抽象工厂模式用来产生套餐，策略模式用来实现不同活动下实现不同价格的算法。抽象工厂模式减少了代码量，更为简单、有效率。策略模式优点： 1、算法可以自由切换，用来实现不同活动下实现不同价格的算法。2、避免使用多重条件判断。 3、扩展性良好。

抽象工场模式下包括一个食物基类baseFood,以及一个食物接口IFood，还有四种具体食物抽象类。  
食物基类包含食物的种类，价格以及数量，还有总价函数和计算价格函数：

//基础食物类

public class \*\*baseFood\*\* implements MemberStrategy{

public String kind;//食物种类

public int num;//食物数量

public float price;//价格

public float calcPrice(float foodPrice) {

return this.num\*this.price;

}

//总价（原价）

public float totalPrice() {

return this.num\*this.price;

}

}

IFood接口：

public interface IFood {

/\*打印输出食物信息\*/

void printMessage();

}

汉堡抽象类：

/\*

\* 汉堡基类

\*/

public abstract class Hamburg extends baseFood implements IFood{

//方法重构

public void printMessage() {

System.out.println("--"+this.kind+"汉堡\t 单价："+this.price+

"数量"+this.num/\*+"合计"+this.totalPrice()\*/);

}

}

鸡翅抽象类：

public abstract class ChickenWings extends baseFood implements IFood{

public void printMessage() {

System.out.println("--"+this.kind+"鸡翅\t 单价："+this.price+

"数量"+this.num/\*+"合计"+this.totalPrice()\*/);

}

}

薯条抽象类：

public abstract class FrenchFires extends baseFood implements IFood{

public void printMessage() {

System.out.println("--"+this.kind+"薯条\t 单价："+this.price+

"数量"+this.num/\*"合计"+this.totalPrice()\*/);

}

}

饮品抽象类：

public abstract class Drink extends baseFood implements IFood{

public void printMessage() {

System.out.println("--"+this.kind+"饮料\t 单价："+this.price+

"数量"+this.num/\*+"合计"+this.totalPrice()\*/);

}

}

抽象工厂类：

public interface IKFCFactory {

// 生产汉堡

public Hamburg createHamburg(int num1);

// 生产薯条

public FrenchFires createFrenchFries(int num2);

// 生产鸡翅

public ChickenWings createChickenWings(int num3);

// 生产饮料

public Drink createBeverage(int num4);

}

工厂实现类：

public class KfcFactory implements IKFCFactory {

//生产汉堡

public Hamburg createHamburg(int num) {

// TODO Auto-generated method stub

return new ChinaHanburm(num);

}

//生产薯条

public FrenchFires createFrenchFries(int num) {

// TODO Auto-generated method stub

return new ChinaFrenchFries(num);

}

//生产鸡翅

public ChickenWings createChickenWings(int num) {

// TODO Auto-generated method stub

return new ChinaChickenWings(num);

}

@Override

public Drink createBeverage(int num) {

// TODO Auto-generated method stub

return new IDrink(num);

}

}

策略模式下有MemberStrategy接口，接口的实现类Activity,baseStrategy和Price类。  
MemberStrategy接口：

/\*策略模式，分出四种计算价格的算法

\* 1.有优惠券，无活动

\* 2.有活动，有优惠券

\* 3.无活动，无优惠劵

\* 4.有活动+有优惠券

\*/

public interface MemberStrategy {

/\*\*

\* 计算餐品的价格

\* @param booksPrice 餐品的原价

\* @return 计算出打折后的价格

\*/

public float calcPrice(float foodPrice);

}

Price类：

//两个实例

private MemberStrategy strategy;

@SuppressWarnings("unused")

private MemberStrategy strategy1;

/\*\*

\* 构造函数，传入一个具体的策略对象

\* @param strategy 具体的策略对象

\*/

public Price(MemberStrategy strategy){

this.strategy = strategy;

this.strategy1=strategy; }

/\*\*

\* 计算餐品的价格

\* @param foodPrice 餐品的原价

\* @return 计算出打折后的价格

\*/

public float quote (float foodPrice ){

return this.strategy.calcPrice(foodPrice);

}

}

baseStrategy类：

/\*

\* 优惠券

\*/

public class baseStrategy implements MemberStrategy{

public float calcPrice(float foodPrice) {//重写计算价格方法

return foodPrice-10;//10元的优惠券

}

}

Activity类：

//活动

/\*

\* 店家有活动时所有商品打85折

\*/

public class Activity implements MemberStrategy{

public float calcPrice(float foodPrice) {//重写计算价格方法

return foodPrice\*0.85f;

}

}