Java设计模式
菜鸟系列

前言

> 原文出处：Java设计模式菜鸟系列
> 作者：lhy_ycu

Java设计模式菜鸟系列

> 主要为Java设计模式的初学者提供帮助，采用uml建模与具体代码实现的方式使内容丰富详实、具体、通俗易懂。

(一)策略模式建模与实现

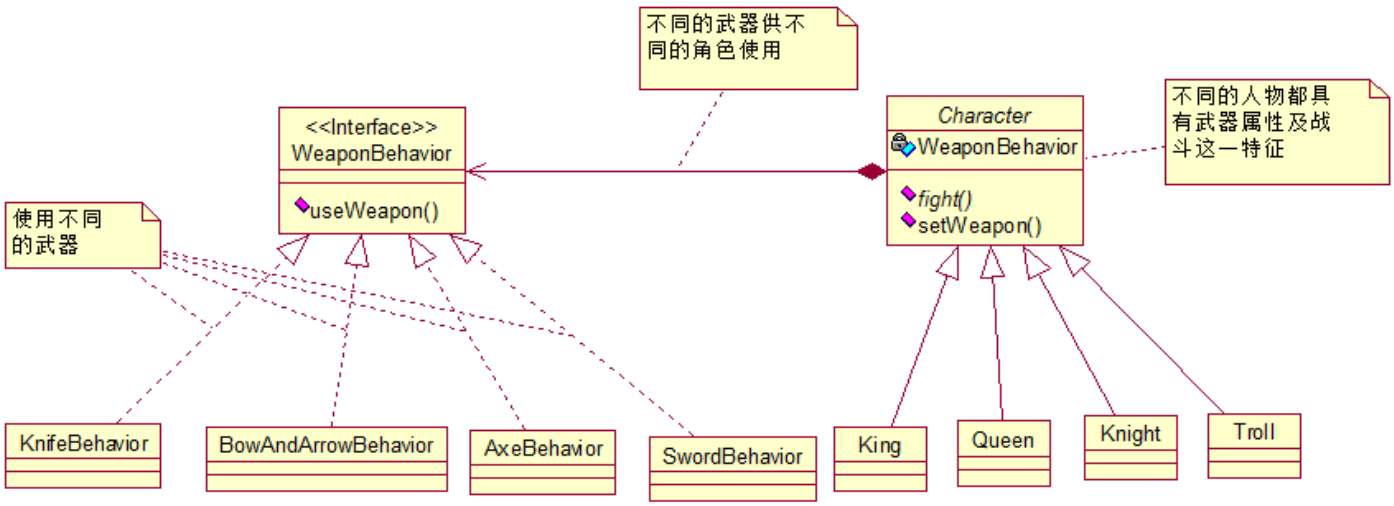
今天开始咱们来谈谈Java设计模式。这里会结合uml图形来讲解，有对uml建模不熟的可以参考我的另一篇博文uml建模。

首先，个人觉得模式的设计就是一个将变化的东西和不变(稳定)的东西分离的过程。咱们的应用中可能有很多需要改变的地方，而模式要做的就是把它们“抽取”出来并进行“封装”和“实现”，因此更多的时候咱们是面向接口编程。下面将结合《Head First 设计模式》这一书中的某些具体案例以及本人的理解进行讲解。如果大家透过看图能说出是哪种模式并能进行具体的代码实现及加以应用，反之也能做到，那么我想关于Java这些常见的设计模式你也就掌握的差不多了。

关于策略模式，网上有的说法是：策略模式让用户可以选择执行一个动作的方法，也就是说，用户可以选择不同的策略来进行操作。个人觉得策略模式可以用这个公式：不同的XXX 拥有不同的XXX供用户选择。比如说：不同的象棋棋子拥有不同的走法供用户选择。

下面根据游戏中的不同人物拥有不同武器供用户选择的简单案例：

一、UML模型图



二、代码实现

```
/**
 * 武器 --模板
 */
interface WeaponBehavior {
    void useWeapon();
}

class KnifeBehavior implements WeaponBehavior {
    @Override
    public void useWeapon() {
```

```
        System.out.println("实现用匕首刺杀...");
    }
}

class BowAndArrowBehavior implements WeaponBehavior {
    @Override
    public void useWeapon() {
        System.out.println("实现用弓箭设计...");
    }
}

class AxeBehavior implements WeaponBehavior {
    @Override
    public void useWeapon() {
        System.out.println("实现用斧头劈砍...");
    }
}

class SwordBehavior implements WeaponBehavior {
    @Override
    public void useWeapon() {
        System.out.println("实现用宝剑挥舞...");
    }
}

/**
 * 角色
 */
abstract class Character {
    // 将接口作为抽象角色的Field以便封装
    protected WeaponBehavior weaponBehavior;

    public void setWeapon(WeaponBehavior w) {
        weaponBehavior = w;
    }

    /**
     * 这里有点类似“代理模式”
     */
    public void performWeapon() {
        // do something...
        weaponBehavior.useWeapon();
        // do something...
    }

    public abstract void fight();
}

/**
 * 国王使用宝剑挥舞
 */
class King extends Character {

    public King() {
        weaponBehavior = new SwordBehavior();
```

```
    }

    @Override
    public void fight() {
        System.out.println("国王使用宝剑挥舞...");
    }

}

/**
 * 皇后使用匕首刺杀
 */
class Queen extends Character {

    public Queen() {
        weaponBehavior = new KnifeBehavior();
    }

    @Override
    public void fight() {
        System.out.println("皇后使用匕首刺杀...");
    }

}

/**
 * Knight和Troll以此类推，这里就不写了
 */

/**
 * 客户端测试
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Character king = new King();
        king.performWeapon();
        // 这里有点类似于“状态模式”
        king.setWeapon(new AxeBehavior());
        king.performWeapon();

        Character queen = new Queen();
        queen.performWeapon();
        queen.setWeapon(new BowAndArrowBehavior());
        queen.performWeapon();
    }
}
```

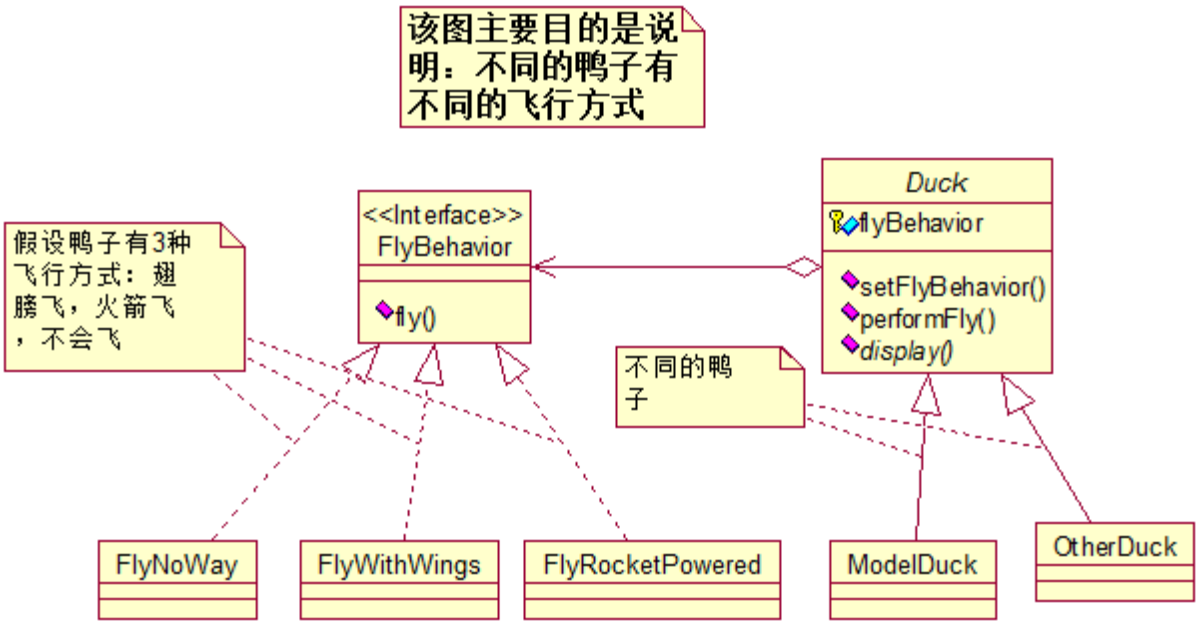
三、应用场景(仅代表个人观点)

游戏中的角色武器、棋类游戏中的棋子走法等。

四、小结

以上内容如果有不同的见解或疏忽的地方，还请大家提出宝贵的建议或意见。

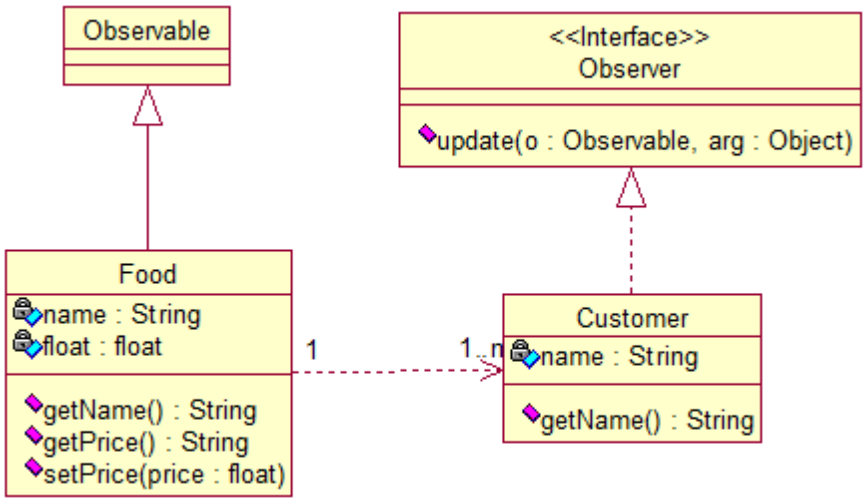
最后大家可以试着将下面的uml模型进行具体的代码实现，如图：附源码



(二)观察者模式建模与实现

观察者(Observer)模式定义：在对象之间定义了一对多的依赖关系，这样一来，当一个对象改变状态时，依赖它的对象都会收到通知并自动跟新。Java已经提供了对观察者Observer模式的默认实现， Java对观察者模式的支持主要体现在Observable类和Observer接口。先看uml模型图：

一、UML模型图



二、代码实现

```
/**示例：咱们去菜市场买菜
 *
 * 小商贩--主题
 */
class Food extends Observable {
    /**菜名 */
    private String name;
    /**菜价 */
    private float price;

    public Food(String name, float price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public float getPrice() {
        return price;
    }
```

```
    public void setPrice(float price) {
        this.price = price;
        /**
         * 设置菜价的状态已经被改变
         */
        this.setChanged();
        /**
         * 通知【所有】正在看菜(已经注册了)的顾客，然后回调Observer的update方法进行更新
         *
         * 这里可以体现对象的一对多：一个小商贩一旦更新价格(即一个对象改变状态)，便会自动通知所有的顾客(依赖它的对象都会收到通知)
         * 并自动update
         */
        this.notifyObservers(price);
    }

}

/**
 * 顾客 --观察者
 */
class Customer implements Observer {
    private String name;

    public Customer(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void update(Observable o, Object arg) {
        if (o instanceof Food && arg instanceof Float) {
            Food food = (Food) o;
            float price = (Float) arg;
            System.out.println("您好：" + this.name + " ," + food.getName()
                    + "的价格已经发生改变，现在的价格为：" + price + "元/斤");
        }
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Food food = new Food("土豆", 1.0f);
        Customer zhangsan = new Customer("张三");
        Customer lisi = new Customer("李四");
        /**
```

```
        /
         * 添加顾客
         */
        food.addObserver(zhangsan);
        food.addObserver(lisi);
        /**
         * 更新价格
         */
        food.setPrice(1.5f);
    }
}
```

三、应用场景

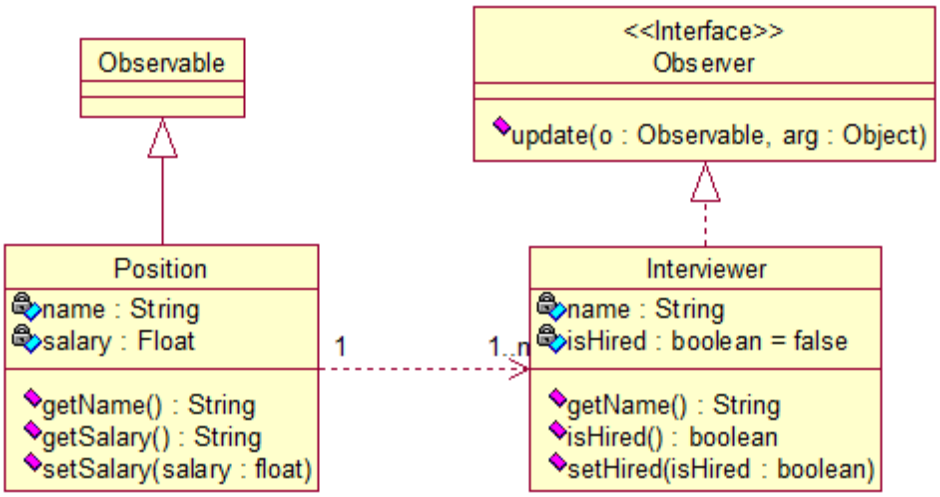
GUI框架、气象观察等

四、小结

在通知所有观察者之前一定要调用 setChanged()方法来设置被观察者的状态已经被改变，这样notifyObservers()才会回调Observer的update方法进行更新。

