23 种设计模式(Java 版)

时间	版本	作者	备注
2017 年 8 月 24 日	V1.0	shenjy	
2017 年 12 月 24 日	V1.1	shenjy	

目录

第一章 设计模式简介

1.1 设计模式简介

设计模式（Design Pattern）是一套被反复使用、多数人软件开发人员所知晓的、经过分类的、代码设计经验的总结。使用设计模式不仅可以避免软件架构设计上的错误，还可以提高编码的效率，较快地看懂一个框架的源代码。

1.2 设计模式分类

设计模式一共分为三大类，分别是创建型模式，结构型模式和行为型模式。其中，创建型模式主要包括工厂模式，单例模式，建造者模式，原型模式；结构型模式主要包括适配器模式，装饰模式，代理模式，外观模式，桥接模式，组合模式，享元模式；行为型模式主要包括策略模式，模板方法模式，观察者模式，迭代器模式，责任链模式，命令模式，备忘录模式，状态模式，访问者模式，中介者模式，解释器模式。

第二章 设计模式六原则

2.1 设计模式六原则简介

在使用设计模式的时候，要遵守下述的六种原则。它们分别是开闭原则，单一职责原则，里氏替换原则，依赖倒转原则，接口隔离原则和迪米特法则。

2.2 详细说明这六种设计原则

开闭原则：开闭原则是模式的最基本原则，一句话概括：对扩展开放，对修改关闭。在需求变更时，系统应该是通过扩展现有系统而不是修改原有逻辑，这是衡量一个架构优劣的最基本的条件。本原则是要求系统灵活性的体现，使系统易于维护和升级。

单一职责原则：又称单一功能原则。它规定一个类应该只有一个发生变化的原因。该原则由罗伯特·C·马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中给出的。

里氏替换原则：任何基类可以出现的地方，子类一定可以出现。 LSP 是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

依赖倒转原则：这个原则中有两重含义。第一，高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象；第二，抽象不应该依赖于具体实现，具体实现应该依赖于抽象。简单的讲，就是面向抽象编程。

接口隔离原则：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。这个看具体的应用上，根据应用来做接口粒度的大小的划分。

迪米特法则：又叫作最少知道原则，就是说一个对象应当对其他对象有尽可能少的了解，不和陌生人说话。也可以说，类尽量不要与其他类相互作用，减少了之间的耦合度。

2.3 面向对象编程思想

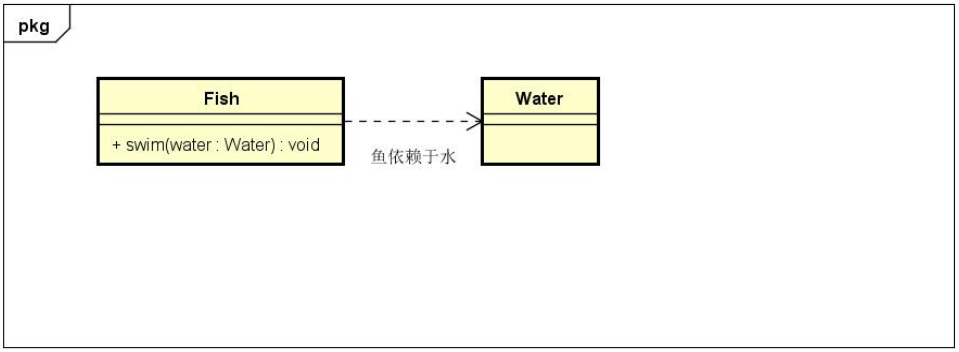
首先，在问题域里或者说在程序里应该具有哪些个对象，在问题域中抽象对象的方法主要注意，第一，一般来说名词是实体类，也有可能是类的属性；第二，形容词是接口；第三，动词是类中方法。

其次，考虑这个类或这个对象应该具有什么样的属性和什么样的方法。

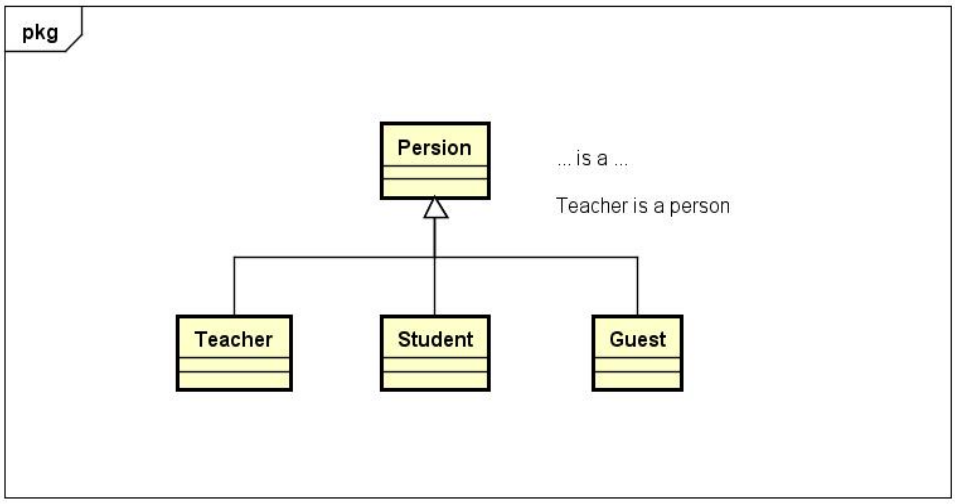
最后，考虑类与类之间的关系。

2.4 类与类之间的关系

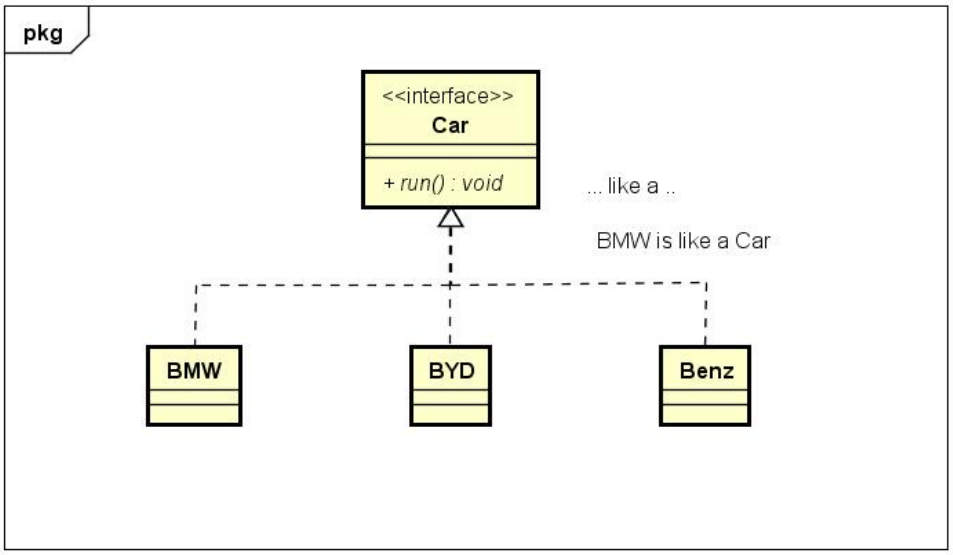
依赖关系(Dependency)：



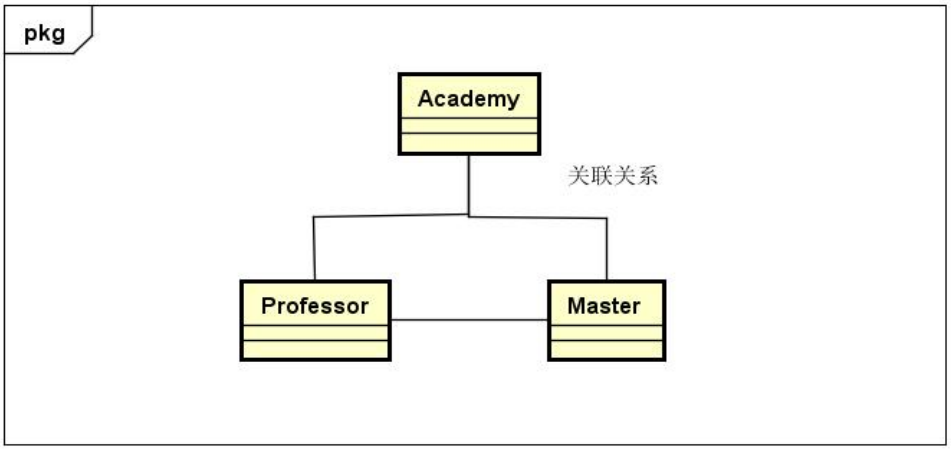
继承关系(Generalization)：



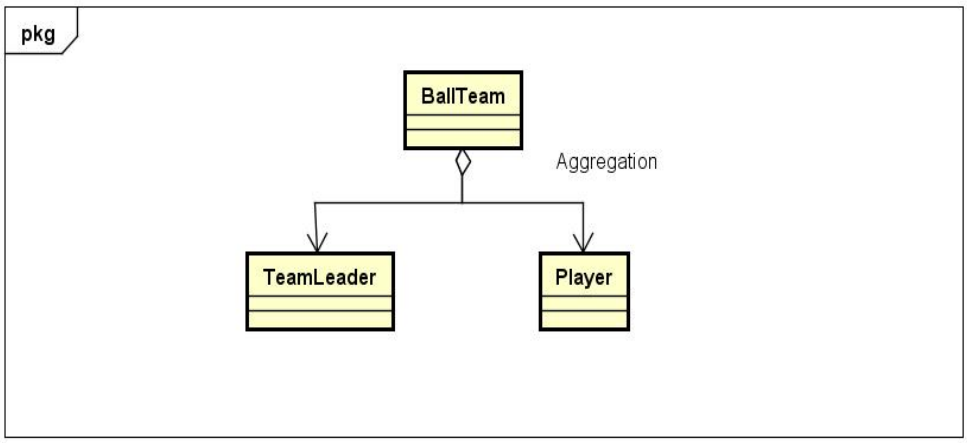
实现关系(Realization)：



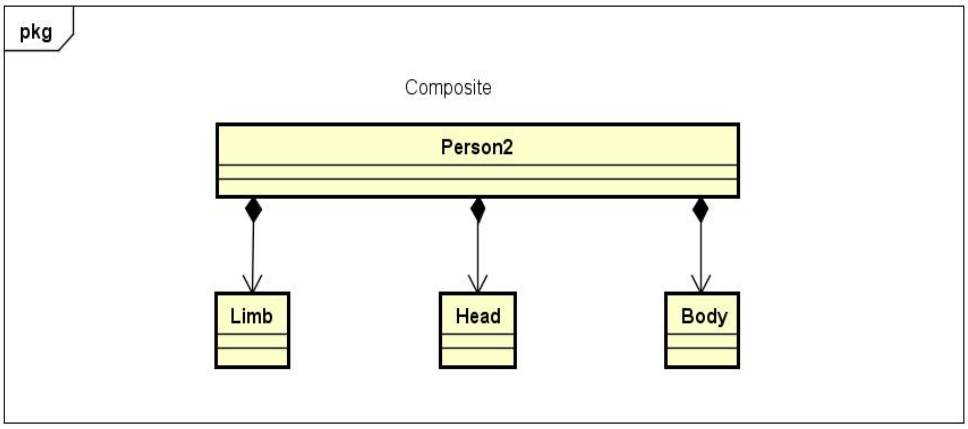
关联关系(Association)：



聚合关系(Aggregation,整体与部分的关系，整体与部分的关系弱):



组合关系（Component,整体与部分的关系，整体与部分依赖关系强）:



总结：

对于继承、实现这两种关系没多少疑问，它们体现的是一种类和类、或者类与接口间的纵向关系。其他的四种关系体现的是类和类、或者类与接口间的引用、横向关系，是比较难区分的，有很多事物间的关系要想准确定位是很难的。前面也提到，这四种关系都是语义级别的，所以从代码层面并不能完全区分各种关系，但总的来说，后几种关系所表现的强弱程度依次为：**组合>聚合>关联>依赖。**

第三章 23 种设计模式

前两章已经介绍什么是设计模式，使用设计模式的好处，设计模式的分类，设计模式的六种原则及其具体内容和类于类之间的关系。基于前两章的内容，本章将主要讲解 23 中设计模式的具体内容，包括每一种模式的内容，代码实现和每一种模式的使用场景。我将会以第一章中介绍的设计模式的分类的顺序来介绍

设计模式，即首先，介绍创建型模式；其次，介绍结构型模式；最后，介绍行为型模式。

3.1 创建型模式

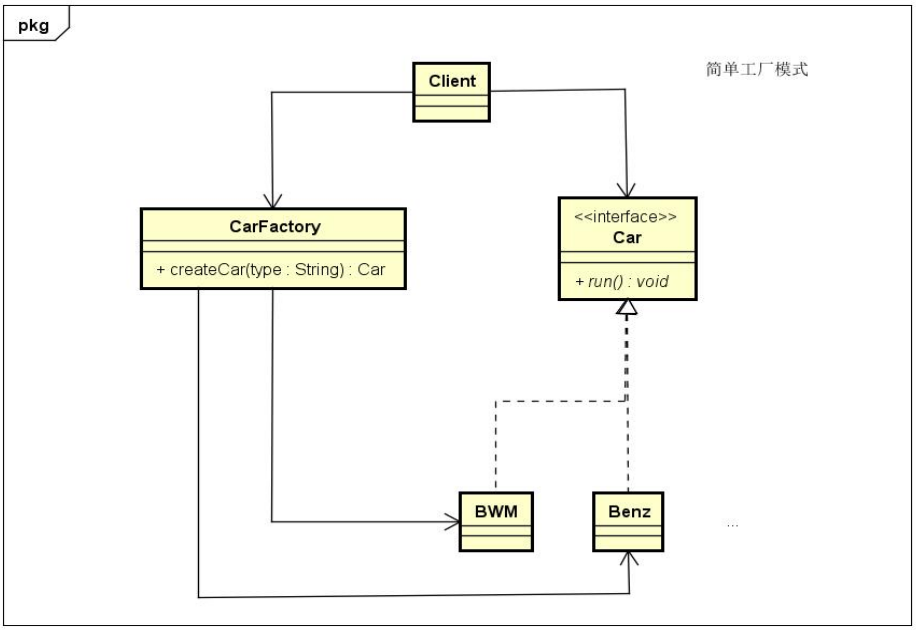
创建型模式包括工厂模式，单例模式，建造者模式和原型模式。

3.1.1 工厂模式

工厂模式包括三种，分别是简单工厂，工厂方法和抽象工厂，我会按照这个顺序来介绍这三种设计模式，工厂模式实现了创建者和调用者的分离。

- 简单工厂模式(Simple Factory)

1） 类图：



2） 代码实现：

SimpleFactory.java

```
package com.alan.test.design.pattern.simple.factory;

public class SimpleFactory {
    public Car createCar(String type) {
        if ("BWM".equals(type)) {
            return new BWM();
        } else if ("Benz".equals(type)) {
            return new Benz();
        } else {
```

```
            return new Benz();
        }
    }
}
```

Car.java

```
package com.alan.test.design.pattern.simple.factory;

public interface Car {
    public void run();
}
```

Benz.java

```
package com.alan.test.design.pattern.simple.factory;

public class Benz implements Car {
    public void run() {
        System.out.println("Benz run...");
    }
}
```

BWM.java

```
package com.alan.test.design.pattern.simple.factory;

public class BWM implements Car {
    public void run() {
        System.out.println("BWM run...");
    }
}
```

Client.java

```
package com.alan.test.design.pattern.simple.factory;

public class Client {
    public static void main(String[] args) {
        SimpleFactory sf = new SimpleFactory();
        Car car = sf.createCar("BWM");
        car.run();

        System.out.println();
        Car car2 = sf.createCar("Benz");
        car2.run();
    }
}
```

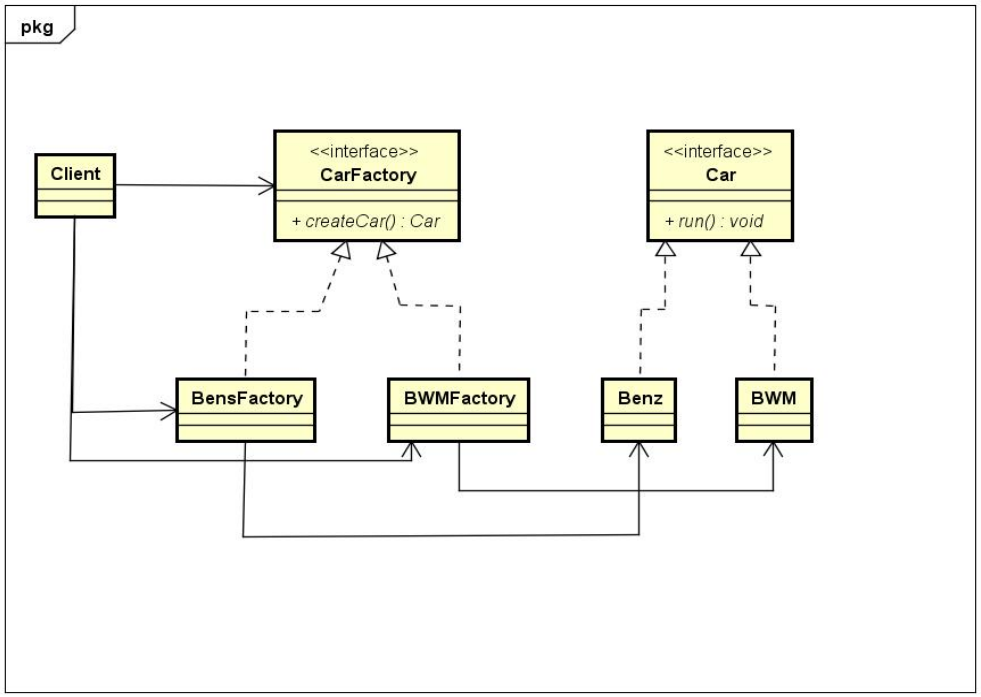
3） 使用场景及优缺点：

a. 使用场景：用于创建对象。

b. 优缺点：最常用的设计模式，但不足是不满足设计模式中的开闭原则。

- 工厂方法模式(Factory Method)

