南京大学

设计模式 –创建型模式(二)

Design Pattern-Creational Pattern (2)

2022/3/31

本章内容

- 建造者模式 (Builder)
- 原型模式 (Prototype)
- 单例模式 (Singleton)

建造者模式概述

- 复杂对象

复杂对象（游戏人物）示意图

游戏角色存在不同类型。且游戏角色是一个复杂对象，它包含属性、技能、外观（性别、面容、服装、发型等）等多个组成部分，必须同时或按先后顺序创建以上组成部分后组合才能得到一个对象。

建造者模式概述

- 分析
  - 如何将这些部件组装成一个完整的游戏对象并返回给用户？

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.

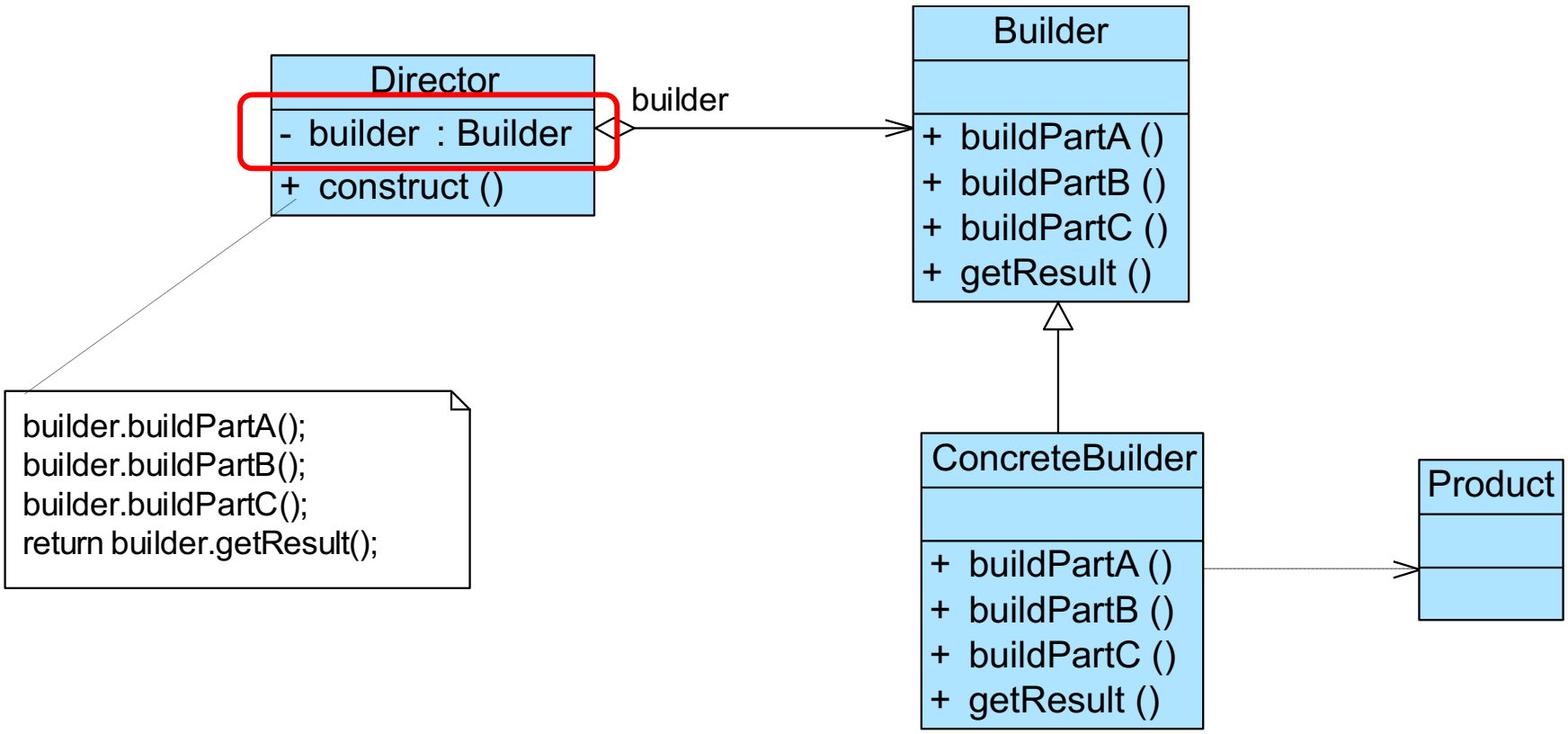
- 对象创建型模式

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

- 建造者模式的实现

```
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

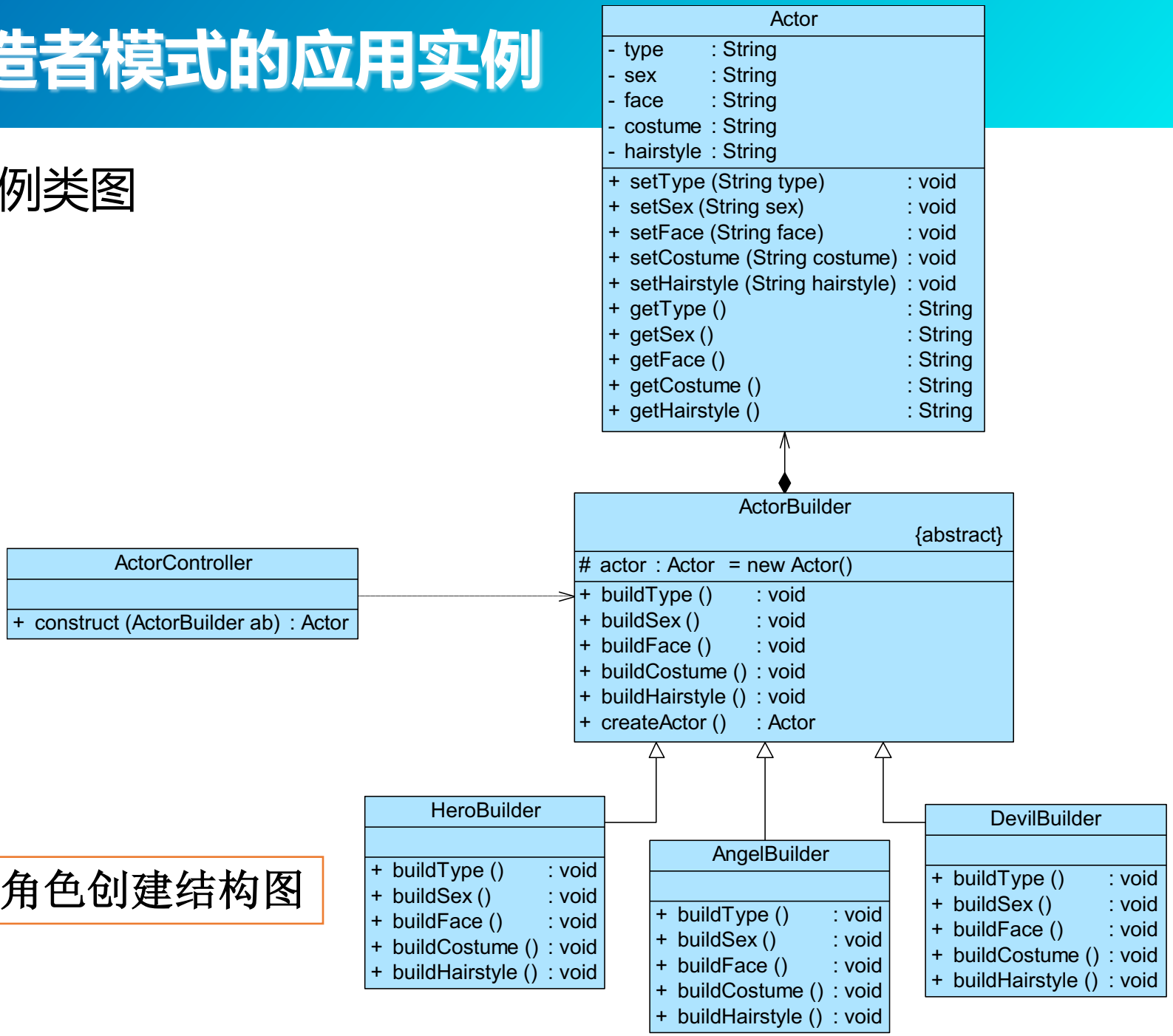
某游戏软件公司决定开发一款基于角色扮演的多人在线网络游戏，玩家可以在游戏中扮演虚拟世界中的一个特定角色，角色根据不同的游戏情节和统计数据（例如力量、魔法、技能等）具有不同的能力，角色也会随着不断升级而拥有更加强大的能力。

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。

建造者模式的应用实例

- 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，充当复杂产品对象
  - (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)

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>
```

指挥者类的深入讨论

- 省略Director
  - 将Director和抽象建造者Builder合并

```
public abstract class ActorBuilder {
   protected static Actor actor = new Actor();

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

   public static Actor construct(ActorBuilder ab) {
      ab.buildType();
      ab.buildSex();
      ab.buildFace();
      ab.buildCostume();
      ab.buildHairstyle();
      return actor;
   }
}
```

```
……
ActorBuilder ab;
