搜索.....

JavaScript

服条端

数据库

XML 教程

ASP.NET

开发工具

网站建设

首面 HTMI CSS JAVASCRIPT JOUERY BOOTSTRAP SQI MYSQI PHP PYTHON2 PYTHON3 C C++ C# JAVA 本地共祭 设计模式 5 ← 单例模式 原型模式.→ 设计模式 设计模式简介 建造者模式 工厂模式 建造者模式 (Builder Pattern) 使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式,它提供了一种创建对 抽象工厂模式 象的最佳方式。 单例模式 一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。 建造者模式 介绍 原型模式 意圖:将一个复杂的构建与其表示相分离,使得同样的构建过程可以创建不同的表示。 活配器模式 主要解决:主要解决在软件系统中,有时候面临着"一个复杂对象"的创建工作,其通常由各个部分的子对象用一定的算法构成;由于需求的变化, 桥接模式 这个复杂对象的各个部分经常面临着剧烈的变化,但是将它们组合在一起的算法却相对稳定。 讨波器模式 何时使用:一些基本部件不会变,而其组合经常变化的时候。 组合模式 如何解决: 将变与不变分离开。 装饰器模式 关键代码: 建造者: 创建和提供实例, 导演: 管理建造出来的实例的依赖关系。 应用实例: 1、去肯德基,汉堡、可乐、薯条、炸鸡翅等是不变的,而其组合是经常变化的,生成出所谓的"套餐"。 2、JAVA 中的 StringBuilder。 外观模式 优点: 1、建造者独立,易扩展。2、便于控制细节风险。 享元模式 缺点: 1、产品必须有共同点,范围有限制。2、如内部变化复杂,会有很多的建造类。 使用场景: 1、需要生成的对象具有复杂的内部结构。2、需要生成的对象内部属性本身相互依赖。 责任链模式 注意事项: 与工厂模式的区别是: 建造者模式更加关注与零件装配的顺序。 命令模式 实现 解释器模式 迭代器模式 我们假设一个快餐店的商业案例,其中,一个典型的套餐可以是一个汉堡(Burger)和一杯冷饮(Cold drink)。汉堡(Burger)可以是素食汉堡 (Veg Burger) 或鸡肉汉堡 (Chicken Burger) ,它们是包在纸盒中。冷饮(Cold drink)可以是可口可乐(coke)或百事可乐(pepsi),它们是 中介者模式 备忘录模式 我们将创建一个表示食物条目(比如汉堡和冷饮)的 Item 接口和实现 Item 接口的实体类,以及一个表示食物包装的 Packing 接口和实现 Packing 观察者模式 接口的实体类,汉堡是包在纸盒中,冷饮是装在瓶子中。 状态模式 然后我们创建一个 Meal 类,带有 Item 的 ArrayList 和一个通过结合 Item 来创建不同类型的 Meal 对象的 MealBuilder。BuilderPatternDemo,我 空对象模式 们的演示类使用 MealBuilder 来创建一个 Meal。 策略模式 Item 模板模式 items : ArrayList <ite 访问者模式 +name(): String +name(): String +prepareVegMeal(): Meal packing(): Packing +packing(): Packing +price(): float MVC 模式 +prepareNonVegMeal() : Meal 业务代表模式 implement 组合实体模式 Packing 数据访问对象模式 BuilderPattern Demo 前端控制器模式 拦截过滤器模式 implement +main(): void ColdDrink 服务定位器模式 Wrapper Bottle 传输对象模式 设计模式其他 设计模式资源 步骤 1 创建---个表示食物条目和食物包装的接口。 Item.java public interface Item {
 public String name();
 public Packing packing(); public float price(); Packing.java public interface Packing { public String pack(); 步骤 2 创建实现 Packing 接口的实体类。 Wrapper.iava public class Wrapper implements Packing { public String pack() { Bottle.java public class Bottle implements Packing { @Override
public String pack() { return "Bottle"; 步骤 3 创建实现 Item 接口的抽象类,该类提供了默认的功能。 Burger.iava public Packing packing() { return new Wrapper(): @Override
public abstract float price();

\*

反馈/建议

```
ColdDrink.java

public abstract class ColdDrink implements Item {

    @Override
    public Packing packing() {
        return new Bottle();
    }

    @Override
    public abstract float price();
}
```

## 步骤 4

```
创建扩展了 Burger 和 ColdDrink 的实体类。
```

```
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";
    }
}
```

## 步骤 5

创建一个 Meal 类,带有上面定义的 Item 对象。

```
Meal.java
import java.util.ArrayList;
import java.util.List;

public class Meal {
    private List(Ttem> 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());
        }
    }
}
```

## 步骤 6

创建一个 MealBuilder 类,实际的 builder 类负责创建 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 ChickenBurger());
        meal.addItem(new Pepsi());
```

反馈/建议