1） 类图：



2） 实现代码：

CarFactory.java

```
package com.alan.test.design.pattern.factory.method;

public interface CarFactory {
    public Car createCar();
}
```

BenzFactory.java

```
package com.alan.test.design.pattern.factory.method;

public class BenzFactory implements CarFactory {
    public Car createCar() {
        return new Benz();
    }
}
```

BWMFactory.java

```java
package com.alan.test.design.pattern.factory.method;

public class BWMFactory implements CarFactory {
    public Car createCar() {
        return new BWM();
    }
}
```

Car.java

```java
package com.alan.test.design.pattern.factory.method;

public interface Car {
    public void run();
}
```

Benz.java

```java
package com.alan.test.design.pattern.factory.method;

public class Benz implements Car {
    public void run() {
        System.out.println("Benz run...");
    }
}
```

BWM.java

```java
package com.alan.test.design.pattern.factory.method;

public class BWM implements Car {
    public void run() {
        System.out.println("BWM run...");
    }
}
```

Client.java

```java
package com.alan.test.design.pattern.factory.method;

public class Client {
    public static void main(String[] args) {
        CarFactory carFactory = new BenzFactory();
        Car car = carFactory.createCar();
        car.run();

        CarFactory carFactory2 = new BWMFactory();
        Car car2 = carFactory2.createCar();
```

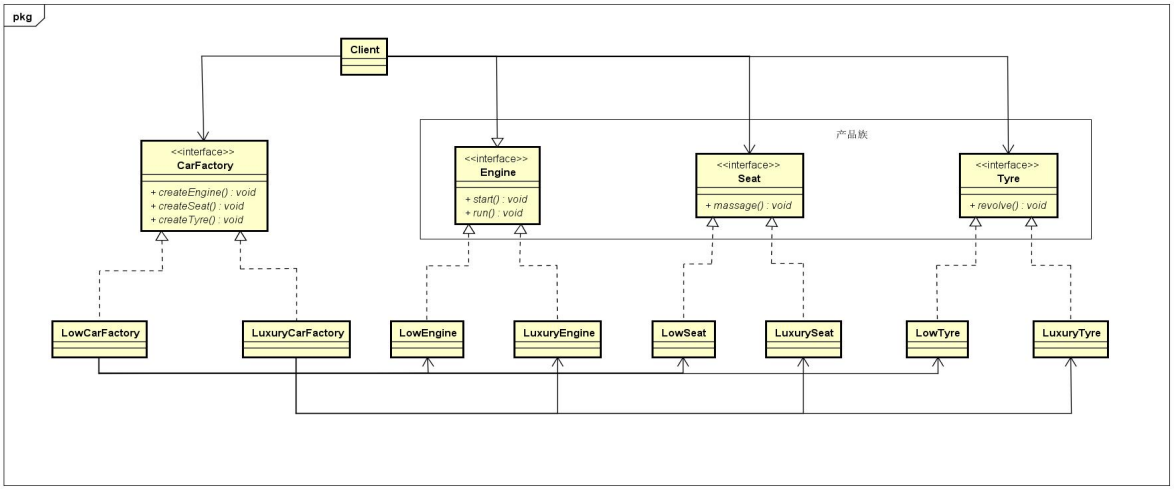
```
        car2.run();
    }
}
```

3） 工厂方法模式与简单工厂模式对比

a. 工厂方法模式避免了简单工厂模式的不足，即简单工厂不满足开闭原则。但是工厂方法模式依然存在缺陷，就是工厂方法模式在类的增长速度较快，即每增加一个新的车，就要增加一个车对应的工厂。

- 抽象工厂模式

1） 类图：



2） 源代码：

CarFactory.java

```
package com.alan.test.design.pattern.abstractfactory;

public interface CarFactory {
    public Engine createEngine();
    public Seat createSeat();
    public Tyre createTyre();
}
```

LowCarFactory.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LowCarFactory implements CarFactory {
    public Engine createEngine() {
        return new LowEngine();
    }
    public Seat createSeat() {
        return new LowSeat();
```

```
    }
    public Tyre createTyre() {
        return new LowTyre();
    }
}
```

LuxuryCarFactory.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LuxuryCarFactory implements CarFactory {
    public Engine createEngine() {
        return new LuxuryEngine();
    }
    public Seat createSeat() {
        return new LuxurySeat();
    }
    public Tyre createTyre() {
        return new LuxuryTyre();
    }
}
```

Engine.java

```
package com.alan.test.design.pattern.abstractfactory;

public interface Engine {
    public void start();
    public void run();
}
```

LowEngine.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LowEngine implements Engine {
    public void start() {
        System.out.println("发动慢...");
    }
    public void run() {
        System.out.println("跑得慢...");
    }
}
```

LuxuryEngine.java

```
package com.alan.test.design.pattern.abstractfactory;
```

```
public class LuxuryEngine implements Engine {
    public void start() {
        System.out.println("发动快...");
    }
    public void run() {
        System.out.println("跑得快...");
    }
}
```

Seat.java

```
package com.alan.test.design.pattern.abstractfactory;

public interface Seat {
    public void massage();
}
```

LowSeat.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LowSeat implements Seat {
    public void massage() {
        System.out.println("不自动按摩...");
    }
}
```

LuxurySeat.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LuxurySeat implements Seat {
    public void massage() {
        System.out.println("自动按摩...");
    }
}
```

Tyre.java

```
package com.alan.test.design.pattern.abstractfactory;

public interface Tyre {
    public void revolve();
}
```

LowTyre.java

```
package com.alan.test.design.pattern.abstractfactory;
```

```
public class LowTyre implements Tyre {
    public void revolve() {
        System.out.println("使用时间短...");
    }
}
```

LuxuryTyre.java

```
package com.alan.test.design.pattern.abstractfactory;

public class LuxuryTyre implements Tyre {
    public void revolve() {
        System.out.println("使用时间长...");
    }
}
```

Client.java

```
package com.alan.test.design.pattern.abstractfactory;

public class Client {
    public static void main(String[] args) {
        CarFactory lowCarFactory = new LowCarFactory();
        Engine e = lowCarFactory.createEngine();
        Seat s = lowCarFactory.createSeat();
        Tyre t = lowCarFactory.createTyre();
        e.start();
        e.run();
        s.massage();
        t.revolve();

        System.out.println();
        CarFactory luxuryCarFactory = new LuxuryCarFactory();
        Engine e2 = luxuryCarFactory.createEngine();
        Seat s2 = luxuryCarFactory.createSeat();
        Tyre t2 = luxuryCarFactory.createTyre();
        e2.start();
        e2.run();
        s2.massage();
        t2.revolve();
    }
}
```

3） 抽象工厂总结：

抽象工厂中存在产品族的概念，比如本例中的引擎，座椅，轮胎就是一个产品族。抽象工厂并不常用，一般会在开源的框架中可以看得到。

总结：

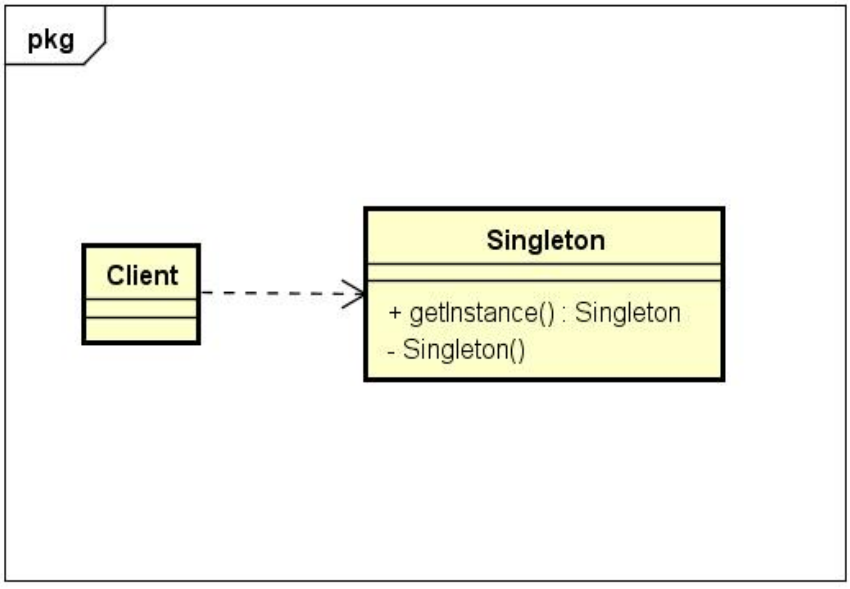
工厂模式要点：

1. 简单工厂模式：
   虽然某种程度上违反设计模式中的开闭原则，但是在实际开发中应用非常多；
2. 工厂方法模式：
   在实现上弥补了简单工厂模式的违反开闭原则的问题，但是其增加一个类的时候，类的增长速度成为增加；
3. 抽象工厂模式：
   抽象工厂中存在产品族的概念，通过其可以创建一系列产品，在实际开发中用的并不多。
4. 应用场景：
   a) JDK 中的 Calendar.getInstance();
   b) JDBC 中 Connection 类的获取；
   c) Hibernate 中的 SessionFactory;
   d) Spring 中的 bean 工厂
   e) XML 解析的时候 DocumentBuilderFactory 常见解析对象；
   f) 反射中 Class 对象的 newInstance();

3.1.2 单例模式

单例模式又分为饿汉式，懒汉式，双重检测锁式，静态内部类和枚举单例。

1. 类图



2. 饿汉式单例模式代码

Singleton1.java

```
package com.alan.test.design.pattern.singleton;

/**
```

```
 * 饿汉式单例模式
 * 特点：线程安全，  类初始化时立即加载
 * @author shenjy
 *
 */
public class Singleton1 {
    private static Singleton1 instance = new Singleton1();

    private Singleton1() {}

    public static Singleton1 getInstance() {
        return instance;
    }
}
```

3. 懒汉式单例模式

Singleton2.java

```
package com.alan.test.design.pattern.singleton;

/**
 * 懒汉式单例模式
 * 特点：线程安全，延时加载,真正用的时候采取加载这个类，资源利用率高
 *       因为需要同步，并发效率较低
 * @author shenjy
 *
 */
public class Singleton2 {
    private static Singleton2 instance;

    private Singleton2() {}

    public synchronized static Singleton2 getInstance() {
        if (null == instance) {
            instance = new Singleton2();
        }
        return instance;
    }
}
```

4. 双重检测锁设计模式

Singleton3.java

```
package com.alan.test.design.pattern.singleton;

/**
 * 双重检测锁模式
```

```
 * 特点： 由于JVM底层内部模型原因， 偶尔会出现问题， 不建议使用
 *
 * @author shenjy
 *
 */
public class Singleton3 {
    private static Singleton3 instance = null;

    private Singleton3() {}

    public static Singleton3 getInstance() {
        if (null == instance) {
            Singleton3 sc;
            synchronized (Singleton3.class) {
                sc = instance;
                if (null == sc) {
                    synchronized (Singleton3.class) {
                        if (null == sc) {
                            sc = new Singleton3();
                        }
                    }
                    instance = sc;
                }
            }
        }
        return instance;
    }
}
```

5. 静态内部类实现

Singleton4.java

```
package com.alan.test.design.pattern.singleton;

/**
 * 静态内部类
 * 特点： 同时具备线程安全， 延迟加载和并发高效的特点
 * @author shenjy
 *
 */
public class Singleton4 {
    private static class SingletonInnerClassInstance {
        private static final Singleton4 instance = new Singleton4();
    }
```

```
    private Singleton4() {}

    public static Singleton4 getInstance() {
        return SingletonInnerClassInstance.instance;
    }
}
```

6. 枚举实现

Singleton5.java

```
package com.alan.test.design.pattern.singleton;

/**
 * 枚举实现
 * 特点：实现简单， 枚举本省就是单例模式，无延时加载
 * @author shenjy
 *
 */
public enum Singleton5 {
    /**
     * 定义一个枚举的元素，它就代表了Singleton的一个实例
     */
    INSTANCE;
    /**
     * 自己的操作
     */
    public void singletonOperate() {
        // 功能处理
    }
}
```

Client.java

```
package com.alan.test.design.pattern.singleton;

public class Client1 {
    public static void main(String[] args) {
        Singleton1 s1 = Singleton1.getInstance();
        Singleton1 s2 = Singleton1.getInstance();
        System.out.println(s1.equals(s2));

        System.out.println();
        Singleton2 s3 = Singleton2.getInstance();
        Singleton2 s4 = Singleton2.getInstance();
        System.out.println(s3.equals(s4));
```

```
        System.out.println();
        Singleton3 s5 = Singleton3.getInstance();
        Singleton3 s6 = Singleton3.getInstance();
        System.out.println(s5.equals(s6));

        System.out.println();
        Singleton4 s7 = Singleton4.getInstance();
        Singleton4 s8 = Singleton4.getInstance();
        System.out.println(s7.equals(s8));

        System.out.println();
        Singleton5 s9 = Singleton5.INSTANCE;
        Singleton5 s0 = Singleton5.INSTANCE;
        System.out.println(s9.equals(s0));
    }
}
```

7. 反射和反序列化破解方式

DeSingleton2.java

```
package com.alan.test.design.pattern.singleton;

import java.io.ObjectStreamException;
import java.io.Serializable;

/**
 * 懒汉式单例模式
 * 特点：线程安全，延时加载,真正用的时候采取加载这个类，资源利用率高
 *          因为需要同步，并发效率较低
 * 如何防止反序列化反射的漏洞
 * @author shenjy
 *
 */
public class DeSingleton2 implements Serializable {
    private static final long serialVersionUID =
2675422551585692602L;
    private static DeSingleton2 instance;

    private DeSingleton2() {}

    public synchronized static DeSingleton2 getInstance() {
        if (null == instance) {
            instance = new DeSingleton2();
        }
        return instance;
```

```
    }

    /**
     * 反序列化时，如果定义了这个方法，则直接返回此方法指定的对象，而不必单独在创建新对象
     * @return
     * @throws ObjectStreamException
     */
    public Object readResolve() throws ObjectStreamException {
        return instance;
    }
}
```

Client2.java

```
package com.alan.test.design.pattern.singleton;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.reflect.Constructor;

/**
 * 测试反射，反序列化破解
 * @author shenjy
 *
 */
public class Client2 {

    public static void main(String[] args) throws Exception {
        DeSingleton2 ds1 = DeSingleton2.getInstance();
        DeSingleton2 ds2 = DeSingleton2.getInstance();
        System.out.println(ds1.equals(ds2));

        Class<DeSingleton2> clazz = (Class<DeSingleton2>) Class.forName("com.alan.test.design.pattern.singleton.DeSingleton2");
        Constructor<DeSingleton2> c = clazz.getDeclaredConstructor(null);
        c.setAccessible(true);
        DeSingleton2 ds3 = c.newInstance();
        DeSingleton2 ds4 = c.newInstance();
        System.out.println(ds3.equals(ds4));
```

```
        FileOutputStream fos = new FileOutputStream("d:/a.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(ds1);
        fos.close();
        oos.close();

        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("d:/a.txt"));
        DeSingleton2 ds5 = (DeSingleton2) ois.readObject();
        System.out.println(ds1.equals(ds5));
        ois.close();
    }
}
```

8. 性能测试（多线程状态下）

```
package com.alan.test.design.pattern.singleton;

import java.util.concurrent.CountDownLatch;

/**
 * 效率测试
 * @author shenjy
 *
 */
public class Client3 {
    public static void main(String[] args) throws
InterruptedException {
        Long start = System.currentTimeMillis();

        int threadNum = 10;
        final CountDownLatch cdl = new CountDownLatch(threadNum);
        for (int i = 0; i < threadNum; i++) {
            new Thread(new Runnable() {
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        Object o = Singleton5.INSTANCE;
                    }
                    cdl.countDown();
                }
            }).start();
        }
        cdl.await();

        Long end = System.currentTimeMillis();
```

```
        System.out.println("所用时间: " + (end - start));
    }
}
```

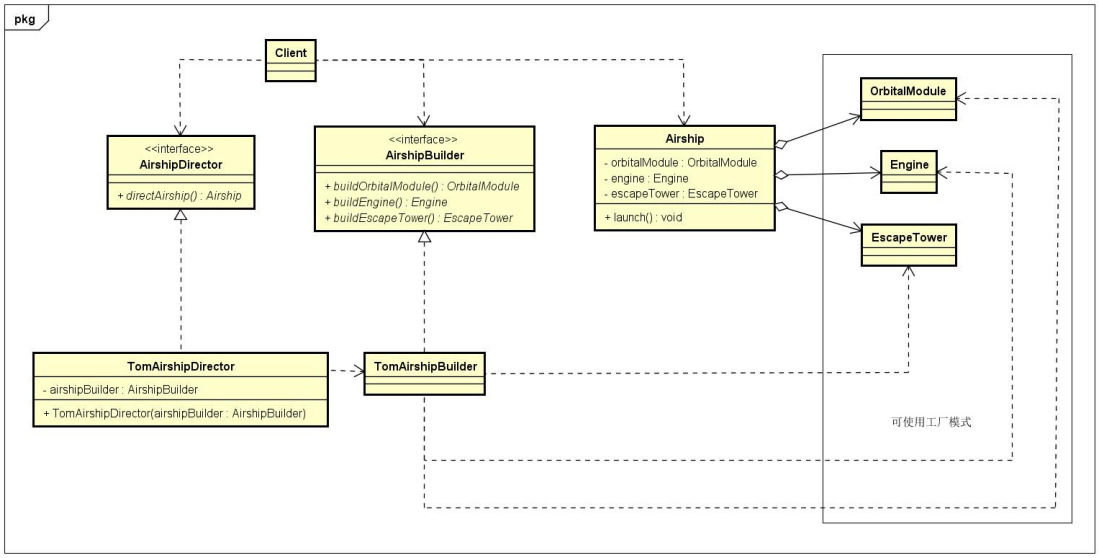
9. 总结

常用单例模式有饿汉式，懒汉式和静态内部类方式。

性能排序	单例模式
1	饿汉式
2	静态内部类
3	枚举
4	双重检查锁
5	懒汉式

3.1.3 建造者模式

1. 类图：



2. 代码实现

Airship.java

```
package com.alan.test.design.pattern.builder;

