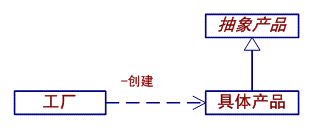
工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。

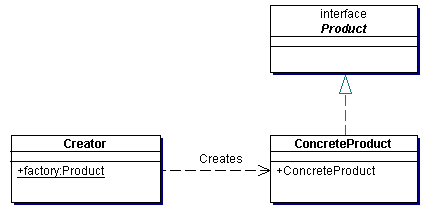
工厂模式主要是为了创建对象提供过渡接口，以便将创建对象的具体过程屏蔽隔离起来，达到提高灵活性的目的

工厂模式模式的几种形态

1. 简单工厂模式(静态工厂方法模式)
2. 工厂方法模式(多态性工厂模式)
3. 抽象工厂模式(工具箱模式)
4. 简单工厂模式

结构

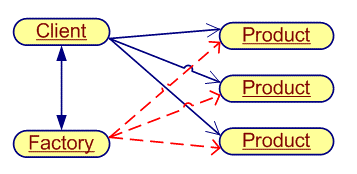




工厂类角色：担任这个角色的是工厂方法模式的核心，含有与应用紧密相关的而商业逻辑;

抽象产品角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或他们共同拥有的接口;

具体产品角色：工厂方法方法模式所创建的任何对象都是这个角色的实例，具体产品角色由一个;



简单模式的优缺点：

优点：模式的核心是工厂类。这个类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例。而客户端则可以免除创建产品对象的责任，而仅仅负责”消费”产品。简单工厂模式通过这种做法实现了对责任的分隔.

缺点：

1. 当产品类有复杂的多层次等级结构时，工厂类只有他自己。以不变应万变，就是模式的缺点;

这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类，若此类出现故障不能正常工作，整个创建产品的都要受到影响

1. 当产品类有不同的接口类时，工厂类需要判断在什么时候创建某种产品;这种对时机的判断和对哪一种具体产品的判断逻辑混在一起，使得系统在将来进行功能扩展时较为困难;
2. 由于简单工厂方法模式使用静态方法作为工厂方法，而静态方法无法由子类继承，因此，工厂角色无法形成基于继承的等级结构

开闭原则(对扩展开放，对修改关闭)

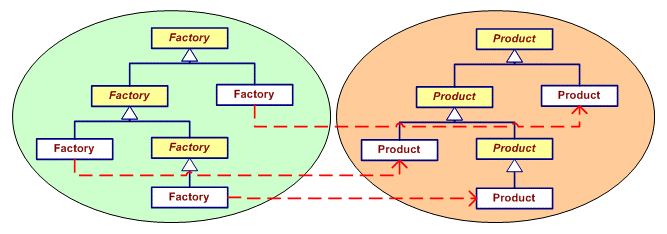
-----“开-闭”原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能;

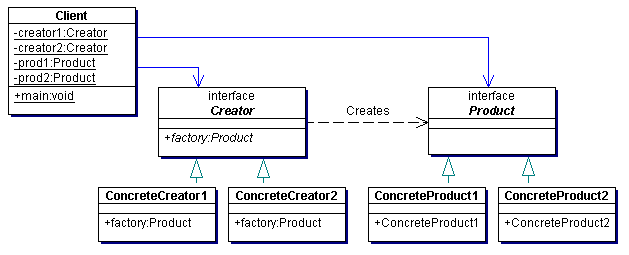
-----简单工厂角色只在有限的程度上支持”开-闭”原则

1. 工厂方法模式

工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

首先，在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做；核心类则变成抽象工厂的角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化;





工厂方法模式涉及到的角色

抽象工厂角色： 担任这个角色的是工厂方法模式的核心，它是与应用程序无关的。任何在模式中创建对象的工厂类必须实现这个接口。在上面的系统中这个角色由java接口Creator扮演;在实际的系统中，这个角色也常常使用抽象Java类实现

具体工厂角色：担任这个角色的是实现抽象工厂接口的具体java类;具体工厂角色含有与应用密切相关的逻辑，并且受到应用程序的调用以创建产品对象;

