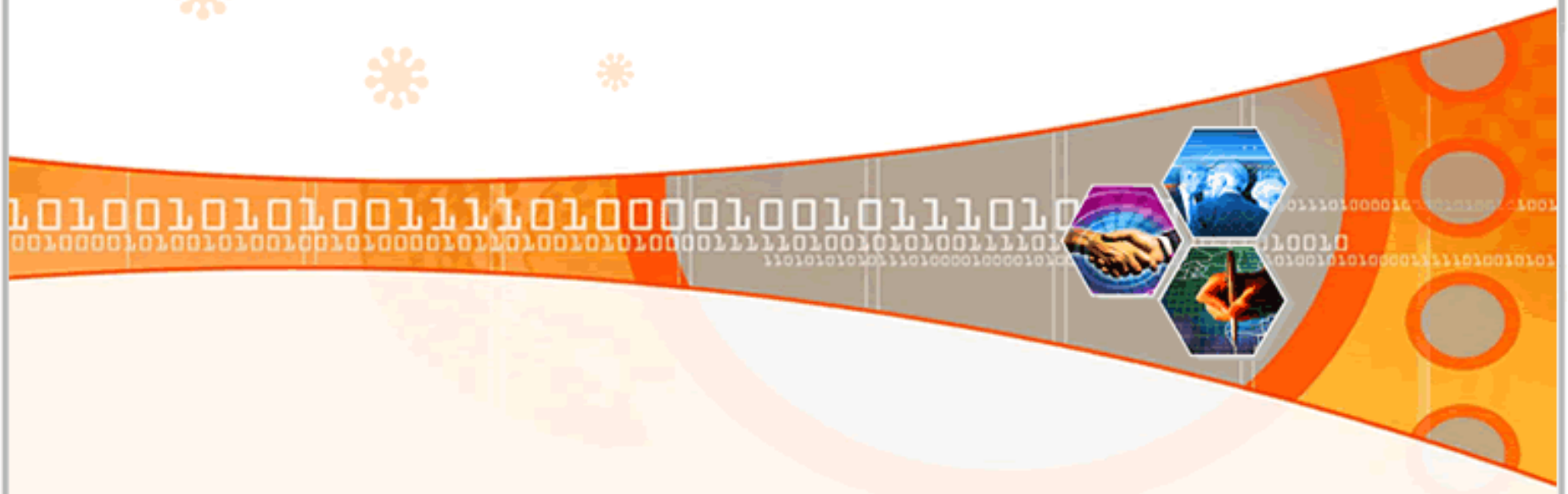
Design Patterns

建造者模式

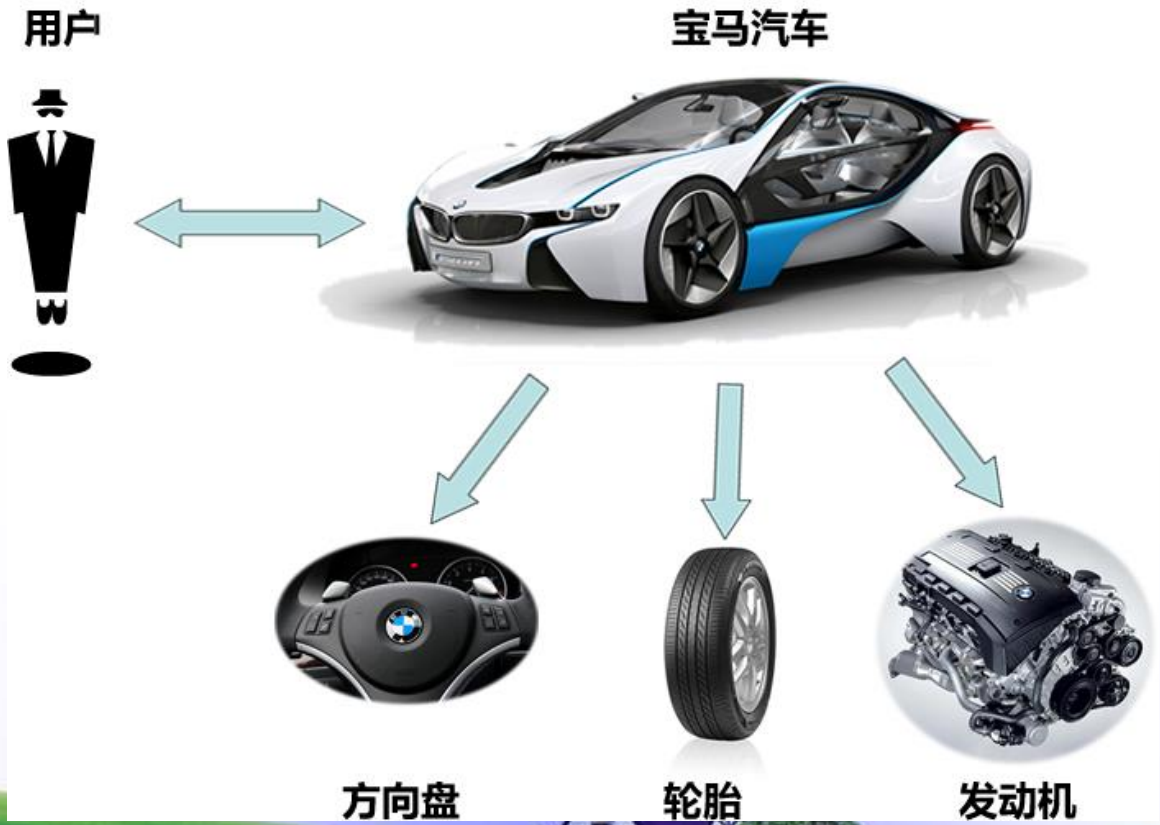


大纲

- 建造者模式概述
- 建造者模式的结构与实现
- 建造者模式的应用实例
- 指挥者类的深入讨论
- 建造者模式的优缺点与适用环境

建造者模式概述

- 复杂对象



复杂对象（汽车）示意图

建造者模式概述

- 分析
  - ✓ 如何将这些部件组装成一辆完整的汽车并返回给用户？

建造者模式

建造者模式可以将部件本身和它们的组装过程分开，关注如何一步步创建一个包含多个组成部分的复杂对象，用户只需要指定复杂对象的类型即可得到该对象，而无须知道其内部的具体构造细节。

建造者模式概述

- 建造者模式的定义

建造者模式：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

Builder Pattern: **Separate the construction of a complex object from its representation** so that the same construction process can create different representations.