/**
 * 宇宙飞船
 *
 * @author shenjy
 *
 */
public class Airship {
```

```java
private OrbitalModule orbitalModule; // 轨道舱
private Engine engine; // 发动机
private EscapeTower escapeTower; // 逃逸塔
/**
 * @return the orbitalModule
 */
public OrbitalModule getOrbitalModule() {
    return orbitalModule;
}
/**
 * @param orbitalModule the orbitalModule to set
 */
public void setOrbitalModule(OrbitalModule orbitalModule) {
    this.orbitalModule = orbitalModule;
}
/**
 * @return the engine
 */
public Engine getEngine() {
    return engine;
}
/**
 * @param engine the engine to set
 */
public void setEngine(Engine engine) {
    this.engine = engine;
}
/**
 * @return the escapeTower
 */
public EscapeTower getEscapeTower() {
    return escapeTower;
}
/**
 * @param escapeTower the escapeTower to set
 */
public void setEscapeTower(EscapeTower escapeTower) {
    this.escapeTower = escapeTower;
}

public void launch() {
    System.out.println("Tom's airship launched...");
}
```

```
}

class OrbitalModule {
    private String name;

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

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
    }

}

class Engine {
    private String name;

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

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
```

```
    }

}

class EscapeTower {
    private String name;

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

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
    }

}
```

AirshipBuilder.java

```
package com.alan.test.design.pattern.builder;

public interface AirshipBuilder {
    public OrbitalModule buildOrbitalModule();
    public Engine buildEngine();
    public EscapeTower buildEscapeTower();
}
```

TomAirshipBuilder.java

```
package com.alan.test.design.pattern.builder;

public class TomAirshipBuilder implements AirshipBuilder {

    public OrbitalModule buildOrbitalModule() {
        System.out.println("构建轨道舱...");
        return new OrbitalModule("轨道舱");  // 可以使用工厂模式来做
    }
```

```
    public Engine buildEngine() {
        System.out.println("构建发动机...");
        return new Engine("发动机");  // 可以使用工厂模式来做
    }

    public EscapeTower buildEscapeTower() {
        System.out.println("构建逃逸塔...");
        return new EscapeTower("逃逸塔");  // 可以使用工厂模式来做
    }

}
```

AirshipDirector.java

```
package com.alan.test.design.pattern.builder;

public interface AirshipDirector {
    public Airship directAirship();
}
```

TomAirshipDirector.java

```
package com.alan.test.design.pattern.builder;

public class TomAirshipDirector implements AirshipDirector {

    // 使用关联的方式更好，更灵活
    private AirshipBuilder airshipBuilder;

    public TomAirshipDirector(AirshipBuilder airshipBuilder) {
        this.airshipBuilder = airshipBuilder;
    }

    public Airship directAirship() {
        Engine e = airshipBuilder.buildEngine();
        OrbitalModule om = airshipBuilder.buildOrbitalModule();
        EscapeTower et = airshipBuilder.buildEscapeTower();

        Airship airship = new Airship();
        airship.setEngine(e);
        airship.setOrbitalModule(om);
        airship.setEscapeTower(et);

        return airship;
    }
}
```

Client.java

```
package com.alan.test.design.pattern.builder;

public class Client {
    public static void main(String[] args) {
        // 构建
        AirshipBuilder airshipBuilder = new TomAirshipBuilder();
        // 装配
              AirshipDirector ad = new
        TomAirshipDirector(airshipBuilder);

        Airship as = ad.directAirship();
        as.launch();
    }
}
```

总结：

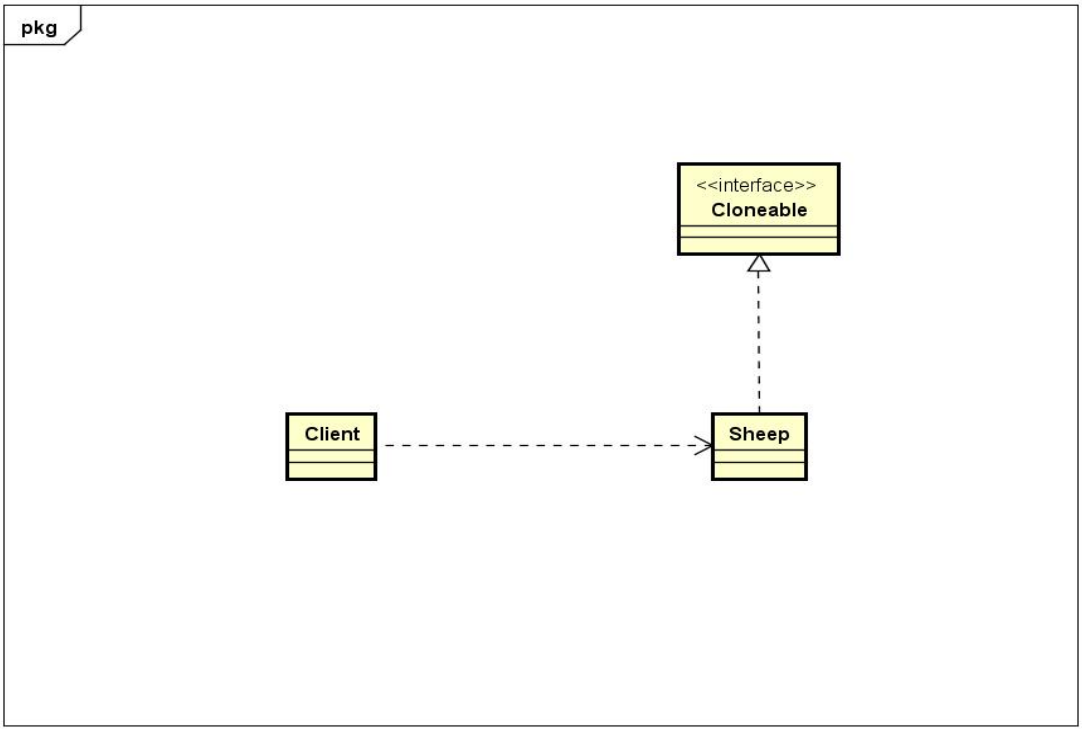
1） 建造一个复杂的产品；

2） 第一步构建零件，这个过程可以和工厂模式，单例模式结合使用；

3） 第二部装配，这个过程涉及到装配的顺序问题；

3.1.4 原型模式

1. 分类：原型模式分为浅克隆，深克隆；
2. 类图：



- 代码实现

1） 浅克隆

Sheep.java

```
package com.alan.test.design.pattern.prototype;

import java.util.Date;

public class Sheep implements Cloneable {
    private String name;
    private Date birthday;

    public Sheep() {}

    public Sheep(String name, Date birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }
    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
    }
    /**
     * @return the birthday
     */
    public Date getBirthday() {
        return birthday;
    }
    /**
     * @param birthday the birthday to set
     */
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    /*
     * @see java.lang.Object#clone()
```

```
     */
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Object obj = super.clone();
        return obj;
    }
}
```

Client.java

```
package com.alan.test.design.pattern.prototype;

import java.util.Date;

public class Client {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Date date = new Date(2111113333333311L);
        Sheep s = new Sheep("少利", date);
        System.out.println(s.getName());
        System.out.println(s.getBirthday());
        s.setBirthday(new Date(2111113333311L));

        Sheep s2 = (Sheep) s.clone();
        System.out.println(s2.getName());
        System.out.println(s2.getBirthday());
    }
}
```

2） 深克隆

```
package com.alan.test.design.pattern.prototype;

import java.util.Date;

public class Sheep2 implements Cloneable {
    private String name;
    private Date birthday;

    public Sheep2() {}

    public Sheep2(String name, Date birthday) {
        this.name = name;
        this.birthday = birthday;
    }
```

```
    /**
     * @return the name
     */
    public String getName() {
        return name;
    }
    /**
     * @param name the name to set
     */
    public void setName(String name) {
        this.name = name;
    }
    /**
     * @return the birthday
     */
    public Date getBirthday() {
        return birthday;
    }
    /**
     * @param birthday the birthday to set
     */
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    /*
     * @see java.lang.Object#clone()
     */
    @Override
    protected Object clone() throws CloneNotSupportedException {
        Object obj = super.clone();
        Sheep2 s = (Sheep2) obj;
        s.birthday = (Date) this.birthday.clone();
        return obj;
    }
}
```

Client2.java

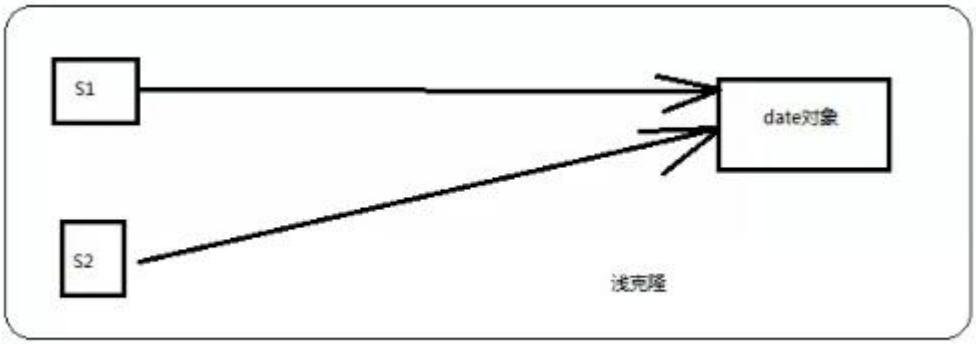
```
package com.alan.test.design.pattern.prototype;
import java.util.Date;
public class Client2 {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Date date = new Date(2111113333333311L);
        Sheep s = new Sheep("少利", date);
```

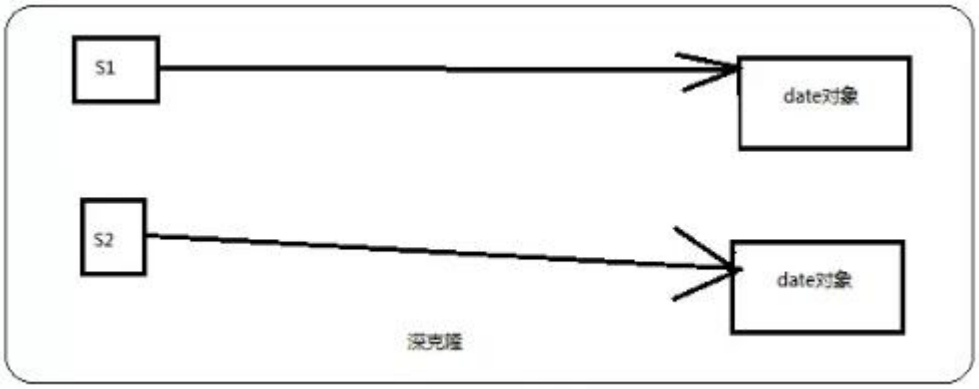
```
        Sheep s2 = (Sheep) s.clone();
        System.out.println(s.getName());
        System.out.println(s.getBirthday());
        s.setBirthday(new Date(2111113333311L));

        System.out.println(s2.getName());
        System.out.println(s2.getBirthday());
    }
}
```

总结：

1） 实现对象之间的 copy

2） 深复制，浅复制原理图



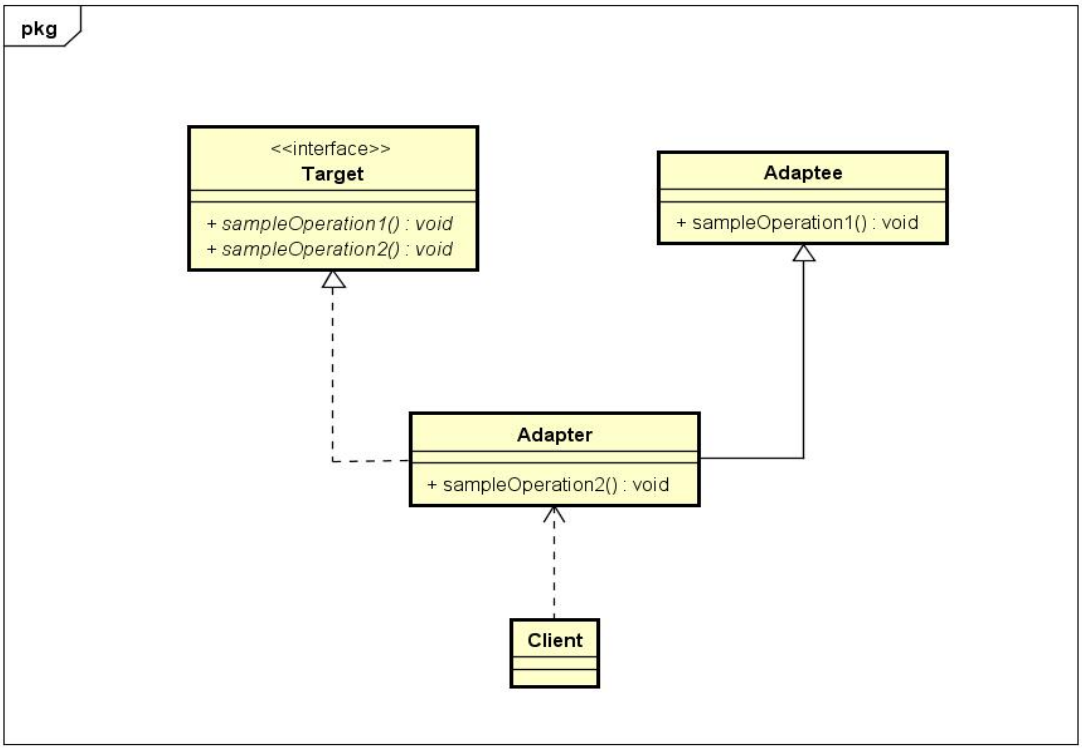


3.2 结构型模式

结构型模式包括适配器模式，桥接模式，组合模式，装饰模式，外观模式，享元模式和代理模式。

3.2.1 适配器模式

1. 类适配器类图



2. 代码实现：

Target.java

```
package design.pattern.structure.classadapter;

public interface Target {
    /**
     * 这是源类Adaptee也有的方法
     */
    public void sampleOperation1();
    /**
     * 这是源类Adapteee没有的方法
     */
    public void sampleOperation2();
}
```

Adaptee.java

```
package design.pattern.structure.classadapter;

public class Adaptee {
    public void sampleOperation1() {
       System.out.println("sampleOperation1...");
    }
}
```

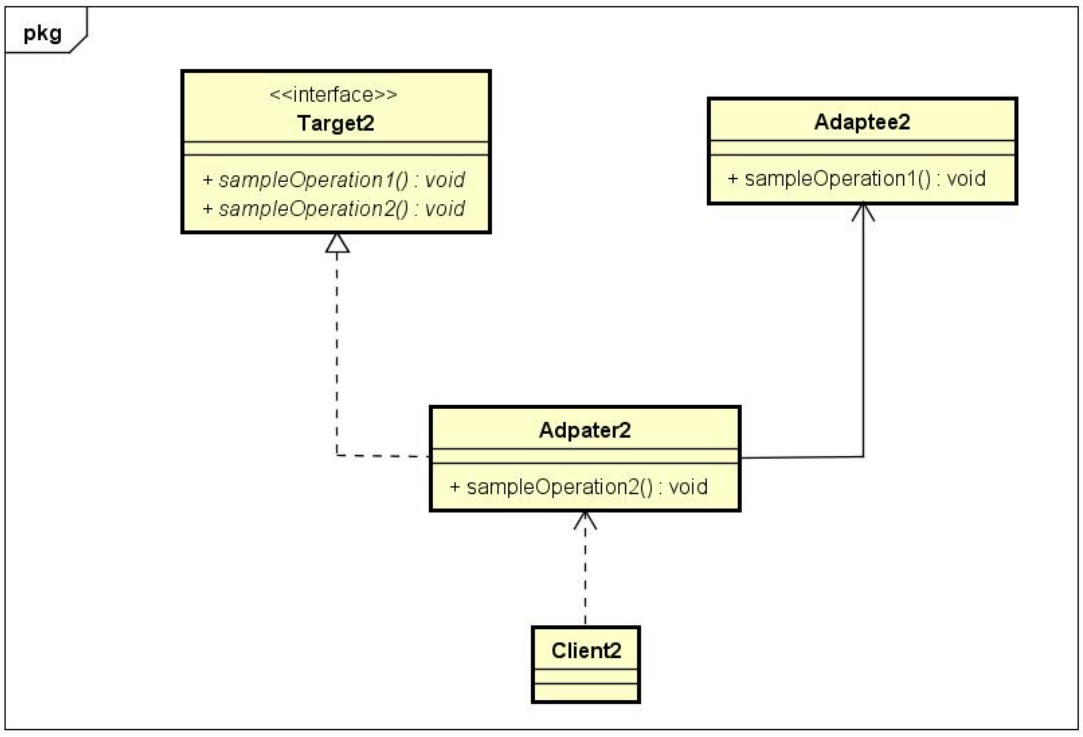
Adapter.java

```
package design.pattern.structure.classadapter;
public class Adapter extends Adaptee implements Target {
    /**
     * 这个方法也可以不重写，在调用的时候直接使用this关键字调用即可
     */
    @Override
    public void sampleOperation1() {
        super.sampleOperation1();
    }
    public void sampleOperation2() {
        System.out.println("sampleOperation2...");
    }
}
```

Client.java

```
package design.pattern.structure.classadapter;
public class Client {
    public static void main(String[] args) {
        Adapter a = new Adapter();
        a.sampleOperation1();
        a.sampleOperation2();
    }
}
```

3. 对象适配器类图



4.代码实现

Target.java

```
package design.pattern.structure.objectadapter;

public interface Target {
    /**
     * 这是源类Adaptee也有的方法
     */
    public void sampleOperation1();
    /**
     * 这是源类Adapteee没有的方法
     */
    public void sampleOperation2();
}
```

Adapter.java

```
package design.pattern.structure.objectadapter;

public class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adapee) {
        this.adaptee = adapee;
    }
    public void sampleOperation1() {
        adaptee.sampleOperation1();
    }
    public void sampleOperation2() {
        System.out.println("sampleOperation2...");
    }
}
```

Adaptee.java

```
package design.pattern.structure.objectadapter;

public class Adaptee {
    public void sampleOperation1() {
        System.out.println("sampleOperation1...");
    }
}
```

Client.java