ab = (ActorBuilder)XMLUtil.getBean();

Actor actor;
actor = ActorBuilder.construct(ab);
……
```

指挥者类的深入讨论

- 省略Director
  - 将construct()方法中的参数去掉，直接在construct()方法中调用buildPartX()方法

```
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();

   public Actor construct() {
      this.buildType();
      this.buildSex();
      this.buildFace();
      this.buildCostume();
      this.buildHairstyle();
      return actor;
   }
}
```

```
……
ActorBuilder ab;
ab = (ActorBuilder)XMLUtil.getBean();

Actor actor;
actor = ab.construct();
……
```

指挥者类的深入讨论

- 钩子方法的引入
  - 钩子方法(Hook Method)：返回类型通常为boolean类型，方法名一般为isXXX()

```
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();

   //钩子方法
    public boolean isBareheaded() {
      return false;
   }

   public Actor createActor() {
      return actor;
   }
}
```

指

- 钩

```
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");
        }

        //覆盖钩子方法
        public boolean isBareheaded() {
                return true;
        }
}
```

指挥者类的深入讨论

- 钩子方法的引入

```
public class ActorController {
    public Actor construct(ActorBuilder ab) {
        Actor actor;
        ab.buildType();
        ab.buildSex();
        ab.buildFace();
        ab.buildCostume();
        //通过钩子方法来控制产品的构建
        if(!ab.isBareheaded()) {
            ab.buildHairstyle();
        }
        actor=ab.createActor();
        return actor;
    }
}
```