以上内容如果有不同的见解或疏忽的地方，还请大家提出宝贵的建议或意见。

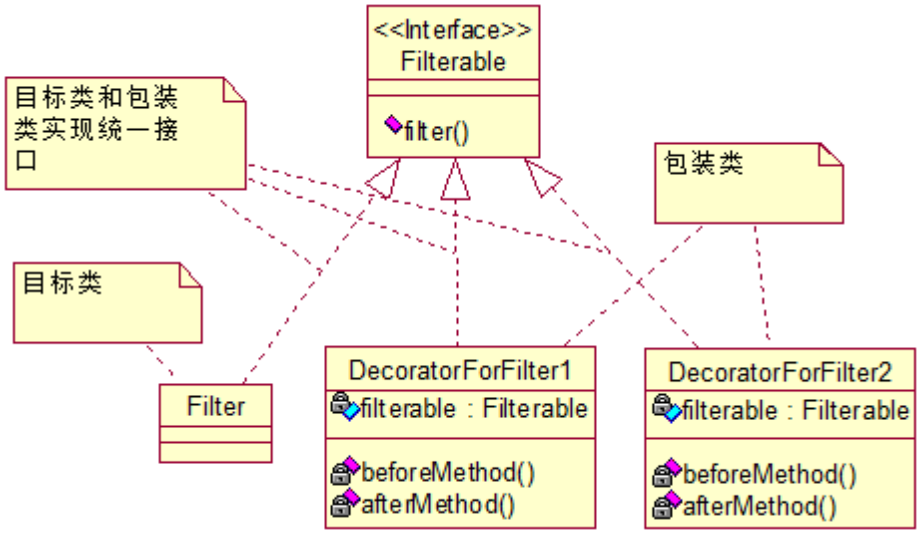
最后大家可以试着将下面的uml模型进行具体的代码实现，如图：附源码



(三)装饰者模式建模与实现

装饰者(Decorator)模式：动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更具有弹性的替代方案。对于装饰者模式，它其实是一种包装，所以我更愿意称它为一种包装。像咱们以前经常使用的Java里面的IO流就用到了装饰者模式。比如：BufferedReader br = new BufferedReader(new InputStreamReader(new FileInputStream(file1)));这里file1为目标对象，而像BufferedReader、InputStreamReader就可以称之为包装类。下面举例说明：

一、UML模型图



二、代码实现

```
/**
 * 对修改关闭，对扩展开放。
 *
 * 统一接口
 */
interface Filterable {
    public void filter();
}

/**
 * 目标类
 */
class Filter implements Filterable {

    @Override
    public void filter() {
        System.out.println("目标类的核心过滤方法...");
    }
}
```

```
/**
 * DecoratorForFilter1包装类与目标类实现相同的接口 --> 织入Log
 */
class DecoratorForFilter1 implements Filterable {
    private Filterable filterable;

    public DecoratorForFilter1(Filterable filterable) {
        this.filterable = filterable;
    }

    private void beforeMethod() {
        System.out.println("DecoratorForFilter1 --> 核心过滤方法执行前执行");
    }

    private void afterMethod() {
        System.out.println("DecoratorForFilter1 --> 核心过滤方法执行后执行");
    }

    @Override
    public void filter() {
        beforeMethod();
        filterable.filter();
        afterMethod();
    }
}

/**
 * DecoratorForFilter2包装类与目标类实现相同的接口 --> 织入Log
 */
class DecoratorForFilter2 implements Filterable {
    private Filterable filterable;

    public DecoratorForFilter2(Filterable filterable) {
        this.filterable = filterable;
    }

    private void beforeMethod() {
        System.out.println("DecoratorForFilter2 --> 核心过滤方法执行前执行");
    }

    private void afterMethod() {
        System.out.println("DecoratorForFilter2 --> 核心过滤方法执行后执行");
    }

    @Override
    public void filter() {
        beforeMethod();
        filterable.filter();
        afterMethod();
    }
}

/**
 * 客户端测试类
```

```
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 目标对象
         */
        Filterable targetObj = new Filter();
        /**
         * 包装对象(对目标对象进行包装)
         */
        Filterable decorObj = new DecoratorForFilter1(new DecoratorForFilter2(
                targetObj));
        /**
         * 执行包装后的业务方法
         */
        decorObj.filter();
    }
}
```

输出：

DecoratorForFilter1 --> 核心过滤方法执行前执行
DecoratorForFilter2 --> 核心过滤方法执行前执行
目标类的核心过滤方法...
DecoratorForFilter2 --> 核心过滤方法执行后执行
DecoratorForFilter1 --> 核心过滤方法执行后执行

三、应用场景(仅代表个人观点)

I/O、过滤器

四、小结

通过输入的Log我们可以看到：输出的过程其实是将包装类“拆包”的过程，就像包装袋一样一层一层的拆开。

设计原则：1)多用组合，少用继承。2)对扩展开放，对修改关闭。

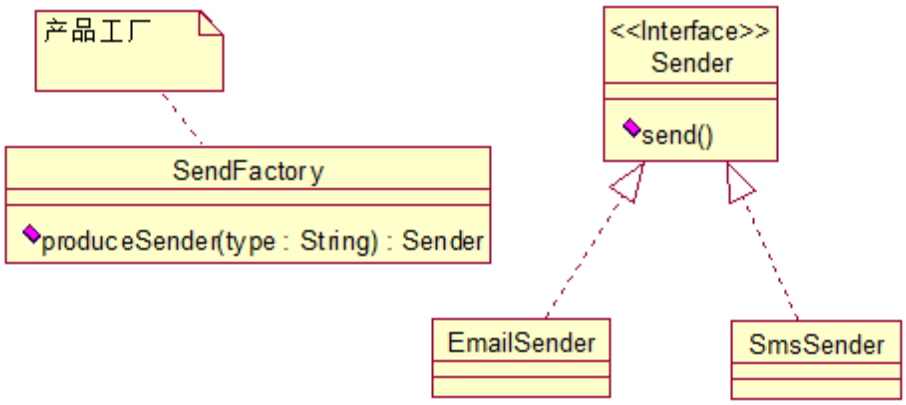
(四)工厂方法模式建模与实现

工厂方法模式（Factory Method）

工厂方法：顾名思义，就是调用工厂里的方法来生产对象(产品)的。
工厂方法实现方式有3种：

一、普通工厂模式。就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

1、uml建模图：



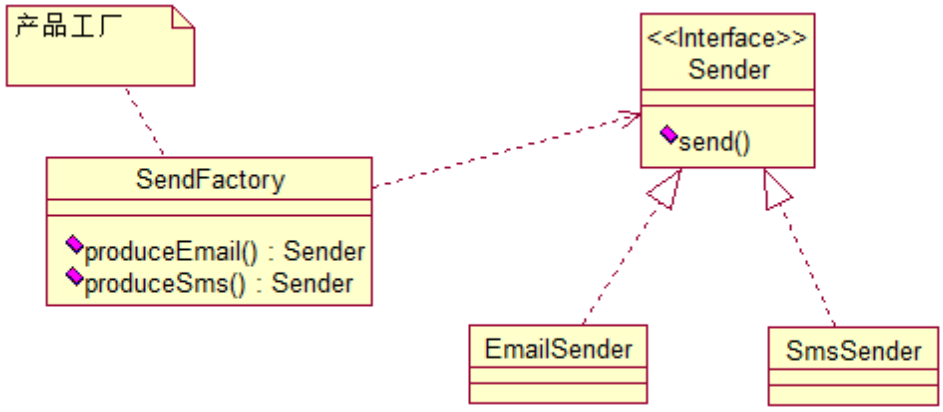
2、代码实现

```
/**
 * 示例(一)：普通工厂方法
 *
 * 缺点：如果传递的字符串出错，则不能正确创建对象
 */
interface Sender {
    public void send();
}

class EmailSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用电子邮箱发送...");
    }

}

class SmsSender implements Sender {

    @Override
```

```
    @Override
    public void send() {
        System.out.println("使用短信发送...");
    }

}

/**
 * 产品工厂
 */
class SendFactory {
    public Sender produceSender(String type) {
        if ("email".equals(type)) {
            return new EmailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("没有这种类型...");
            return null;
        }

    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        // 创建工厂
        SendFactory sendFactory = new SendFactory();
        // 生产产品
        Sender sender = sendFactory.produceSender("email");
        // 发货
        sender.send();
    }

}
```

二、多个工厂方法模式。是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

1、uml建模图：



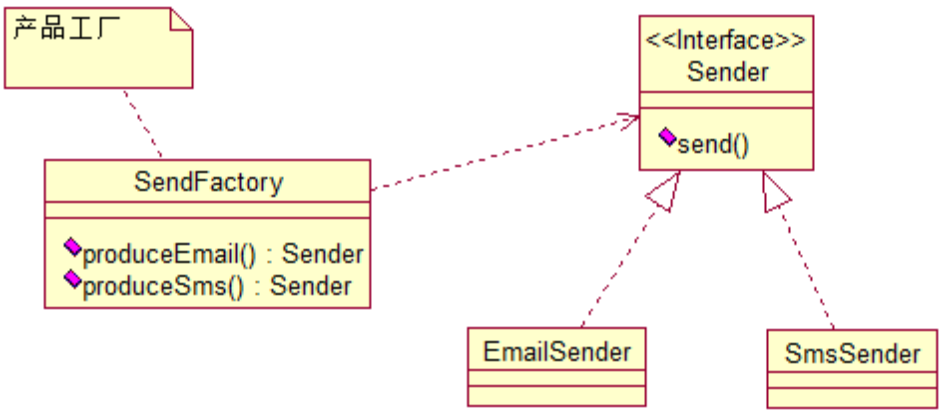
2、代码实现

```
/**
 * 示例(二)：多个工厂方法
 *
 * 优点：多个工厂方法模式是提供多个工厂方法，分别创建对象
 */
interface Sender {
    public void send();
}

class EmailSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用电子邮箱发送...");
    }

}

class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用短信发送...");
    }

}

/**
 * 不同方法分别生产相应的产品
 */
class SendFactory {
    public Sender produceEmail() {
        return new EmailSender();
    }

    public Sender produceSms() {
```

```
        return new SmsSender();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        // 创建工厂
        SendFactory sendFactory = new SendFactory();
        // 生产产品
        Sender senderEmail = sendFactory.produceEmail();
        // 发货
        senderEmail.send();
    }

}
```

三、静态工厂方法模式。将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

1、uml建模图：



2、代码实现

```
/**
 * 示例(三)：静态工厂方法
 *
 * 优点：多个工厂方法模式是提供多个工厂方法，分别创建对象
 */
interface Sender {
    public void send();
}
```

```
class EmailSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用电子邮箱发送...");
    }

}

class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用短信发送...");
    }

}

/**
 * 静态工厂：不同实例化工厂
 *
 * 不同方法分别生产相应的产品
 */
class SendFactory {
    public static Sender produceEmail() {
        return new EmailSender();
    }

    public static Sender produceSms() {
        return new SmsSender();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        // 直接生产产品
        Sender senderEmail = SendFactory.produceEmail();
        // 发货
        senderEmail.send();
    }

}
```

