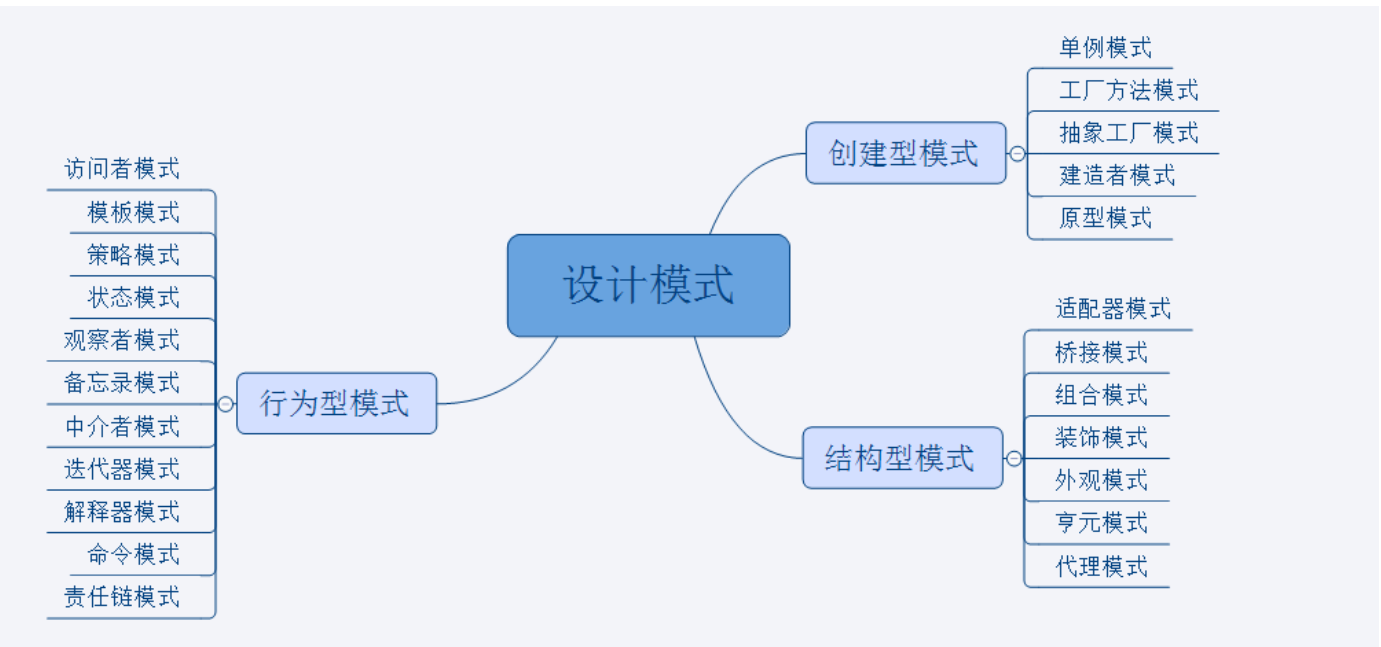
23种设计模式

参考：<https://www.cnblogs.com/pony1223/p/7608955.html>

## **各分类中模式的关键点**



单例模式：某个类只能有一个实例，提供一个全局的访问点。

简单工厂：一个工厂类根据传入的参量决定创建出那一种产品类的实例。

工厂方法：定义一个创建对象的接口，让子类决定实例化那个类。

抽象工厂：创建相关或依赖对象的家族，而无需明确指定具体类。

建造者模式：封装一个复杂对象的构建过程，并可以按步骤构造。

原型模式：通过复制现有的实例来创建新的实例。

适配器模式：将一个类的方法接口转换成客户希望的另外一个接口。

组合模式：将对象组合成树形结构以表示“”部分-整体“”的层次结构。

装饰模式：动态的给对象添加新的功能。

代理模式：为其他对象提供一个代理以便控制这个对象的访问。

亨元（蝇量）模式：通过共享技术来有效的支持大量细粒度的对象。

外观模式：对外提供一个统一的方法，来访问子系统中的一群接口。

桥接模式：将抽象部分和它的实现部分分离，使它们都可以独立的变化。

模板模式：定义一个算法结构，而将一些步骤延迟到子类实现。

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器。

策略模式：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

状态模式：允许一个对象在其对象内部状态改变时改变它的行为。

观察者模式：对象间的一对多的依赖关系。

备忘录模式：在不破坏封装的前提下，保持对象的内部状态。

中介者模式：用一个中介对象来封装一系列的对象交互。

命令模式：将命令请求封装为一个对象，使得可以用不同的请求来进行参数化。

访问者模式：在不改变数据结构的前提下，增加作用于一组对象元素的新功能。

责任链模式：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

迭代器模式：一种遍历访问聚合对象中各个元素的方法，不暴露该对象的内部结构。

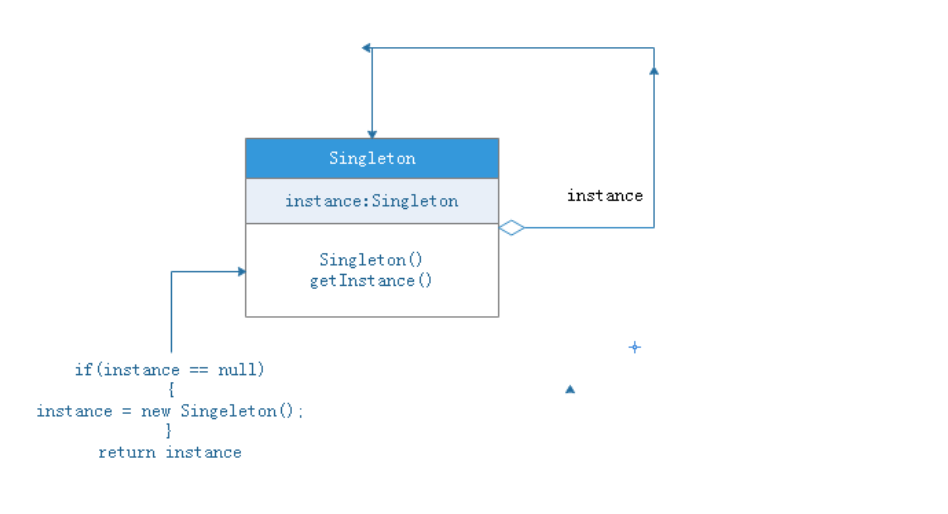
## 分类

### **创建型模式：**

**对象实例化的模式，创建型模式用于解耦对象的实例化过程。**

#### 单例模式

##### 结构图



确保某个类只能有一个实例

应用：

当连接数据库的时候，代码相同为了每次new一个实例带来的系统和内存资源的消耗

申明一个static getInstance方法，在该方法里创建该对象的实例。如果该实例已经存在，则不创建。

注：共用一个实例后通过其中一个修改实例变量其他的也改变

class Single {

private $name;//声明一个私有的实例变量

private function \_\_construct(){//声明私有构造方法为了防止外部代码使用new来创建对象。

}

static public $instance;//声明一个静态变量（保存在类中唯一的一个实例）

static public function getinstance(){//声明一个getinstance()静态方法，用于检测是否有实例对象

if(!self::$instance) self::$instance = new self();

return self::$instance;

}

public function setname($n){ $this->name = $n; }