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

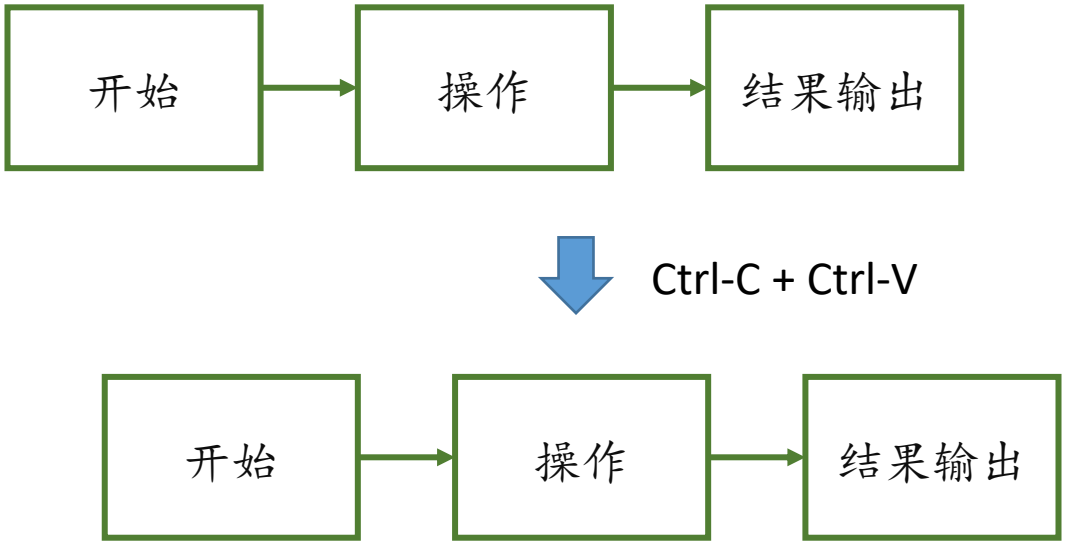
- 模式适用环境
  - 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员变量
  - 需要生成的产品对象的属性相互依赖，需要指定其生成顺序
  - 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类和客户类中
  - 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品

相关模式

- Abstract Factory
  - 相似：可以创建复杂对象
  - 区别：(1) Builder着重于一步步构造一个复杂对象，而Abstract Factory着重于多个系列的产品对象；(2) Builder在最后的一步返回产品，而Abstract Factory中产品是立即返回的。
- Composite通常是用Builder生成的

原型模式概述

- Ctrl-C与Ctrl-V实现



原型模式概述

- 分析
  - 软件开发问题：通过复制一个原型对象得到多个与原型对象一模一样的新对象

原型模式概述

- 原型模式的定义

原型模式：使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象。

Prototype Pattern: Specify the kinds of objects to create using a prototypical instance, and **create new objects by copying this prototype**.