```
return meal;
步骤 7
BuiderPatternDemo 使用 MealBuider 来演示建造者模式 (Builder Pattern)。
 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.showttems();
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());
步骤 8
执行程序,输出结果:
  Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
   Non-Veg Meal
  Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
   Total Cost: 85.5
   ◆ 单例模式
                                                                                                                           原型模式 🗲
② 2 篇笔记
                                                                                                                            ☞ 写笔记
建造者模式举例:去肯德基点餐,我们可以认为点餐就属于一个建造订单的过程。我们点餐的顺序是无关的,点什么东西也是没有要求的,可以单点,也可以点套餐,也可以套餐加单点,但是最后一定要点确认来完成订单。
          public class OrderBuilder{
              private Burger mBurger;
              private Suit mSuit:
              public OrderBuilder burger(Burger burger   int num){
                 mBurger = burger;
             //点套餐,实际中套餐也可以点多份
              public OrderBuilder suit(Suit suit, int num){
              mSuit = suit;
              //完成订单
              public Order build(){
                  Order order = new Order();
                  order.setBurger(mBurger);
                   order.setSuit(mSuit);
                  return order;
       另外适用于快速失败,在 build 时可以做校验,如果不满足必要条件,则可以直接抛出创建异常,在 OkHttp3 中的
       Request.Builder 中就是这样用的。
         public Request build() {
              if (url == null) throw new IllegalStateException("url == null");
             return new Request(this);
       例如订单要求价格至少达到 30 块
          //完成订单
             Order order = new Order():
              order.setBurger(mBurger);
              order.setSuit(mSuit);
             if(order.getPrice() < 30){
throw new BuildException("订单金额未达到30元");
       另外,在构建时如果有必传参数和可选参数,可以为 Builder 类添加构造函数来保证必传参数不会遗漏,例如在构建一
          public class RequestBuilder
             private final String mUrl;
              private Map<String, String> mHeaders = new HashMap<String, String>();
             private RequestBuilder(String url){
                 return new RequestBuilder(url);
             public RequestBuilder addHeader(String key, String value){
    mHeaders.put(key, value);
              public Request build(){
                  Request request = new Request();
request.setUrl(mUrl);
                   request.setHeaders(mHeaders);
```

反馈/建议

```
牟春平 3个月前 (07-26)
建造者模式,又称生成器模式:将一个复杂的构建与其表示相分离,使得同样的构建过程可以创建不同的表示。
     三个角色:建造者、具体的建造者、监工、使用者 (严格来说不算)
      ■ 建造者角色: 定义生成实例所需要的所有方法;
      具体的建造者角色:实现生成实例所需要的所有方法,并且定义获取最终生成实例的方法;
      监工角色: 定义使用建造者角色中的方法来生成实例的方法;
      ● 使用者: 使用建造者模式。
        注意: 定义中"将一个复杂的构建过程与其表示相分离",表示并不是由建造者负责一切,而是由监工负责控制(定义)一
         个复杂的构建过程,由各个不同的建造者分别负责实现构建过程中所用到的所有构建步骤。不然,就无法做到"使得同样
        的构建过程可以创建不同的表示"这一目标。
       public abstract class Builder {
          public abstract void buildPart1();
public abstract void buildPart2();
public abstract void buildPart3();
      监丁鱼色:
       public class Director {  // 将一个复杂的构建过程与其表示相分离
private Builder builder;  // 针对接口编程·而不是针对实现编程
public Director(Builder builder) {
              this.builder = builder;
           public void setBuilder(Builder builder) {
              this.builder = builder;
           public \ void \ construct() \ \{ \ \ // 控制 (定义) 一个复杂的构建过程
              具体的建造者角色:
         * 此处实现了建造纯文本文档的具体建造者。

    可以考虑再实理一个建造HTML文档、XML文档、或者其它什么文档的具体建造者。
    这样,就可以使得同样的构建过程可以创建不同的表示。

       public class ConcreteBuilder1 extends Builder {
         private StringBuffer buffer = new StringBuffer();//假设 buffer.toString() 就是最终生成的产品
           public void buildPart1() {//实现构建最终实例需要的所有方法
              buffer.append("Builder1 : Part1\n");
          }
           public void buildPart2() {
              buffer.append("Builder1 : Part2\n");
           public void buildPart3() {
              buffer.append("Builder1 : Part3\n");
           public String getResult() {//定义获取最终生成实例的方法
              return buffer.toString();
      客户角色:
       public class Client {
          public void testBuilderPattern() {
    ConcreteBuilder1 bl = new ConcreteBuilder1();//逆遊者
    Director director = new Director(bl)://宣正
    director.construct();//接接支幣(正了的報告等:建始者实际建始)
    String result = bl.getResult();//接收最終生成结果
               System.out.printf("the result is :%n%s", result);
            iade 3周前 (09-20)
```

```
在线实例
                     字符集&工具
                                           最新更新
                     · HTML 字符集设置
                                                                 ・意见反馈
· HTML 实例
                                           · NumPy 迭代数组
                     · HTML ASCII 字符集
· CSS 实例
                                           · NumPv 广播(Broa...
                                                                  免责声明
                     · HTML ISO-8859-1
· JavaScript 空例
                                           · NumPy 高级索引
                                                                  关于我们
Aiax 实例
                     · HTML 实体符号
                                           · NumPy 切片和索引
                                                                  文章归档
 jQuery 实例
                     · HTML 拾色器
                                           NumPy 从数值范...
 XML 实例
                     · JSON 格式化工具
                                           · NumPy 从已有的..
Java 实例
                                           · NumPy 创建数组
```

关注微信



Copyright © 2013-2018 菜鸟教程 runoob.com All Rights Reserved. 备案号: 闽ICP备15012807号-1