```
package design.pattern.structure.objectadapter;
```

```
public class Client {
    public static void main(String[] args) {
        Adaptee adaptee = new Adaptee();
        Adapter a = new Adapter(adaptee);
        a.sampleOperation1();
        a.sampleOperation2();
    }
}
```

5. 总结

1） 现实中三孔插座转两孔插座。

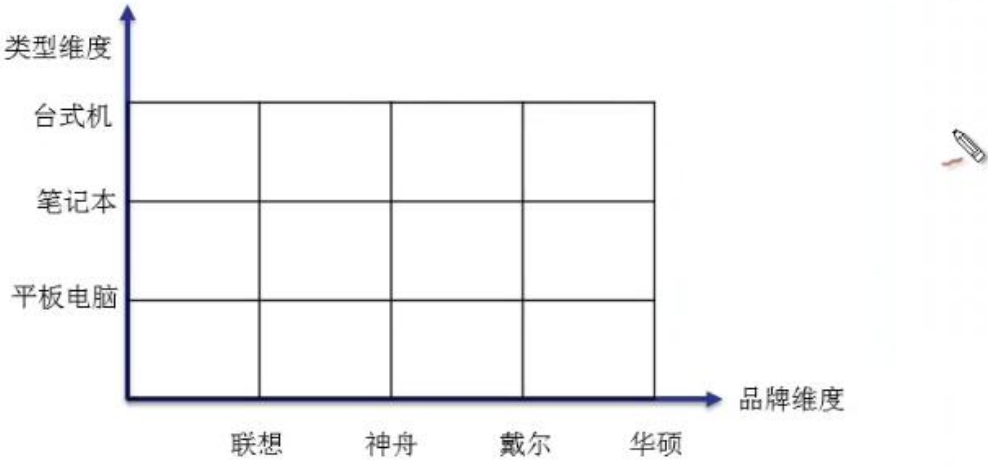
3.2.2 桥接模式

1. 问题描述：

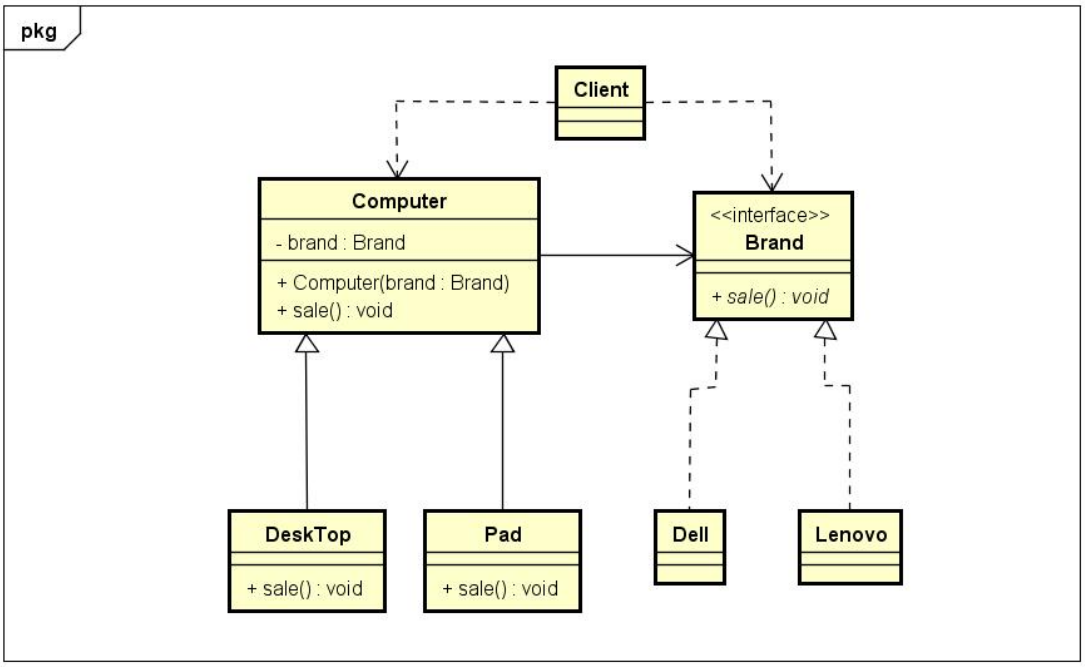
2. 问题解决方案：

商城系统中常见的商品分类，以电脑为类，如何良好的处理商品分类销售的问题？

这个场景中有两个变化的维度：电脑类型、电脑品牌。



3. 类图描述



4. 代码实现

Brand.java

```
package design.pattern.structure.bridge;

public interface Brand {
    public void sale();
}
```

Dell.java

```
package design.pattern.structure.bridge;

public class Dell implements Brand {
    public void sale() {
        System.out.println("销售戴尔电脑...");
    }
}
```

Lenovo.java

```
package design.pattern.structure.bridge;

public class Lenovo implements Brand {
    public void sale() {
        System.out.println("销售联想电脑...");
    }
}
```

Computer.java

```
package design.pattern.structure.bridge;

public class Computer {
    private Brand brand;

    public Computer(Brand brand) {
        this.brand = brand;
    }

    public void sale() {
        brand.sale();
    }
}
```

DeskTopComputer.java

```
package design.pattern.structure.bridge;

public class DeskTopComputer extends Computer {

    public DeskTopComputer(Brand brand) {
        super(brand);
    }

    @Override
    public void sale() {
        super.sale();
        System.out.println("销售台式电脑...");
    }
}
```

PadComputer.java

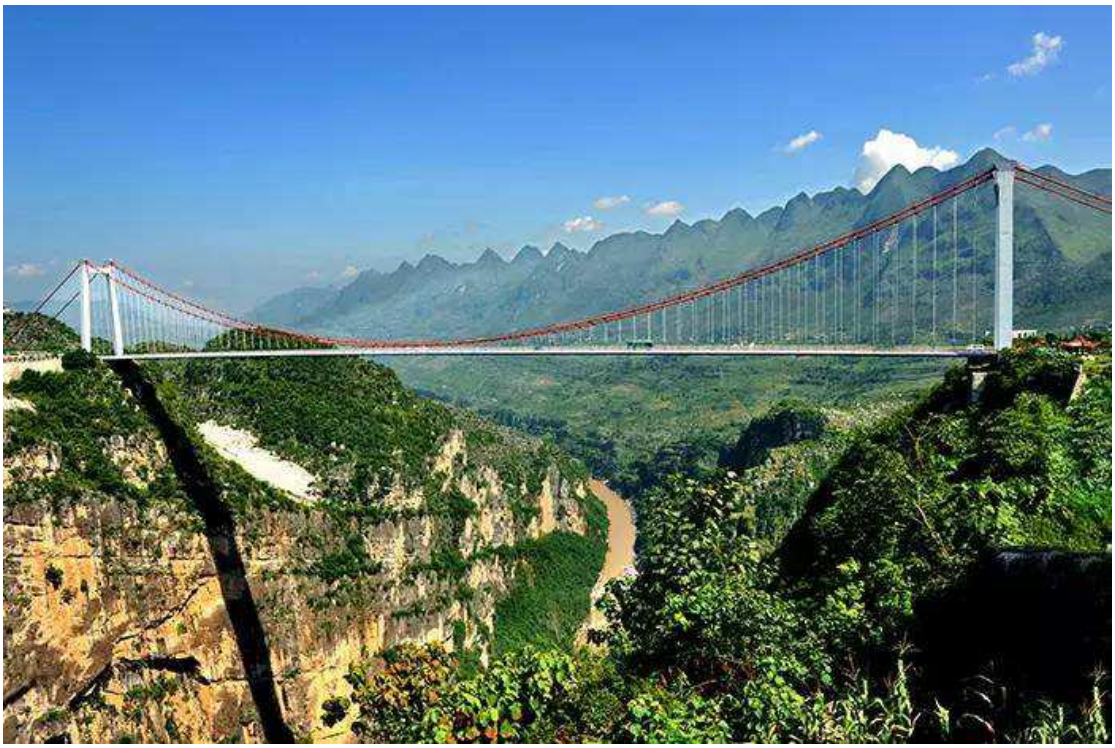
```
package design.pattern.structure.bridge;

public class PadComputer extends Computer {
    public PadComputer(Brand brand) {
        super(brand);
    }
    @Override
    public void sale() {
        super.sale();
        System.out.println("销售Pad电脑...");
    }
}
```

Client.java

```
package design.pattern.structure.bridge;

public class Client {
    public static void main(String[] args) {
        Brand brand = new Lenovo();
        Computer c = new DeskTopComputer(brand);
        c.sale();

        System.out.println();
        Brand brand2 = new Lenovo();
        Computer c2 = new DeskTopComputer(brand2);
        c2.sale();
    }
}
```

5.总结

桥接模式实现了两个维度上的一个连接，如下图。

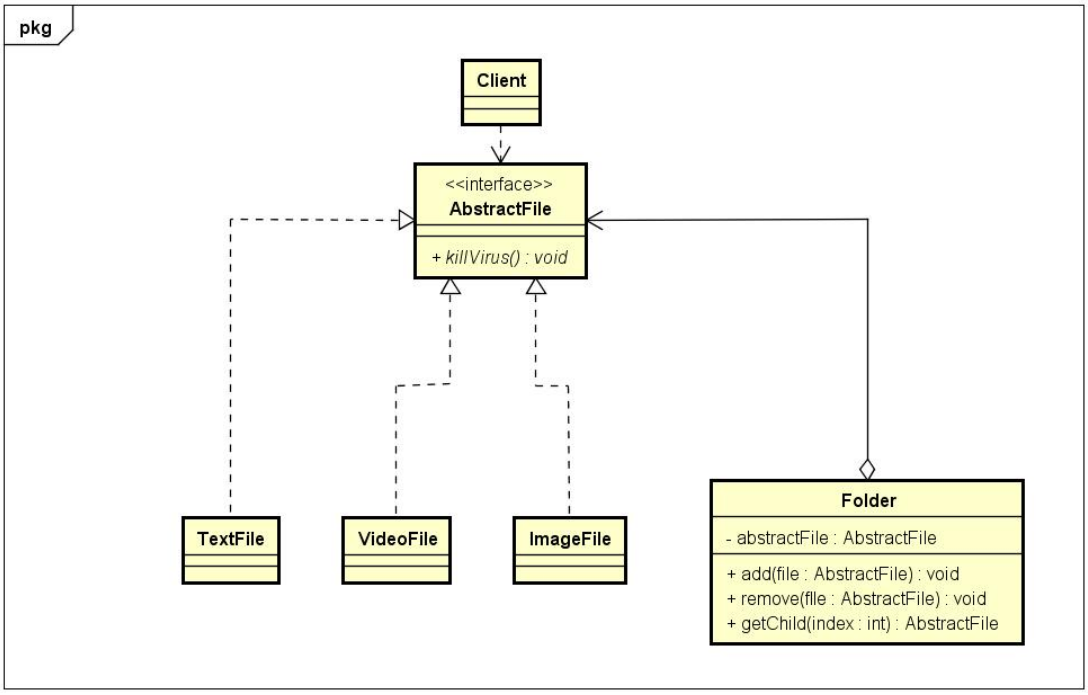


3.2.3 组合模式

1. 介绍

部分与整体的关系，用属性结构表示。例如，计算机中的文件系统，数据结构中的树。

2. 类图描述



3.代码实现

Component.java

```
package design.pattern.structure.composite;

public interface Component {
    public void operation();
}

interface Leaf extends Component {

}

interface Composite extends Component {
    public void add(Component component);
    public void remove(Component component);
    public Component getChild(int index);
}
```

AbstractFile.java

```
package design.pattern.structure.composite;
/**
 * 相当于Component
 * @author shenjy
 *
```

```
 */
public interface AbstractFile {
    public void killVirus();
}
```

TextFile.java

```
package design.pattern.structure.composite;
/**
 * 相当于Leaf，  具体文件
 * @author shenjy
 *
 */
public class TextFile implements AbstractFile {
    private String name;
    public TextFile(String name) {
        super();
        this.name = name;
    }
    public void killVirus() {
        System.out.println("文本文件: " + name + ", 开始查杀!");
    }
}
```

VideoFile.java

```
package design.pattern.structure.composite;

/**
 * 相当于Leaf，  具体文件
 * @author shenjy
 *
 */
public class VideoFile implements AbstractFile {
    private String name;

    public VideoFile(String name) {
        super();
        this.name = name;
    }
    public void killVirus() {
        System.out.println("视频文件: " + name + ", 开始查杀!");
    }
}
```

ImageFile.java

```java
package design.pattern.structure.composite;

/**
 * 相当于Leaf， 具体文件
 * @author shenjy
 *
 */
public class ImageFile implements AbstractFile {
    private String name;

    public ImageFile(String name) {
        super();
        this.name = name;
    }
    public void killVirus() {
        System.out.println("图像文件: " + name + ", 开始查杀!");
    }
}
```

Folder.java

```java
package design.pattern.structure.composite;
import java.util.ArrayList;
import java.util.List;
public class Folder implements AbstractFile {
    private String name;
    private List<AbstractFile> abstractFileList = new
ArrayList<AbstractFile>();

    public Folder(String name) {
        this.name = name;
    }
    public void add(AbstractFile abstractFile) {
        abstractFileList.add(abstractFile);
    }
    public void remove(AbstractFile abstractFile) {
        abstractFileList.remove(abstractFile);
    }
    public void killVirus() {
        System.out.println("文件夹: " + name + "杀毒开始...");
        for (AbstractFile abstractFile : abstractFileList) {
            abstractFile.killVirus();
        }
    }
}
```

Client.java

```
package design.pattern.structure.composite;

public class Client {
    public static void main(String[] args) {
        Folder f = new Folder("学习资料");
        Folder f2 = new Folder("程序设计");
        Folder f3 = new Folder("文档");
        Folder f4 = new Folder("类图");
        Folder f5 = new Folder("视频教程");

        TextFile doc = new TextFile("Java编程思想.pdf");
        VideoFile video = new VideoFile("设计模式.mp4");
        ImageFile image = new ImageFile("设计模式整体关系图.png");
        f3.add(doc);
        f4.add(image);
        f5.add(video);

        // 组合各个文件中的内容
        f2.add(f3);
        f2.add(f4);
        f2.add(f5);
        f.add(f2);

        f.killVirus();
    }
}
```

4.总结

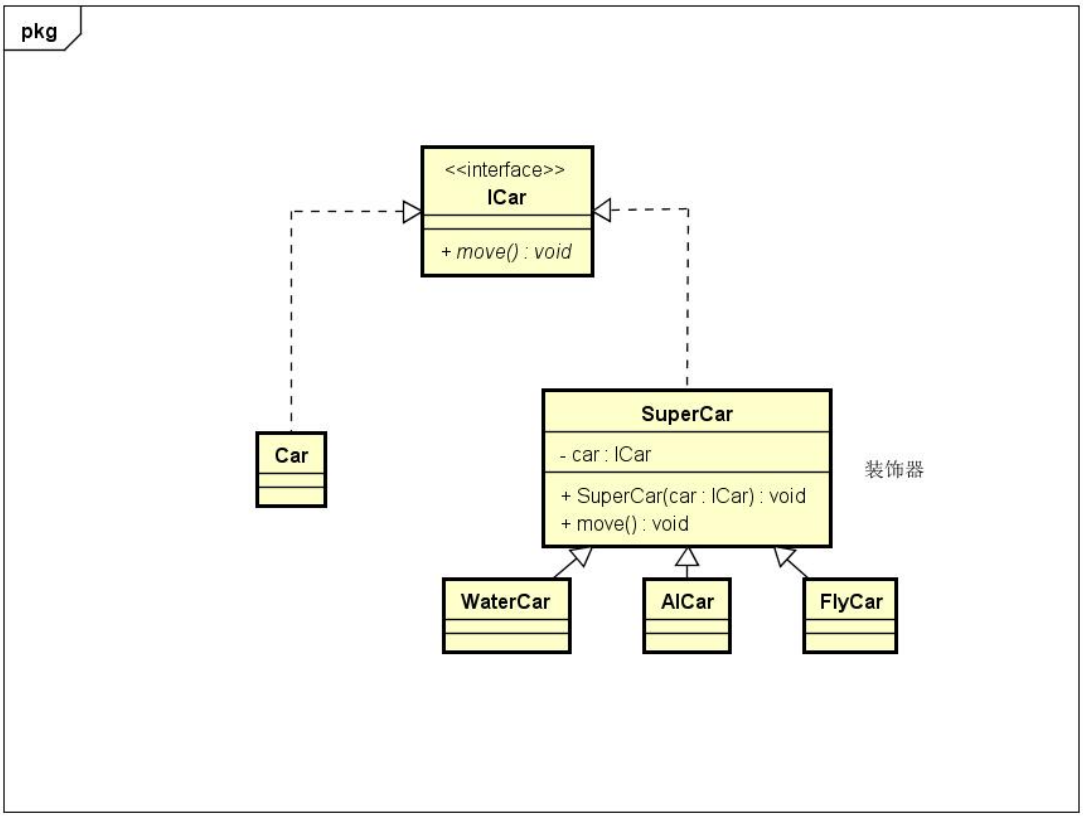
组合模式用于树形数据结构的处理；

3.2.4 装饰模式

1. 角色

- Component抽象构件角色：
  - 真实对象和装饰对象有相同的接口。这样，客户端对象就能够以与真实对象相同的方式同装饰对象交互。
- ConcreteComponent 具体构件角色(真实对象)：
  - io流中的FileInputStream、FileOutputStream
- Decorator装饰角色：
  - 持有一个抽象构件的引用。装饰对象接受所有客户端的请求，并把这些请求转发给真实的对象。这样，就能在真实对象调用前后增加新的功能。
- ConcreteDecorator具体装饰角色：
  - 负责给构件对象增加新的责任。

2. 类图



3.实现代码

ICar.java

```
package design.pattern.structure.decorator;

/**
 * 抽象组件
 * @author shenjy
 *
 */
public interface ICar {
    public void move();
}
```

Car.java

```
package design.pattern.structure.decorator;

/**
 * 具体构件对象，一般车所具备的功能
 * @author shenjy
 *
 */
```

```java
public class Car implements ICar {
    public void move() {
        System.out.println("陆地上跑...");
    }
}
```

SuperCar.java