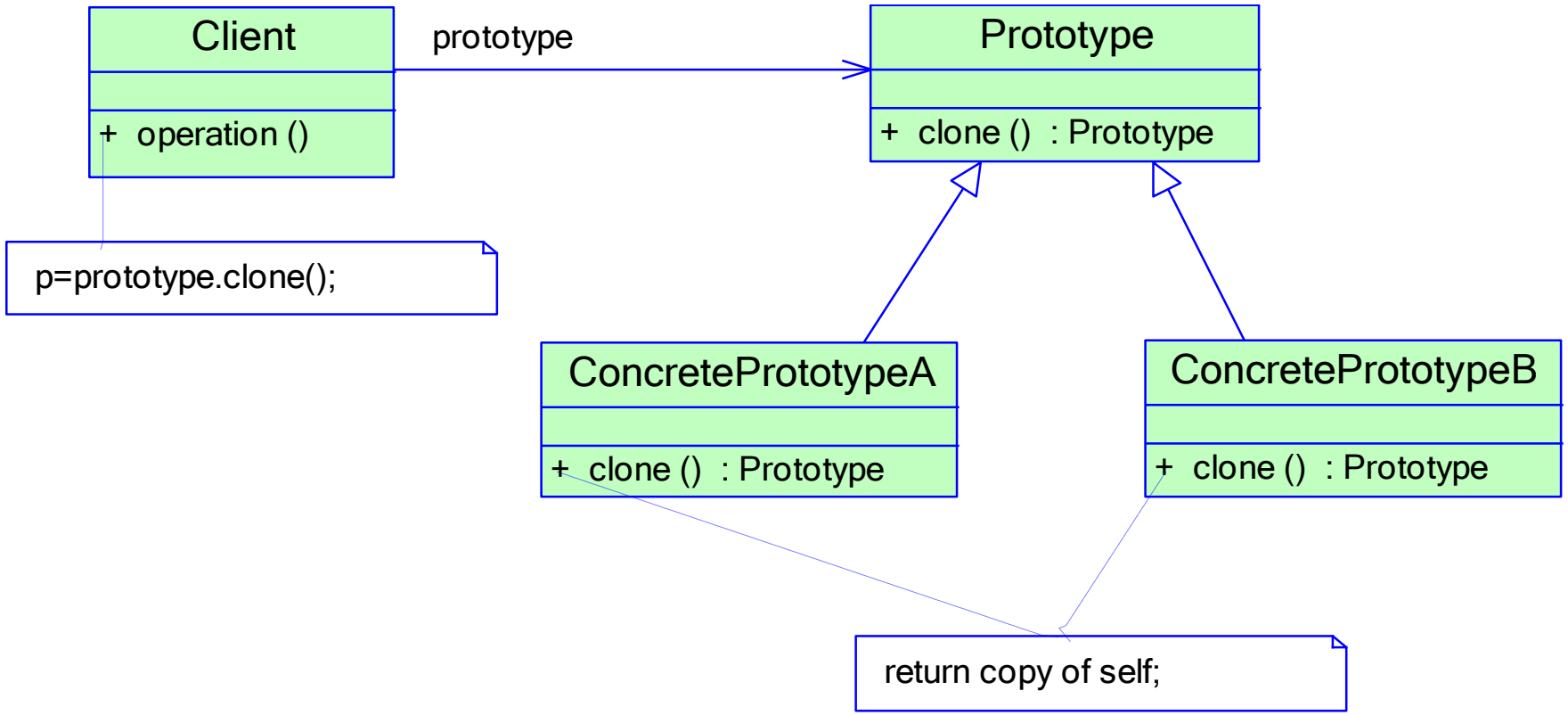
- 对象创建型模式

原型模式概述

- 原型模式的定义
  - 工作原理：将一个原型对象传给要发动创建的对象（即客户端对象），这个要发动创建的对象通过请求原型对象复制自己来实现创建过程
  - 创建新对象（也称为克隆对象）的工厂就是原型类自身，工厂方法由负责复制原型对象的克隆方法来实现
  - 通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，每一个克隆对象都是独立的
  - 通过不同的方式对克隆对象进行修改以后，可以得到一系列相似但不完全相同的对象

原型模式的结构与实现

- 原型模式的结构

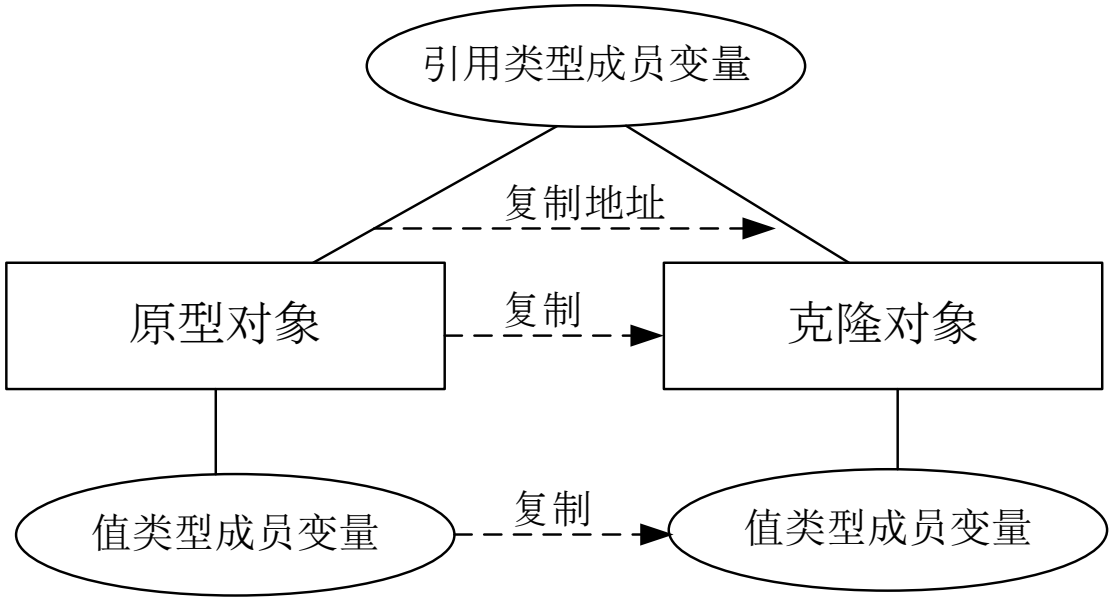


原型模式的结构与实现

- 原型模式的结构
  - 原型模式包含以下3个角色：
    - Prototype（抽象原型类）
    - ConcretePrototype（具体原型类）
    - Client（客户类）

