# 设计模式

# 2.1 设计原则

## 单一职责原则

* 不要存在多于一个导致类变更的原因。
* 总结：一个类只负责一项职责。

## 里氏替换原则

* 1.子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法。
* 2.子类中可以增加自己特有的方法。
* 3.当子类的方法重载父类的方法时，方法的前置条件（即方法的形参)要比父类方法的输入参数更宽松。
* 4.当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值)要比父类更严格。
* 总结：所有引用父类的地方必须能透明地使用其子类对象

## 依赖倒置原则/面向接口编程

* 高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

## 接口隔离原则

* 使用多个专门的接口来替代一个统一的接口；
* 一个类对另一个类的依赖应建立在最小的接口上

## 迪米特法则

* 一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部

## 开闭原则

* 对扩展开放，对修改关闭
* 用抽象构建框架，用实现扩展细节

## 合成复用原则/组合优于继承

* 尽量多使用组合和聚合，尽量少使用甚至不使用继承关系

# 2.2 分类

* 创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
* 结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。
* 行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

# 2.3 创建型设计模式

## 工厂方法模式

### 介绍

* 工厂模式分为简单（静态)工厂模式、工厂方法模式和抽象工厂模式
  + 1) 工厂类角色：这是本模式的核心，含有一定的商业逻辑和判断逻辑。在java中它往往由一个具体类实现。
  + 2) 抽象产品角色：它一般是具体产品继承的父类或者实现的接口。在java中由接口或者抽象类来实现。
  + 3) 具体产品角色：工厂类所创建的对象就是此角色的实例。在java中由一个具体类实现。
* 简单工厂模式：一个工厂类处于对产品类实例化调用的中心位置上,它决定那一个产品类应当被实例化,
* 工厂方法模式：一个抽象产品类，可以派生出多个具体产品类。
* 一个抽象工厂类，可以派生出多个具体工厂类。
* 每个具体工厂类只能创建一个具体产品类的实例。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)客户不需要知道要使用的对象的创建过程
  + 2)客户使用的对象存在变动的可能，或者根本就不知道使用哪一个具体对象
* 缺点：
* 类的数量膨胀

## 抽象工厂模式

### 介绍

* 抽象工厂模式：多个抽象产品类，每个抽象产品类可以派生出多个具体产品类。
* 一个抽象工厂类，可以派生出多个具体工厂类。
* 每个具体工厂类可以创建多个具体产品类的实例。
* 区别： 工厂方法模式只有一个抽象产品类，而抽象工厂模式有多个。
* 工厂方法模式的具体工厂类只能创建一个具体产品类的实例，而抽象工厂模式可以创建多个。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)系统中有多个产品族，而系统一次只能消费其中一族产品
  + 2)同属于同一个产品族的产品一起使用

## 单例模式

### 介绍

* 通过单例模式可以保证系统中，应用该模式的类一个类只有一个实例。即一个类只有一个对象实例

### UML

### 适用场景与优缺点

* 使用场景：
  + 1)当类只有一个实例且客户可以从一个众所周知的访问点 访问它
  + 2)当这个唯一实例应该是通过子类化可扩展的，且客户应该无序更改代码就能使用一个扩展的实例
* 优点：
  + 1)对唯一实例的受控访问
  + 2)缩小命名空间，避免命名污染
  + 3)允许单例有子类
  + 4)允许可变数目的实例

public class Car{

* //懒汉式，线程不安全

private static Car instance;

private Car() {}

public static Car getInstance() {

* if(instance == null) {
* instance = new Car();
* }
* return instance;
* }
* 这种写法lazy loading很明显，但是致命的是在多线程不能正常工作。
* //懒汉式，线程安全

private static Car instance ;

private Car() {}

public static synchronized Car getInstance(){

* if(instance == null) {
* instance = new Car();
* }
* return instance;
* }
* 这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。
* //饿汉式

private static Car instance = new Car();

private Car() {}

public static Car getInstance() {

* return instance;
* }
* 这种方式基于classloder机制避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法)导致类装载，这时候初始化instance显然没有达到lazy loading的效果。
* //饿汉式变种

private static Car instance;