抽象产品角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口。

具体产品角色：这个角色实现了抽象产品角色所声明的接口，工厂方法模式所创建的每一个对象都是某个具体产品角色的实例

使用场景：

1当客户程序不需要知道要使用对象的创建过程

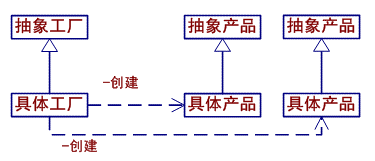
2客户程序使用的对象存在变动的可能，或者根本不知道使用哪个具体的对象

工厂方法模式 VS 简单工厂模式

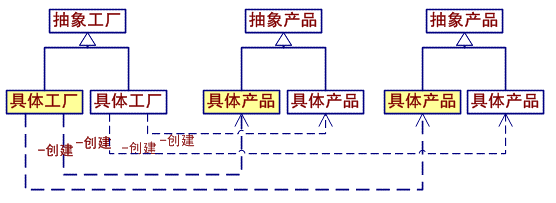
——工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上

——工厂方法模式可以允许很多具体工厂类从抽象工厂类中将创建行为继承下来，从而可以成为多个简单工厂模式的综合，进而推广了简单工厂模式;

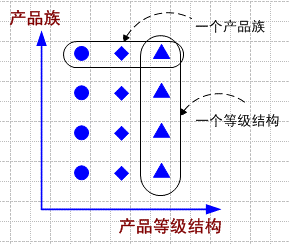
——对于增加新的产品类而言，这个系统完全支持“开-闭”原则;

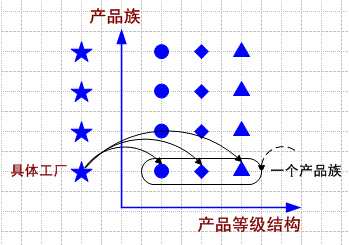


抽象工厂模式的引入



抽象工厂模式与工厂方法模式的最大区别就在于，工厂方法模式针对的是一个产品等级结构；而抽象工厂模式则需要面对多个产品等级结构

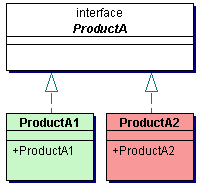


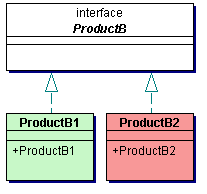


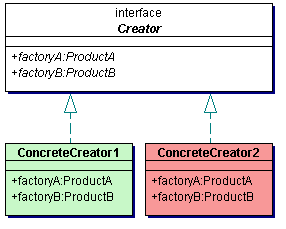
抽象工厂模式的结构

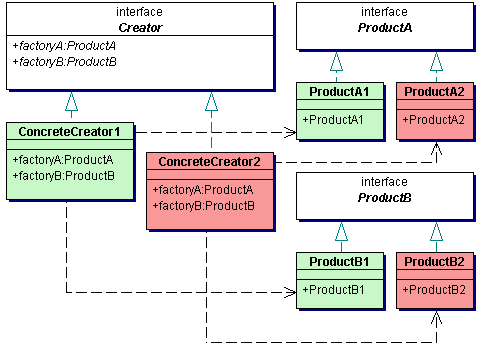
——抽象工厂模式是对象的创建模式，它是工厂方法模式的进一步推广

——假如一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级结构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引进抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只是要向一个公用的工厂接口请求所需要的产品;









抽象工厂模式涉及到的角色

——抽象工厂角色：担任这个角色的是工厂模式的核心，它是与应用系统的商业逻辑无关的。

——具体工厂角色：这个角色直接在客户端的调用下创建产品的实例。这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的

——抽象产品角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或它们共同拥有的接口;

——具体产品角色：抽象工厂模式所创建的任何对象都是某一个具体产品类的实例。其内部一定充满了应用系统的商业逻辑;

\*\*\*\*\*\*

一般而言，有多少个产品等级结构，就会在工厂角色中发现多少个工厂方法；

每个产品等级结构中有多少具体产品，就有多少个产品族，也就会在工厂等级结构中发现多少具体工厂；

抽象工厂模式的使用场景：

1. 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节。这对于所有形态的工厂模式都是重要的
2. 这个系统的产品有多于一个的产品族，而系统只消费其中某一族的产品;
3. 同属于同一个产品族的产品是在一起使用的，这一约束必须要在系统的设计中体现出来
4. 系统提供一个产品类的库，所有的产品以同样的接口出现，从而客户端不依赖于实现