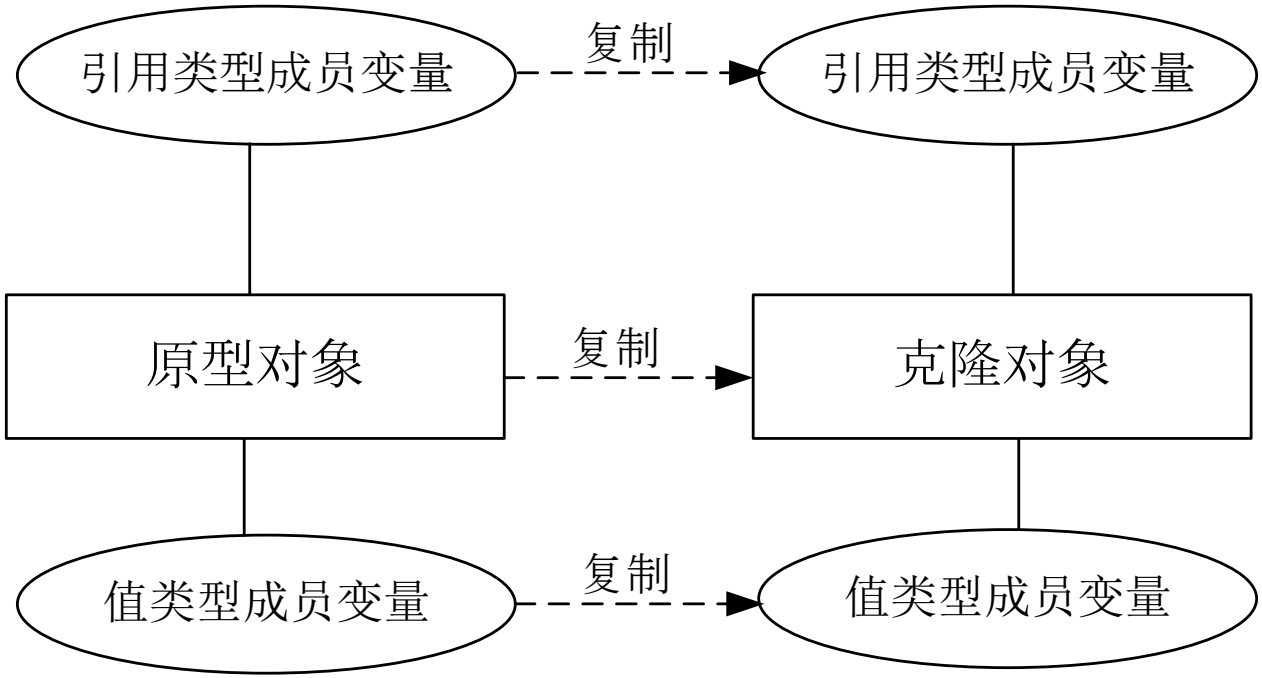
原型模式的结构与实现

- 浅克隆与深克隆
  - 浅克隆(Shallow Clone)：当原型对象被复制时，只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制



原型模式的结构与实现

- 浅克隆与深克隆
  - 深克隆(Deep Clone)：除了对象本身被复制外，对象所包含的所有成员变量也将被复制



```
public interface Prototype {
    public Prototype clone();
}

public class ConcretePrototype implements Prototype {
    private String attr;

    public void setAttr(String attr) {
        this.attr = attr;
    ……
    ConcretePrototype prototype = new ConcretePrototype();
    prototype.setAttr("Sunny");
    ConcretePrototype copy = (ConcretePrototype)prototype.clone();
    ……

    //克隆方法
    public Prototype clone() {
        Prototype prototype = new ConcretePrototype(); //创建新对象
        prototype.setAttr(this.attr);
        return prototype;
    }
}
```

原型模式的结构与实现

- 原型模式的实现
  - Java语言中的clone()方法和Cloneable接口
    - 在Java语言中，提供了一个clone()方法用于实现浅克隆，该方法使用起来很方便，直接调用super.clone()方法即可实现克隆

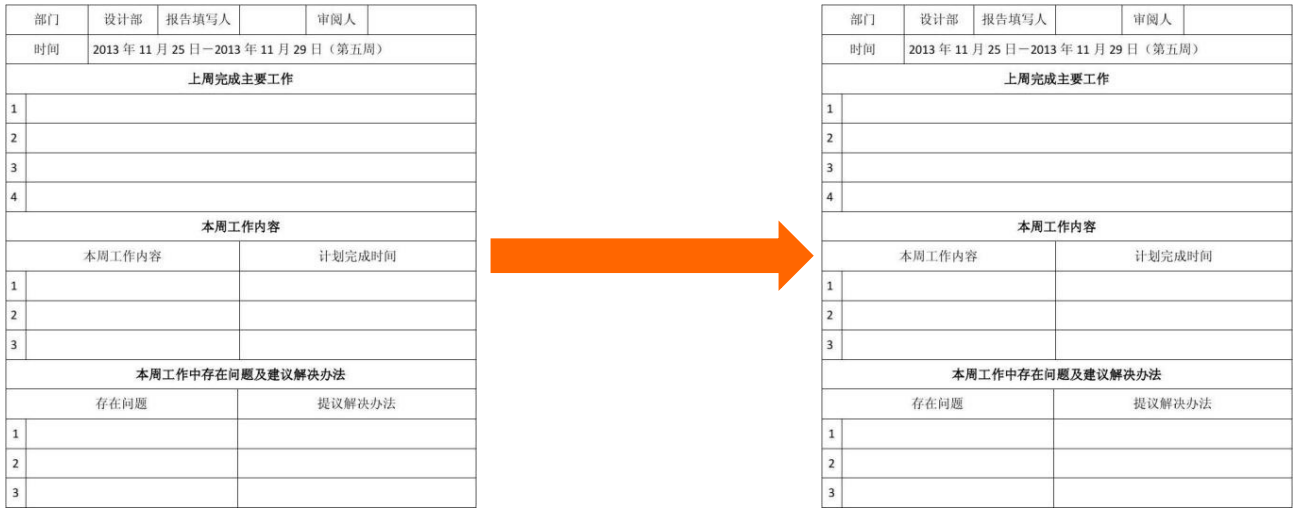
```
public class ConcretePrototype implements Cloneable {
    ……
    //Shallow Clone
    public Prototype clone() {
        Prototype protptype = new ConcretePrototype();
        Prototype copy = protptype.clone();

        }
        catch (CloneNotSupportedException exception) {
            System.err.println("Not support cloneable");
        }
        return (Prototype )object;
    }
    ……
}
```