* static {
* instance = new Car();
* }

private Car() {}

public static Car getInstance() {

* return instance;
* }
* 表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。
* //静态内部类（类的加载是线程安全的)

private static class CarHolder{

private static final Car instance = new Car();

* }

private Car() {}

public static Car getInstance() {

* return CarHolder.instance;
* }
* 这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别)：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果)，而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。
* // 枚举

public enum Car {

* INSTANCE;
* }
* //双重校验锁

private volatile static Car instance;

private Car() {}

public static Car getInstance() {

* if(instance == null) {
* synchronized(Car.class) {
* if(instance == null) {
* instance = new Car();
* }
* }
* }
* return instance;
* }
* 这个是第二种方式的升级版，俗称双重检查锁定。
* 所谓双重检查加锁机制，指的是：并不是每次进入getInstance方法都需要同步， 而是先不同步，进入方法过后，先检查实例是否存在，如果不存在才进入下面的同步块， 这是第一重检查。进入同步块后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。
* 双重检查加锁机制的实现会使用一个关键字volatile，它的意思是：被volatile
* 修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。
* 说明：由于volatile关键字可能会屏蔽掉虚拟机中的一些必要的代码优化，所以运行效率并不是很高。因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用”双重检查加锁“机制来实现线程安全的单例，但并不建议大量采用，可以根据情况来选用。
* }

## 建造者模式

### 介绍

* 将一个复杂对象的创建和它的表示分离，使得同样的创建过程可以创建不同的表示。
* Builder用于构建组件
* Director负责装配
* 客户端通过Director来获得最终产品，Director与Builder打交道，持有一个Builder的引用。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)当创建复杂对象的算法应该独立于该对象的组成部分以及它们的赚个屁方式时
  + 2)当构造过程必须允许被构造的对象有不同的表示时
* 优点：
  + 1)可以改变一个对象的内部表示：Builder对象提供给Director一个构造产品的抽象接口，该接口使得Buildewr可以隐藏这个产品的表示和内部结构，同时隐藏了该产品是如何装配的。
  + 2)将构造代码与表示代码分离
  + 3)可以对构造过程进行更精细化的控制

## 原型模式

### 介绍

### UML

### 适用场景与优缺点

* • 当要实例化的类是在运行时刻指定时，例如，通过动态装载；
* • 为了避免创建一个与产品类层次平行的工厂类层次时；
* • 当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。
* 优点：
* 性能优良。原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。
* 缺点：
* 逃避构造函数的约束。这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的。优点就是减少了约束，缺点也是减少了约束，需要大家在实际应用时考虑。

# 2.4 结构型设计模式

## 适配器模式

### 介绍

* 将一个类的接口转换成客户所期待的另一种接口
* Adapter可以组合+实现（对象适配器方式)，也可以继承+实现（类适配器方式)。但是继承不如组合好，因此尽量使用组合+实现。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)想使用一个已存在的类，而它的接口不符合你的需求
  + 2)想创建一个可以复用的类，该类可以与不相关的类或不可预见的类协同工作

## 装饰器模式

### 介绍

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)在不影响其他对象的情况下，以动态透明的方式给单个对象添加职责
  + 2)处理那些可以撤销的职责
  + 3)当不能通过生成子类的方法进行扩充时
* 优点：
  + 1)比继承更加灵活，可以用添加和分离的方式，用装饰在运行时 增加和删除职责
  + 2)避免在层次结构高的类有太多特征，用装饰器为其逐渐地添加功能