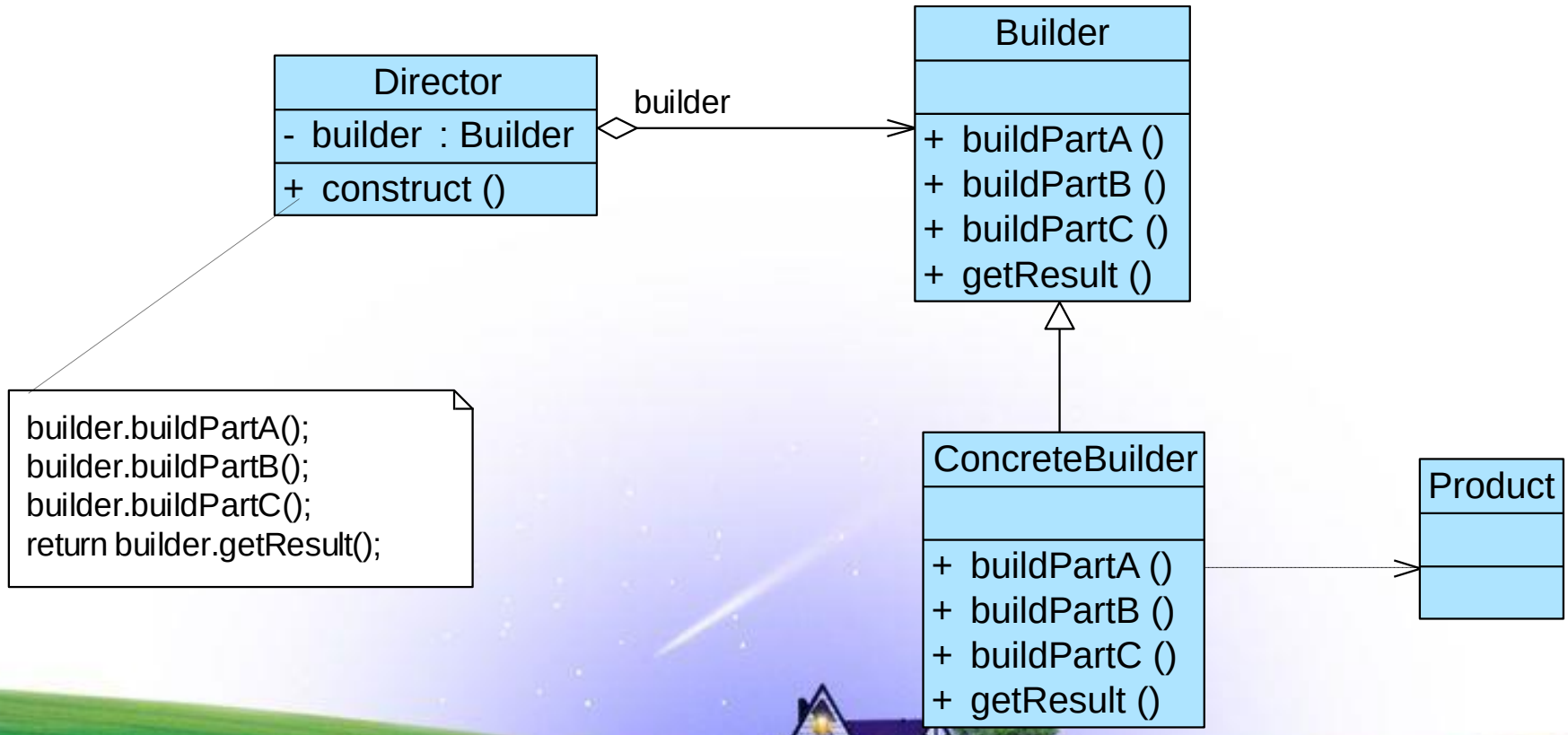
✓对象创建型模式

建造者模式概述

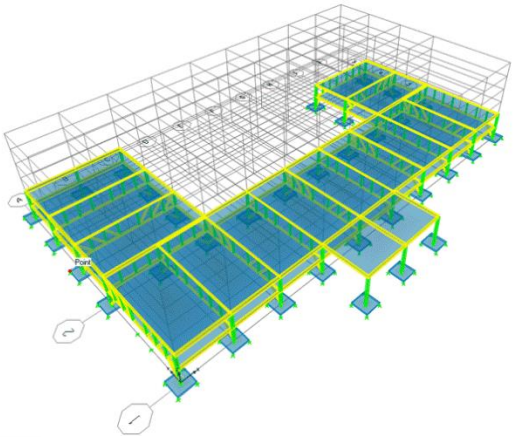
- 建造者模式的定义
  - ✓ 将客户端与包含多个部件的复杂对象的创建过程分离，客户端无须知道复杂对象的内部组成部分与装配方式，只需要知道所需建造者的类型即可
  - ✓ 关注如何逐步创建一个复杂的对象，不同的建造者定义了不同的创建过程

建造者模式的结构与实现

- 建造者模式的结构



建造者模式的结构与实现



- 建造者模式的结构
  - ✓ 建造者模式包含以下4个角色:
    - Builder（抽象建造者）
    - ConcreteBuilder（具体建造者）
    - Product（产品）
    - Director（指挥者）

建造者模式的结构与实现

- 建造者模式的实现
  - ✓典型的复杂对象类代码:

```
public class Product {
        private String partA; //定义部件，部件可以是任意类型，包括值类型和引用类型
        private String partB;
        private String partC;

        //partA的Getter方法和Setter方法省略
        //partB的Getter方法和Setter方法省略
        //partC的Getter方法和Setter方法省略
}
```

建造者模式的结构与实现

- 建造者模式的实现

  ✓典型的抽象建造者类代码:

```
public abstract class Builder {
    //创建产品对象
    protected Product product=new Product();
    public abstract void buildPartA();
    public abstract void buildPartB();
    public abstract void buildPartC();

    //返回产品对象
    public Product getResult() {
        return product;
    }
}
```