原型模式的应用实例

- 实例说明

在使用某OA系统时，有些岗位的员工发现他们每周的工作都大同小异，因此在填写工作周报时很多内容都是重复的，为了提高工作周报的创建效率，大家迫切希望有一种机制能够快速创建相同或者相似的周报，包括创建周报的附件。

试使用原型模式对该OA系统中的工作周报创建模块进行改进。

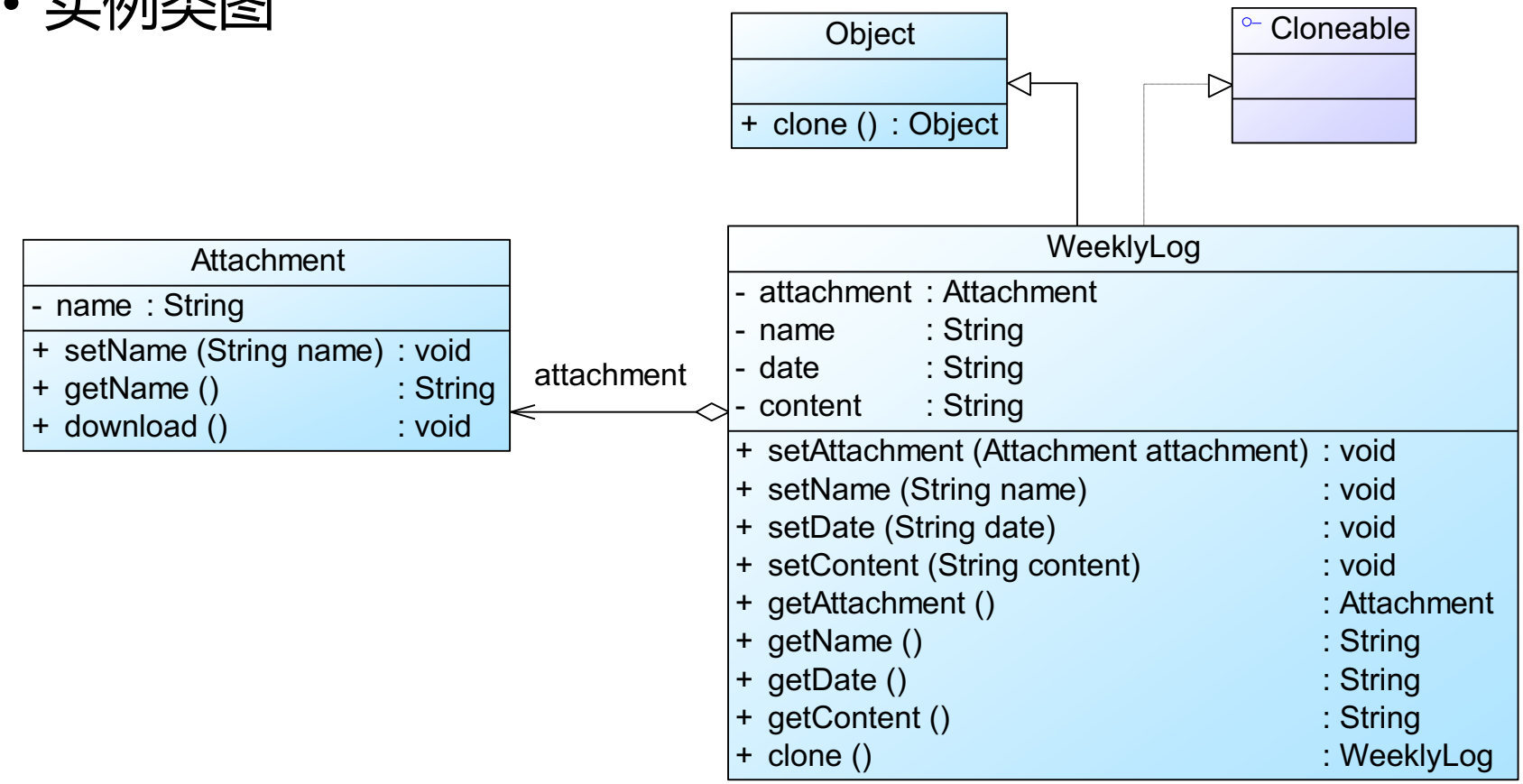


部门	设计部	报告填写人		审阅人	
时间	2013 年 11 月 25 日－2013 年 11 月 29 日（第五周）				
上周完成主要工作					
1					
2					
3					
4					
本周工作内容					
本周工作内容			计划完成时间		
1					
2					
3					
本周工作中存在问题及建议解决办法					
存在问题			提议解决办法		
1					
2					
3					

部门	设计部	报告填写人		审阅人	
时间	2013 年 11 月 25 日－2013 年 11 月 29 日（第五周）				
上周完成主要工作					
1					
2					
3					
4					
本周工作内容					
本周工作内容			计划完成时间		
1					
2					
3					
本周工作中存在问题及建议解决办法					
存在问题			提议解决办法		
1					
2					
3					