public function getname(){ return $this->name; }

}

$oa = Single::getinstance();

$ob = Single::getinstance();

$oa->setname('hello world');

$ob->setname('good morning');

echo $oa->getname();//good morning

echo $ob->getname();//good morning

#### 工厂模式

##### 定义：

工厂模式属于创建型模式，它提供了一种创建对象的最佳方法。

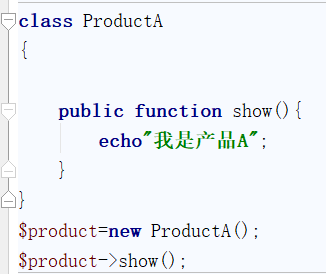
通常把用类创建对象的过程称为实例化

最后这边再说一下什么是实例化对象。小编突然像想到这样一个问题我的第一反应是不就是new嘛。可是真让我说我竟以实语塞。后来仔细想想。这几个字要分开来想 $student =new student();前面的$ student是我们的类对象而用new 关键字创建类对象的过程就是我们所说的实例化。

##### 为什么要使用工厂模式

我们制造一双鞋不用知道这双写是怎们知道的要用哪写材料，只需要交给工厂类，说我们要那种写尺寸多少颜是什么，然后工厂类就会帮你造好鞋

什么是工厂模式呢，很多地方有这样说道：工厂模式就是为了把对象的创建与使用的过程分开或者是将“类实例化的操作”与“使用对象的操作”分开。那我们怎么来理解这一句话呢。举个简单的例子。我们像调用ProductA类里面的show方法，我们一般的的做法是下面这样



这就是我们平时用new来实例化对象进而使用类中的方法。这样子的确没有什么问题。

但是如果一个类的构造函数很复杂，需要用到其他的类

**class** size  
{  
 **protected $size**;  
 **public function** \_\_construct($size)  
 {  
 $this->**size**=$size;  
 }  
 **public function** size(){  
 **echo** $this->**size**;  
 }  
}  
**class** name  
{  
 **protected $name**;  
 **public function** \_\_construct($name)  
 {  
 $this->**name**=$name;  
 }  
 **public function** name(){  
 **echo** $this->**name**;  
 }  
}  
**class** ProductA  
{  
 **protected $name**,**$size**;  
 **public function** \_\_construct(name $name,size $size)  
 {  
 $this->**name**=$name;  
 $this->**size**=$size;  
 }  
 **public function** show(){  
 **echo** $this->**name**->name();  
 **echo** $this->**size**->size();  
 }  
}  
$name=**new** name(**'产品A'**);  
$size=**new** size(**'50X50'**);  
$product=**new** ProductA($name,$size);  
$product->show();