## 代理模式

### 介绍

* 代理可以分为静态代理和动态代理
* 为其他对象提供一种代理以控制对这个对象的访问。  
  为了一个对象提供一个替身或者占位符，以控制对这个对象的访问
  + 远程代理能够控制访问远程对象(RMI)
  + 虚拟代理控制访问创建开销大的资源（先创建一个资源消耗较小的对象表示，真实对象只在需要时才会被创建)
  + 保护代理基于权限控制对资源的访问

### UML

### 适用场景与优缺点

* 使用场景：
* 按职责来划分，通常有以下使用场景： 1、远程代理。 2、虚拟代理。 3、Copy-on-Write 代理。 4、保护（Protect or Access)代理。 5、Cache代理。 6、防火墙（Firewall)代理。 7、同步化（Synchronization)代理。 8、智能引用（Smart Reference)代理。
* 优点：
* 1、职责清晰。 2、高扩展性。 3、智能化。
* 缺点：
* 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
* 2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

## 外观模式

### 介绍

* 为子系统的一组接口提供一个一致的界面，定义了一个高层接口，这一接口使得子系统更加容易使用。
* 遵循了迪米特法则：

通俗的来讲，就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地的将逻辑封装在类的内部，对外除了提供的public方法，不对外泄漏任何信息。

* 外观模式就是一种较好的封装
* 是整体和子组件之间的关系，外部类不应该与一个类的子组件过多的接触，应该尽可能与整体打交道。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)为一个复杂子系统提供一个简单接口
  + 2)客户与抽象类的实现部分之间存在着很大的依赖性，引入Facade将子系统与客户解耦，提高了子系统的独立性和可移植性
* 优点：
  + 1)对客户屏蔽了子系统组件，减少客户处理的对象数目，并使得子系统使用起来更加容易
  + 2)实现了子系统与客户之间的松耦合
  + 3)降低了大型软件中的编译依赖性
  + 4)只是提供了一个访问子系统的统一入口，并不影响客户直接使用子系统

## 桥接模式

### 介绍

* 处理多层继承结构，处理多维度变化的场景，将各个维度设计成独立的继承结构，使各个维度可以独立地扩展，在抽象层建立关联。
* 一个维度的父类持有另一个维度的接口的引用（使用组合代替了继承)
* 希望有一个Bridge类来将类型维度和品牌维度连接起来，这样增加类型和增加品牌不会影响对方。
* 两种变化以上的情况应该考虑桥接模式

### UML

### 适用场景与优缺点

* 适用场景：
* 1．如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。
* 2．设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。
* 3．一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。
* 4．虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

## 组合模式（树形结构)

### 介绍

* 无子节点的是叶子，有子节点的是容器
* 叶子和容器的共同点抽象为Component组件
* 每个容器持有一个Component的List引用，包含它的所有子节点。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)想表示对象的层次结构
  + 2)希望客户忽略组合对象与单一对象的不同，用户将统一使用组合结构中的所有对象
* 优点：
  + 1)定义了包含基本对象和组合对象的类层次结构
  + 2)简化客户代码，客户可以一致地使用组合结构和单个对象
  + 3)更容易添加新类型的组件

## 享元模式

### 介绍

* 将相同部分放在一个类中，工厂持有一个Map，可以创建相同部分，如果已持有那么直接返回。
* 不同部分单独设计一个类，可以作为相同部分类的方法的参数传入
* 将一个对象拆成两部分（成员变量拆成两部分)：相同部分和不同部分。相同部分使用工厂创建，进行共享；不同部分作为参数传入

### UML

### 适用场景与优缺点

* 适用场景：池化 内存池 数据库连接池 线程池
* 优点：
  + 1)极大减少内存中对象的数量
  + 2)相同或相似对象内存中只存一份，节省内存
  + 3)外部状态相对独立，不影响内部状态
* 缺点：
  + 1)模式较复杂，使程序逻辑复杂化
  + 2)读取外部状态使运行时间变长，用时间换取了空间