```java
package design.pattern.structure.decorator;

/**
 * 装饰器对象，给一般的车添加新的功能
 * @author shenjy
 *
 */
public class SuperCar implements ICar {
    protected ICar car;

    public SuperCar(ICar car) {
        this.car = car;
    }
    public void move() {
        car.move();
    }
}
```

WaterCar.java

```java
package design.pattern.structure.decorator;
/**
 * 具体装饰角色
 * @author shenjy
 *
 */
public class WaterCar extends SuperCar {
    public WaterCar(ICar car) {
        super(car);
    }
    public void swim() {
        System.out.println("水里开...");
    }
    public void move() {
        super.move();
        this.swim();
    }
}
```

FlyCar.java

```
package design.pattern.structure.decorator;

/**
 * 具体装饰角色
 * @author shenjy
 *
 */
public class FlyCar extends SuperCar {
    public FlyCar(ICar car) {
        super(car);
    }
    public void fly() {
        System.out.println("天上飞...");
    }
    public void move() {
        super.move();
        this.fly();
    }
}
```

AICar.java

```
package design.pattern.structure.decorator;

/**
 * 具体装饰角色
 * @author shenjy
 *
 */
public class AICar extends SuperCar {
    public AICar(ICar car) {
        super(car);
    }

    public void autoDrive() {
        System.out.println("自动驾驶...");
    }
    public void move() {
        super.move();
        this.autoDrive();
    }
}
```

Client.java

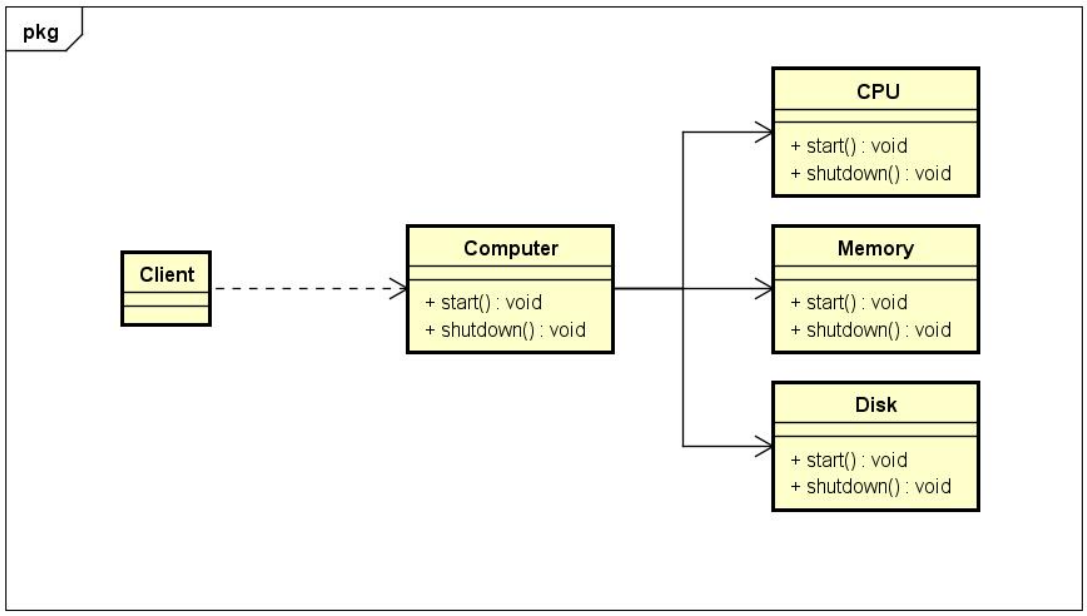
```java
package design.pattern.structure.decorator;

public class Client {
    public static void main(String[] args) {
        // 具体被装饰构建，将Car换成ICar，结果是一样的
        Car car = new Car();

        // 给汽车加在水上开的功能
        WaterCar waterCar = new WaterCar(car);
        // 给汽车加上天上飞的功能
        FlyCar flyCar = new FlyCar(waterCar);
        // 给车添加自动驾驶的功能
        AICar aiCar = new AICar(flyCar);
        aiCar.move();
    }
}
```

4.总结：

装饰器模式是一种用于代替继承的技术，无须通过继承增加子类就可以扩展对象的新功能。是对象的关系代替继承关系，更加灵活，同时避免类型体系的快速膨胀。其功能是动态地位一个对象添加一个新的功能。

3.2.5 外观模式

1. 类图



2.代码实现

Computer.java

```java
package design.pattern.structure.facade;

public class Computer {
    private Disk disk;
    private CPU cpu;
    private Memory memory;

    public Computer() {
        disk = new Disk();
        cpu = new CPU();
        memory = new Memory();
    }
    public void start() {
        System.out.println("电脑启动...");
        disk.start();
        cpu.start();
        memory.start();
        System.out.println("电脑已启动...");
    }
    public void shutdown() {
        System.out.println("电脑关机...");
        disk.shutdown();
        cpu.shutdown();
        memory.shutdown();
        System.out.println("电脑已关机...");
    }
}
```

Disk.java

```java
package design.pattern.structure.facade;

public class Disk {
    public void start() {
        System.out.println("disk start...");
    }
    public void shutdown() {
        System.out.println("disk shutdown...");
    }
}
```

CPU.java

```java
package design.pattern.structure.facade;
```

```java
public class CPU {
    public void start() {
        System.out.println("cpu start...");
    }
    public void shutdown() {
        System.out.println("cpu shutdown...");
    }
}
```

Memory.java

```java
package design.pattern.structure.facade;

public class Memory {
    public void start() {
        System.out.println("memory start...");
    }
    public void shutdown() {
        System.out.println("memory shutdown...");
    }
}
```

Computer.java

```java
package design.pattern.structure.facade;

public class Computer {
    private Disk disk;
    private CPU cpu;
    private Memory memory;

    public Computer() {
        disk = new Disk();
        cpu = new CPU();
        memory = new Memory();
    }
    public void start() {
        System.out.println("电脑启动...");
        disk.start();
        cpu.start();
        memory.start();
        System.out.println("电脑已启动...");
    }
    public void shutdown() {
        System.out.println("电脑关机...");
        disk.shutdown();
```

```
        cpu.shutdown();
        memory.shutdown();
        System.out.println("电脑已关机...");
    }
}
```

Client.java

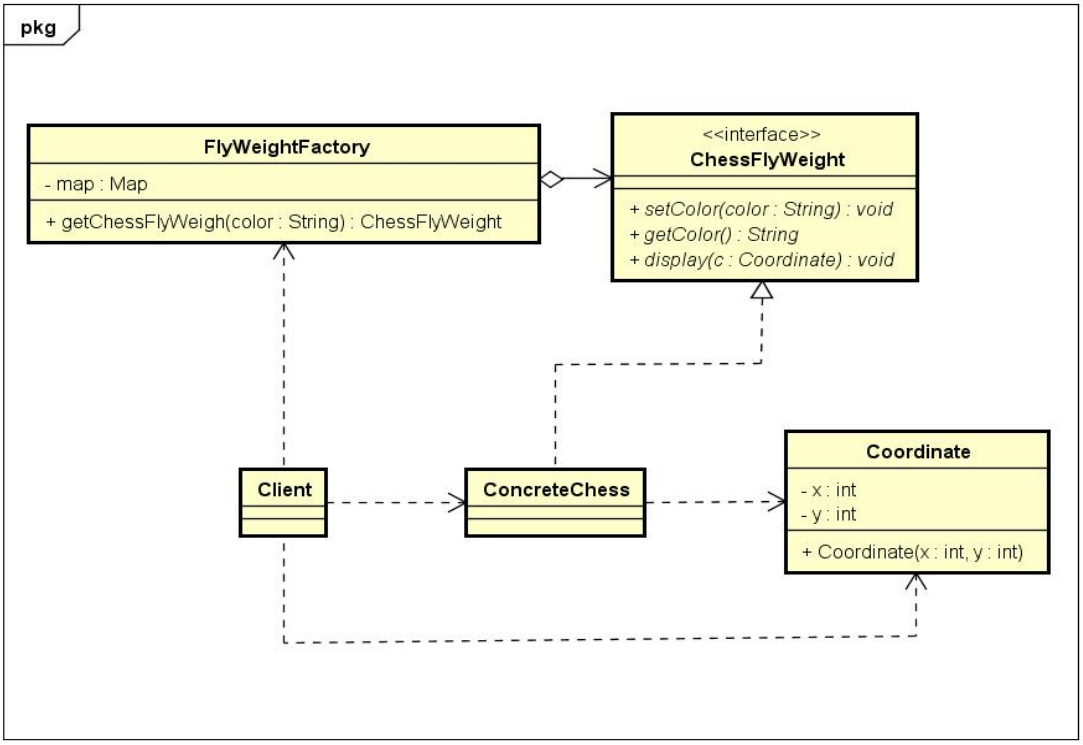
```
package design.pattern.structure.facade;

public class Client {
    public static void main(String[] args) {
        Computer c = new Computer();
        c.start();
        c.shutdown();
    }
}
```

3. 总结

a）隐藏了系统的复杂性，并向客户端提供了一个可以访问系统的接口。使用了该外观模式后，会使用户和部件之间解耦。

3.2.6 享元模式

1. 类图



2.代码实现

ChessFlyWeight.java

```
package design.pattern.structure.flyweight;

/**
 * 享元类
 * @author shenjy
 *
 */
public interface ChessFlyWeight {
    public void setColor(String color);
    public String getColor();
    public void display(Coordinate c);
}
```

ConcreteChess.java

```
package design.pattern.structure.flyweight;

/**
 * 具体享元类
 * @author shenjy
 *
 */
public class ConcreteChess implements ChessFlyWeight {

    private String color;

    public ConcreteChess(String color) {
        this.color = color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void display(Coordinate c) {
        System.out.println("棋子颜色: " + color);
        System.out.println("棋子位置: " + c.getX() + ", " + c.getY());
    }
}
```

Coordinate.java

```java
package design.pattern.structure.flyweight;

/**
 * 外部状态UnSharedConcreteFlyWeight
 * @author shenjy
 *
 */
public class Coordinate {
    int x, y;

    public Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /**
     * @return the x
     */
    public int getX() {
        return x;
    }

    /**
     * @param x the x to set
     */
    public void setX(int x) {
        this.x = x;
    }

    /**
     * @return the y
     */
    public int getY() {
        return y;
    }

    /**
     * @param y the y to set
     */
    public void setY(int y) {
        this.y = y;
    }
}
```

ChessFlyWeightFactory.java

```
package design.pattern.structure.flyweight;

import java.util.HashMap;
import java.util.Map;

/**
 * 享元工厂类
 * @author shenjy
 *
 */
public class ChessFlyWeightFactory {
    private static Map<String, ChessFlyWeight> map = new HashMap<String, ChessFlyWeight>();

    public static ChessFlyWeight getChess(String color) {
        if (map.get(color) != null) {
            return map.get(color);
        } else {
            ChessFlyWeight cfw = new ConcreteChess(color);
            map.put(color, cfw);
            return cfw;
        }
    }
}
```

Client.java

```
package design.pattern.structure.flyweight;

public class Client {
    public static void main(String[] args) {
        ChessFlyWeight cfw = ChessFlyWeightFactory.getChess("黑色");
        ChessFlyWeight cfw2 = ChessFlyWeightFactory.getChess("黑色");
        System.out.println(cfw.equals(cfw2));

        // 增加外部状态的处理
        System.out.println("外部状态的处理");
        cfw.display(new Coordinate(10, 10));
        cfw.display(new Coordinate(20, 20));
    }
}
```

3.总结：

- FlyweightFactory享元工厂类
  - 创建并管理享元对象，享元池一般设计成键值对
- FlyWeight抽象享元类
  - 通常是一个接口或抽象类，声明公共方法，这些方法可以向外界提供对象的内部状态，设置外部状态。
- ConcreteFlyWeight具体享元类
  - 为内部状态提供成员变量进行存储
- UnsharedConcreteFlyWeight非共享享元类
  - 不能被共享的子类可以设计为非共享享元类

a）内存输入稀缺资源，不要随便浪费。如果有很多完全相同或相似的对象，我们可以通过享元模式来节省内存，以共享的方式高效地支持大量细粒度对象的重用。

b） 关键：

a) 内部状态：可共享，不会随环境的变化而变化；

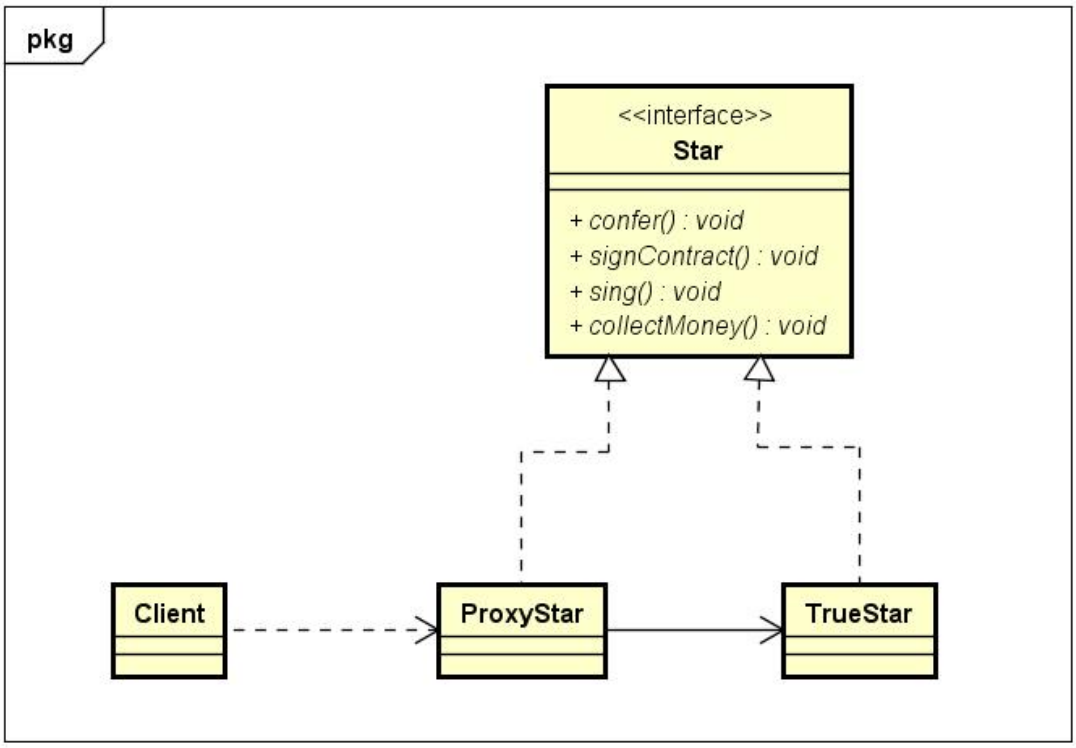
b) 外部状态：不可共享，会随环境的变化而变化；

3.2.7 代理模式

1.代理模式分为静态代理和动态代理。

2.静态代理模式

1） 类图



Star.java

```
package design.pattern.structure.staticproxy;

public interface Star {
    /**
     * 洽谈
     */
    public void confer();
    /**
     * 签合同
     */
    public void signContract();
    /**
     * 订票
     */
    public void bookTickit();
    /**
     * 唱歌
     */
    public void sing();
    /**
     * 收钱
     */
    public void collectMoney();
}
```

ProxyStar.java

```
package design.pattern.structure.staticproxy;

public class ProxyStar implements Star {

    private Star star;

    public ProxyStar(Star star) {
        this.star = star;
    }
    public void confer() {
        System.out.println("Proxy confer...");
    }
    public void signContract() {
        System.out.println("Proxy signContract...");
    }
    public void bookTickit() {
        System.out.println("Proxy bookTickit...");
```

```
    }
    public void sing() {
        star.sing();
    }
    public void collectMoney() {
        System.out.println("Proxy collectMoney...");
    }
}
```

TrueProxy.java

```
package design.pattern.structure.staticproxy;

public class TrueStar implements Star {

    public void confer() {
        System.out.println("TrueStar confer...");
    }

    public void signContract() {
        System.out.println("TrueStar signContract...");
    }

    public void bookTickit() {
        System.out.println("TrueStar bookTickit...");
    }

    public void sing() {
        System.out.println("TrueStar sing...");
    }

    public void collectMoney() {
        System.out.println("TrueStar collectMoney...");
    }
}
```

Client.java

```
package design.pattern.structure.staticproxy;

public class Client {
    public static void main(String[] args) {
        Star star = new TrueStar();
        ProxyStar proxy = new ProxyStar(star);

        proxy.confer();
```

```
        proxy.signContract();
        proxy.bookTickit();
        proxy.sing();
        proxy.collectMoney();
    }
}
```

3.动态代理代码实现

Star.java

```
package design.pattern.structure.dynamic.proxy;

public interface Star {
    /**
     * 洽谈
     */
    public void confer();
    /**
     * 签合同
     */
    public void signContract();
    /**
     * 订票
     */
    public void bookTickit();
    /**
     * 唱歌
     */
    public void sing();
    /**
     * 收钱
     */
    public void collectMoney();
}
```

StarHandler.java

```
package design.pattern.structure.dynamic.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class StarHandler implements InvocationHandler {
    /**
     * 声明被代理对象
     */
```

```
    private Star star;

    public StarHandler(Star star) {
        this.star = star;
    }

    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        method.invoke(star, args);
        return null;
    }
}
```

TrueStar.java

```
package design.pattern.structure.dynamic.proxy;

public class TrueStar implements Star {

    public void confer() {
        System.out.println("TrueStar confer...");
    }

    public void signContract() {
        System.out.println("TrueStar signContract...");
    }

    public void bookTickit() {
        System.out.println("TrueStar bookTickit...");
    }

    public void sing() {
        System.out.println("TrueStar sing...");
    }

    public void collectMoney() {
        System.out.println("TrueStar collectMoney...");
    }
}
```

Client.java