这里面我们可以看到ProductA类的构造需要用到name和size类，也许这样这看起来有人说不是很复杂可以直接new。但是如ProductA类的构造函数需要很多个其他类的参数，$product=**new** ProductA($name,$size,$color……)

那我每new一个ProductA类就要new很多构造函数用到的其他类，一但代码中用的需要实例化很多ProductA类，那么此时就会有很多重复的代码，假设我们上面一样写了很多的代码，有一天发现ProductA类的构造函数的参数需要修改增或删除一个参数，或者修改ProductA类的名字为ProductB。那么我们就要一个个去修改我们new ProductA类的地方。这显然不现实，很费时间。那么造成这样的原因就是new 把对象的创建和使用的过程放在了一起，每次使用一个类时我们都要去创建一个类对象，耦合度高。此外我们还可以看到每一次new我们都要知道ProductA的构造函数有哪些参数，即对象的创建逻辑与过程。如果

对象的创建逻辑与过程很复杂很繁琐，那么我们每一次都要去理一遍显然不合理，也很麻烦。

一次我们使用工厂模式。

通过上面的的例子我们可以总结出来使用

##### 工厂模式的好处就是：

1. 解耦，将“类实例化的操作”与“使用对象的操作”分开。
2. 降低代码重复
3. 然使用者不需要知道类对象创建的逻辑，直接使用其中方法就好了。

##### 简单工厂

简单工厂又叫做静态方法模式。工厂类中定义了静态方法用来创建产品类实例。使用的时候直接调用静态方法。

* 创建**抽象产品类** & 定义具体产品的公共接口；
* 创建**具体产品类**（继承抽象产品类） & 定义生产的具体产品；
* 创建**工厂类**，通过创建静态方法根据传入不同参数从而创建不同具体产品类的实例；
* 外界通过调用工厂类的静态方法，**传入不同参数**从而创建不同**具体产品类的实例**
* **<?php  
    
  class** size  
  {  
   **protected $size**;  
   **public function** \_\_construct($size)  
   {  
   $this->**size**=$size;  
   }  
   **public function** size(){  
   **echo** $this->**size**;  
   }  
  }  
  **class** name  
  {  
   **protected $name**;  
   **public function** \_\_construct($name)  
   {  
   $this->**name**=$name;  
   }  
   **public function** name(){  
   **echo** $this->**name**;  
   }  
  }  
    
  *//具体产品的公共接口***interface** Shoes{  
   **public function** show();  
  }  
  *//具体产品类***class** Shoes**A implements** Shoes  
  {  
   **protected $name**,**$size**;  
   **public function** \_\_construct(name $name,size $size)  
   {  
   $this->**name**=$name;  
   $this->**size**=$size;  
   }  
   **public function** show()  
   {  
   *//* ***TODO: Implement show() method.* echo** $this->**name**->name();  
   **echo** $this->**size**->size();  
   }  
  }  
  **class** ShoesB **implements** Shoes  
  {  
   **protected $name**,**$size**;  
   **public function** \_\_construct(name $name,size $size)  
   {  
   $this->**name**=$name;  
   $this->**size**=$size;  
   }  
   **public function** show(){  
   **echo** $this->**name**->name();  
   **echo** $this->**size**->size();  
   }  
  }  
  *//工厂类*

**class** Factory {  
 **static function** createshoes($name,$size,$class){  
  
 **if**($class==**"A"**){  
 $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesA($name1,$size1);  
 }  
 **else if**($class==**'B'**){  
 $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesB($name1,$size1);  
 }  
 }  
}  
 Factory:: createshoes(**'鞋子A'**,**'50x50'**,**"A"**)->show();  
  