# 2.5 行为型设计模式

## 策略模式

### 介绍

* 策略模式定义了一系列的算法，并将每一个算法封装起来，而且使他们可以相互替换，让算法独立于使用它的客户而独立变化。
* 环境类(Context):用一个ConcreteStrategy对象来配置。维护一个对Strategy对象的引用。可定义一个接口来让Strategy访问它的数据。
* 抽象策略类(Strategy):定义所有支持的算法的公共接口。 Context使用这个接口来调用某ConcreteStrategy定义的算法。
* 具体策略类(ConcreteStrategy):以Strategy接口实现某具体算法。

### UML

### 适用场景与优缺点

* 适用场景：
* 实现某一个功能由多种算法或者策略，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
* 优点：
  + 1)Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。
  + 2)提供了可以替换继承关系的方法
  + 3)消除if-else

## 模板方法模式

### 介绍

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)一次性实现一个算法的不变部分，并将可变部分留给子类来实现
  + 2)个子类中公共的行为提取出来并集中到一个公共父类中以避免重复
  + 3)控制子类扩展，只允许在某些点进行扩展
* 优点：
  + 1)在一个父类中形式化地定义算法，由它的子类实现细节的处理
  + 2)是一个代码复用的基本技术
  + 3)控制翻转（好莱坞原则)，父类调用子类的操作，通过对子类的扩展来增加新的行为，符合开闭原则

## 观察者模式

### 介绍

* 也称为发布-订阅模式。
* 在此种模式中，一个目标物件管理所有相依于它的观察者物件，并且在它本身的状态改变时主动发出通知。这通常透过呼叫各观察者所提供的方法来实现。此种模式通常被用来实现事件处理系统。
* 与Reactor模式非常类似，不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)当一个对象的改变需要通知其他对象，而且它不知道具体有多少个对象有待通知时
  + 2)当一个抽象模型有两个方面，其中一个方面依赖于另一方面，将这二者封装在独立的对象中国以使它们可以独立地改变和服用
* 优点：
  + 1)独立地改变目标和观察者，解耦
  + 2)是吸纳表示层和数据逻辑层分离（表示层是观察者，逻辑层是主题)

## 迭代器模式

### 介绍

* 找到一种不同容器的统一的遍历方式，定义一个接口，所有可以提供遍历方式的容器都实现这个接口，返回一个迭代器，然后所有的迭代器的接口是一致的。
* 所有的容器都可以通过iterator方法返回一个迭代器Iterator，这个迭代器对外暴露的接口是一致的，因此可以保证对所有的容器遍历方法是一致的，仅需得到这个容器的迭代器即可，而各个容器对迭代器的实现是不同的，即遍历方式是不同的。迭代器模式可以将各个容器的遍历方式的调用方式统一起来，隐藏了内部遍历的实现细节。

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)访问一个聚合对象的内容而无需暴露它的内部表示
  + 2)需要为聚合对象提供多种遍历方式
  + 3)为遍历不同的聚合结构提供一个统一的接口
* 优点：
  + 1)支持以不同的方式遍历一个聚合对象
  + 2)简化聚合接口
  + 3)方便添加新的聚合类和迭代器类

## 责任链模式

### 介绍

* 使多个处理器对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系，将这些处理器对象连成一条链，并沿这条链传递请求，直到有一个处理器对象处理它为止

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)有多个处理器对象可以处理一个请求，哪个处理器对象处理该请求在运行时动态确定
  + 2)在不明确指定接收者的情况下，向多个处理器对象中的一个提交请求
  + 3)可以动态指定一组处理器对象处理请求
* 优点：
  + 1)降低耦合，使得 请求发送者无需知道是哪个处理器对象处理请求
  + 2)简化对象的相互连接
  + 3)增强了给对象指派责任的灵活性
  + 4)方便添加新的请求处理类