```
package design.pattern.structure.dynamic.proxy;

import java.lang.reflect.Proxy;
```

```
public class Client {
    public static void main(String[] args) {
        Star star = new TrueStar();
        StarHandler sh = new StarHandler(star);
        Star proxy = (Star)
Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(), new
Class[] {Star.class}, sh);
        proxy.confer();
        proxy.sing();
    }
}
```

4.动态代理总结

1）可对任意对象，任意接口中的方法，添加任意的实现逻辑。

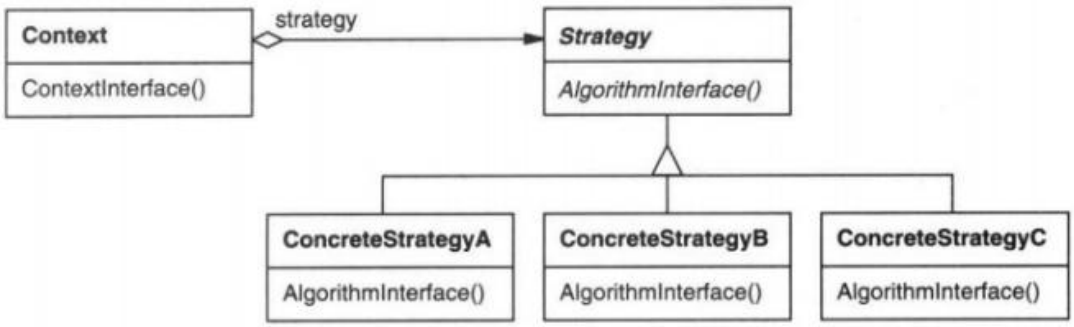
2）动态代理的应用： AOP，日志，事务配置。

3.3 行为型模式

行为型模式包括策略模式，模板方法模式，观察者模式，迭代器模式，责任链模式，命令模式，备忘录模式，状态模式，访问者模式，中介者模式，解释器模式。

3.3.1 策略模式

1.类图



2.代码实现

Strategy.java

```
package design.pattern.behavior.strategy;
public interface Strategy {
    public int operate(int a, int b);
}
```

AddStrategy.java

```java
package design.pattern.behavior.strategy;

public class AddStrategy implements Strategy {
    public int operate(int a, int b) {
        return a + b;
    }
}
```

MinusStrategy.java

```java
package design.pattern.behavior.strategy;

public class MinusStrategy implements Strategy {
    public int operate(int a, int b) {
        return a - b;
    }
}
```

MultiplyStrategy.java

```java
package design.pattern.behavior.strategy;

public class MultiplyStrategy implements Strategy {
    public int operate(int a, int b) {
        return a * b;
    }
}
```

DivideStrategy.java

```java
package design.pattern.behavior.strategy;

public class DivideStrategy implements Strategy {
    public int operate(int a, int b) {
        if (b == 0) {
            return -1;
        }
        return a / b;
    }
}
```

OperateContext.java

```java
package design.pattern.behavior.strategy;

public class OperateContext {
    private Strategy strategy;
```

```
    public OperateContext(Strategy strategy) {
        this.strategy = strategy;
    }

    public int compute(int a, int b) {
        return strategy.operate(a, b);
    }
}
```

3.总结

环境类(Context):用一个 ConcreteStrategy 对象来配置。维护一个对 Strategy 对象的引用。可定义一个接口来让 Strategy 访问它的数据。

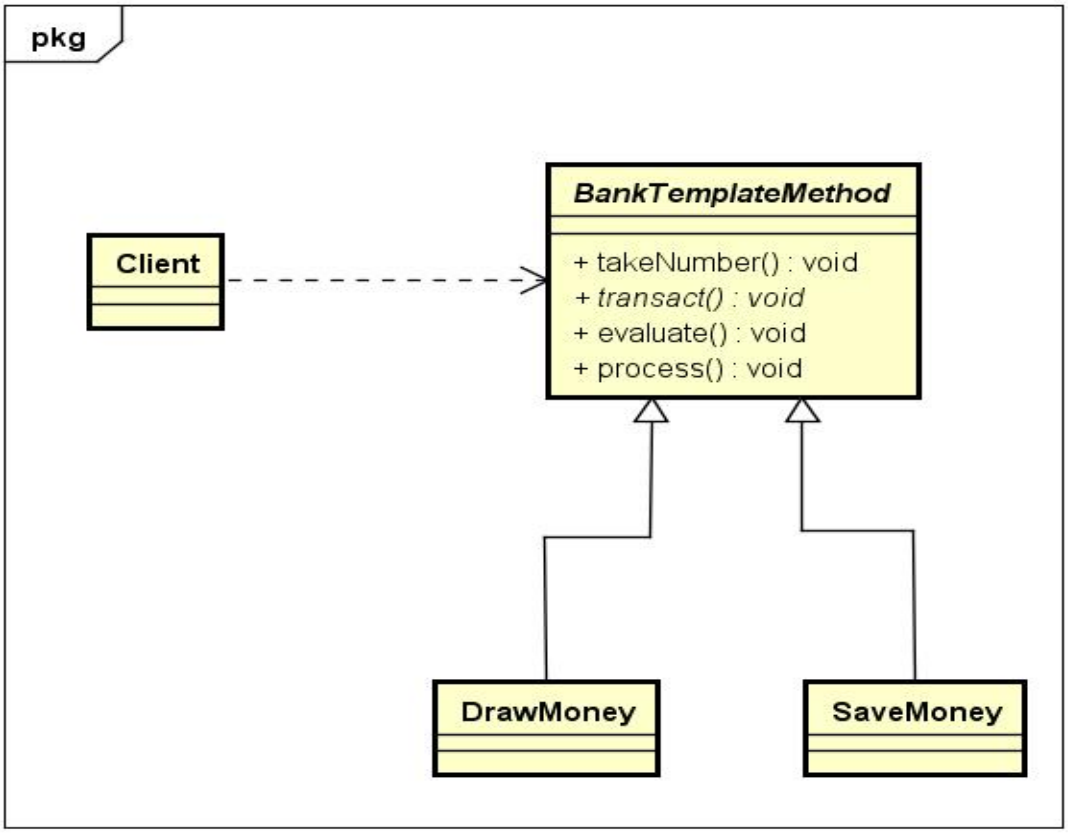
抽象策略类(Strategy):定义所有支持的算法的公共接口。 Context 使用这个接口来调用某 ConcreteStrategy 定义的算法。

具体策略类(ConcreteStrategy):以 Strategy 接口实现某具体算法。

有过有 if…else…，可以考虑使用策略模式解决。

3.3.2 模板方法模式

1.类图



2.代码实现

BankTemplateMethod.java

```
package design.pattern.behavior.template.method;

public abstract class BankTemplateMethod {
    // 具体方法
    public void takeNumber() {
        System.out.println("取号排队...");
    }

    public abstract void transact(); //办理具体的业务

    public void evaluate() {
        System.out.println("反馈评分...");
    }

    // 模板方法，把基本操作组合到一起
    public final void process() {
        this.takeNumber();
        this.transact();  // 钩子函数
        this.evaluate();
    }
}
```

DrawMoney.java

```
package design.pattern.behavior.template.method;

public class DrawMoney extends BankTemplateMethod {
    @Override
    public void transact() {
        System.out.println("取钱...");
    }
}
```

SaveMoney.java

```
package design.pattern.behavior.template.method;

public class SaveMoney extends BankTemplateMethod {
    @Override
    public void transact() {
        System.out.println("存钱...");
    }
}
```

Client.java

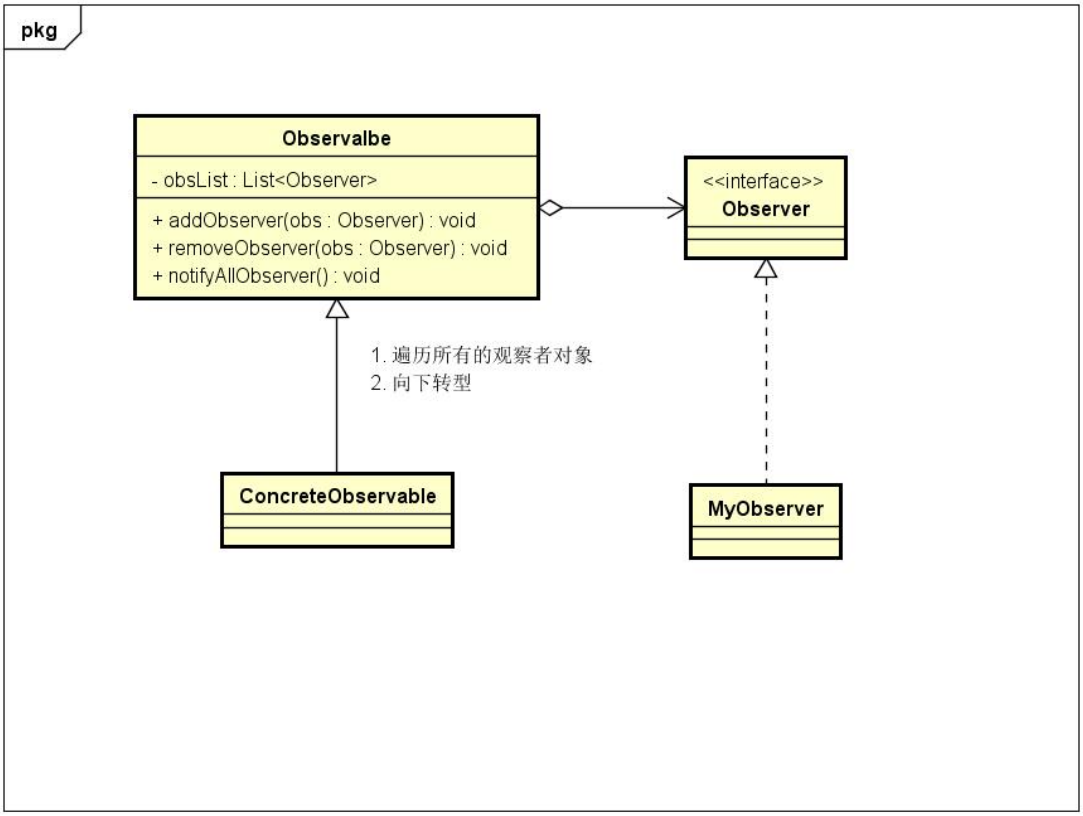
```
package design.pattern.behavior.template.method;

public class Client {
    public static void main(String[] args) {
        BankTemplateMethod btm = new DrawMoney();
        btm.process();

        System.out.println();
        BankTemplateMethod btm2 = new SaveMoney();
        btm2.process();
    }
}
```

3.适用场景

实现一个算法时，整体步骤很固定，但某些部分易变。易变的部分抽象出来，供子类实现。

3.3.3 观察者模式

1.类图



2.代码实现之自己实现观察者模式

Subject.java

```
package design.pattern.behavior.myobserver;

import java.util.ArrayList;
import java.util.List;

public class Subject {
	protected List<Observer> list = new ArrayList<Observer>();

	public void register(Observer o) {
		list.add(o);
	}

	public void removeObserver(Observer o) {
		list.remove(o);
	}

	public void notifyAllObserver() {
		for (Observer observer : list) {
			observer.update(this);
		}
	}
}
```

Observer.java

```
package design.pattern.behavior.myobserver;

public interface Observer {
    public void update(Subject s);
}
```

MyObserver.java

```
package design.pattern.behavior.myobserver;
public class MyObserver implements Observer {
    private int state;

    public int getState() {
        return state;
    }

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

```
    public void update(Subject s) {
        state = ((ConcreteSubject)s).getState();
    }
}
```

ConcreteSubject.java

```
package design.pattern.behavior.myobserver;

public class ConcreteSubject extends Subject {
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        this.notifyAllObserver();
    }
}
```

Client.java

```
package design.pattern.behavior.myobserver;

public class Client {
    public static void main(String[] args) {
        MyObserver o1 = new MyObserver();
        MyObserver o2 = new MyObserver();
        MyObserver o3 = new MyObserver();
        System.out.println(o1.getState());
        System.out.println(o2.getState());
        System.out.println(o3.getState());

        ConcreteSubject s = new ConcreteSubject();
        s.register(o1);
        s.register(o2);
        s.register(o3);
        s.setState(3000);
        System.out.println("state changed...");
        System.out.println(o1.getState());
        System.out.println(o2.getState());
        System.out.println(o3.getState());
    }
}
```

3.代码实现-基于 JDK 中 Observer 接口和 Observable.java 类

MyObserver.java

```
package design.pattern.behavior.observer;

import java.util.Observable;
import java.util.Observer;

public class MyObserver implements Observer {

    private int state;

    public int getState() {
        return state;
    }

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

    public void update(Observable o, Object arg) {
        state = ((ConcreteSubject)o).getState();
    }
}
```

ConcreteSubject.java

```
package design.pattern.behavior.observer;
import java.util.Observable;

public class ConcreteSubject extends Observable {
    private int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
        this.setChanged();
        this.notifyObservers();
    }
}
```

Client.java

```
package design.pattern.behavior.observer;
```

```java
public class Client {
    public static void main(String[] args) {
        MyObserver o1 = new MyObserver();
        MyObserver o2 = new MyObserver();
        MyObserver o3 = new MyObserver();

        System.out.println(o1.getState());
        System.out.println(o2.getState());
        System.out.println(o3.getState());

        ConcreteSubject s = new ConcreteSubject();
        s.addObserver(o1);
        s.addObserver(o2);
        s.addObserver(o3);

        s.setState(3000);

        System.out.println("state changed...");
        System.out.println(o1.getState());
        System.out.println(o2.getState());
        System.out.println(o3.getState());
    }
}
```

4.使用场景

1)Servlet 中的监听器

2)Zookeeper 框架中

3)游戏中的广播机制

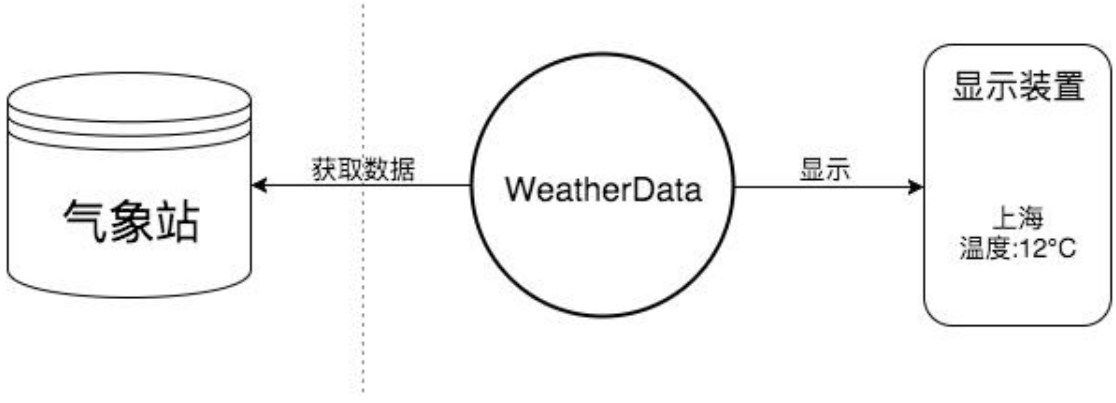
4)事件监听机制

5.补充内容

时间、地点、任务、事件的起因、经过和结果。

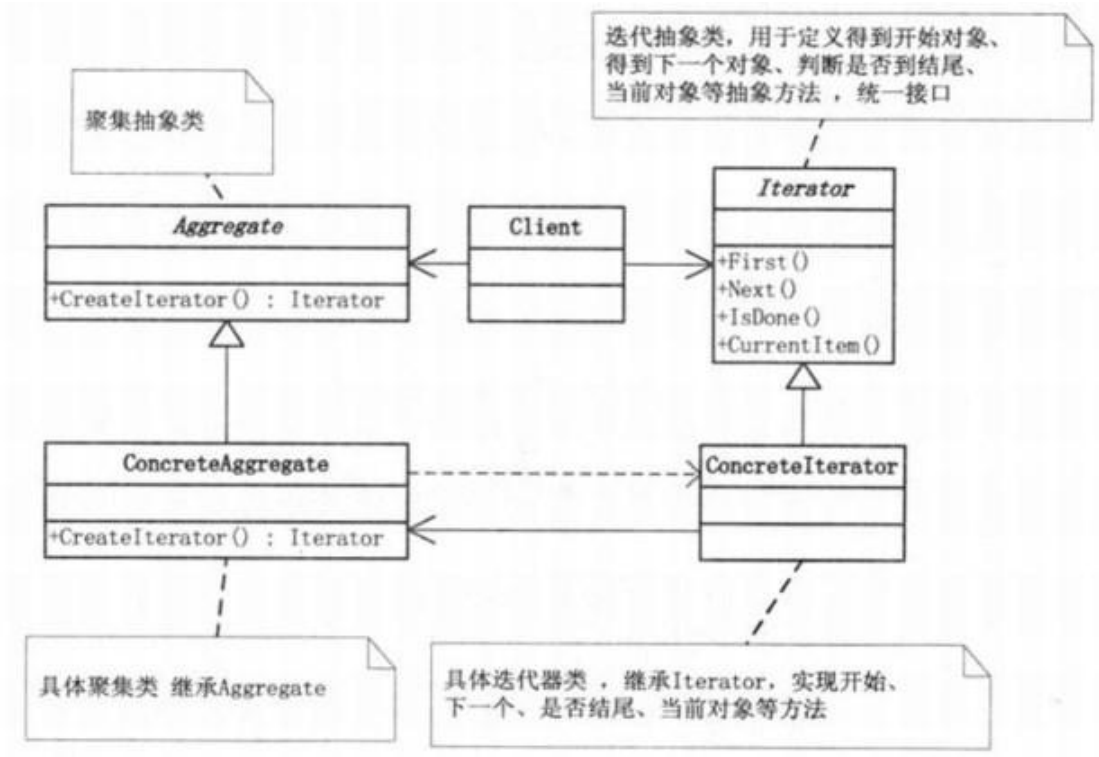
根据事情的具体情况来做相应的处理。事件处理

缓存



3.3.4 迭代器模式

1.类图



2.代码实现

MyIterator.java

```
package design.pattern.behavior.iterator;
public interface MyIterator {
    public void first();// 将游标指向第一个元素
    public void next();// 将游标指向下一个元素
    public boolean hasNext();// 判断是否有下一个元素
    public boolean isFirst();// 判断是否是第一个元素
    public boolean isLast();// 判断是否是最后一个元素
    public Object getCurrentObject();// 得到当前对象
}
```