简单工厂的缺点：

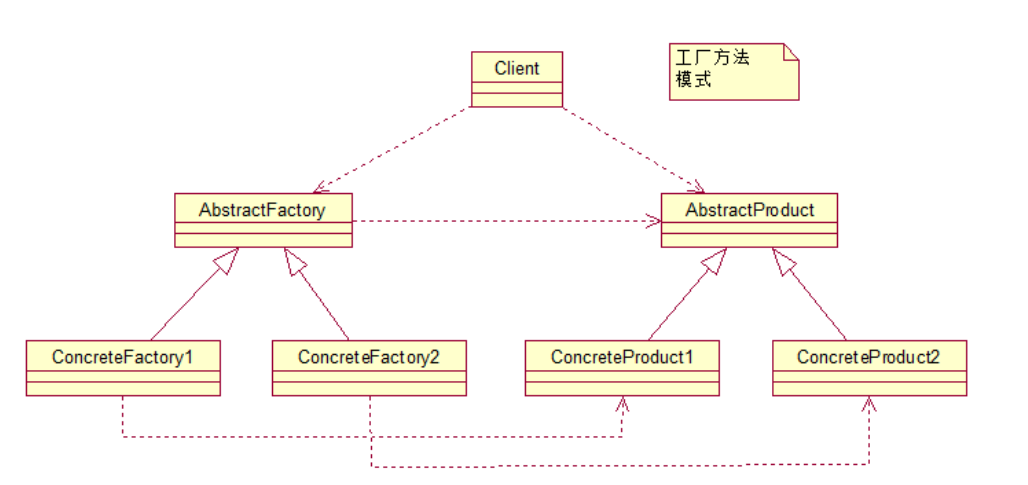
将所有的产品即实例放到一个工厂中，一但这个工厂出错那么所有的产品都无法生产

因此为了克服这个确定我们产生了工厂方法模式。

##### 工厂方法模式

工厂方法模式是将类的实例化延迟到工厂类的子类中完成，即由子类来决定应该实例化哪一个类，哪个产品由哪个工厂生产。

结构图



抽象工厂类

具体某个工厂类---只能生产某个特定的产品

抽象产品类

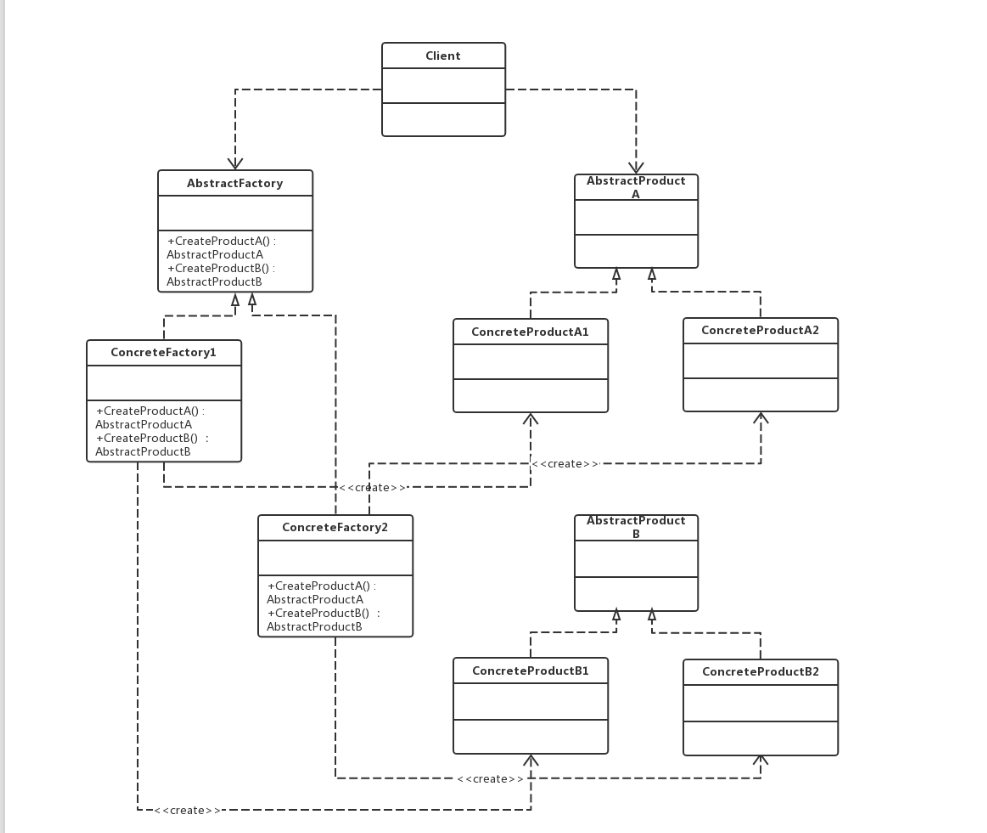
具体某个产品类

**<?php  
  
class** size  
{  
 **protected $size**;  
 **public function** \_\_construct($size)  
 {  
 $this->**size**=$size;  
 }  
 **public function** size(){  
 **echo** $this->**size**;  
 }  
}  
**class** name  
{  
 **protected $name**;  
 **public function** \_\_construct($name)  
 {  
 $this->**name**=$name;  
 }  
 **public function** name(){  
 **echo** $this->**name**;  
 }  
}  
  
*//具体产品的公共接口***interface** Shoes{  
 **public function** show();  
}  
*//具体产品类***class** ShoesA **implements** Shoes  
{  
 **protected $name**,**$size**;  
 **public function** \_\_construct(name $name,size $size)  
 {  
 $this->**name**=$name;  
 $this->**size**=$size;  
 }  
 **public function** show()  
 {  
 *//* ***TODO: Implement show() method.* echo** $this->**name**->name();  
 **echo** $this->**size**->size();  
 }  
}  
**class** ShoesB **implements** Shoes  
{  
 **protected $name**,**$size**;  
 **public function** \_\_construct(name $name,size $size)  
 {  
 $this->**name**=$name;  
 $this->**size**=$size;  
 }  
 **public function** show(){  
 **echo** $this->**name**->name();  
 **echo** $this->**size**->size();  
 }  
}  
*//工厂类***interface** Factory {  
 **public function** createshoes($name,$size);  
}  
**class** FactoryA **implements** Factory {  
  
 **public function** createshoes($name, $size)  
 {  
 *//* ***TODO: Implement create() method.*** $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesA($name1,$size1);  
  
 }  
}  
**class** FactoryB **implements** Factory {  
  
 **public function** createshoes($name, $size)  
 {  
 *//* ***TODO: Implement create() method.*** $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesB($name1,$size1);  
  
