getInterfaces(), handler);

proxy.sing();//调用代理类的唱歌方法：其实调用的是真实角色的唱歌方法

proxy.signContract();

}

代理类在程序运行时创建的代理方式被成为动态代理。也就是说，这种情况下，代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的指示动态生成的。(Proxy静态方法newProxyInstance,传入ClassLoader,被代理类的接口Class对象,代理类对象)

动态代理的优势是当Star有多个实现类时不必为每个实现类提供一个具体的代理方法，传入接口类的getClass().getInterfaces()即可实现代理。

### Proxy.newInstance

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h) throws IllegalArgumentException

loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载

interfaces:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了.

h:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上.

## 观察者模式

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象；观察者模式属于行为型模式。意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

异步实现的观察者模式

import java.util.Iterator;

import java.util.Vector;

/\*\*

\*

\* @author Seastar

\*/

interface Observed {//被观察者接口

public void addObserver(Observer o);

public void removeObserver(Observer o);

public void update();

}

interface Observer {//观察者接口

public void takeAction();

}

class Invoker {//调用观察者完成操作的代理类

private Observer o;

Handler handler;

public Invoker(Observer o) {

new Handler();

this.o = o;

}

private class Handler extends Thread {//内部开启线程 调用被观察者的类

public Handler() {

handler = this;

}

@Override

public void run() {

o.takeAction();

}

}

public boolean TestSameObserver(Observer o) {

return o == this.o;

}

public void invoke() {

handler.start();

}

}

class ObservedObject implements Observed {

//被观察者维护一个调用观察者的Invoke列表，invoke对象以观察者初始化

private Vector<Invoker> observerList = new Vector<Invoker>();

public void addObserver(Observer o) {

observerList.add(new Invoker(o));

}

public void removeObserver(Observer o) {

Iterator<Invoker> it = observerList.iterator();

while (it.hasNext()) {

Invoker i = it.next();

if (i.TestSameObserver(o)) {

observerList.remove(i);

break;

}

}

}

public void update() {

for (Invoker i : observerList) {

i.invoke();//使用invoke对象通知观察者

}

}

}

class ObserverA implements Observer {

public void takeAction() {

System.out.println("I am Observer A ,state changed");

}

}

public class Main {

public static void main(String[] args) {

ObserverA a = new ObserverA();

ObservedObject oo = new ObservedObject();

oo.addObserver(a);

oo.update();

}

}

## 迭代器模式

提供一种方法顺序访问一个聚合对象中各个元素, 而又无须暴露该对象的内部表示。

public interface Iterator {//迭代器接口

public boolean hasNext();

public Object next();

}

public interface Container {//容器接口

public Iterator getIterator();

}

public class NameRepository implements Container {//具体容器

public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

@Override

public Iterator getIterator() {

return new NameIterator();

}

private class NameIterator implements Iterator {

int index;

@Override

public boolean hasNext() {

if(index < names.length){

return true;

}

return false;

}

@Override

public Object next() {

if(this.hasNext()){

return names[index++];

}

return null;

}

}

}

public class IteratorPatternDemo {

public static void main(String[] args) {

NameRepository namesRepository = new NameRepository();

for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){

String name = (String)iter.next();

System.out.println("Name : " + name);

}

}

}

## 外观模式

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性实现了子系统与客户之间的松耦合关系。

public class Television {

public void on(){

System.out.println("打开了电视");

}

public void off(){

System.out.println("关闭了电视");

}

}

public class Light {

public void on(){

System.out.println("打开了电灯");

}

public void off(){

System.out.println("关闭了电灯");

}

}

public class Screen {

public void up(){

System.out.println("升起银幕");

}

public void down(){

System.out.println("下降银幕");

}

}

public class WatchTvSwtichFacade {//统一的外观

Light light;

Television tv;

Screen screen;

public WatchTvSwtichFacade(Light light, Television tv,Screen screen){

this.light = light;

this.tv = tv;

this.screen = screen;

}

public void on(){

light.on(); //首先开灯

screen.down(); //把银幕降下来

tv.on(); //最后是打开电视

}

public void off(){

tv.off(); //首先关闭电视机

screen.up(); //银幕升上去

light.off(); //最后关灯

}

}

客户端

public class Client {

public static void main(String[] args) {

//实例化组件

Light light = new Light();

Television tv = new Television();

Screen screen = new Screen();

WatchTvSwtichFacade watchTv = new WatchTvSwtichFacade(light,tv,screen);

watchTv.on();

watchTv.off();

}

}

## 模板方法

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。即定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。

模板方法：一个模板方法是定义在抽象类中的、把基本操作方法组合在一起形成一个总算法或一个总行为的方法。

基本方法：基本方法是实现算法各个步骤的方法，是模板方法的组成部分。

public abstract class Game {//模板类

abstract void initialize();

abstract void startPlay();

abstract void endPlay();

public final void play(){//模板方法,用final修饰,保证子类逻辑被父类控制

initialize();

startPlay();

endPlay();

}

}

public class Cricket extends Game {//具体子类

@Override

void endPlay() {//具体方法

System.out.println("Cricket Game Finished!");

}

@Override

void initialize() {

System.out.println("Cricket Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Cricket Game Started. Enjoy the game!");

}

}

public class Football extends Game {

@Override

void endPlay() {

System.out.println("Football Game Finished!");

}

@Override

void initialize() {

System.out.println("Football Game Initialized! Start playing.");

}

@Override

void startPlay() {

System.out.println("Football Game Started. Enjoy the game!");

}

}

public class TemplatePatternDemo {

public static void main(String[] args) {

Game game = new Cricket();

game.play();

game = new Football();

game.play();

}

}

## 设计模式的六大原则

### 1、开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

### 2、里氏代换原则（Liskov Substitution Principle）

任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

### 3、依赖倒置原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。

### 4、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

### 5、迪米特法则（最少知道原则）（Demeter Principle）

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

### 6、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。