四、总结

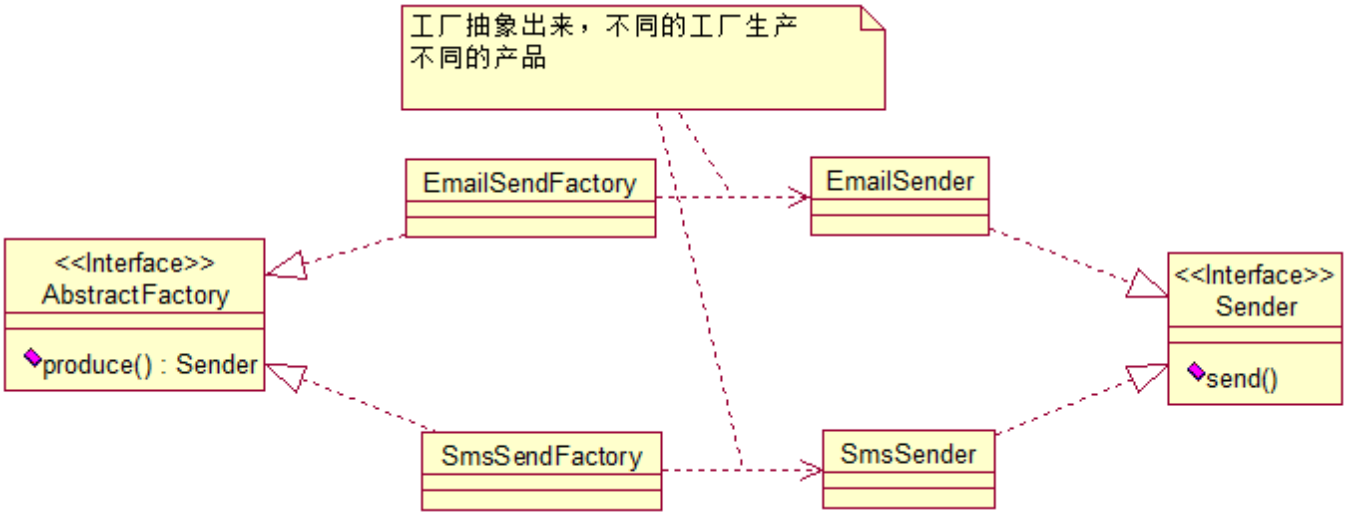
总体来说，凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要

实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式。

(五)抽象工厂模式建模与实现

抽象工厂模式（Abstract Factory）：抽象工厂--顾名思义，就是把工厂抽象出来，不同的工厂生产不同的产品。这样做有个好处：一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码。

一、uml建模图：



二、代码实现

```
/**
 * 示例：抽象工厂--顾名思义，就是把工厂抽象出来，不同的工厂生产不同的产品
 *
 * 优点：一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码
 */
interface Sender {
    public void send();
}

class EmailSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is a email...");
    }
}

class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("this is a sms...");
    }
```

```
}

/**
 * 角色：抽象工厂
 */
interface AbstractFactory {
    public Sender produce();
}

/**
 * 邮件工厂
 */
class EmailSendFactory implements AbstractFactory {

    @Override
    public Sender produce() {
        return new EmailSender();
    }
}

/**
 * 短信工厂
 */
class SmsSendFactory implements AbstractFactory {

    @Override
    public Sender produce() {
        return new SmsSender();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建工厂
         */
        AbstractFactory factory = new SmsSendFactory();
        /**
         * 生产产品
         */
        Sender sender = factory.produce();
        /**
         * 执行业务逻辑
         */
        sender.send();
    }

}
```

三、总结

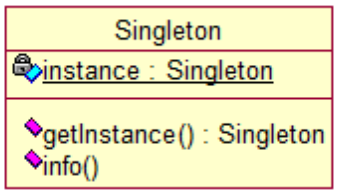
如果你现在想增加一个功能：给特别关注的好友发信息，则只需做一个实现类，实现Sender接口，同时做一个工厂类，实现AbstractFactory接口，就可以了，无需修改现有代码。这样做拓展性较好！

(六)单例模式建模与实现

单例模式（Singleton）：是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。好处主要有：1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。

单例模式主要有两种实现方式：1、懒汉式；2、饿汉式

一、uml建模



二、代码实现：

1、懒汉式

```
/**
 * 示例：单例--单例对象能保证在一个JVM中，该对象只有一个实例存在。
 *
 * 缺点：这种做法在多线程环境下，不安全
 *
 * 懒汉式
 */

class Singleton {
    /**
     * 持有私有静态变量(也称类变量)，防止被引用
     *
     * 此处赋值为null，目的是实现延迟加载 (因为有些类比较庞大，所以延迟加载有助于提升性能)
     */
    private static Singleton instance = null;

    /**私有构造方法，防止被实例化 */
    private Singleton() {

    }

    /**静态工厂方法，创建实例 --只不过这里是创建自己，而且只能创建一个 */
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
```

```
        return instance;
    }

    public void info() {
        System.out.println("this is a test method...");
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        /**
         * 调用普通方法
         */
        s1.info();
        Singleton s2 = Singleton.getInstance();
        /**
         * 运行结果为true，说明s1、s2这两个类变量都指向内存中的同一个对象
         */
        System.out.println(s1 == s2);
    }
}
```

2、饿汉式

```
/**
 * 饿汉式
 */

class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {

    }

    public static Singleton getInstance() {
        return instance;
    }

    public void info() {
        System.out.println("this is a test method...");
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        /**
         * 调用普通方法
         */
        s1.info();
        Singleton s2 = Singleton.getInstance();
        /**
         * 运行结果为true，说明s1、s2这两个类变量都指向内存中的同一个对象
         */
        System.out.println(s1 == s2);
    }

}
```

3、如果考虑多线程，那么getInstance()方法要加同步synchronized，这时饿汉式比懒汉式要好，尽管资源利用率要差，但是不用同步。

```
/**
 *
 * 考虑多线程的时候，下面这种做法可以参考一下：--懒汉式
 *
 * 在创建类的时候进行同步，所以只要将创建和getInstance()分开，单独为创建加synchronized关键字
 *
 * 这种做法考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。
 *
 * @author Leo
 */
public class SingletonTest {

    private static SingletonTest instance = null;

    private SingletonTest() {
    }

    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }
    }

    public static SingletonTest getInstance() {
        if (instance == null) {
            syncInit();
        }
        return instance;
    }
}
```

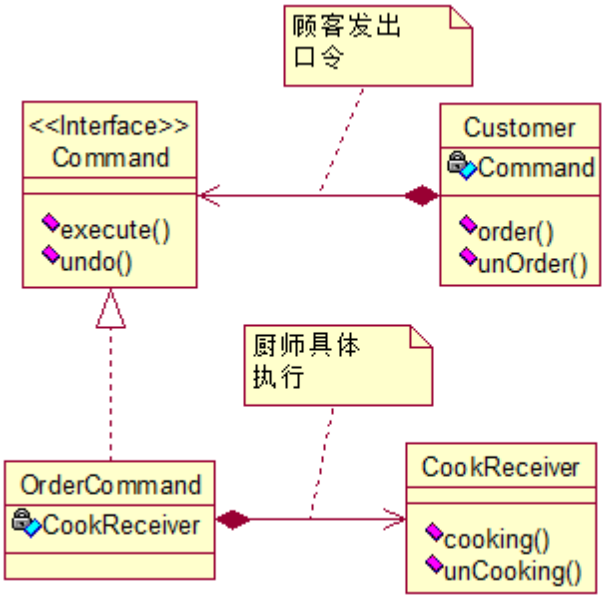
三、总结

单例模式保证了一个类只有一个实例，且提供一个访问全局点的方式，更加灵活的保证了实例的创建和访问约束。系统中只有一个实例，因此构造方法应该为私有饿汉式：类加载时直接创建静态实例；懒汉式：第一次需要时才创建一个实例，那么getInstance方法要加同步 饿汉式比懒汉式要好，尽管资源利用率要差，但是不用同步。

(七)命令模式建模与实现

命令模式(Command)：将“请求”(命令/口令)封装成一个对象，以便使用不同的请求、队列或者日志来参数化其对象。命令模式也支持撤销操作。命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开。

一、uml建模



二、代码实现：

```
/**
 * 示例：以咱去餐馆吃饭为例，分为3步
 *
 * 1、和小二说，来个宫保鸡丁 --> 顾客发出口令
 *
 * 2、小二来了一句：宫保鸡丁一份。 这时命令被传递到了厨师。--> 口令传递到了厨师
 *
 * 3、然后厨师就开始做宫保鸡丁去了。 --> 厨师根据口令去执行
 *
 * 从这3步可以看到，宫保鸡丁并不是我想吃就我来做，而是传达给别人去做。
 *
 * 我要的是一个结果——宫保鸡丁这道菜做好了，而我无需去关系这道菜是怎么去做的。
 */
interface Command {
    /**
     * 口令执行
     */
    public void execute();

    /**
     * 口令撤销
```

```
     */
    public void undo();
}

/**
 * 口令 -- 经小二传递
 */
class OrderCommand implements Command {
    private CookReceiver cook;

    public OrderCommand(CookReceiver cook) {
        this.cook = cook;
    }

    @Override
    public void execute() {
        cook.cooking();
    }

    @Override
    public void undo() {
        cook.unCooking();
    }
}

/**
 * 厨师--真正的口令执行者
 */
class CookReceiver {
    public void cooking() {
        System.out.println("开始炒宫保鸡丁了...");
    }

    public void unCooking() {
        System.out.println("不要炒宫保鸡丁了...");
    }
}

/**
 * 顾客--真正的口令发出者
 */
class Customer {
    private Command command;

    public Customer(Command command) {
        this.command = command;
    }

    /**
     * 将命令的发出与执行分开
     */
    public void order() {
        command.execute();
    }
```

```
    public void unOrder() {
        command.undo();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 等待口令的执行者 --炒菜总得有个厨师吧.
         */
        CookReceiver receiver = new CookReceiver();
        /**
         * 等待将口令传达给厨师 --因为顾客要什么菜还不知道，但口令始终要传达到厨师耳朵里这是肯定的。
         */
        Command cmd = new OrderCommand(receiver);
        Customer customer = new Customer(cmd);
        /**
         * 执行口令
         */
        customer.order();
        /**
         * 撤销口令
         */
        customer.unOrder();
    }
}
```

三、应用场景

菜馆点餐、遥控器、队列请求、日志请求。

四：小结

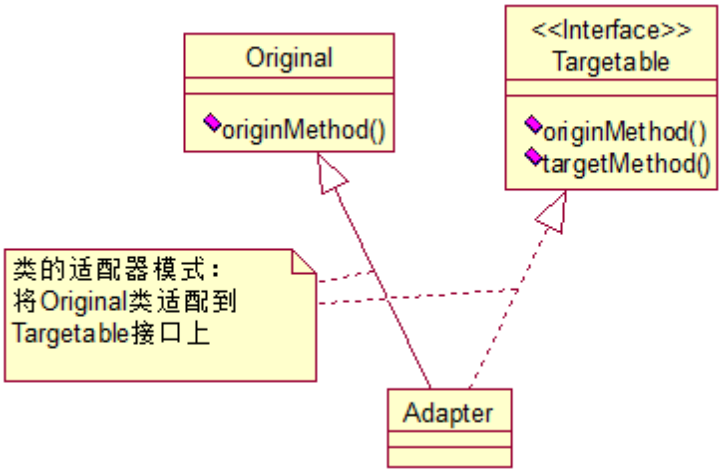
从上面的示例可以看到：命令模式将“动作的请求者”从“动作的执行者”对象中解耦出来，这就是将方法的调用封装起来的好处。

(八)适配器模式建模与实现

适配器模式(Adapter)：将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。
主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。

一、类的适配器模式

1、uml建模：



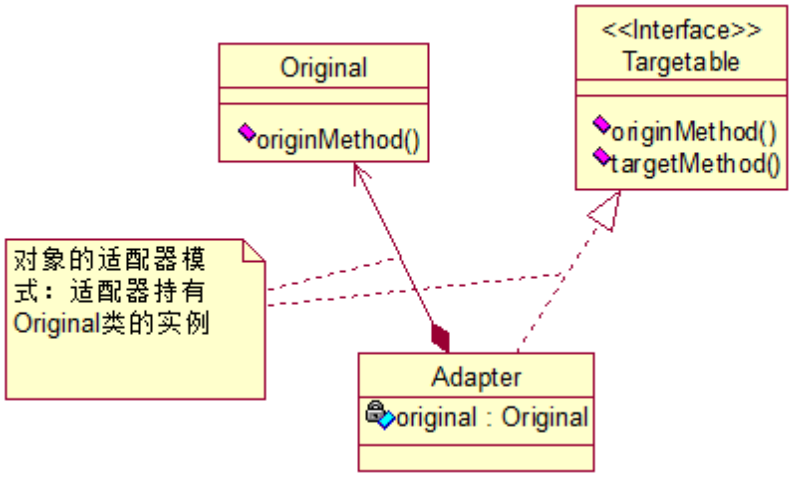
2、代码实现

```
/**
 * 示例(一)：类的适配器模式
 *
 * 原类拥有一个待适配的方法originMethod
 */
class Original {
    public void originMethod() {
        System.out.println("this is a original method...");
    }
}

interface Targetable {
    /**
     * 与原类的方法相同
     */
    public void originMethod();

    /**
     * 目标类的方法
     */
    public void targetMethod();
```

```
    public void targetMethod();
}

/**
 * 该Adapter类的目的：将Original类适配到Targetable接口上
 */
class Adapter extends Original implements Targetable {
    /**
     * 可以看到该类只需要实现targetMethod即可。
     *
     * 因为Targetable接口里的originMethod方法已经由Original实现了。
     *
     * 这就是Adapter适配器这个类的好处：方法实现的转移(或称嫁接) --> 将Adapter的责任转移到Original身
上
     *
     * 这样就实现了类适配器模式 --> 将Original类适配到Targetable接口上
     *
     * 如果Original又添加了一个新的方法originMethod2，那么只需在Targetable接口中声明即可。
     */
    @Override
    public void targetMethod() {
        System.out.println("this is a target method...");
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Targetable target = new Adapter();
        target.originMethod();
        target.targetMethod();
    }
}
```