 }  
}  
 $factorya=**new** FactoryA();  
$factorya->createshoes(**'鞋子A'**,**'50x50'**)->show();  
$factoryb=**new** FactoryA();  
$factoryb->createshoes(**'鞋子B'**,**'100x100'**)->show();  
  
缺点：每个工厂只能创建一类产品

##### 抽象工厂模式

从工厂方法模式中我们可以看到FactoryA，FactoryB只能生产不同品牌的鞋子要是我们想让它们生产汽车它们就没有办法了。因此我们使用工厂模式用于每个工厂可以创建多种类的产品

结构图



抽象工厂类

具体某个工厂类---可以生产不同种类的产品

抽象产品类

具体某个产品类

**<?php  
  
class** size  
{  
 **protected $size**;  
 **public function** \_\_construct($size)  
 {  
 $this->**size**=$size;  
 }  
 **public function** size(){  
 **echo** $this->**size**;  
 }  
}  
**class** name  
{  
 **protected $name**;  
 **public function** \_\_construct($name)  
 {  
 $this->**name**=$name;  
 }  
 **public function** name(){  
 **echo** $this->**name**;  
 }  
}  
  
*//具体产品的公共接口***interface** Shoes{  
 **public function** show();  
}  
*//具体产品类***class** ShoesA **implements** Shoes  
{  
 **protected $name**,**$size**;  
 **public function** \_\_construct(name $name,size $size)  
 {  
 $this->**name**=$name;  
 $this->**size**=$size;  
 }  
 **public function** show()  
 {  
 *//* ***TODO: Implement show() method.* echo** $this->**name**->name();  
 **echo** $this->**size**->size();  
 }  
}  
**class** ShoesB **implements** Shoes  
{  
 **protected $name**,**$size**;  
 **public function** \_\_construct(name $name,size $size)  
 {  
 $this->**name**=$name;  
 $this->**size**=$size;  
 }  
 **public function** show(){  
 **echo** $this->**name**->name();  
 **echo** $this->**size**->size();  
 }  
}  
**interface** Car{  
 **public function** show();  
}  
**class** CarA **implements** Car  
{  
  
 **public function** show()  
 {  
 **echo"生产汽车A"**;  
 }  
}  
**class** CarB **implements** Car  
{  
 **public function** show(){  
 **echo"生产汽车B"**;  
 }  
}  
*//工厂类***interface** Factory {  
 **public function** createshoes($name,$size);  
 **public function** createcars();  
}  
**class** FactoryA **implements** Factory {  
  
 **public function** createshoes($name, $size)  
 {  
 *//* ***TODO: Implement create() method.*** $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesA($name1,$size1);  
  
 }  
  
 **public function** createcars()  
 {  
 *//* ***TODO: Implement createcars() method.* return new** CarA();  
 }  
}  
**class** FactoryB **implements** Factory {  
  
 **public function** createshoes($name, $size)  
 {  
 *//* ***TODO: Implement create() method.*** $name1=**new** name($name);  
 $size1=**new** size($size);  
 **return new** ShoesB($name1,$size1);  
  
 }  
  
 **public function** createcars()  
 {  
 *//* ***TODO: Implement createcars() method.* return new** CarB();  
 }  
}  
 $factorya=**new** FactoryA();  
$factorya->createshoes(**'鞋子A'**,**'50x50'**)->show();  
$factorya->createcars()->show();  
$factoryb=**new** FactoryA();  
$factoryb->createshoes(**'鞋子B'**,**'100x100'**)->show();  
$factoryb->createcars()->show();

#### 建造者模式

参考：<https://www.jianshu.com/p/3d1c9ffb0a28>

##### 使用场景

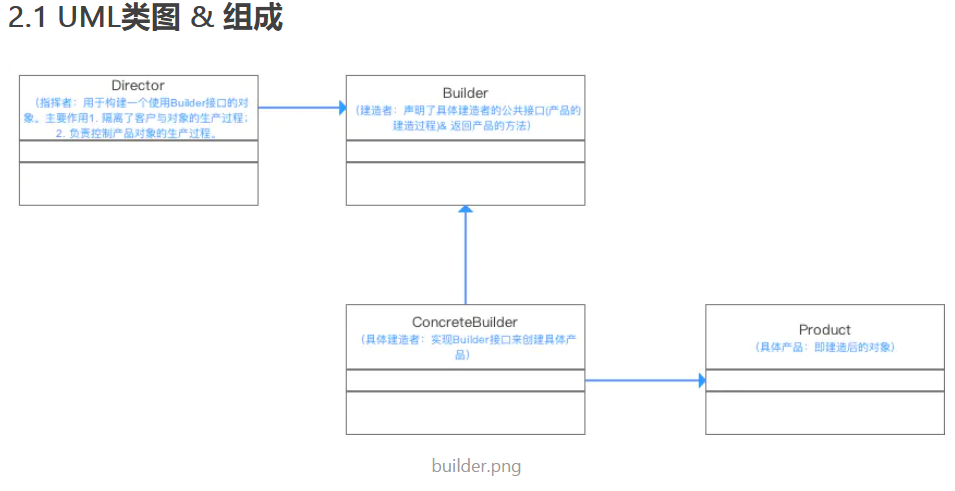
**在用户不知道对象的建造过程和细节的情况下就可以直接创建复杂的对象**。

例子：造汽车 & 买汽车。

工厂（建造者模式）：负责制造汽车（组装过>程和细节在工厂内）

汽车购买者（用户）：你只需要说出你需要的>型号（对象的类型和内容），然后直接购买就可>>以使用了  
（不需要知道汽车是怎么组装的（车轮、车门、>发动机、方向盘等等））