原型模式的应用实例

- 实例类图



工作周报创建模块结构图：浅克隆

原型模式的应用实例

- 实例代码
  - Object：抽象原型角色

```
package java.lang;
public class Object {
    ……
    protected native Object clone() throws CloneNotSupportedException;
    ……
}
```

演示……

Code (designpatterns.prototype.shallowclone)

原型模式的应用实例

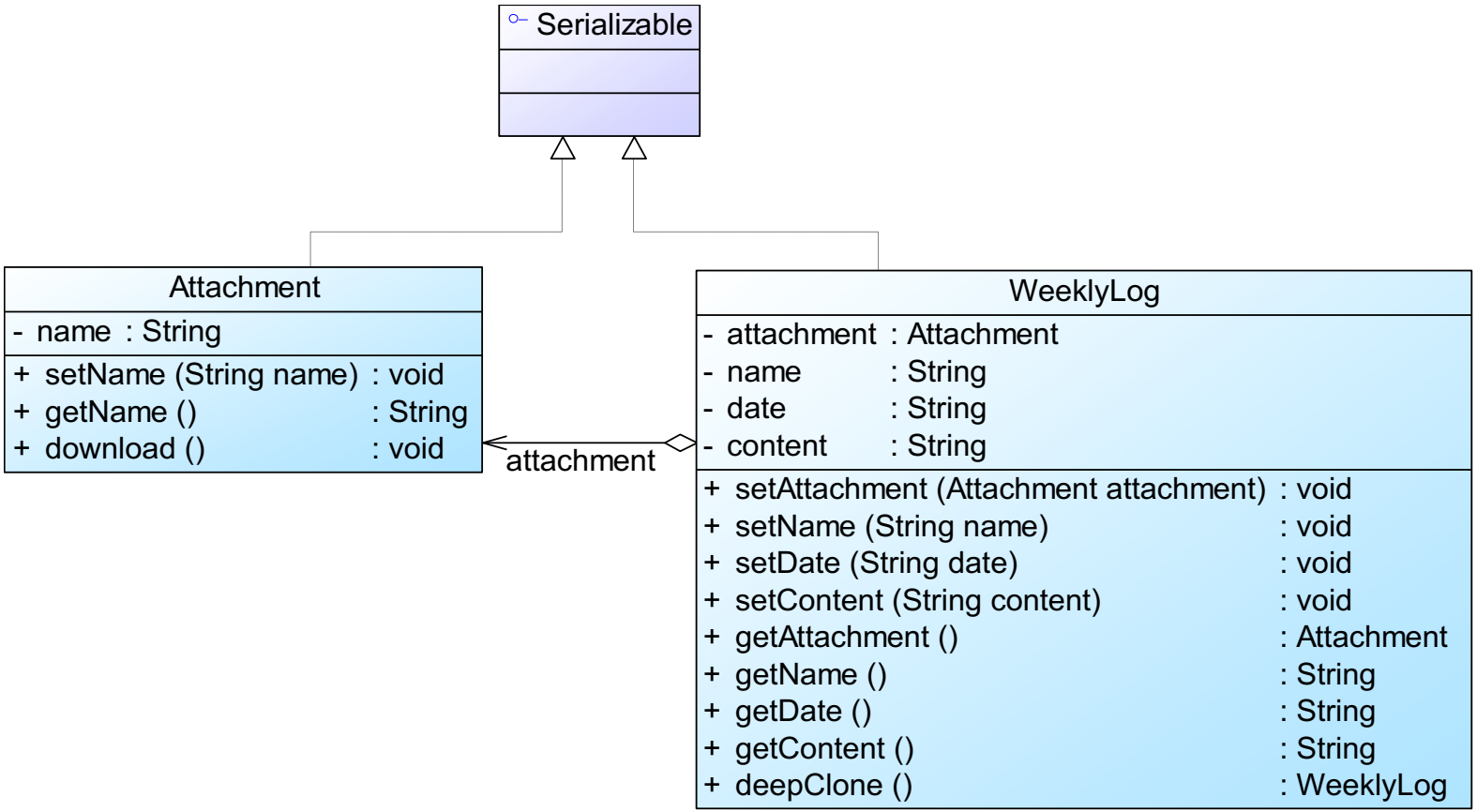
- 结果及分析

```
周报是否相同？ false
附件是否相同？ true
```

  - 工作周报对象被成功复制，但是附件对象并没有复制，实现了浅克隆

原型模式的应用实例

- 深克隆解决方案
  - 工作周报类WeeklyLog和附件类Attachment实现Serializable接口



原型模式的应用实例

- 深克隆解决方案
  - 修改工作周报类WeeklyLog的clone()方法
  - (1) WeeklyLog: 具体原型类
  - (2) Attachment: 具体原型类
  - (3) Client

演示……

Code (designpatterns.prototype.deepclone)