## 命令模式

### 介绍

* 命令模式把一个请求或者操作封装到一个对象中，把发出命令的责任和执行命令的责任分割开，委派给不同的对象，可降低行为请求者与行为实现者之间耦合度。从使用角度来看就是请求者把接口实现类作为参数传给使用者，使用者直接调用这个接口的方法，而不用关心具体执行的那个命令。
* Command模式将操作的执行逻辑封装到一个个Command对象中，解耦了操作发起者和操作执行逻辑之间的耦合关系：操作发起者要进行一个操作，不用关心具体的执行逻辑，只需创建一个相应的Command实例，调用它的执行接口即可。而在swing中，与界面交互的各种操作，比如插入，删除等被称之为Edit，实际上就是Command。
* 使用undo包很简单，主要操作步骤如下：
* 1、创建CommandManager实例（持有Command的undo栈和redo栈)；
* 2、创建各种实现Command的具体操作类；
* 3、调用某种操作时，创建一个具体操作类的实例，加入CommandManager；
* 4、在Undo/Redo时，直接调用CommandManager的undo/redo方法。

### UML

* 黑色箭头表示持有，关联关系 Client持有Invoker
* 菱形箭头也是持有，聚合关系 Invoker持有Command
* 白色箭头是继承，ConcreteCommand继承了Command

### 适用场景与优缺点

* 适用场景：
  + 1)系统需要将请求调用者和请求接收者解耦，使得调用者和接收者不直接交互
  + 2)系统需要在不同的时间制定请求，将请求排序和执行请求
  + 3)系统需要支持undo和redo操作
  + 4)系统需要将一组操作组合在一起
* 优点：
  + 1)降低系统的耦合度，调用者和接收者解耦
  + 2)Command是头等对象，可以被操纵和扩展
  + 3)组合命令
  + 4)方便实现undo和redo

## 备忘录模式

### 介绍

* Originate是实体类，并负责创建和恢复Memento（比如JavaBean)
* Memento负责保存对象的状态
* CareTaker 负责存储Memento（一个或一系列)（多条历史记录)
* Originate除了对象的属性和setter getter之外，还有创建和恢复Memento的方法
* Memento也持有对象的所有属性和setter getter，它的构造方法是由Originate对象得到其内部状态。
* CareTaker持有一个或一组Memento，并提供setter and getter

### UML

### 适用场景与优缺点

* 适用场景：
* 1、需要保存/恢复数据的相关状态场景。
* 2、提供一个可回滚的操作。
* 优点：
* 1、给用户提供了一种可以恢复状态的机制，可以使用户能够比较方便地回到某个历史的状态。
* 2、实现了信息的封装，使得用户不需要关心状态的保存细节。
* 缺点：消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。

## 状态模式

### 介绍

### UML

### 适用场景与优缺点

* 适用场景：
  + 1)一个对象的行为取决于它的状态
  + 2)代码中包含大量的与对象状态有关的条件语句
* 优点：
  + 1)将与特定状态相关的行为局部化，并且将不同状态的行为分割开来
  + 2)使得状态转换显式化
  + 3)State对象可被共享

## 访问者模式

### 介绍

### UML

### 适用场景与优缺点

## 中介者模式

### 介绍

* 解耦多个同事对象之间的交互关系。
* 每个同事对象都持有中介者对象的引用，只跟中介者打交道。我们通过中介者统一管理这些交互关系。
* 每个同事类都持有一个中介者类的引用。

### UML

* 将多对多的关系解耦后转为一对多的关系，每个对象和中介者打交道，不直接和其他对象打交道。
* 如果关系比较简单，那么没有必要使用中介者模式，反而会复杂化。

### 适用场景与优缺点

* MVC中的C就是中介者
* 适用场景：
  + 1)系统中对象之间存在着复杂的引用关系
  + 2)一组对象以定义良好但复杂的方式进行通信
  + 3)一个对象引用其他很多对象并直接与这些对象通信，导致难以复用该对象
* 优点：
  + 1)减少子类生成
  + 2)简化同事类的设计和实现
  + 3)简化对象协议（一对多代替多对多)
  + 4)对对象如何协作进行了抽象
  + 5)使控制集中化（将交互复杂性变为中介者的复杂性)