原型模式

模式动机：

——在面向对象系统中，使用原型模式来复制一个对象自身，从而克隆出多个与原型对象一模一样的对象

——在如那件系统中，有些对象的创建过程较为复杂，而且有时候需要频繁的创建，原型模式通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象;

模式定义：

原型模式是一种对象创建型模式，用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。原型模式允许一个对象再创建另外一个可定制的对象，无需知道任何创建的细节

基本工作原理

通过一个原型对象传给那个要发动创建的对象，这个要发动的对象通过请求原型对象拷贝原型自己来实现创建过程

模式结构

抽象原型类：这是一个抽象角色，此角色给出所有的具体原型类所需的接口

具体原型类：被复制的对象。此角色需要实现抽象的原型角色所要求的接口

客户类：提出创建对象的请求

模式分析

在远行模式结构定义了一个抽象原型类，所有的Java类都继承自Object，Object类提供一个clone()方法, 可以将一个Java对象复制一份。因此在Java中可以直接使用Object提供的clone()方法来实现对象的克隆

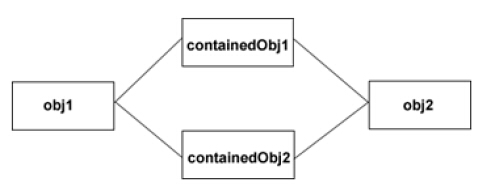
能够实现克隆的Java类必须实现一个表示接口Cloneable;如果一个类没有实现这个接口但是调用了clone()会抛异常CloneNotSupportedException;

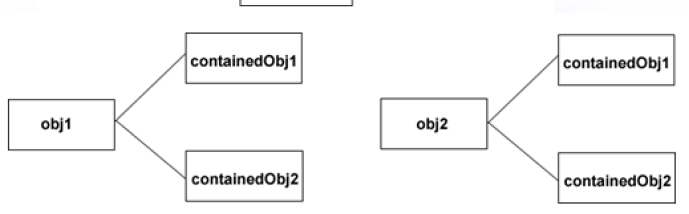
情景分析

通常情况下，一个类包含一些成员对象，在使用原型模式克隆对象时，根据其成员对象是否也克隆，原型模式可以分为两种形式： 深克隆和浅克隆

浅克隆: 仅仅复制所考虑的对象，不复制它所引用的成员对象

深克隆: 所有的都复制





模式分析

Java语言提供的clone()方法将对象复制了一份并返回给调用者。一般而言clone()方法满足：

1. 对任何对象的x，都有x.clone() != x，即克隆对象与源对象不是同一个对象
2. 对任何的对象x，都有x.clone().getclass() ==x.get(),即克隆对象与源对象类型一样
3. 如果对象x的equals()方法定义恰当，那么x.clone().equals(x)成立

原型模式的优点

1. 当创建新的对象实例为较复杂时，使用原型模式可以简化对象的创建过程，通过一个已有实例可以提高新实例的创建效率
2. 可以动态增加或减少产品类
3. 原型模式提供了简化的创建结构
4. 可以使用深克隆的方式保存对象的状态

缺点：

1. 需要为每一个类配备一个克隆方法，而且这个克隆方法需要对类额功能惊醒通盘考虑，这对全新的类来说不是很难，但对已有的类进行改造时，不一定是件容易的事，必须修改其代码，违背了”开闭原则”
2. 在实现深克隆时需要编写较为复杂的代码;

适用环境

1. 创建新对象成本较大，新的对象可以通过原型模式对已有对象进行复制来获得，如果是相似对象，则可以对其属性稍作修改
2. 如果系统要保存对象的状态，而对象的状态变化很小，或者对象本身占内存不大的时候，也可以使用原型模式配合备忘录模式来应用。
3. 需要避免使用分层次的工厂类来创建分层次的对象，并且类的实例对象只有一个或很少的几个组合状态，通过复制原型对象得到新实例可能比使用构造函数创建一个新实例更加的方便;
4. 原型模式应用于很多软件中，如果每次创建一个对象要花大量时间，原型是最好的解决方案。很多软件提供的复制和粘贴操作就是原型模式的应用，复制得到的对象与原型对象是两个类型相同但内存地址不同的对象，通过原型模式可以大大的提高对象的创建效率;