Design Patterns - Builder Pattern

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

生成器模式使用简单的对象来逐步的构建一个复杂的对象。这种类型的设计模式是创建型模式中创建对象最好的方式之一。

一个生成器类会逐步的构建这个最终的对象。这个生成器与其他对象是相互独立的。

Implementation

实现

We have considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

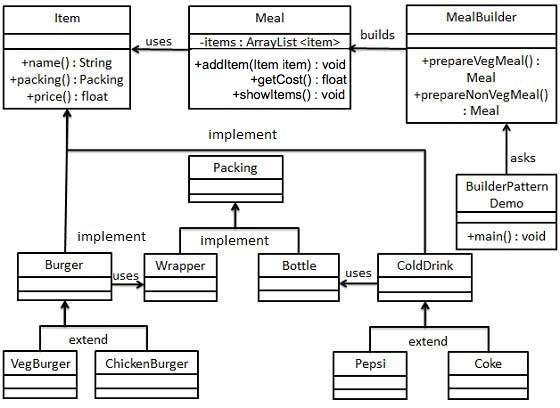
我们举一个快餐店的案例，这个快餐店的典型饮食风格是一个汉堡加一杯可乐。这里的汉堡可以是蔬菜汉堡也可以是鸡肉汉堡，它们将使用包装纸来包装；冷饮可以是可口可乐或者是百事可乐，它们将使用瓶子来包装。

We are going to create an *Item* interface representing food items such as burgers and cold drinks and concrete classes implementing the *Item* interface and a *Packing* interface representing packaging of food items and concrete classes implementing the *Packing* interface as burger would be packed in wrapper and cold drink would be packed as bottle.

我们将创建一个Item接口代表食品元素如汉堡和冷饮，创建具体的类实现这个Item接口；Packing接口代表包装食品元素，创建具体的类实现这个Packing接口，像汉堡可以通过包装纸来包装，冷饮可以通过瓶子来包装。

We then create a *Meal* class having *ArrayList* of *Item* and a *MealBuilder* to build different types of *Meal* objects by combining *Item*. *BuilderPatternDemo*, our demo class will use *MealBuilder* to build a *Meal*.

我们随后将创建一个Meal类，它有一个存储Item类型的ArrayList，然后我们使用一个MealBuilder来结合Item来构建不同类型的Meal对象。BuilderPatternDemo，我们的demo类将使用MealBuilder将会构建一个Meal。



Step 1

第一步

Create an interface Item representing food item and packing.

创建一个接口Item代表食品元素和一个Packing接口代表包装的情况。

*Item.java*

public interface Item {

public String name();

public Packing packing();

public float price();

}

*Packing.java*

public interface Packing {

public String pack();

}

Step 2

第二步

Create concrete classes implementing the Packing interface.

创建具体的类来实现Packing 接口。

*Wrapper.java*

public class Wrapper implements Packing {

@Override

public String pack() {

return "Wrapper";

}

}

*Bottle.java*

public class Bottle implements Packing {

@Override

public String pack() {

return "Bottle";

}

}

Step 3

第三步

Create abstract classes implementing the item interface providing default functionalities.

创建多个抽象类实现Item接口提供默认的功能。

*Burger.java*

public abstract class Burger implements Item {

@Override

public Packing packing() {

return new Wrapper();

}

@Override

public abstract float price();

}

*ColdDrink.java*

public abstract class ColdDrink implements Item {

@Override

public Packing packing() {

return new Bottle();

}

@Override

public abstract float price();

}

Step 4

Create concrete classes extending Burger and ColdDrink classes

创建具体的类继承Burger和ColdDrink类。

*VegBurger.java*

public class VegBurger extends Burger {

@Override

public float price() {

return 25.0f;

}

@Override

public String name() {

return "Veg Burger";

}

}

*ChickenBurger.java*

public class ChickenBurger extends Burger {

@Override

public float price() {

return 50.5f;

}

@Override

public String name() {

return "Chicken Burger";

}

}

*Coke.java*

public class Coke extends ColdDrink {

@Override

public float price() {

return 30.0f;

}

@Override

public String name() {

return "Coke";

}

}

*Pepsi.java*

public class Pepsi extends ColdDrink {

@Override

public float price() {

return 35.0f;

}

@Override

public String name() {

return "Pepsi";

}

}

Step 5

Create a Meal class having Item objects defined above.

创建一个Meal类,该类包含一个Item的集合。

*Meal.java*

import java.util.ArrayList;

import java.util.List;

public class Meal {

private List<Item> items = new ArrayList<Item>();

public void addItem(Item item){

items.add(item);

}

public float getCost(){

float cost = 0.0f;

for (Item item : items) {

cost += item.price();

}

return cost;

}

public void showItems(){

for (Item item : items) {

System.out.print("Item : " + item.name());

System.out.print(", Packing : " + item.packing().pack());

System.out.println(", Price : " + item.price());

}

}

}

Step 6

Create a MealBuilder class, the actual builder class responsible to create Meal objects.

创建一个MealBuilder类，这个类负责实际创建Meal对象。

*MealBuilder.java*

public class MealBuilder {

public Meal prepareVegMeal (){

Meal meal = new Meal();

meal.addItem(new VegBurger());

meal.addItem(new Coke());

return meal;

}

public Meal prepareNonVegMeal (){

Meal meal = new Meal();

meal.addItem(new ChickenBurger());

meal.addItem(new Pepsi());

return meal;

}

}

Step 7

第七步

BuiderPatternDemo uses MealBuider to demonstrate builder pattern.

BuilderPatternDemo使用MealBuilder来演示生成器模式

*BuilderPatternDemo.java*

public class BuilderPatternDemo {

public static void main(String[] args) {

MealBuilder mealBuilder = new MealBuilder();

Meal vegMeal = mealBuilder.prepareVegMeal();

System.out.println("Veg Meal");

vegMeal.showItems();

System.out.println("Total Cost: " + vegMeal.getCost());

Meal nonVegMeal = mealBuilder.prepareNonVegMeal();

System.out.println("\n\nNon-Veg Meal");

nonVegMeal.showItems();

System.out.println("Total Cost: " + nonVegMeal.getCost());

}

}

Step 8

第八步

Verify the output.

校验输出

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5