二、对象的适配器模式

1、uml建模：



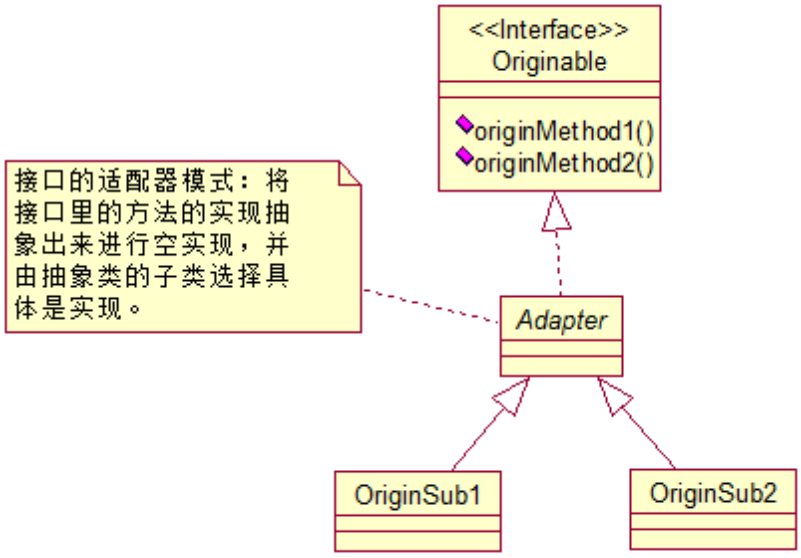
2、代码实现：

```
/**
 * 示例(二)：对象的适配器模式
 *
 * 原类拥有一个待适配的方法originMethod
 */
class Original {
    public void originMethod() {
        System.out.println("this is a original method...");
    }
}

interface Targetable {
    /**
     * 与原类的方法相同
     */
    public void originMethod();

    /**
     * 目标类的方法
     */
    public void targetMethod();
}

/**
 * 持有Original类的实例
 */
class Adapter implements Targetable {
    private Original original;

    public Adapter(Original original) {
        this.original = original;
    }

    @Override
    public void targetMethod() {
        System.out.println("this is a target method...");
```

```
        System.out.println("this is a target method...");
    }

    @Override
    public void originMethod() {
        original.originMethod();
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Original original = new Original();
        Targetable target = new Adapter(original);
        target.originMethod();
        target.targetMethod();
    }
}
```

三、接口的适配器模式

1、uml建模：



2、代码实现

```
/**
 * 示例(三)：接口的适配器模式
 *
 * 这次咱们直接将原类做成一个接口 --> 原始接口
 */
```

```
interface Originable {
    public void originMethod1();

    public void originMethod2();
}

/**
 * 该抽象类实现了原始接口，实现了所有的方法。
 *
 * 空实现即可，具体实现靠子类，子类只需实现自身需要的方法即可。
 *
 * 以后咱们就不用跟原始的接口打交道，只和该抽象类取得联系即可。
 */
abstract class Adapter implements Originable {
    public void originMethod1() {

    }

    public void originMethod2() {

    }
}

/**
 * 子类只需选择你所需要的方法进行实现即可
 */
class OriginSub1 extends Adapter {
    @Override
    public void originMethod1() {
        System.out.println("this is Originable interface's first sub1...");
    }

    /**
     * 此时：originMethod2方法默认空实现
     */
}

class OriginSub2 extends Adapter {
    /**
     * 此时：originMethod1方法默认空实现
     */

    @Override
    public void originMethod2() {
        System.out.println("this is Originable interface's second sub2...");
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
```

```
    public static void main(String[] args) {
        Originable origin1 = new OriginSub1();
        Originable origin2 = new OriginSub2();
        origin1.originMethod1();
        origin1.originMethod2();
        origin2.originMethod1();
        origin2.originMethod2();
    }
}
```

四、总结

1、类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

2、对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Adapter类，持有原类的一个实例，在Adapter类的方法中，调用实例的方法就行。

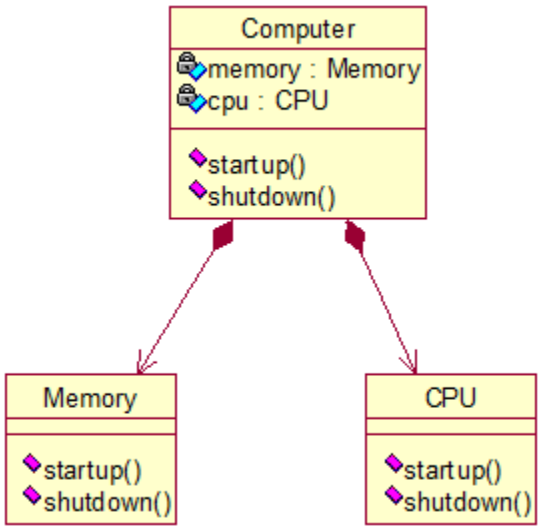
3、接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Adapter实现所有方法，我们写别的类的时候，继承抽象类即可。

(九)外观模式建模与实现

外观模式(Facade)：是为了解决类与类之间的依赖关系的，像spring一样，可以将类与类之间的关系配置到配置文件中，而外观模式就是将他们的

关系放在一个Facade类中，降低了类与类之间的耦合度，该模式中没有涉及到接口。

一、uml建模：



二、代码实现：

```
/**
 * 示例：外观模式，也称门面模式
 *
 * 优点：为了解决类与类之间的依赖关系，降低了类与类之间的耦合度
 *
 * 该模式中没有涉及到接口
 */

class Memory {
    public void startup() {
        System.out.println("this is memory startup...");
    }

    public void shutdown() {
        System.out.println("this is memory shutdown...");
    }

}

class CPU {
    public void startup() {
```

```
        System.out.println("this is CPU startup...");
    }

    public void shutdown() {
        System.out.println("this is CPU shutdown...");
    }
}

/**
 * 作为facade，持有Memory、CPU的实例
 *
 * 任务让Computer帮咱们处理，我们无需直接和Memory、CPU打交道
 *
 * 这里有点像去商店里买东西：咱们买东西只需要到商店去买，而无需去生产厂家那里买。
 *
 * 商店就可以称为是一个facade外观(门面)模式。--> 商品都在商店里
 */
class Computer {
    private Memory memory;
    private CPU cpu;

    public Computer() {
        memory = new Memory();
        cpu = new CPU();
    }

    public void startup() {
        System.out.println("begin to start the computer...");
        memory.startup();
        cpu.startup();
        System.out.println("computer start finished...");
    }

    public void shutdown() {
        System.out.println("begin to close the computer...");
        memory.shutdown();
        cpu.shutdown();
        System.out.println("computer close finished...");
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.startup();
        System.out.println("\n");
        computer.shutdown();
    }
}
```

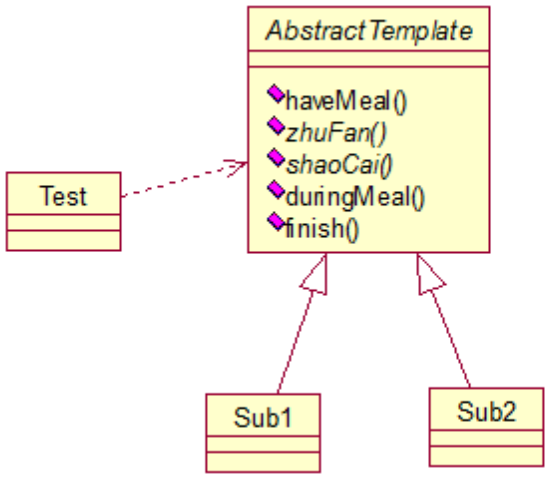
三、总结

如果我们没有Computer类，那么，CPU、Memory他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是咱们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这就是外观Facade模式。

(十)模板方法模式建模与实现

模板方法模式(Template Method)：在一个方法中定义了一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以再不改变算法结构的情况下，重新定义算法中的某些步骤。简而言之：模板方法定义了一个算法的步骤，并允许子类为一个或多个步骤提供实现。

一、uml建模：



二、代码实现：

```
/**
 * 示例：模板方法定义了一个算法的步骤，并允许子类为一个或多个步骤提供实现。
 *
 * 以吃饭为例：有几个步骤 --> 煮饭+烧菜+吃饭中+吃完了
 */
abstract class AbstractTemplate {

    public final void haveMeal() {
        zhuFan();
        shaoCai();
        duringMeal();
        finish();
    }

    public abstract void zhuFan();

    public abstract void shaoCai();

    public void duringMeal() {
        System.out.println("吃饭中...");
    }

    public void finish() {
```

```
        System.out.println("吃完了...");
    }
}

class Sub1 extends AbstractTemplate {

    @Override
    public void zhuFan() {
        System.out.println("使用电饭煲煮饭...");

    }

    @Override
    public void shaoCai() {
        System.out.println("使用炉灶烧菜...");
    }
}

class Sub2 extends AbstractTemplate {

    @Override
    public void zhuFan() {
        System.out.println("使用高压锅煮饭...");

    }

    @Override
    public void shaoCai() {
        System.out.println("使用电磁炉烧菜...");
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        AbstractTemplate at1 = new Sub1();
        at1.haveMeal();
        System.out.println("\n");
        AbstractTemplate at2 = new Sub2();
        at2.haveMeal();
    }
}
```

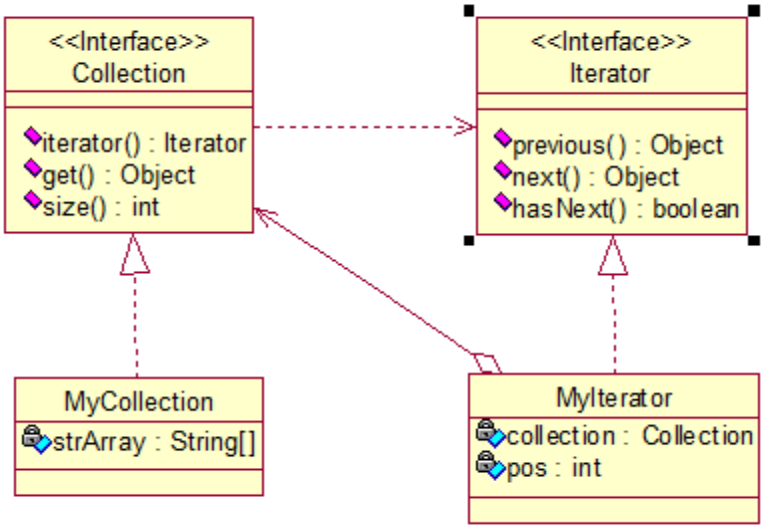
三、总结

模板方法模式：一个抽象类中，有一个主方法，再定义1...n个方法，可以抽象，可以不抽象，定义子类继承该抽象类，重写抽象方法，通过调用抽象类，实现对子类的调用。

(十一)迭代器模式建模与实现

迭代器模式(Iterator)：提供了一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

一、uml建模：



二、代码实现

```
/**
 * 示例：迭代器模式
 *
 */

interface Iterator {
    /**前移 */
    public Object previous();

    /**后移 */
    public Object next();

    /**判断是否有下一个元素 */
    public boolean hasNext();
}

interface Collection {
    public Iterator iterator();

    /**取得集合中的某个元素 */
    public Object get(int i);

    /**取得集合大小 */
    public int size();
```

```
}

/**
 * 集合
 */
class MyCollection implements Collection {
    private String[] strArray = { "aa", "bb", "cc", "dd" };

    @Override
    public Iterator iterator() {
        return new MyIterator(this);
    }

    @Override
    public Object get(int i) {
        return strArray[i];
    }

    @Override
    public int size() {
        return strArray.length;
    }

}

/**
 * 迭代器
 */
class MyIterator implements Iterator {
    private Collection collection;
    private int pos = -1;

    public MyIterator(Collection collection) {
        this.collection = collection;
    }

    @Override
    public Object previous() {
        if (pos > 0) {
            pos--;
        }
        return collection.get(pos);
    }

    @Override
    public Object next() {
        if (pos < collection.size() - 1) {
            pos++;
        }
        return collection.get(pos);
    }

    @Override
    public boolean hasNext() {
        if (pos < collection.size() - 1) {
```

```
        if (pos < collection.size() - 1) {
            return true;
        }
        return false;
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 实例化容器
         */
        Collection collection = new MyCollection();
        /**
         * 创建迭代器
         */
        Iterator iterator = collection.iterator();
        /**
         * 遍历集合中的元素
         */
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

三、应用场景

遍历、访问集合中的某个元素等

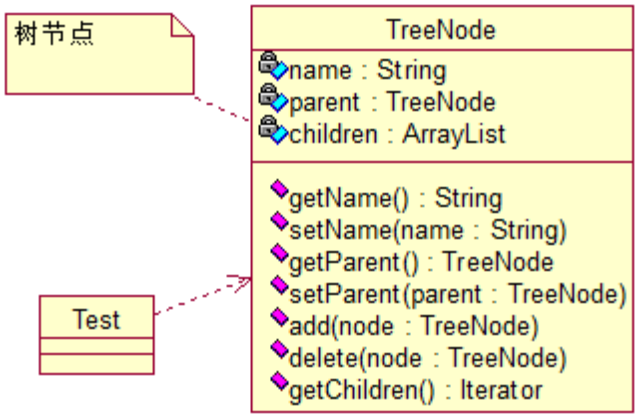
四、总结

迭代器模式就是顺序访问集合中的对象，这句话包含两层意思：一是需要遍历的对象，即集合对象，二是迭代器对象，用于对集合对象进行遍历访问。

(十二)组合模式建模与实现

组合模式(Composite)：组合模式有时又叫部分-整体模式，将对象组合成树形结构来表示“部分-整体”层次结构。组合模式在处理树形结构的问题时比较方便。

一、uml建模：

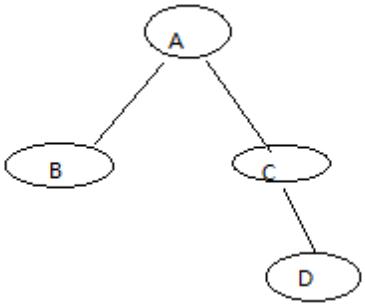


二、代码实现