建造者模式的结构与实现

- 建造者模式的实现

  ✓典型的具体建造者类代码:

```
public class ConcreteBuilder1 extends Builder{
    public void buildPartA() {
        product.setPartA("A1");
    }

    public void buildPartB() {
        product.setPartB("B1");
    }

    public void buildPartC() {
        product.setPartC("C1");
    }
}
```

建造者模式的结构与实现

```
public class Director {
    private Builder builder;

    public Director(Builder builder) {
        this.builder=builder;
    }

    public void setBuilder(Builder builder) {
        this.builder=builer;
    }

    //产品构建与组装方法
    public Product construct() {
        builder.buildPartA();
        builder.buildPartB();
        builder.buildPartC();
        return builder.getResult();
    }
}
```

建造者模式的结构与实现

- 建造者模式的实现
  - ✓客户类代码片段:

```
……
Builder builder = new ConcreteBuilder1(); //可通过配置文件实现
Director director = new Director(builder);
Product product = director.construct();
……
```

建造者模式的应用实例

实例说明

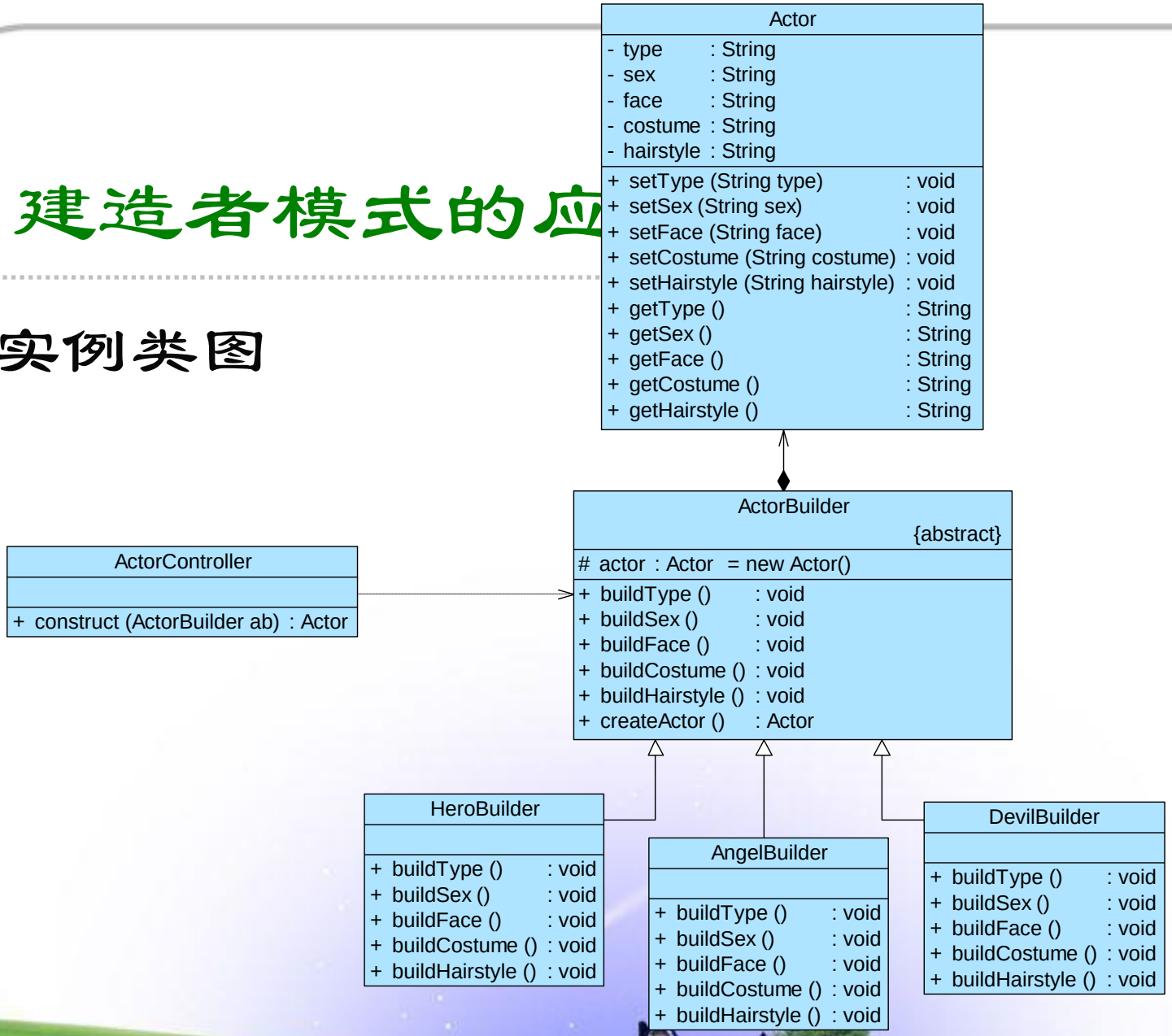
某游戏软件公司决定开发一款基于角色扮演的多人在线网络游戏，玩家可以在游戏中扮演虚拟世界中的一个特定角色，角色根据不同的游戏情节和统计数据（例如力量、魔法、技能等）具有不同的能力，角色也会随着不断升级而拥有更加强大的能力。