原型模式的应用实例

- 深克隆解决方案

```
周报是否相同？  false
附件是否相同？  false
```

  - 工作周报对象和附件对象都成功复制，实现了深克隆

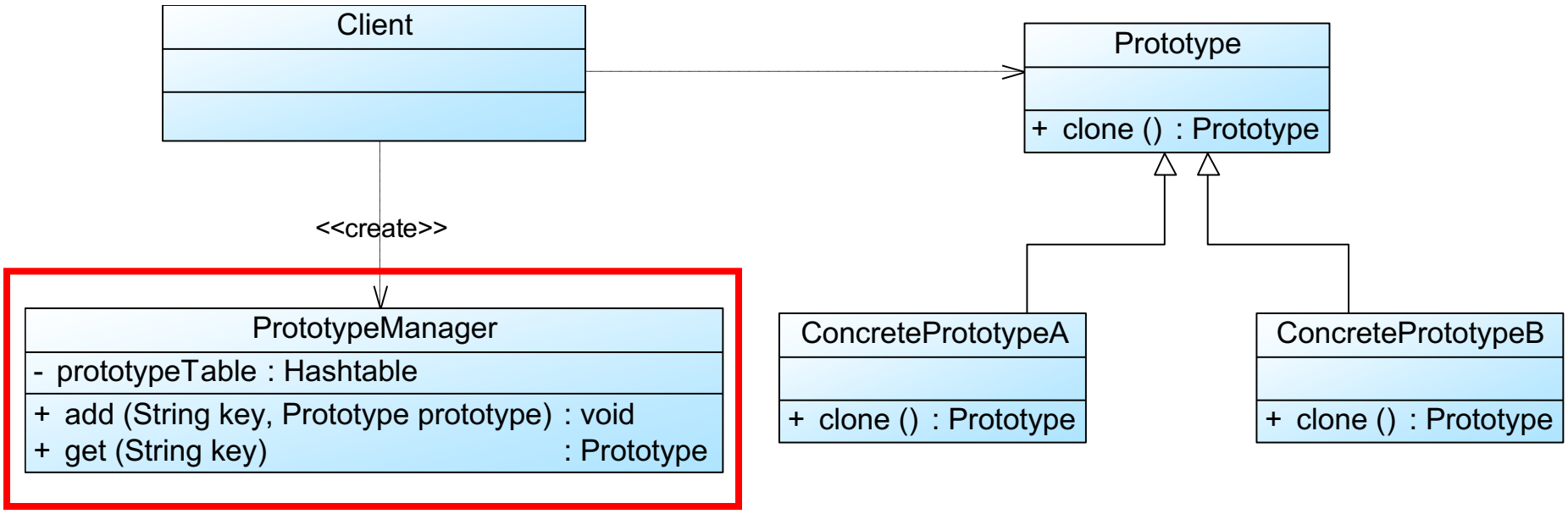
Shallow Clone
VS.
Deep Clone

原型管理器

- 定义
  - 原型管理器(Prototype Manager)将多个原型对象存储在一个集合中供客户端使用，它是一个专门负责克隆对象的工厂，其中定义了一个集合用于存储原型对象，如果需要某个原型对象的一个克隆，可以通过复制集合中对应的原型对象来获得

原型管理器

- 结构



带原型管理器的原型模式

原型管理器

```
import java.util.*;

public class PrototypeManager {
    private Hashtable prototypeTable=new Hashtable();  //使用Hashtable存储原型对象
    public PrototypeManager() {
        prototypeTable.put("A", new ConcretePrototypeA());
        prototypeTable.put("B", new ConcretePrototypeB());
    }

    public void add(String key, Prototype prototype) {
        prototypeTable.put(key,prototype);
    }

    public Prototype get(String key) {
        Prototype clone = null;
        clone = ((Prototype) prototypeTable.get(key)).clone(); //通过克隆方法创建新对象
        return clone;
    }
}
```

原型模式的优缺点与适用环境

- 模式优点
  - 简化对象的创建过程，通过复制一个已有实例可以提高新实例的创建效率
  - 扩展性较好
  - 提供了简化的创建结构，原型模式中产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品
  - 可以使用深克隆的方式保存对象的状态，以便在需要的时候使用，可辅助实现撤销操作

原型模式的优缺点与适用环境

- 模式缺点
  - 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了开闭原则
  - 在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重的嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来可能会比较麻烦

原型模式的优缺点与适用环境

- 模式适用环境
  - 创建新对象成本较大，新对象可以通过复制已有对象来获得，如果是相似对象，则可以对其成员变量稍作修改
  - 系统要保存对象的状态，而对象的状态变化很小
  - 需要避免使用分层次的工厂类来创建分层次的对象
  - Ctrl + C → Ctrl + V

作业

- 假设有一销售管理系统其中包括一个客户类Customer，在客户类中包含一个名为客户地址的成员变量，客户地址类型为Address，两个类型的原始定义如下：

```
Class Customer{
    String name;
    int age;
    bool gender;
    Address address;
    //getter, setter 函数省略
}
```

```
Class Address{
    String street1;
    String city;
    String province;
    String country;
    String postcode;
    //getter, setter 函数省略
}
```

- 请尝试使用Java中的Cloneable接口以及Serializable接口实现浅拷贝以及深拷贝的原型模式以对Customer进行拷贝。对实现的两种原型模式，客户端测试代码中比较拷贝前后Customer对象是否相同， 拷贝前后的Address对象是否相同。
- 另尝试不采用Java中的接口实现深拷贝以及浅拷贝的原型模式，客户端测试代码与上述要求相同。

作业

- Factory Method模式和Abstract Factory模式的区别在哪？一般哪些情况下适合用前者，哪些情况下适合用后者？

提交作业到教学立方（4月14号24点截止）