```
/**
 * 示例：组合模式有时也称“整合-部分”模式
 *
 * 组合模式在处理树形结构的问题时比较方便
 *
 * 节点
 */
class TreeNode {
    /**节点名称 */
    private String name;
    private TreeNode parent;
    private ArrayList<TreeNode> children = new ArrayList<TreeNode>();

    public TreeNode(String name) {
        this.name = name;
    }

    /**
     * 对相关属性进行封装
     */
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    }

    public TreeNode getParent() {
        return parent;
    }

    public void setParent(TreeNode parent) {
        this.parent = parent;
    }

    /**
     * 对孩子节点的增删查操作
     */
    public void add(TreeNode node) {
        children.add(node);
    }

    public void delete(TreeNode node) {
        children.add(node);
    }

    public Iterator<TreeNode> getChildren() {
        return children.iterator();
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        TreeNode rootNode = new TreeNode("A");
        TreeNode bNode = new TreeNode("B");
        TreeNode cNode = new TreeNode("C");
        TreeNode dNode = new TreeNode("D");
        rootNode.add(bNode);
        rootNode.add(cNode);
        cNode.add(dNode);
        Iterator<TreeNode> iterator = rootNode.getChildren();
        while (iterator.hasNext()) {
            System.out.println(iterator.next().getName());
        }
    }
}
```

说明，这里构造了这样一棵树：



三、应用场景

将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树等。

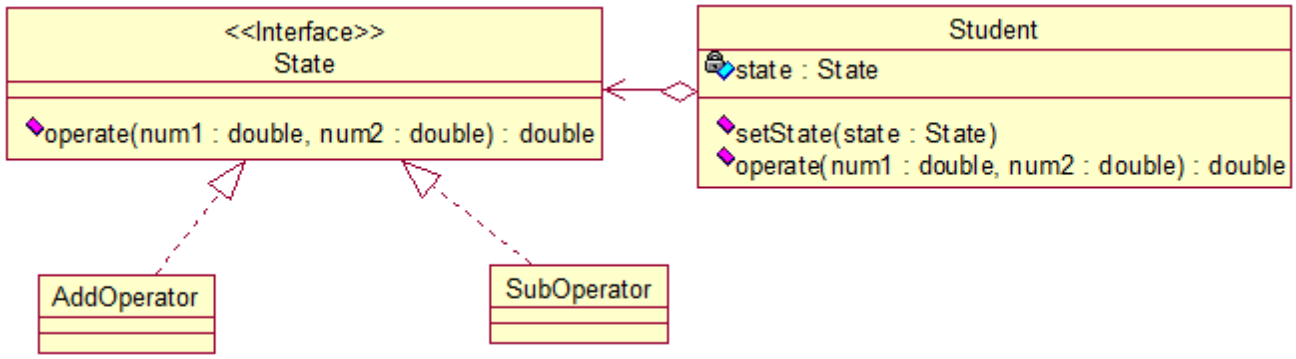
四、总结

组合能让客户以一致的方式处理个别对象以及对象组合。

(十三)状态模式建模与实现

状态模式(State)：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。状态模式说白了就是一个对象有不同的状态，不同的状态对应不同的行为，它其实是对switch case这样的语句的拓展。

一、uml建模：



二、代码实现

```
/**
 * 示例：状态模式-- 一个对像有不同的状 态，不同的状态对应不同的行为
 *
 * 下面四则运算为例
 */

interface State {
    public double operate(double num1, double num2);
}

/**
 * 加法
 */
class AddOperator implements State {

    @Override
    public double operate(double num1, double num2) {
        return num1 + num2;
    }
}

/**
 * 减法
 */
class SubOperator implements State {

    @Override
```

```
    public double operate(double num1, double num2) {
        return num1 - num2;
    }
}

/**
 * 学生
 */
class Student {
    private State state;

    public Student(State state) {
        this.state = state;
    }

    /**
     * 设置状态
     */
    public void setState(State state) {
        this.state = state;
    }

    public double operate(double num1, double num2) {
        return state.operate(num1, num2);
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Student s1 = new Student(new AddOperator());
        System.out.println(s1.operate(12, 23));
        /**
         * 改变状态，即改变了行为 --> 加法运算变成了减法运算
         */
        s1.setState(new SubOperator());
        System.out.println(s1.operate(12, 23));
    }
}
```

三、总结

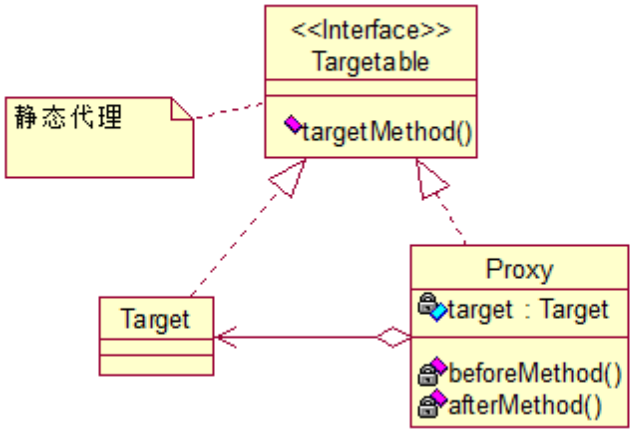
封装基类状态的行为，并将行为委托到当前状态。

(十四)代理模式建模与实现

代理模式(Proxy)：代理模式其实就是多一个代理类出来，替原对象进行一些操作。比如咱有的时候打官司需要请律师，因为律师在法律方面有专长，可以替咱进行操作表达咱的想法，这就是代理的意思。代理模式分为两类：1、静态代理(不使用jdk里面的方法)；2、动态代理(使用jdk里面的InvocationHandler和Proxy)。下面请看示例：

一、静态代理

1、uml建模：



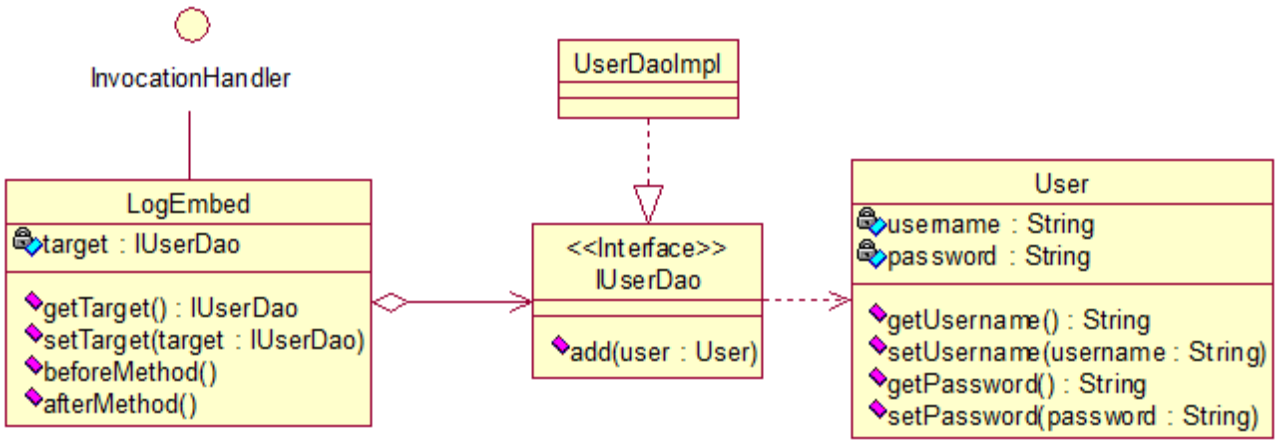
2、代码实现

```
/**
 * 示例(一)：代理模式 --静态代理(没有调用JDK里面的方法)
 *
 * 目标接口
 */

interface Targetable {
    public void targetMethod();
}

class Target implements Targetable {

    @Override
    public void targetMethod() {
        System.out.println("this is a target method...");
    }
}

class Proxy implements Targetable {
    private Target target;
```

```
    public Proxy() {
        this.target = new Target();
    }

    private void beforeMethod() {
        System.out.println("this is a method before proxy...");
    }

    private void afterMethod() {
        System.out.println("this is a method after proxy...");
    }

    /**
     * 在执行目标方法前后加了逻辑
     */
    @Override
    public void targetMethod() {
        beforeMethod();
        target.targetMethod();
        afterMethod();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建代理对象
         */
        Targetable proxy = new Proxy();
        /**
         * 执行代理方法
         */
        proxy.targetMethod();
    }
}
```

二、动态代理

1、uml建模：



2、代码实现

```
/**
 * 示例(二)：代理模式 --动态代理
 *
 * 以添加用户为例
 */
class User {
    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

```
/**
 * 目标接口
 */
interface IUserDao {
    public void add(User user);
}

class UserDaoImpl implements IUserDao {
    @Override
    public void add(User user) {
        System.out.println("add a user successfully...");
    }
}

/**
 * 日志类 --> 待织入的Log类
 */
class LogEmbed implements InvocationHandler {
    private IUserDao target;

    /**
     * 对target进行封装
     */
    public IUserDao getTarget() {
        return target;
    }

    public void setTarget(IUserDao target) {
        this.target = target;
    }

    private void beforeMethod() {
        System.out.println("add start...");
    }

    private void afterMethod() {
        System.out.println("add end...");
    }

    /**
     * 这里用到了反射
     *
     * proxy 代理对象
     *
     * method 目标方法
     *
     * args 目标方法里面参数列表
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
        beforeMethod();
        // 回调目标对象的方法
        method.invoke(target, args);
```

```
        method.invoke(target, args);
        System.out.println("LogEmbed --invoke-> method = " + method.getName());
        afterMethod();
        return null;
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        IUserDao userDao = new UserDaoImpl();
        LogEmbed log = new LogEmbed();
        log.setTarget(userDao);
        /**
         * 根据实现的接口产生代理
         */
        IUserDao userDaoProxy = (IUserDao) Proxy.newProxyInstance(userDao
                .getClass().getClassLoader(), userDao.getClass()
                .getInterfaces(), log);
        /**
         * 注意：这里在调用IUserDao接口里的add方法时，
         * 代理对象会帮我们调用实现了InvocationHandler接口的LogEmbed类的invoke方法。
         *
         * 这样做，是不是有点像Spring里面的拦截器呢？
         */
        userDaoProxy.add(new User("张三", "123"));
    }
}
```