作为该游戏的一个重要组成部分，需要对游戏角色进行设计，而且随着该游戏的升级将不断增加新的角色。通过分析发现，游戏角色是一个复杂对象，它包含性别、面容等多个组成部分，不同类型的游戏角色，其性别、面容、服装、发型等外部特性都有所差异，例如“天使”拥有美丽的面容和披肩的长发，并身穿一袭白裙；而“恶魔”极其丑陋，留着光头并穿一件刺眼的黑衣。

无论是何种造型的游戏角色，它的创建步骤都大同小异，都需要逐步创建其组成部分，再将各组成部分装配成一个完整的游戏角色。现使用建造者模式来实现游戏角色的创建。

建造者模式的应

- 实例类图



游戏角色创建结构图

建造者模式的应用实例

- 实例代码
  - (1) Actor：游戏角色类，充当产品对象
  - (2) ActorBuilder：游戏角色建造者，充当抽象建造者
  - (3) HeroBuilder：英雄角色建造者，充当具体建造者
  - (4) AngelBuilder：天使角色建造者，充当具体建造者
  - (5) DevilBuilder：恶魔角色建造者，充当具体建造者
  - (6) ActorController：角色控制器，充当指挥者
  - (7) Client：客户端测试类

演示.....

Code (designpatterns.builder)

```
public class Actor {//充当产品对象
    private String type; //角色类型
    private String sex; //性别
    private String face; //脸型
    private String costume; //服装
    private String hairstyle; //发型
    public void setType(String type) {
            this.type = type;
    }
    public void setSex(String sex) {
            this.sex = sex;
    }
    public void setFace(String face) {
            this.face = face;
    }
    public void setCostume(String costume) {
            this.costume = costume;
    }
    public void setHairstyle(String hairstyle) {
            this.hairstyle = hairstyle;
    }
```

```
    public String getType() {
            return (this.type);
     }
     public String getSex() {
            return (this.sex);
     }
     public String getFace() {
            return (this.face);
    }
    public String getCostume() {
            return (this.costume);
    }
    public String getHairstyle() {
            return (this.hairstyle);
    }
}
```

```java
//游戏角色建造者，充当抽象建造者
public abstract class ActorBuilder {
    protected Actor actor = new Actor();

    public abstract void buildType();
    public abstract void buildSex();
    public abstract void buildFace();
    public abstract void buildCostume();
    public abstract void buildHairstyle();

    //返回1个完整的游戏角色对象
    public Actor createActor() {
        return actor;
    }
}
```

```java
//天使角色建造者，充当具体建造者
public class AngelBuilder extends ActorBuilder {
    public void buildType() {
        actor.setType("天使");
    }
    public void buildSex() {
        actor.setSex("女");
    }
    public void buildFace() {
        actor.setFace("漂亮");
    }
    public void buildCostume() {
        actor.setCostume("白裙");
    }
    public void buildHairstyle() {
        actor.setHairstyle("披肩长发");
    }
}
```

```
//游戏角色建造者，充当抽象建造者
public abstract class ActorBuilder {
    protected Actor actor = new Actor();

    public abstract void buildType();
    public abstract void buildSex();
    public abstract void buildFace();
    public abstract void buildCostume();
    public abstract void buildHairstyle();

    //返回1个完整的游戏角色对象
    public Actor createActor() {
        return actor;
    }
}
```

```
//恶魔角色建造者，充当具体建造者
public class DevilBuilder extends ActorBuilder {
        public void buildType() {
                actor.setType("恶魔");
        }
        public void buildSex() {
                actor.setSex("妖");
        }
        public void buildFace() {
                actor.setFace("丑陋");
        }
        public void buildCostume() {