ConcreteMyAggregate.java

```
package design.pattern.behavior.iterator;

import java.util.ArrayList;
import java.util.List;

public class ConcreteMyAggregate {
    private List<Object> list = new ArrayList<Object>();
```

```
public void addObject(Object obj) {
    this.list.add(obj);
}
public void removeObject(Object obj) {
    this.list.remove(obj);
}
public List<Object> getList() {
    return list;
}
public void setList(List<Object> list) {
    this.list = list;
}
// 获得迭代器
public MyIterator createIterator() {
    return new ConcreteIterator();
}
// 使用内部类来定义迭代器，好处就是可以直接使用外部类的属性
private class ConcreteIterator implements MyIterator {
    private int cursor;// 定义一个迭代器游标
    public void first() {
        cursor = 0;
    }
    public void next() {
        if (cursor < list.size()) {
            cursor++;
        }
    }
    public boolean hasNext() {
        // 如果游标<list 的大小，则说明还有下一个
        if (cursor < list.size()) {
            return true;
        }
        return false;
    }
    public boolean isFirst() {
        return cursor == 0 ? true : false;
    }
    public boolean isLast() {
        // 判断游标是否是容器的最后一个
        return cursor == (list.size() - 1) ? true : false;
    }
    public Object getCurrentObject() {
        return list.get(cursor);// 获取当前游标指向的元素
    }
```

```
    }
}
```

Client.java

```
package design.pattern.behavior.iterator;

public class Client {
    public static void main(String[] args) {
        ConcreteMyAggregate cma = new ConcreteMyAggregate();
        cma.addObject("111");
        cma.addObject("222");
        cma.addObject("333");
        cma.addObject("444");

        MyIterator iterator = cma.createIterator();
        // 如果删除一个元素的话，迭代的时候也同样会被删除
        cma.removeObject("111");
        while (iterator.hasNext()) {
            // 获取当前对象
            System.out.println(iterator.getCurrentObject());
            iterator.next();// 将游标向下移
        }
    }
}
```

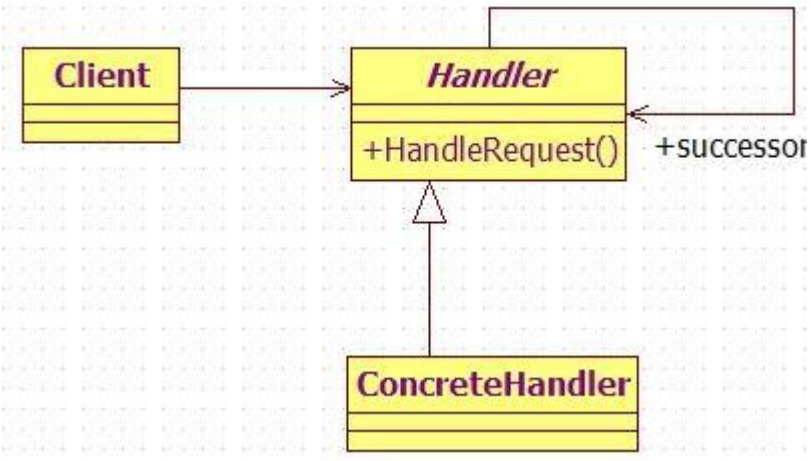
3.总结

聚合对象的方式，游标模式；

聚合对象的作用是存数据，迭代器的作用是遍历的对象；

3.3.5 责任链模式

1.类图



2.代码实现

Leader.java

```
package design.pattern.behavior.responsibilitychain;

/**
 * 抽象类
 * @author shenjy
 *
 */
public abstract class Leader {
    protected String name;
    protected Leader nextLeader;
    public Leader(String name) {
        super();
        this.name = name;
    }

    /**
     * 后继对象
     * @param nextLeader
     */
    public void setNextLeader(Leader nextLeader) {
        this.nextLeader = nextLeader;
    }

    public abstract void handleRequest(LeaveRequest
leaveRequest);

}
```

Director.java

```
package design.pattern.behavior.responsibilitychain;

public class Director extends Leader {
    public Director(String name) {
        super(name);
    }
    @Override
    public void handleRequest(LeaveRequest leaveRequest) {
        if (leaveRequest.getDays() < 3) {
            // 自己审批
            System.out.println("Director agreed...");
        } else if (this.nextLeader != null) {
            this.nextLeader.handleRequest(leaveRequest);
```

```
        }
    }
}
```

Manager.java

```
package design.pattern.behavior.responsibilitychain;

public class Manager extends Leader {
    public Manager(String name) {
        super(name);
    }

    @Override
    public void handleRequest(LeaveRequest leaveRequest) {
        if (leaveRequest.getDays() >= 3 && leaveRequest.getDays()
< 10) {
            // 自己审批
            System.out.println("Manager agreed...");
        } else if (this.nextLeader != null) {
            this.nextLeader.handleRequest(leaveRequest);
        }
    }
}
```

GeneralManger.java

```
package design.pattern.behavior.responsibilitychain;

public class GeneralManager extends Leader {
    public GeneralManager(String name) {
        super(name);
    }

    @Override
    public void handleRequest(LeaveRequest leaveRequest) {
        if (leaveRequest.getDays() < 30) {
            // 自己审批
            System.out.println("GeneralManager agreed...");
        } else {
            System.out.println("离职...");
        }
    }
}
```

LeaveRequest.java

```java
package design.pattern.behavior.responsibilitychain;

/**
 * 封装请假的基本信息
 * @author shenjy
 *
 */
public class LeaveRequest {
    private String employeeName;
    private int days;
    private String reasionMsg;

    public LeaveRequest(String employeeName, int days, String
reasionMsg) {
        super();
        this.employeeName = employeeName;
        this.days = days;
        this.reasionMsg = reasionMsg;
    }

    public String getEmployeeName() {
        return employeeName;
    }

    public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
    }

    public int getDays() {
        return days;
    }

    public void setDays(int days) {
        this.days = days;
    }

    public String getReasionMsg() {
        return reasionMsg;
    }
    public void setReasionMsg(String reasionMsg) {
        this.reasionMsg = reasionMsg;
    }

}
```

Client.java

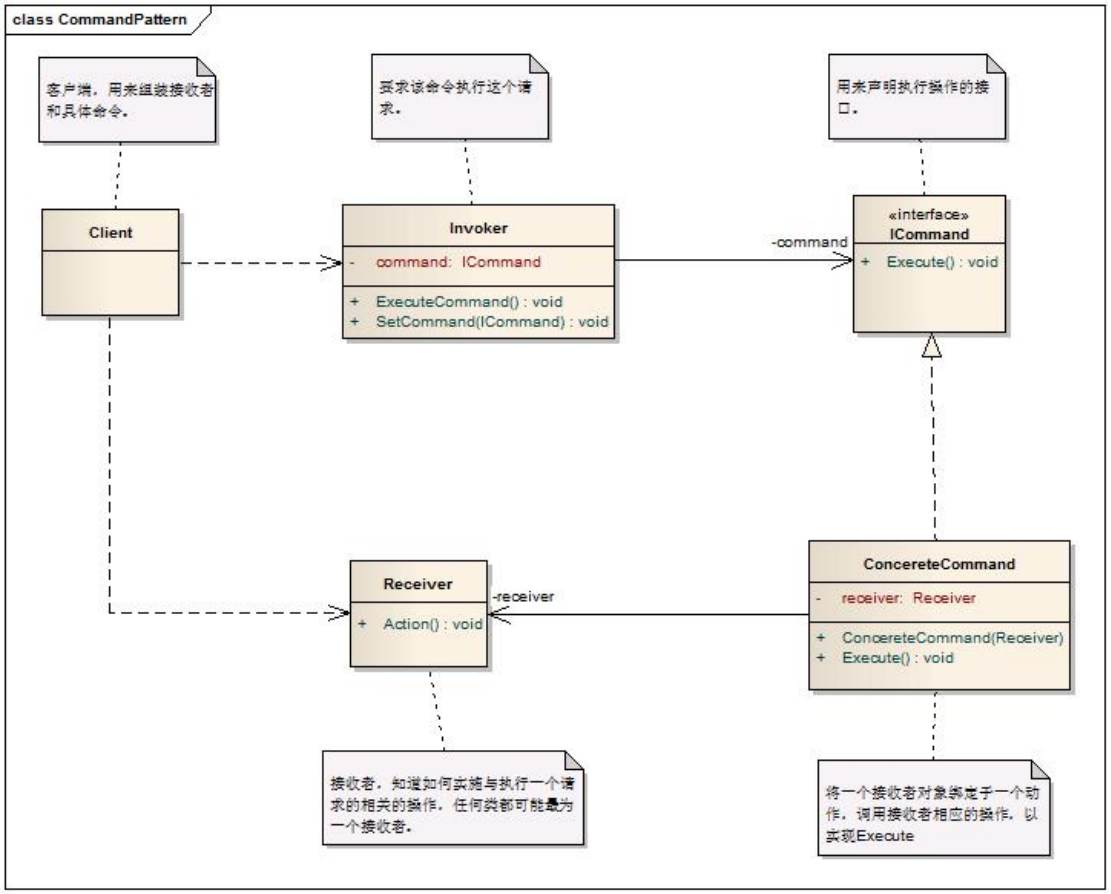
```
package design.pattern.behavior.responsibilitychain;

public class Client {
    public static void main(String[] args) {
        LeaveRequest lr = new LeaveRequest("alan", 3, "休假调
整...");
        Leader d = new Director("张三");
        Leader m = new Manager("李四");
        Leader gm = new GeneralManager("王五");

        d.setNextLeader(m);
        m.setNextLeader(gm);
        d.handleRequest(lr);
    }
}
```

3.应用场景：

Servlet 中的 Filter

Struts 中的过滤器

3.3.6 命令模式



2.代码实现

Invoke.java

```
package design.pattern.behavior.command;
/**
 * 命令发起者
 * @author shenjy
 */
public class Invoke {
    // 也可以是多个命令（多条sql语句）
    private Command command;
    public Invoke(Command command) {
        this.command = command;
    }
    // 业务方法，调用命令类的方法
    public void call() {
        command.execute();
    }
}
```

Command.java

```java
package design.pattern.behavior.command;
public interface Command {
    public void execute();
}
```

ConcreteCommand.java

```java
package design.pattern.behavior.command;
public class ConcreteCommand implements Command {
    private Receiver receiver; // 命令真正执行者
    public ConcreteCommand(Receiver receiver) {
        super();
        this.receiver = receiver;
    }
    public void execute() {
        // 命令执行前后，执行相关处理
        receiver.action();
    }
}
```

Receiver.java

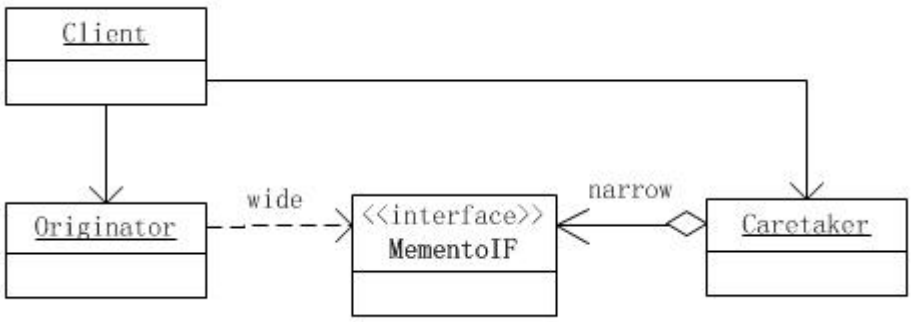
```java
package design.pattern.behavior.command;
/**
 * 真正命令执行者
 * @author shenjy
 *
 */
public class Receiver {
    public void action() {
        System.out.println("Receiver.action...");
    }
}
```

Client.java

```java
package design.pattern.behavior.command;

public class Client {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        command.execute();
    }
}
```

3.3.7 备忘录模式

1.类图



2.代码实现

BrowserHistory.java

```
package design.pattern.behavior.memento;

/**
 * 浏览器的浏览历史(源发器类)
 * @author shenjy
 *
 */
public class BrowserHistory {
    private String title;
    private String url;
    // 备忘操作，返回备忘操作
    public BrowserHistoryMemento memento() {
        return new BrowserHistoryMemento(this);
    }
    // 恢复成指定的值
    public void recovery(BrowserHistoryMemento bhm) {
        this.title = bhm.getTitle();
        this.url = bhm.getUrl();
    }

    public BrowserHistory(String title, String url) {
        this.title = title;
        this.url = url;
    }

    public String getTitle() {
        return title;
    }
```

```
    public void setTitle(String title) {
        this.title = title;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }
}
```

BrowserHistoryMemento.java

```
package design.pattern.behavior.memento;

public class BrowserHistoryMemento {
    private String title;
    private String url;

    public BrowserHistoryMemento(BrowserHistory bh) {
        this.title = bh.getTitle();
        this.url = bh.getUrl();
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

}
```

CareTaker.java

```
package design.pattern.behavior.memento;

public class CareTaker {
    private BrowserHistoryMemento bhm;
    public BrowserHistoryMemento getBhm() {
        return bhm;
    }
    public void setBhm(BrowserHistoryMemento bhm) {
        this.bhm = bhm;
    }
}
```

Client.java

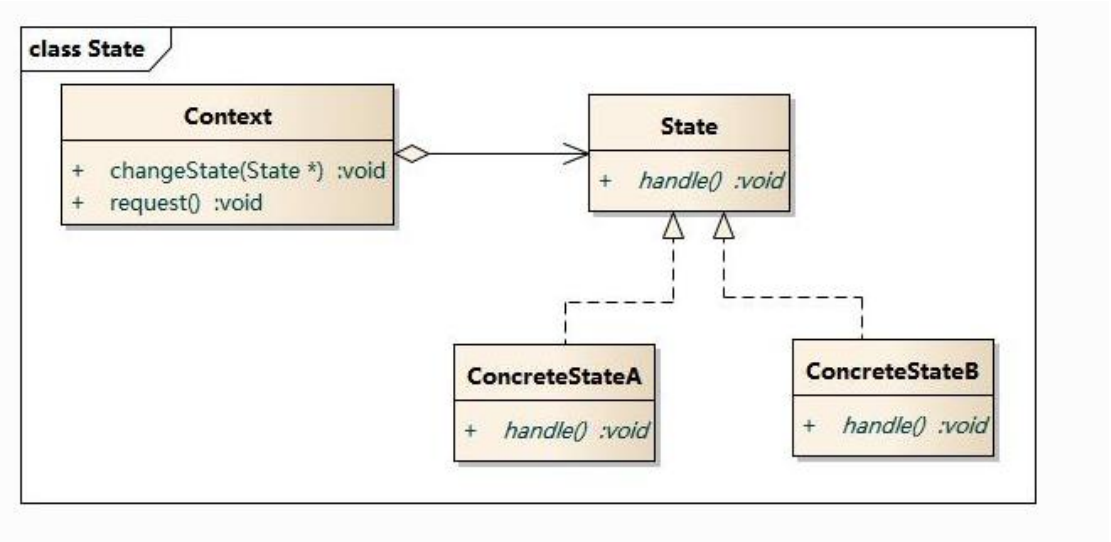
```
package design.pattern.behavior.memento;

public class Client {
    public static void main(String[] args) {
        CareTaker ct = new CareTaker();

        BrowserHistory bh = new BrowserHistory("倚天屠龙记",
"aiqiyi.com");
        System.out.println(bh.getTitle() + bh.getUrl());
        ct.setBhm(bh.memento());

        bh.setTitle("爱情睡醒了");
        bh.setUrl("baofengyingyin.com");
        System.out.println(bh.getTitle() + bh.getUrl());

        bh.recovery(ct.getBhm());
        System.out.println(bh.getTitle() + bh.getUrl());
    }
}
```

3.应用场景

事务回滚操作；
象棋游戏中悔棋；
软件中撤销；
浏览器中的浏览历史；
Maven 的快照
所有软件中的 ctrl + z 操作

3.3.8 状态模式

1.类图



2.代码实现

IWaterState.java

```
package design.pattern.behavior.state;

public interface IWaterState {
    void printState();
}
```

IceWaterState.java

```
package design.pattern.behavior.state;

public class IceWaterState implements IWaterState {
    public void printState() {
        System.out.println("Now state: Ice Water");
    }
}
```

WarmWaterState.java

```
package design.pattern.behavior.state;

public class WarmWaterState implements IWaterState {
    public void printState() {
        System.out.println("Now state: Warm Water");
    }
}
```

BoilingWaterState.java

```
package design.pattern.behavior.state;

public class BoilingWaterState implements IWaterState {
    public void printState() {
        System.out.println("Now state: Boiling Water");
    }
}
```

WaterContext.java

```
package design.pattern.behavior.state;

public class WaterContext {
    private IWaterState mIWaterState;
    public IWaterState getIWaterState() {
        return mIWaterState;
    }
    public void setIWaterState(int i) {
        if (i == 0) {
            mIWaterState = new IceWaterState();
            return;
        }
        if (i == 1) {
            mIWaterState = new WarmWaterState();
            return;
        }
        if (i == 2) {
            mIWaterState = new BoilingWaterState();
            return;
        }
    }
}
```

Client.java

```
package design.pattern.behavior.state;

public class Client {
    public static void main(String[] args) {
        IWaterState iWaterState;
        WaterContext waterContext = new WaterContext();
        // 模拟状态改变
        for (int i = 0; i < 3; i++) {
            waterContext.setIWaterState(i);
            iWaterState = waterContext.getIWaterState();
```

```
            System.out.println("i=" + i);
            iWaterState.printState();
        }
    }
}
```