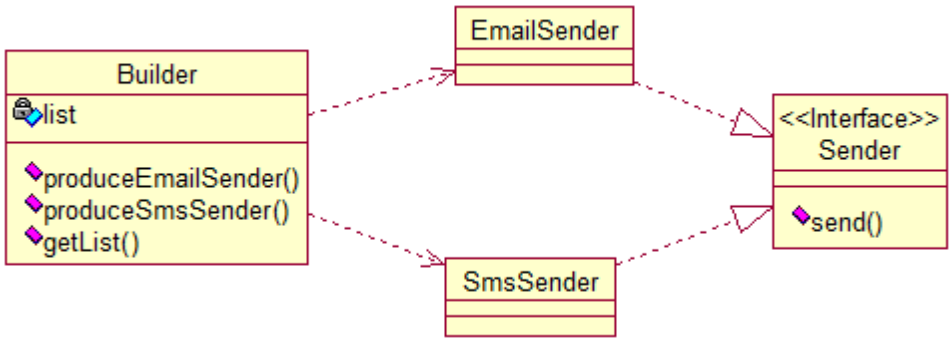
三、总结

代理模式好处：1、一个代理类调用原有的方法，且对产生的结果进行控制。2、可以将功能划分的更加清晰，有助于后期维护。

(十五)建造者模式建模与实现

建造者模式(Builder)：工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理。简单起见，就拿之前的工厂方法模式进行修改一下就可以得到建造者模式。

一、uml建模：



二、代码实现

```
/**
 * 示例：建造者模式
 *
 * 与工厂模式的区别：工厂类模式提供的是创建单个类，而建造者模式则是将各种产品集中起来进行管理
 */
interface Sender {
    public void send();
}

class EmailSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用电子邮箱发送...");
    }

}

class SmsSender implements Sender {

    @Override
    public void send() {
        System.out.println("使用短信发送...");
    }
}

class Builder {
    private List<Sender> list = new ArrayList<Sender>();
```

```
    private List<Sender> list = new ArrayList<Sender>();

    public List<Sender> getList() {
        return list;
    }

    public void produceEmailSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new EmailSender());
        }
    }

    public void produceSmsSender(int count) {
        for (int i = 0; i < count; i++) {
            list.add(new SmsSender());
        }
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        Builder builder = new Builder();
        builder.produceEmailSender(5);
        builder.produceSmsSender(5);
        /**
         * 遍历list中的元素
         */
        List<Sender> list = builder.getList();
        for (int i = 0; i < list.size(); i++) {
            Sender sender = list.get(i);
            System.out.println(sender);
            sender.send();
        }
    }
}
```

三、总结

建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工厂模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建复合对象，多个部分。

(十六)原型模式建模与实现

原型模式(Prototype)：该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。而这里的复制有两种：浅复制、深复制。

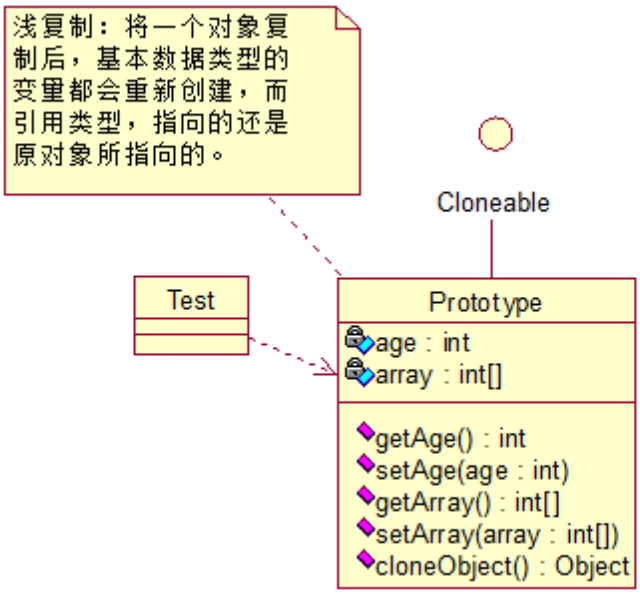
浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

下面通过示例进行说明：

一、浅复制

1、uml建模：



2、代码实现

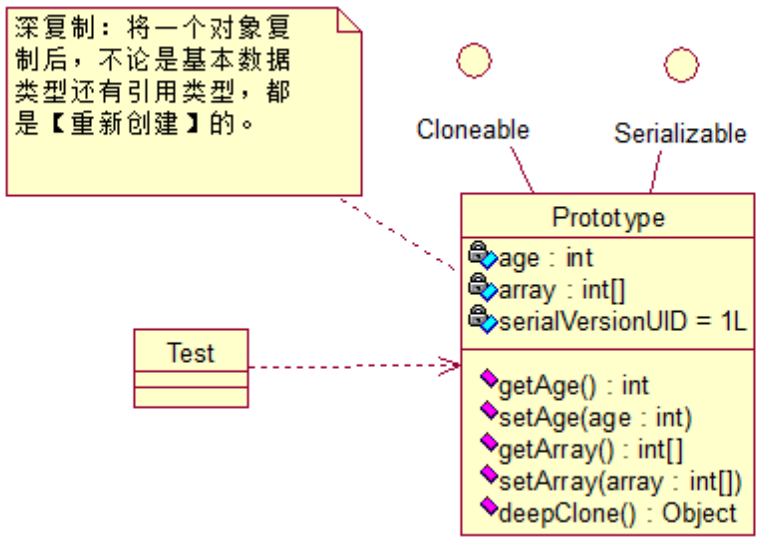
```
/**
 * 原型模式：将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的【新对象】。
 *
 * 而这里的复制有两种：浅复制、深复制
 *
 * 示例(一) 浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，
 *
 * 而引用类型，指向的还是原对象所指向的，【不会重新创建】。
 */
class Prototype implements Cloneable {
```

```
    private int age;
    private int[] array = new int[] { 1, 2, 3 };

    public Prototype() {
    }

    public Prototype(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int[] getArray() {
        return array;
    }

    public void setArray(int[] array) {
        this.array = array;
    }

    /**
     * 因为Cloneable接口是个空接口
     *
     * 此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的
     *
     * 这就涉及到JNI，关于JNI还有NDK以后会讲到，这里你只要记住浅复制的核心是super.clone()。
     */
    public Object cloneObject() throws CloneNotSupportedException {
        Prototype prototype = (Prototype) super.clone();
        return prototype;
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) throws CloneNotSupportedException {
        Prototype prototype = new Prototype(20);
        Prototype cloneProto = (Prototype) prototype.cloneObject();
        /**
         * 通过打印可以看到：prototype和cloneProto这两个同一类型的变量指向的是两个不同的内存地址
         *
         * 这说明克隆成功
         */
        System.out.println("prototype = " + prototype);
```

```
        System.out.println("cloneProto = " + cloneProto);
        /**
         * 要完全复制一个对象的话，那么它的引用类型变量array指向的肯定是不同的内存地址
         *
         * 而这里的引用类型变量array，指向的还是原对象所指向的。可以看到打印的内存地址是相同的。
         *
         * 这说明对象复制不彻底
         */
        System.out.println("prototype.getArray() = " + prototype.getArray());
        System.out.println("cloneProto.getArray() = " + cloneProto.getArray());
        /**
         * 透过这个例子可以看到：浅复制并没有将对象进行完全复制
         */
    }
}
```

二、深复制

1、uml建模：



2、代码实现

```
/**
 * 示例(二) 深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是【重新创建】的。
 *
 * 简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。
 *
 * 由于这里涉及到对对象的读写，所以这里用到了对象的序列化--实现了Serializable接口
 */
class Prototype implements Cloneable, Serializable {
    private static final long serialVersionUID = 1L;
    private int age;
    private int[] array = new int[] { 1, 2, 3 };
```

```
    public Prototype() {
    }

    public Prototype(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int[] getArray() {
        return array;
    }

    public void setArray(int[] array) {
        this.array = array;
    }

    /* 深复制 */
    public Object deepClone() throws IOException, ClassNotFoundException {

        /* 写入当前对象的二进制流 */
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);
        oos.writeObject(this);

        /* 读出二进制流产生的新对象 */
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }

}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) throws IOException,
            ClassNotFoundException {
        Prototype prototype = new Prototype(20);
        Prototype cloneProto = (Prototype) prototype.deepClone();
        /**
         * 通过打印可以看到：prototype和cloneProto这两个同一类型的变量指向的是两个不同的内存地址
         *
         * 这说明克隆成功
         */
```

```
        */
        System.out.println("prototype = " + prototype);
        System.out.println("cloneProto = " + cloneProto);
        /**
         * 通过打印可以看到，两个对象的引用类型变量array指向的是不同的内存地址
         *
         * 这说明对象进行了完全彻底的复制
         */
        System.out.println("prototype.getArray() = " + prototype.getArray());
        System.out.println("cloneProto.getArray() = " + cloneProto.getArray());

        /**
         * 当然我们也可以试着打印一下引用变量的内容，
         *
         * 可以看到：内容是不变的(1 2 3)，改变的只是引用变量指向的内存地址。
         */
        int[] proArray = prototype.getArray();
        int[] cloneProtoArray = cloneProto.getArray();
        for (int p : proArray) {
            System.out.print(p + "\t");
        }
        System.out.println();
        for (int p : cloneProtoArray) {
            System.out.print(p + "\t");
        }

    }
}
```

三、总结

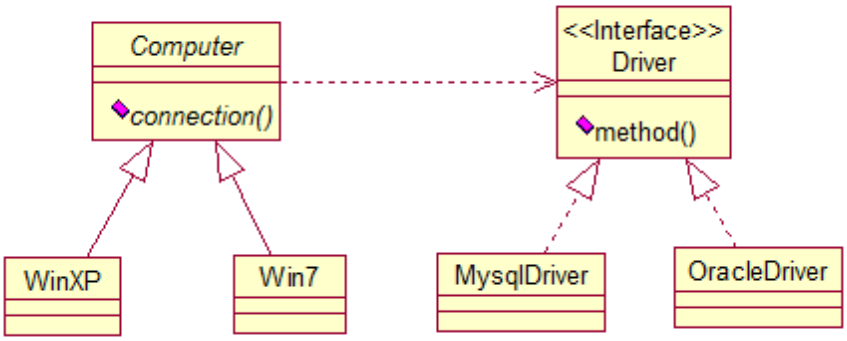
1、浅复制的核心是super.clone()，它调用的是Object的clone()方法，而在Object类中，clone()是native的。

2、要实现深复制，需要采用二进制流的形式写入当前对象，再对其进行读取。

(十七)桥接模式建模与实现

桥接模式(Bridge)：把事物和其具体实现分开(抽象化与实现化解耦)，使他们可以各自独立的变化。假设你的电脑是双系统(WinXP、Win7)，而且都安装了mysql、oracle、sqlserver、DB2这4种数据库,那么你有2*4种选择去连接数据库。按平常的写法，咱要写2*4*个类，但是使用了桥接模式，你只需写2+4个类,可以看出桥接模式其实就是一种将N*M转化成N+M组合的思想。

一、uml建模：



二、代码实现

```
/**
 * 桥接模式(Bridge)： 把事物和其具体实现分开(抽象化与实现化解耦)，使他们可以各自独立的变化。
 *
 * 假设你的电脑是双系统(WinXP、Win7)，而且都安装了mysql、oracle、sqlserver、DB2这4种数据库
 *
 * 那么你有2*4种选择去连接数据库。按平常的写法，咱要写2*4个类，但是使用了桥接模式，你只需写2+4个
类
 *
 * 可以看出桥接模式其实就是一种将N*M转化成N+M组合的思想。
 */
interface Driver {
    public void method();
}

class MysqlDriver implements Driver {

    @Override
    public void method() {
        System.out.println("use mysql driver to connection db...\n");
    }
}

class OracleDriver implements Driver {
```

```
    @Override
    public void method() {
        System.out.println("use oracle driver to connection db...\n");
    }
}

/**
 * 这里你还可以写SqlserverDriver、DB2Driver...
 */

abstract class Computer {
    public abstract void connection(Driver driver);
}

class WinXP extends Computer {
    @Override
    public void connection(Driver driver) {
        System.out.println("WinXP Computer");
        driver.method();
    }
}

class Win7 extends Computer {
    @Override
    public void connection(Driver driver) {
        System.out.println("Win7 Computer");
        driver.method();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 第一种组合：winXP使用mysql驱动连接数据库
         */
        Computer winxp = new WinXP();
        winxp.connection(new MysqlDriver());
        /**
         * 第二种组合：win7使用mysql驱动连接数据库
         */
        Computer win7 = new Win7();
        win7.connection(new MysqlDriver());
        /**
         * 第三种组合：winXP使用oracle驱动连接数据库
         */
        Computer cwinxp = new WinXP();
        cwinxp.connection(new OracleDriver());
        /**
         * 第四种组合：winXP使用oracle驱动连接数据库
         */
```

```
        Computer cwin7 = new Win7();
        cwin7.connection(new OracleDriver());

    }
}
```