##### 结构图



建造者（Builder）：对复杂对象的创建过程加以抽象，给出一个抽象接口，以规范产品对象的各个组成部分的建造。

具体创建者（ConcreateBuilder）：实现Builder接口，针对不同的业务逻辑，具体化复杂对象的各个部分的创建。在建造过程完成后，提供产品的实例

指导者（Dierctor）：调用具体建造者来创建复杂对象的各个部分，在指导者中不设计具体产品的信息，只负责保证对象各部分完整创建或者按某种顺序创建。

产品（Product）：要创建的复杂对象，一般来说包含多个部分。

##### 代码

.用 builder 模式创建共享单车为例子

###### 产品类：

public class Bike {

private IFrame frame;

private ISeat seat;

private ITire tire;

public IFrame getFrame() {

return frame;

}

public void setFrame(IFrame frame) {

this.frame = frame;

}

public ISeat getSeat() {

return seat;

}

public void setSeat(ISeat seat) {

this.seat = seat;

}

public ITire getTire() {

return tire;

}

public void setTire(ITire tire) {

this.tire = tire;

}

}

###### Builder 类：

// 抽象 builder 类

public abstract class Builder {

abstract void buildFrame();

abstract void buildSeat();

abstract void buildTire();

abstract Bike createBike();

}

###### ConcreteBuilder 类 :

// 具体 builder 类

public class MobikeBuilder extends Builder{

private Bike mBike = new Bike();

@Override

void buildFrame() {

mBike.setFrame(new AlloyFrame());

}

@Override

void buildSeat() {

mBike.setSeat(new DermisSeat());

}

@Override

void buildTire() {

mBike.setTire(new SolidTire());

}

@Override

Bike createBike() {

return mBike;

}

}

public class OfoBuilder extends Builder{

private Bike mBike = new Bike();

@Override

void buildFrame() {

mBike.setFrame(new CarbonFrame());

}

@Override

void buildSeat() {

mBike.setSeat(new RubberSeat());

}

@Override

void buildTire() {

mBike.setTire(new InflateTire());

}

@Override

Bike createBike() {

return mBike;

}

}

###### 指挥者类：

指挥如何生产产品

public class Director {

private Builder mBuilder = null;

public Director(Builder builder) {

mBuilder = builder;

}

public Bike construct() {

mBuilder.buildFrame();

mBuilder.buildSeat();

mBuilder.buildTire();

return mBuilder.createBike();

}

}

###### 客户端使用：

告诉指挥者要生产什么样的产品

public class Click {

public static void main(String[] args) {

showBike(new OfoBuilder());

showBike(new MobikeBuilder());

}

private void showBike(Builder builder) {

Director director = new Director(builder);

Bike bike = director.construct();

bike.getFrame().frame();

bike.getSeat().seat();

bike.getTire().tire();

}

}

#### 原型模式

通过复制现有的实例来创建新的实例。

##### 结构图

未完待续

### **结构型模式：**

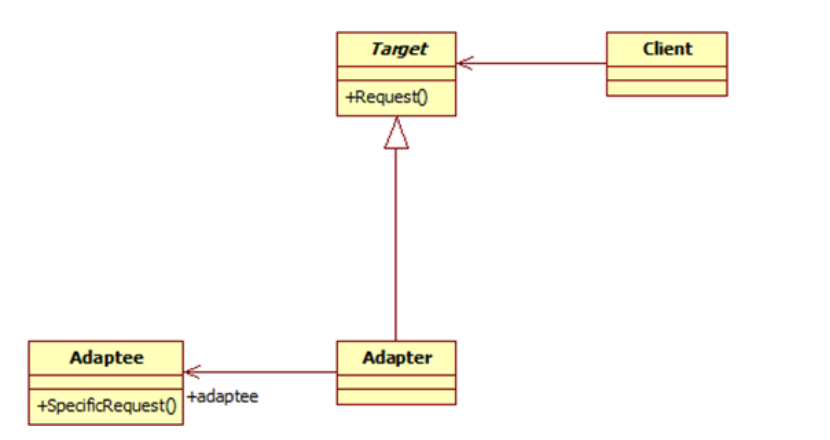
**把类或对象结合在一起形成一个更大的结构。**

#### 适配器模式：

当引用的第三方类库或APi改变时不可能去改大量的代码一次产生了适配器模式

其实换成现实中的另一种说法：当你将程序完整的写结束时，你的客户突然说：听说pdo比mysql好，要不用pdo吧？客户笑眯眯的看着你，你傻乎乎的望着客户，哎，改吧！  
总不见得要所有代码都修改一遍吧，这时候适配器模式就派上了用场！