## 解释器模式

### 介绍

### UML

### 适用场景与优缺点

# 2.6 设计模式的区分

## 代理模式和装饰器区别

* 装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。换句话说，用代理模式，代理类（proxy class)可以对它的客户隐藏一个对象的具体信息。
* 因此，当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。
* 相同点：都是为被代理(被装饰)的类扩充新的功能。
* 不同点：代理模式具有控制被代理类的访问等性质，而装饰模式紧紧是单纯的扩充被装饰的类。所以区别仅仅在是否对被代理/被装饰的类进行了控制而已。

## 适配器模式和代理模式的区别

* 适配器模式，一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。
* 装饰器模式，原有的不能满足现有的需求，对原有的进行增强。
* 代理模式，同一个类而去调用另一个类的方法，不对这个方法进行直接操作，控制访问。

## 抽象工厂和工厂方法模式的区别

* 工厂方法：创建某个具体产品
* 抽象工厂：创建某个产品族中的系列产品

工厂方法模式 抽象工厂模式  
针对的是一个产品等级结构 针对的是面向多个产品等级结构  
一个抽象产品类 多个抽象产品类  
可以派生出多个具体产品类 每个抽象产品类可以派生出多个具体产品类  
一个抽象工厂类，可以派生出多个具体工厂类 一个抽象工厂类，可以派生出多个具体工厂类  
每个具体工厂类只能创建一个具体产品类的实例 每个具体工厂类可以创建多个具体产品类的实例

# 2.7 JDK中的设计模式（17)

## 创建型

- 1)工厂方法

* Collection.iterator() 由具体的聚集类来确定使用哪一个Iterator
  + 2)单例模式
* Runtime.getRuntime()
  + 3)建造者模式
* StringBuilder
  + 4)原型模式
* Java中的Cloneable

## 结构性

- 1)适配器模式

* InputStreamReader
* OutputStreamWriter
* RunnableAdapter
  + 2)装饰器模式
* io包 FileInputStream BufferedInputStream
  + 3)代理模式
* 动态代理；RMI
  + 4)外观模式
* java.util.logging
  + 5)桥接模式
* JDBC
  + 6)组合模式
* dom
  + 7)享元模式
* Integer.valueOf

## 行为型

- 1)策略模式

* 线程池的四种拒绝策略
  + 2)模板方法模式
* AbstractList、AbstractMap等
* InputStream、OutputStream
* AQS
  + 3)观察者模式
* Swing中的Listener
  + 4)迭代器模式
* 集合类中的iterator
  + 5)责任链模式
* J2EE中的Filter
  + 6)命令模式
* Runnable、Callable，ThreadPoolExecutor
  + 7)备忘录模式
  + 8)状态模式
  + 9)访问者模式
* 10)中介者模式
  + 11)解释器模式

# 2.8 Spring中的设计模式（6)

- 1)抽象工厂模式：

* BeanFactory
  + 2)代理模式：
* AOP
  + 3)模板方法模式：
* AbstractApplicationContext中定义了一系列的抽象方法，比如refreshBeanFactory、closeBeanFactory、getBeanFactory。
  + 4)单例模式：
* Spring可以管理单例对象，控制对象为单例
  + 5)原型模式：
* Spring可以管理多例对象，控制对象为prototype
  + 6)适配器模式：
* Advice与Interceptor的适配
* Adapter类接口：Target

public interface AdvisorAdapter {

* boolean supportsAdvice(Advice advice);
* MethodInterceptor getInterceptor(Advisor advisor);
* } MethodBeforeAdviceAdapter类，Adapter
* class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

      public boolean supportsAdvice(Advice advice) {

* return (advice instanceof MethodBeforeAdvice);
* }

      public MethodInterceptor getInterceptor(Advisor advisor) {

* MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
* return new MethodBeforeAdviceInterceptor(advice);
* }
* }