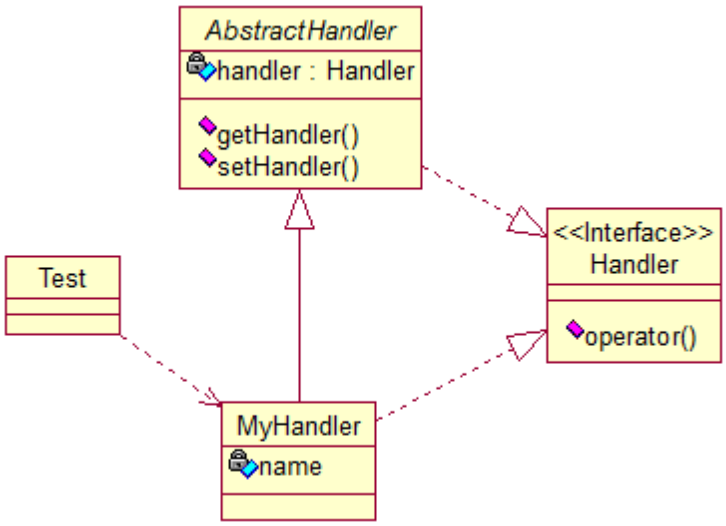
三、总结

桥接的核心思想是：将抽象化与实现化解耦，使得二者可以独立变化。

(十八)责任链模式建模与实现

责任链模式(ChainOfResponsibility)：有多个对象，每个对象持有下一个对象的引用，形成一条链，请求在这条链上传递，直到某一对象决定处理该请求，但是发出者并不清楚最终哪个对象会处理该请求。

一、uml建模：



二、代码实现

```
/**
 * 责任链模式：有多个对象，每个对象持有下一个对象的引用，形成一条链，
 *
 * 请求在这条链上传递，直到某一对象决定处理该请求，
 *
 * 但是发出者并不清楚最终哪个对象会处理该请求。
 */
interface Handler {
    public void operator();
}

/**
 * 这里单独对Handler进行封装，方便修改引用对象
 */
abstract class AbstractHandler implements Handler {
    private Handler handler;

    public Handler getHandler() {
        return handler;
    }

    public void setHandler(Handler handler) {
        this.handler = handler;
    }
```

```
}

class MyHandler extends AbstractHandler implements Handler {
    private String name;

    public MyHandler(String name) {
        this.name = name;
    }

    @Override
    public void operator() {
        if (getHandler() != null) {
            System.out.print(name + "，将BUG交给——>");
            /**
             * 这里是关键。【注意1】这里不是递归哦~
             *
             * 递归：A(operator)——>A(operator)——>A(operator)
             *
             * 责任链：A(operator)——>B(operator)——>C(operator)
             */
            getHandler().operator();
        } else {
            System.out.println(name + "处理BUG...\n");
        }
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        MyHandler handler1 = new MyHandler("技术总监");
        MyHandler handler2 = new MyHandler("项目经理");
        MyHandler handler3 = new MyHandler("程序员");
        /**
         * 如果没有下家，将会自行处理
         *
         * 打印结果：技术总监处理BUG...
         */
        handler1.operator();
        /**
         * 只要有下家，就传给下家处理
         *
         * 下面的打印结果：技术总监，将BUG交给——>项目经理，将BUG交给——>程序员处理BUG...
         *
         * 就这样，原本是技术总监自行处理的BUG，现在一层一层的把责任推给了程序员去处理
         */
        handler1.setHandler(handler2);
        handler2.setHandler(handler3);
        /**
         * 透过打印结果可以知道：MyHandler实例化后将生成一系列相互持有的对象(handler)，构成一条链。
         */
```

```
        handler1.operator();
        /**
         * 【注意2】责任链不是链表：链表有个头结点，咱每次必须通过头结点才能访问后面的节点
         *
         * 而责任链它可以从头访问，也可以从中间开始访问，如：handler2.operator();
         */

    }
}
```

三、总结

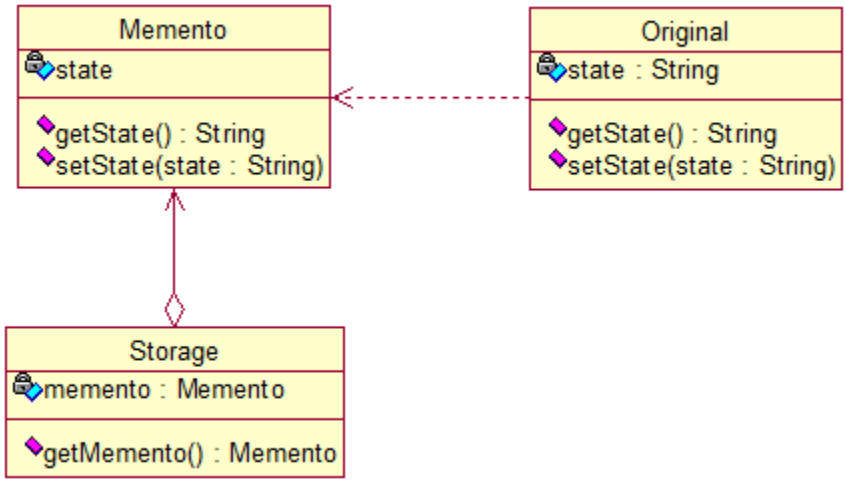
1、责任链模式可以实现，在隐瞒客户端(不知道具体处理的人是谁)的情况下，对系统进行动态的调整。

2、链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象。

(十九)备忘录模式建模与实现

备忘录模式(Memento)：主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象。

一、uml建模：



二、代码实现

```
/**
 * 备忘录模式(Memento)：主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象
 *
 * 示例：原始类--> 创建、恢复备忘录
 */
class Original {
    private String state;

    public Original(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }

    /**
     * 创建备忘录
     */
    public Memento createMemento() {
        return new Memento(state);
```

```
    }

    /**
     * 恢复备忘录
     */
    public void recoverMemento(Memento memento) {
        this.state = memento.getState();
    }
}

/**
 * 备忘录
 */
class Memento {
    private String state;

    public Memento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
    }
}

/**
 * 用来存储备忘录(持有备忘录实例)：只能存储，不能修改
 */
class Storage {
    private Memento memento;

    public Storage(Memento memento) {
        this.memento = memento;
    }

    public Memento getMemento() {
        return memento;
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建原始对象
         */
        Original original = new Original("白天模式");
```

```
        System.out.println("original初始状态为：" + original.getState());
        /**
         * 创建备忘录
         *
         * 注意：original.createMemento()会将初始state(白天模式)传给Memento对象
         *
         * 以备需要的时候可以调用storage.getMemento()来拿到该state(白天模式)状态
         *
         * 相当于state(白天模式)这个状态已经委托给了storage这个对象来保存
         */
        Storage storage = new Storage(original.createMemento());
        original.setState("夜间模式");
        System.out.println("original修改后的状态为：" + original.getState());
        /**
         * 恢复备忘录
         */
        original.recoverMemento(storage.getMemento());
        System.out.println("original恢复后的状态为：" + original.getState());
    }
}
```

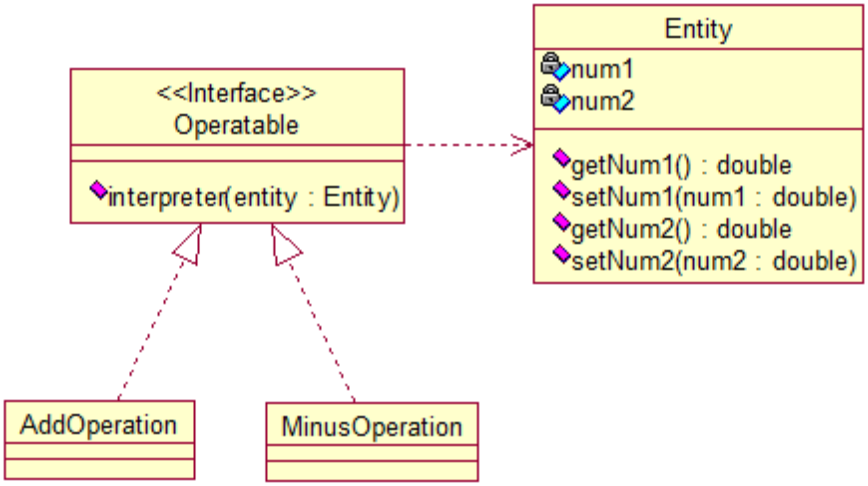
三、总结

Memento备忘录设计模式用来备份一个对象的当前状态，当需要的时候，用这个备份来恢复这个对象在某一个时刻的状态。

(二十)解释器模式建模与实现

解释器模式(Interpreter)：它定义了对象与对象之间进行某种操作之后会得到什么值。一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。

一、uml建模：



二、代码实现

```
/**
 * 解释器模式(Interpreter)：它定义了对象与对象之间进行某种操作之后会得到什么值。
 *
 * 一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。
 *
 * 示例：先定义一个实体类，封装两个变量num1、num2
 */
class Entity {
    private double num1;
    private double num2;

    public Entity(double num1, double num2) {
        this.num1 = num1;
        this.num2 = num2;
    }

    public double getNum1() {
        return num1;
    }

    public void setNum1(double num1) {
        this.num1 = num1;
    }

    public double getNum2() {
```

```
    public double getNum2() {
        return num2;
    }

    public void setNum2(double num2) {
        this.num2 = num2;
    }
}

/**
 * 运算接口
 */
interface Operatable {
    public double interpreter(Entity entity);
}

/**
 * 加法运算
 */
class AddOperation implements Operatable {

    @Override
    public double interpreter(Entity entity) {
        return entity.getNum1() + entity.getNum2();
    }
}

/**
 * 减法运算
 */
class MinusOperation implements Operatable {

    @Override
    public double interpreter(Entity entity) {
        return entity.getNum1() - entity.getNum2();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建加法、减法运算
         */
        AddOperation addOperation = new AddOperation();
        MinusOperation minusOperation = new MinusOperation();
        /**
         * 一、分步运算
         */
        double addResult = addOperation.interpreter(new Entity(20, 30));
        double minusResult = minusOperation.interpreter(new Entity(20, 30));
```

```
        System.out.println("addResult = " + addResult);
        System.out.println("minusResult = " + minusResult);
        /**
         * 二、混合运算
         */
        double mixResult = new AddOperation().interpreter(new Entity(
                addOperation.interpreter(new Entity(20, 30)), minusOperation
                        .interpreter(new Entity(40, 50))));
        System.out.println("mixResult = " + mixResult);
    }
}
```

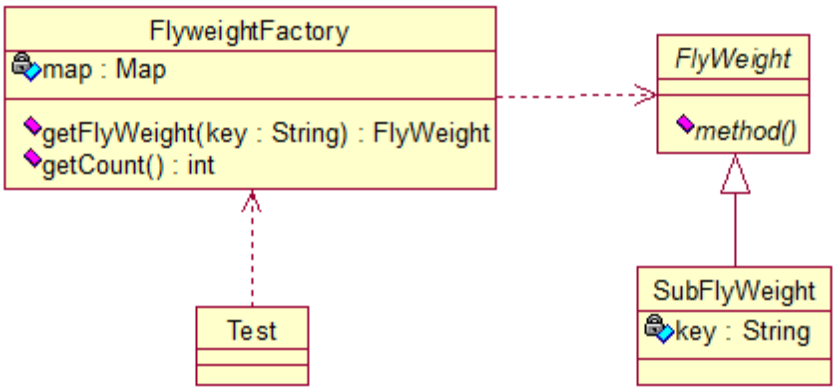
三、总结

解释器模式用来做各种各样的解释器，如正则表达式的解释器等等。

(二十一)享元模式建模与实现

享元模式（Flyweight）：运用共享的技术有效地支持大量细粒度的对象。主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销。在某种程度上，你可以把单例看成是享元的一种特例。

一、uml建模：



二、代码实现

```
/**
 * 享元模式(Flyweight)：运用共享的技术有效地支持大量细粒度的对象。
 *
 * 主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销。
 */
abstract class FlyWeight {
    public abstract void method();
}

/**
 * 创建持有key的子类
 */
class SubFlyWeight extends FlyWeight {
    private String key;

    public SubFlyWeight(String key) {
        this.key = key;
    }

    @Override
    public void method() {
        System.out.println("this is the sub method，and the key is " + this.key);
    }
}

/**
```

```
/**
 * 享元工厂：负责创建和管理享元对象
 */
class FlyweightFactory {
    private Map<String, FlyWeight> map = new HashMap<String, FlyWeight>();

    /**
     * 获取享元对象
     */
    public FlyWeight getFlyWeight(String key) {
        FlyWeight flyWeight = map.get(key);
        if (flyWeight == null) {
            flyWeight = new SubFlyWeight(key);
            map.put(key, flyWeight);
        }
        return flyWeight;
    }

    /**
     * 获取享元对象数量
     */
    public int getCount() {
        return map.size();
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建享元工厂
         */
        FlyweightFactory factory = new FlyweightFactory();
        /***第一种情况：key相同时 *********/
        FlyWeight flyWeightA = factory.getFlyWeight("aaa");
        FlyWeight flyWeightB = factory.getFlyWeight("aaa");
        /**
         * 透过打印结果为true可以知道： 由于key都为"aaa"，所以flyWeightA和flyWeightB指向同一块内存地
址
         */
        System.out.println(flyWeightA == flyWeightB);
        flyWeightA.method();
        flyWeightB.method();
        /**
         * 享元对象数量：1
         */
        System.out.println(factory.getCount());

        /***第二种情况：key不相同时 *********/
        System.out.println("\n=======================================");
        FlyWeight flyWeightC = factory.getFlyWeight("ccc");
```

```
        /**
         * 打印结果为false
         */
        System.out.println(flyWeightA == flyWeightC);
        flyWeightC.method();
        /**
         * 享元对象数量：2
         */
        System.out.println(factory.getCount());
    }
}
```