                actor.setCostume("黑衣");
        }
        public void buildHairstyle() {
                actor.setHairstyle("光头");
        }
}
```

```java
//游戏角色建造者，充当抽象建造者
public abstract class ActorBuilder {
    protected Actor actor = new Actor();

    public abstract void buildType();
    public abstract void buildSex();
    public abstract void buildFace();
    public abstract void buildCostume();
    public abstract void buildHairstyle();

    //返回1个完整的游戏角色对象
    public Actor createActor() {
        return actor;
    }
}
```

```java
//英雄角色建造者，充当具体建造者
public class HeroBuilder extends ActorBuilder {
        public void buildType() {
                actor.setType("英雄");
        }
        public void buildSex() {
                actor.setSex("男");
        }
        public void buildFace() {
                actor.setFace("英俊");
        }
        public void buildCostume() {
                actor.setCostume("盔甲");
        }
        public void buildHairstyle() {
                actor.setHairstyle("飘逸");
        }
}
```

建造者模式的应用实例

```
//角色控制器，充当指挥者
public class ActorController {
   //逐步构建产品对象
        public Actor construct(ActorBuilder ab) {
                Actor actor;
                ab.buildType();
                ab.buildSex();
                ab.buildFace();
                ab.buildCostume();
                ab.buildHairstyle();
                actor=ab.createActor();
                return actor;
        }
}
```

建造者模式的应用实例

```
public class Client {
        public static void main(String args[]) {
                ActorBuilder ab; //针对抽象建造者编程
                ab = new AngelBuilder();//为构建一个天使做准备
                //ab = (ActorBuilder)XMLUtil.getBean(); //反射生成具体建造者对象
                 ActorController ac = new ActorController();
                Actor actor;
                actor = ac.construct(ab); //通过指挥者创建完整的建造者对象

                String type = actor.getType();
                System.out.println(type + "的外观：");
                System.out.println("性别：" + actor.getSex());
                System.out.println("面容：" + actor.getFace());
                System.out.println("服装：" + actor.getCostume());
                System.out.println("发型：" + actor.getHairstyle());
        }
}
```

建造者模式的应用实例

- 结果及分析
  - ✓ 如果需要更换具体角色建造者，只需要修改配置文件
  - ✓ 当需要增加新的具体角色建造者时，只需将新增具体角色建造者作为抽象角色建造者的子类，然后修改配置文件即可，原有代码无须修改，完全符合开闭原则

```
<?xml version="1.0"?>
<config>
   <className>designpatterns.builder.AngelBuilder</className>
</config>
```

建造者模式的优缺点与适用环境

- 模式优点
  - ✓ 客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象
  - ✓ 每一个具体建造者都相对独立，与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，扩展方便，符合开闭原则
  - ✓ 可以更加精细地控制产品的创建过程

建造者模式的优缺点与适用环境

- 模式缺点
  - ✓ 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，不适合使用建造者模式，因此其使用范围受到一定的限制
  - ✓ 如果产品的内部变化复杂，可能会需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加了系统的理解难度和运行成本

建造者模式的优缺点与适用环境

- 模式适用环境
  - ✓ 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员变量
  - ✓ 需要生成的产品对象的属性相互依赖，需要指定其生成顺序
  - ✓ 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类和客户类中
  - ✓ 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品

END