##### 结构图



Target：目标抽象类  
Adapter：适配器类  
Adaptee：适配者类  
Client：客户类

**<?php  
interface** toy{  
 **public function** openMouse();  
 **public function** closeMouse();  
}  
**class** dog **implements** toy{  
 **public function** openMouse()  
 {  
 **echo"小狗张嘴"**;  
 }  
  
 **public function** closeMouse()  
 {  
 **echo"小狗闭嘴"**;  
 }  
}  
  
  
*//目标接口***interface** target {  
 **public function** doopenMouse();  
 **public function** docloseMouse();  
}  
*//适配器类***class** adapter **implements** target{  
 **private $swe**;  
 **function** \_\_construct(toy $data){  
 $this->**swe**=$data;  
 }  
 *//委派$swe调用TOY的openMouse closeMouse方法* **public function** doopenMouse()  
 {  
 $this->**swe**->openMouse();  
 }  
  
 **public function** docloseMouse()  
 {  
 $this->**swe**->closeMouse();  
 }  
}  
$dog=**new** dog();  
$myadapter=**new** adapter($dog);  
$myadapter->doopenMouse();  
**?>**

### **行为型模式：**

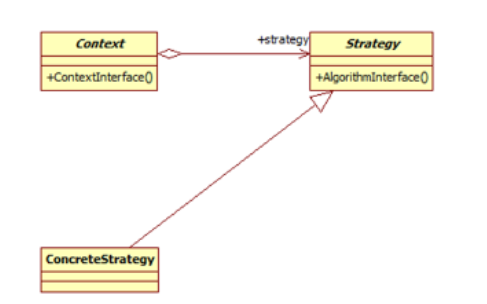
**类和对象如何交互，及划分责任和算法。**

#### **策略模式**

**简单的说策略模式就是**对一组算法，将每个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换

适用场景：根据不同等级的客户或根据不同的节日 打折不同8折,9折

##### 结构图



Context: 环境类  
Strategy: 抽象策略类  
ConcreteStrategy: 具体策略类

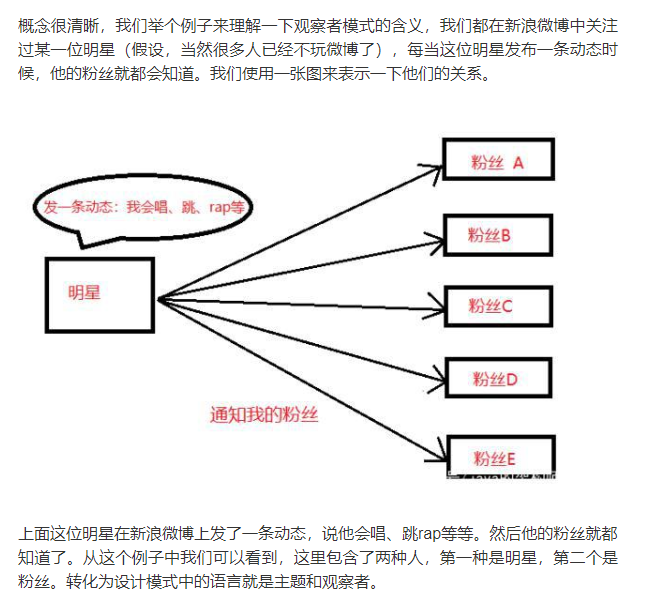
**<?php***/\*\*  
 \* Created by PhpStorm.  
 \* User: SWESWE  
 \* Date: 2019/10/12  
 \* Time: 13:42  
 \*/  
//抽象策略角色---通常由一个接口或抽象类实现---作用：给具体的决策类提供接口***interface** strategy  
{  
 **public function** countPrice($price);  
}  
*//具体决策角色--用于实现具体的算法***class** StrategyA **implements** strategy{  
  
 **public function** countPrice($price)  
 {  
 *//* ***TODO: Implement countPrice() method.* echo"A方案价格+100"**;  
 **return** $price+100;  
 }  
}  
  
**class** StrategyB **implements** strategy{  
  
 **public function** countPrice($price)  
 {  
 *//* ***TODO: Implement countPrice() method.* echo"B方案价格+120"**;  
 **return** $price+120;  
 }  
}  
**class** StrategyC **implements** strategy{  
  
 **public function** countPrice($price)  
 {  
 *//* ***TODO: Implement countPrice() method.* echo"C方案价格+140"**;  
 **return** $price+140;  
 }  
}  
*//环境角色--简单的说这一角色就是告诉你那个算法要干什么  
//持有抽象策略角色的引用***class** setPrice{  
 **private $strategy**;*//用于接收传入的策略对象* **public function** \_\_construct($instance)  
 {  
 $this->**strategy**=$instance;  
 }  
 **public function** setPriceShow($price){  
 **return** $this->**strategy**->countPrice($price);  
 }  
}  
*//客户端操作*$client=**new** StrategyA();  
$setprice=**new** setPrice($client);  
$data=$setprice->setPriceShow(100);  
**echo** $data;