打印结果：

```
true
this is the sub method，and the key is aaa
this is the sub method，and the key is aaa
1
=====================================
false
this is the sub method，and the key is ccc
2
```

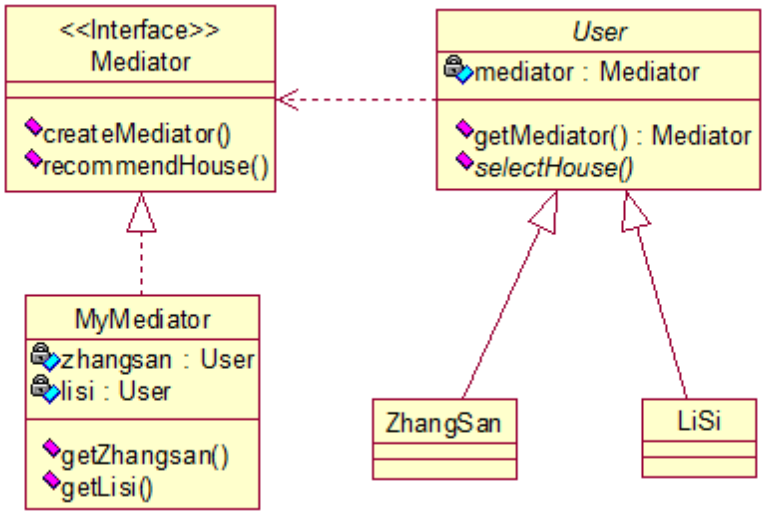
三、总结

享元与单例的区别：1、与单例模式不同，享元模式是一个类可以有很多对象(共享一组对象集合)，而单例是一个类仅一个对象；2、它们的目的也不一样，享元模式是为了节约内存空间，提升程序性能(避免大量的new操作)，而单例模式则主要是共享单个对象的状态及特征。

(二十二)中介者模式建模与实现

中介者模式(Mediator)：主要用来降低类与类之间的耦合的，因为如果类与类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。

一、uml建模：



二、代码实现

```
/**
* 中介者模式(Mediator)：主要用来降低类与类之间的耦合的，因为如果类与类之间有依赖关系的话，
*
* 不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。
*
* 示例：下面以房屋(出租)中介为例
*/
interface Mediator {
void createMediator();

void recommendHouse();
}

/**
* 咱(User)让中介帮我们推荐房子
*
* 所以咱需要持有一个中介实例
*/
abstract class User {
private Mediator mediator;

public User(Mediator mediator) {
    this.mediator = mediator;
```

```
}

public Mediator getMediator() {
    return mediator;
}

public abstract void selectHouse();
}

class ZhangSan extends User {

public ZhangSan(Mediator mediator) {
    super(mediator);
    // TODO Auto-generated constructor stub
}

@Override
public void selectHouse() {
    System.out.println("张三在选房子...");
}
}

class LiSi extends User {

public LiSi(Mediator mediator) {
    super(mediator);
    // TODO Auto-generated constructor stub
}

@Override
public void selectHouse() {
    System.out.println("李四在选房子...");
}
}

/**
* 房屋中介：中介向要找房子的那些人推荐房子，由他们选择自己想要的房子。
*
* 因此中介得持有那些实例，才有可能把房子租出去。
*/
class MyMediator implements Mediator {
private User zhangsan;
private User lisi;

public User getZhangsan() {
    return zhangsan;
}

public User getLisi() {
    return lisi;
}

@Override
public void createMediator() {
    zhangsan = new ZhangSan(this);
```

```
    lisi = new LiSi(this);
}

/**
 * 中介向要找房子的那些人推荐房子，由他们选择自己想要的房子
 */
@Override
public void recommendHouse() {
    zhangsan.selectHouse();
    lisi.selectHouse();
}
}

/**
* 客户端测试类
*
* @author Leo
*/
public class Test {
public static void main(String[] args) {
    Mediator mediator = new MyMediator();
    mediator.createMediator();
    mediator.recommendHouse();
}
}
```

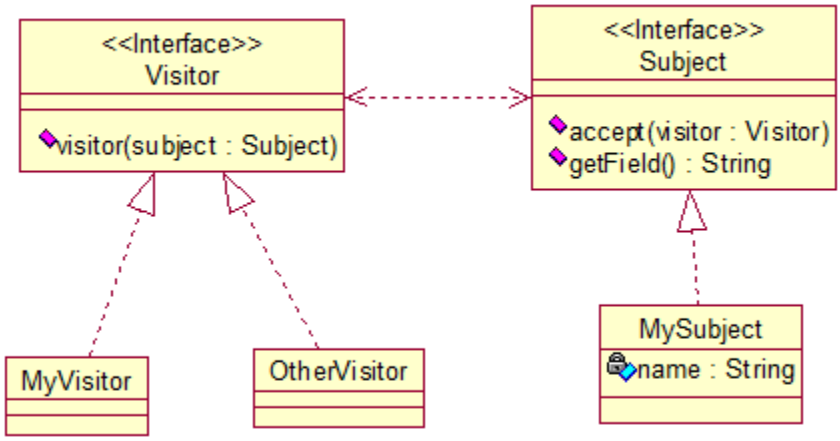
三、总结

中介者模式只需关心和Mediator类的关系，具体类与类之间的关系及调用交给Mediator就行。

(二十三)访问者模式建模与实现

访问者模式(Visitor)：把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定而算法又容易变化的系统。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者；而它的缺点就是增加新的数据结构很困难。

一、uml建模：



二、代码实现

```
/**
 * 访问者模式(Visitor)：把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。
 *
 * 访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，
 *
 * 可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。
 */
interface Visitor {
    /**
     * 访问对象
     *
     * @param subject
     *            待访问的对象
     */
    public void visitor(Subject subject);
}

class MyVisitor implements Visitor {
    @Override
    public void visitor(Subject subject) {
        System.out.println("MyVisitor 访问的属性值为：" + subject.getField());
    }
}
```

```
class OtherVisitor implements Visitor {
    @Override
    public void visitor(Subject subject) {
        System.out.println("OtherVisitor 访问的属性值为：" + subject.getField());
    }
}

interface Subject {
    /**接受将要访问它的对象 */
    public void accept(Visitor visitor);

    /**获取将要被访问的属性 */
    public String getField();
}

class MySubject implements Subject {
    private String name;

    public MySubject(String name) {
        this.name = name;
    }

    /**
     * 这是核心：接收【指定的访问者】来访问咱自身的MySubject类的状态或特征
     */
    @Override
    public void accept(Visitor visitor) {
        visitor.visitor(this);
    }

    @Override
    public String getField() {
        return name;
    }
}

/**
 * 客户端测试类
 *
 * @author Leo
 */
public class Test {
    public static void main(String[] args) {
        /**
         * 创建待访问的对象
         */
        Subject subject = new MySubject("张三");
        /**
         * 接受访问对象：这里只接收MyVisitor访问者对象，不接收OtherVisitor访问者对象
         */
        subject.accept(new MyVisitor());
    }
}
```

三、总结

访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。

Java设计模式博客全目录

今天来对这23种设计模式做个总结。咱使用设计模式的目的是为了可重用代码、让代码更容易被他人理解、保证代码可靠性，当然设计模式并不是万能的，项目中的实际问题还有具体分析。咱不能为了使用设计模式而使用，而是在分析问题的过程中，想到使用某种设计模式能达到咱需要的效果，而且比不使用设计模式更有优势，那么咱该考虑使用设计模式了。

一、设计模式的一般分类

创建型(Creator)模式（共5种）：单例、原型、建造者、工厂方法、抽象工厂。

结构型(Structure)模式（共7种）：适配器、代理、外观、桥接、组合、享元、装饰者。

行为型(Behavior)模式（共11种）：策略、观察者、模板方法、迭代器、责任链、命令、备忘录、状态、访问者、中介者、解释器。

二、下面谈谈我对这23种设计模式的理解

1、创建型(Creator)模式（共5种）

①单例（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，该对象只有一个实例存在。实现方式主要有饿汉式和懒汉式两种。

②原型(Prototype)：该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。实现方式主要有浅复制和深复制两种。浅复制的关键是super.clone()；而深复制，需要采用二进制流的形式写入当前对象，再对其进行读取。

③建造者(Builder)：该模式是将各种产品集中起来进行管理。将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。它关注的是创建复合对象，多个部分。

④工厂方法(Factory method)：调用工厂里的方法来生产对象(产品)的。它有3种实现方式：1)普通工厂模式：就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建)。2)多个工厂方法模式：是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。3)静态工厂方法模式：将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。如果你想使用工厂方法模式可以优先选择：静态工厂方法模式。

⑤抽象工厂(Abstract factory)：顾名思义，就是把工厂抽象出来，不同的工厂生产不同的产品。

2、结构型(Structure)模式（共7种）

①适配器(Adapter)：将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。1)类

的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。2)对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Adapter类，持有原类的一个实例，在Adapter类的方法中，调用实例的方法就行。3)接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Adapter实现所有方法，我们写别的类的时候，继承抽象类即可。

②代理(Proxy)：代理模式其实就是多一个代理类出来，替原对象进行一些操作。比如咱有的时候打官司需要请律师，因为律师在法律方面有专长，可以替咱进行操作表达咱的想法，这就是代理的意思。有两种实现方式：静态代理(不使用JDK里面的方法)、动态代理(InvocationHandler和Proxy)。

③外观(Facade)：也称门面模式。外观模式是为了解决类与类之间的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口。

④桥接(Bridge)： 把事物和其具体实现分开(抽象化与实现化解耦)，使他们可以各自独立的变化。桥接模式其实就是一种将N*M转化成N+M组合的思想。

⑤组合(Composite)：组合模式有时又叫部分-整体模式，将对象组合成树形结构来表示“部分-整体”层次结构。

⑥享元(Flyweight ）：运用共享的技术有效地支持大量细粒度的对象。主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销。在某种程度上，你可以把单例看成是享元的一种特例。

⑦装饰者(Decorator)：动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更具有弹性的替代方案。保持接口，增强性能。

行为型(Behavior)模式（共11种）

①策略(Strategy)让用户可以选择执行一个动作的方法，也就是说，用户可以选择不同的策略来进行操作。个人觉得策略模式可以用这个公式：不同的XXX 拥有不同的XXX供用户选择。比如说：不同的象棋棋子拥有不同的走法供用户选择。

②观察者(Observer)：在对象之间定义了一对多的依赖关系，这样一来，当一个对象改变状态时，依赖它的对象都会收到通知并自动跟新。Java已经提供了对观察者Observer模式的默认实现，Java对观察者模式的支持主要体现在Observable类和Observer接口。

③模板方法(Template method)：在一个方法中定义了一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以再不改变算法结构的情况下，重新定义算法中的某些步骤。简而言之：模板方法定义了一个算法的步骤，并允许子类为一个或多个步骤提供实现。

④迭代器(Iterator)：提供了一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示。

⑤责任链(ChainOfResponsibility)： 有多个对象，每个对象持有下一个对象的引用，形成一条链，请求在这条链上传递，直到某一对象决定处理该请求，但是发出者并不清楚最终哪个对象会处理该请求。

⑥命令(Command)：将“请求”(命令/口令)封装成一个对象，以便使用不同的请求、队列或者日志来参数化其对象。命令模式也支持撤销操作。

⑦备忘录(Memento)： 主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象。

⑧状态(State)：允许对象在内部状态改变时改变它的行为，对象看起来好像修改了它的类。状态模式说白了就是一个对像有不同的状态，不同的状态对应不同的行为，它其实是对switch case这样的语句的拓展。

⑨解释器(Interpreter)：它定义了对象与对象之间进行某种操作之后会得到什么值。一般主要应用在OOP开发中的编译器的开发中，所以适用面比较窄。

⑩中介者(Mediator)：主要用来降低类与类之间的耦合的，因为如果类与类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。

□访问者(Visitor)：把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定而算法又容易变化的系统。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者；而它的缺点就是增加新的数据结构很困难。

三、Java设计模式菜鸟系列目录

Java设计模式菜鸟系列(二十三)访问者模式建模与实现

Java设计模式菜鸟系列教程PDF下载地址：http://download.csdn.net/detail/jave_lover/8714347