我们之前在选择出行方式的时候，往往会使用if-else语句，也就是用户不选择A那么就选择B这样的一种情况。这种情况耦合性太高了，而且代码臃肿，有了策略模式我们就可以避免这种现象，策略模式遵循开闭原则，实现代码的解耦合。扩展新的方法时也比较方便，只需要继承策略接口就好了上面列出的这两点算是策略模式的优点了，但是不是说他就是完美的，有很多缺点仍然需要我们去掌握和理解，

客户端必须知道所有的策略类，并自行决定使用哪一个策略类。策略模式会出现很多的策略类。

#### 观察者模式

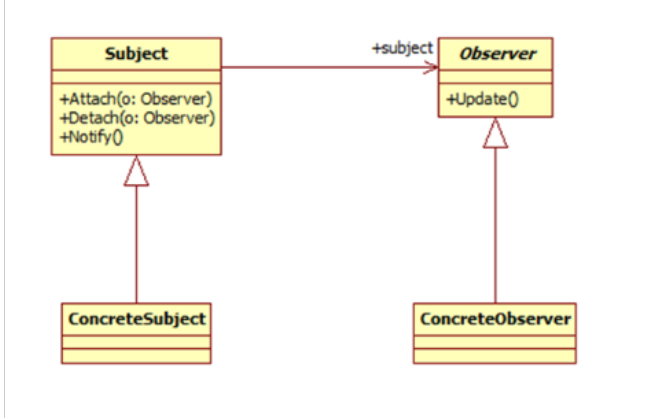
##### 使用场景



##### 什么是观察者模式

观察者模式又叫发布订阅模式属于行为模式的一种，它定义了一对多的依赖关系，让多个观察者对象同时监听某个主题，主题对象状态发生变化时，会通知所有的观察者对象，是他们能够自动更新。

##### 结构图



Subject: 目标  
ConcreteSubject: 具体目标  
Observer: 观察者  
ConcreteObserver: 具体观察者

参考：<https://baijiahao.baidu.com/s?id=1639044219412817957&wfr=spider&for=pc>

<https://blog.csdn.net/leo_jk/article/details/84635305>

<?php

// 被观察者接口

interface Subject{

public function register(Observer $observer); //添加（注册）观察者对象

public function detach(Observer $observer); //删除观察者对象

public function notify(); //通知观察者执行相应功能

}

// 观察者接口

interface Observer{

public function watch(); //观察者要执行的方法

}

<?php

// 被观察者继承类

class Action implements Subject{

public $\_observers=array(); //用于存储观察者对象

//用于添加（注册）观察者对象

public function register(Observer $observer){

$this->\_observers[]=$observer;

}

//用于删除观察者对象

public function detach(Observer $observer){

$index = array\_search($observer, $this->\_observers);

if ($index === FALSE || ! array\_key\_exists($index, $this->\_observers)) {

return FALSE;

}

unset($this->\_observers[$index]);

return TRUE;

}

//通知各观察者

public function notify(){

//\*\*\*\*重点，其实就是循环中执行各观察这对象的watch方法，不同功能方法内容不同但方法名相同

foreach ($this->\_observers as $observer) {

$observer->watch();

}

}

}

// cat观察者继承类

class Cat implements Observer{

public function watch(){

echo "Cat watches TV<hr/>";

}

}

//dog观察者继承类

class Dog implements Observer{

public function watch(){

echo "Dog watches TV<hr/>";

}

}

//people观察者继承类

class People implements Observer{

public function watch(){

echo "People watches TV<hr/>";

}

}

## 注册模式：

<https://www.cnblogs.com/DeanChopper/p/4767181.html>

不管你是通过单例模式还是工厂模式还是二者结合生成的对象，都统统给我“插到”注册树上。我用某个对象的时候，直接从注册树上取一下就好。这和我们使用全局变量一样的方便实用。注册树应该由一个静态变量来充当。而且这个注册树应该是一个二维数组。这个类应该有一个插入对象实例的方法（set()），当让相对应的就应该有一个撤销对象实例的方法（\_unset()）。当然最重要的是还需要有一个读取对象的方法（get()）

<?php

//创建单例

class Single{

public $hash;

static protected $ins=null;

final protected function \_\_construct(){

$this->hash=rand(1,9999);

}

static public function getInstance(){

if (self::$ins instanceof self) {

return self::$ins;

}

self::$ins=new self();

return self::$ins;

}

}

//工厂模式

class RandFactory{

public static function factory(){

return Single::getInstance();

}

}

//注册树

class Register{

protected static $objects;

public static function set($alias,$object){

self::$objects[$alias]=$object;

}

public static function get($alias){

return self::$objects[$alias];

}

public static function \_unset($alias){

unset(self::$objects[$alias]);

}

}

Register::set('rand',RandFactory::factory());

$object=Register::get('rand');

print\_r($object);