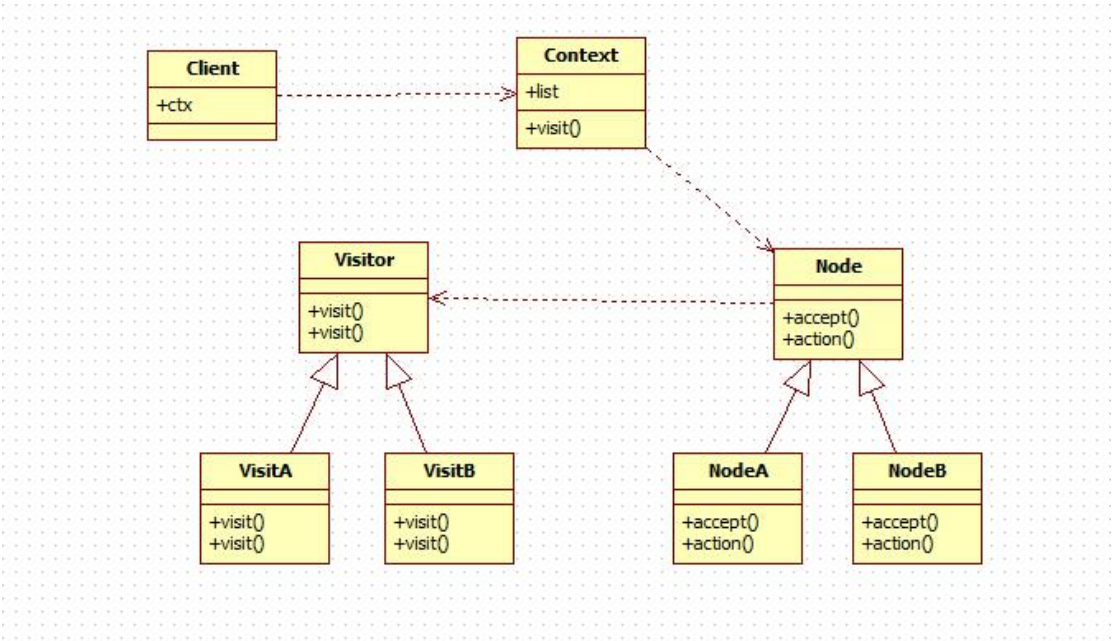
3.应用场景

订单状态；

水的状态；

3.3.9 访问者模式

1.类图



2.代码实现

Context.java

```
package design.pattern.behavior.visitor;

import java.util.List;
import java.util.ArrayList;

public class Context {
    List<Node> list = new ArrayList<Node>();

    public void add(Node node) {
        list.add(node);
    }
```

```
    public void visit(Visitor visitor) {
        for(Node node : list){
            node.accept(visitor);
        }
    }
}
```

Node.java

```
package design.pattern.behavior.visitor;

public interface Node {
   public void accept(Visitor visitor);
}
```

NodeA.java

```
package design.pattern.behavior.visitor;

public class NodeA implements Node {
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
   public void action(){
       System.out.println("NodeA visited");
    }
}
```

NodeB.java

```
package design.pattern.behavior.visitor;
public class NodeB implements Node {
   public void accept(Visitor visitor) {
      visitor.visit(this);
   }
   public void action() {
      System.out.println("NodeB visited");
   }
}
```

Visitor.java

```
package design.pattern.behavior.visitor;
public interface Visitor {
   public void visit(NodeA nodeA);
    public void visit(NodeB nodeB);
}
```

VisitA.java

```
package design.pattern.behavior.visitor;

public class VisitA implements Visitor {
    public void visit(NodeA nodeA) {
        System.out.println("***visitA***");
        nodeA.action();
    }

    public void visit(NodeB nodeB) {
        System.out.println("***visitA***");
        nodeB.action();
    }
}
```

VisitB.java

```
package design.pattern.behavior.visitor;

public class VisitB implements Visitor {
    public void visit(NodeA nodeA) {
        System.out.println("***visitB***");
        nodeA.action();
    }

    public void visit(NodeB nodeB) {
        System.out.println("***visitB***");
        nodeB.action();
    }
}
```

Client.java

```
package design.pattern.behavior.visitor;
public class Client {
    private static Context ctx = new Context();

    public static void main(String[] args) {
        ctx.add(new NodeA());
        ctx.add(new NodeB());
        ctx.visit(new VisitA());
        ctx.visit(new VisitB());
    }
}
```

3.应用场景

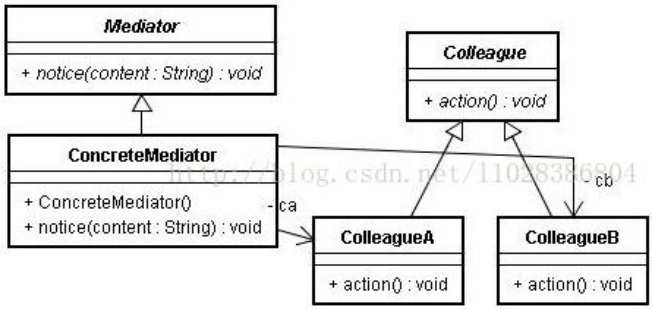
1）不同的子类，依赖于不同的其他对象

2）需要对一组对象，进行许多不相关的操作，又不想在类中是现在这些方法

3）定义的类很少改变，但是执行的操作却经常发生改变。

3.3.10 中介者模式

1.类图



2.代码实现

Colleague.java

```
package design.pattern.behavior.mediator;
public abstract class Colleague {
    public abstract void action();
}
```

ColleagueA.java

```
package design.pattern.behavior.mediator;
public class ColleagueA extends Colleague {
    @Override
    public void action() {
        System.out.println("普通员工努力工作...");
    }
}
```

ColleagueB.java

```
package design.pattern.behavior.mediator;

public class ColleagueB extends Colleague {
    @Override
    public void action() {
        System.out.println("前台注意了!");
    }
}
```

Mediator.java

```java
package design.pattern.behavior.mediator;
/**
 * Mediator
 * @author shenjy
 */
public abstract class Mediator {
    public abstract void notice(String content);
}
```

ConcreteMediator.java

```java
package design.pattern.behavior.mediator;

public class ConcreteMediator extends Mediator {
    private ColleagueA ca;
    private ColleagueB cb;
    public ConcreteMediator() {
        ca = new ColleagueA();
        cb = new ColleagueB();
    }
    @Override
    public void notice(String content) {
        if (content.equals("boss")) {
            //老板来了，通知员工A
            ca.action();
        }
        if (content.equals("client")) {
            //客户来了，通知前台B
            cb.action();
        }
    }
}
```

Client.java

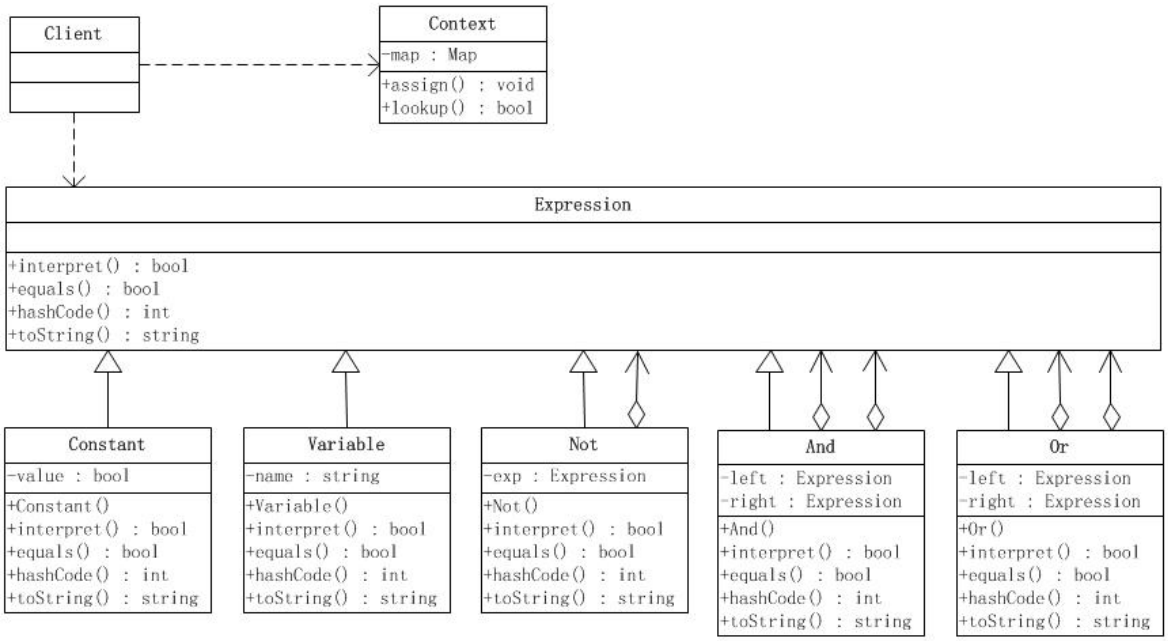
```java
package design.pattern.behavior.mediator;
public class Client {
    public static void main(String[] args) {
        Mediator med = new ConcreteMediator();
        // 老板来了
        med.notice("boss");
        // 客户来了
        med.notice("client");
    }
}
```

3．应用场景

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

3.3.11 解释器模式

1. 类图



2. 实现代码

Expression.java

```
package design.pattern.behavior.interpreter;

public abstract class Expression {
    /**
     * 以环境为准，本方法解释给定的任何一个表达式
     */
    public abstract boolean interpret(Context ctx);

    /**
     * 检验两个表达式在结构上是否相同
     */
    public abstract boolean equals(Object obj);

    /**
     * 返回表达式的hash code
     */
    public abstract int hashCode();
```

```
    /**
     * 将表达式转换成字符串
     */
    public abstract String toString();
}
```

Constant.java

```
package design.pattern.behavior.interpreter;

public class Constant extends Expression {

    private boolean value;

    public Constant(boolean value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj != null && obj instanceof Constant) {
            return this.value == ((Constant) obj).value;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {
        return value;
    }

    @Override
    public String toString() {
        return new Boolean(value).toString();
    }

}
```

Variable.java

```
package design.pattern.behavior.interpreter;
```

```java
public class Variable extends Expression {

    private String name;

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

    @Override
    public boolean equals(Object obj) {

        if (obj != null && obj instanceof Variable) {
            return this.name.equals(((Variable) obj).name);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

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

    @Override
    public boolean interpret(Context ctx) {
        return ctx.lookup(this);
    }

}
```

And.java

```java
package design.pattern.behavior.interpreter;

public class And extends Expression {

    private Expression left, right;

    public And(Expression left, Expression right) {
        this.left = left;
```

```
        this.right = right;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj != null && obj instanceof And) {
            return left.equals(((And) obj).left) &&
right.equals(((And) obj).right);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {

        return left.interpret(ctx) && right.interpret(ctx);
    }

    @Override
    public String toString() {
        return "(" + left.toString() + " AND " + right.toString()
+ ")";
    }
}
```

Or.java

```
package design.pattern.behavior.interpreter;

public class Or extends Expression {
    private Expression left, right;

    public Or(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj != null && obj instanceof Or) {
```

```
            return this.left.equals(((Or) obj).left) &&
this.right.equals(((Or) obj).right);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {
        return left.interpret(ctx) || right.interpret(ctx);
    }

    @Override
    public String toString() {
        return "(" + left.toString() + " OR " + right.toString() +
")";
    }

}
```

Not.java

```
package design.pattern.behavior.interpreter;

public class Not extends Expression {

    private Expression exp;

    public Not(Expression exp) {
        this.exp = exp;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj != null && obj instanceof Not) {
            return exp.equals(((Not) obj).exp);
        }
        return false;
    }

    @Override
```

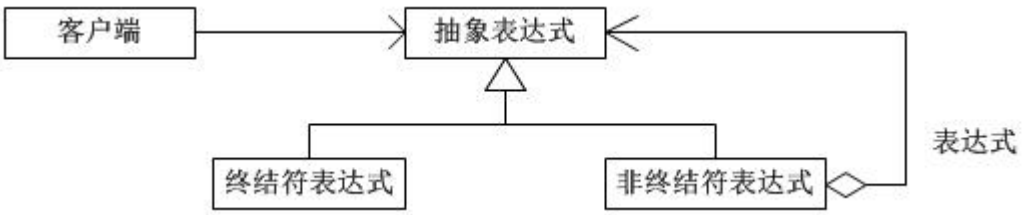
```
    public int hashCode() {
        return this.toString().hashCode();
    }

    @Override
    public boolean interpret(Context ctx) {
        return !exp.interpret(ctx);
    }

    @Override
    public String toString() {
        return "(Not " + exp.toString() + ")";
    }

}
```

Client.java

```
package design.pattern.behavior.interpreter;

public class Client {
    public static void main(String[] args) {
        Context ctx = new Context();
        Variable x = new Variable("x");
        Variable y = new Variable("y");
        Constant c = new Constant(true);
        ctx.assign(x, false);
        ctx.assign(y, true);

        Expression exp = new Or(new And(c, x), new And(y, new
Not(x)));
        System.out.println("x=" + x.interpret(ctx));
        System.out.println("y=" + y.interpret(ctx));
        System.out.println(exp.toString() + "=" +
exp.interpret(ctx));
    }
}
```

3. 用处：

1) 先介绍解释器模式的结构：



抽象表达式(Expression)角色：声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个 interpret()方法，称做解释操作。

终结符表达式(Terminal Expression)角色：实现了抽象表达式角色所要求的接口，主要是一个 interpret()方法；文法中的每一个终结符都有一个具体终结表达式与之相对应。比如有一个简单的公式 R=R1+R2，在里面 R1 和 R2 就是终结符，对应的解析 R1 和 R2 的解释器就是终结符表达式。

非终结符表达式(Nonterminal Expression)角色：文法中的每一条规则都需要一个具体的非终结符表达式，非终结符表达式一般是文法中的运算符或者其他关键字，比如公式 R=R1+R2 中，“+"就是非终结符，解析“+”的解释器就是一个非终结符表达式。

环境(Context)角色：这个角色的任务一般是用来存放文法中各个终结符所对应的具体值，比如 R=R1+R2，我们给 R1 赋值 100，给 R2 赋值 200。这些信息需要存放到环境角色中，很多情况下我们使用 Map 来充当环境角色就足够了。

为了说明解释器模式的实现办法，这里给出一个最简单的文法和对应的解释器模式的实现，这就是模拟 Java 语言中对布尔表达式进行操作和求值。

在这个语言中终结符是布尔变量，也就是常量 true 和 false。非终结符表达式包含运算符 and，or 和 not 等布尔表达式。这个简单的文法如下：

```
Expression  ::= Constant | Variable | Or | And | Not

And         ::= Expression 'AND' Expression

Or          ::= Expression 'OR' Expression

Not         ::= 'NOT' Expression

Variable    ::= 任何标识符

Constant    ::= 'true' | 'false'
```

第四章 设计模式之间关系图

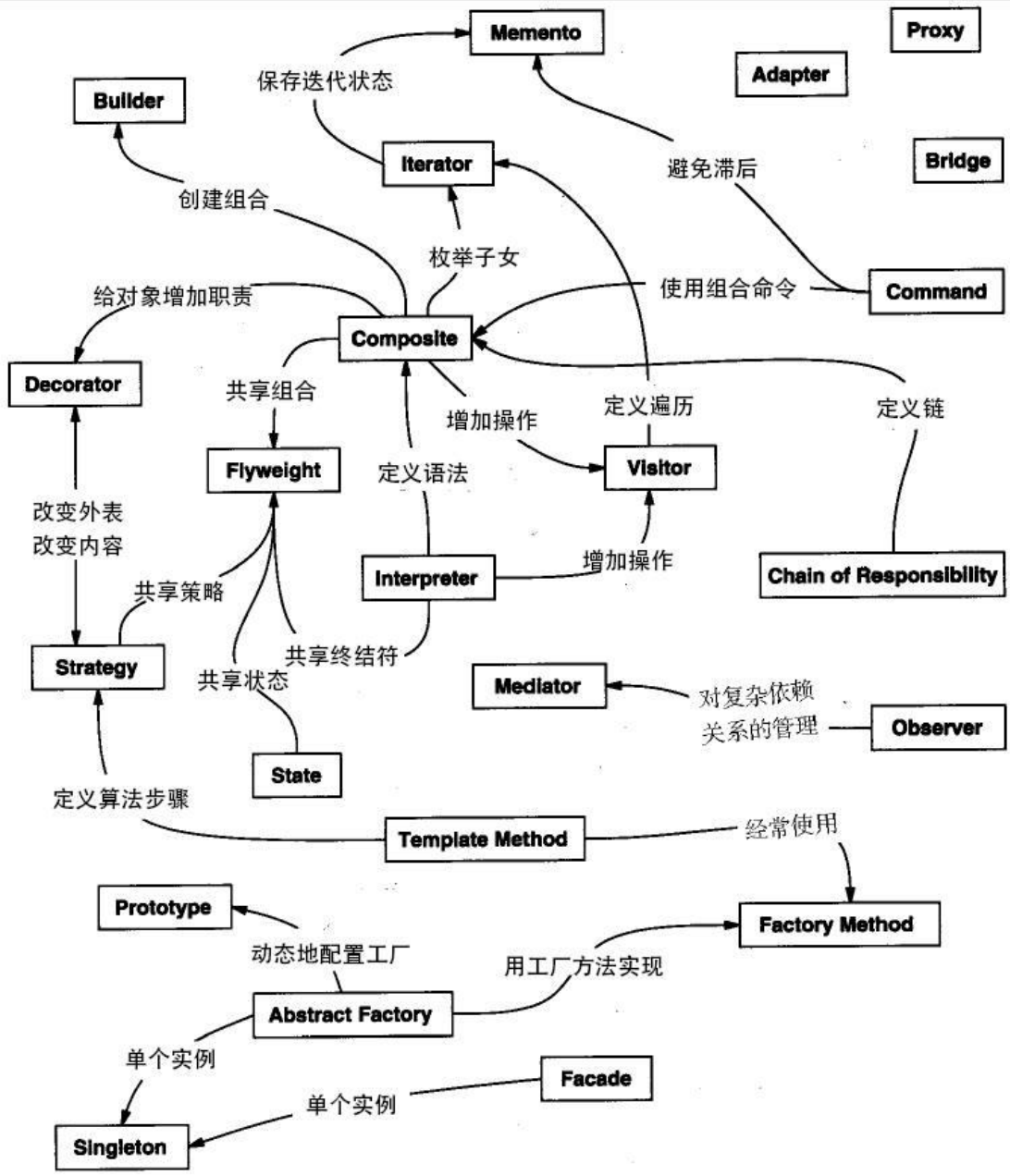


图　设计模式之间的关系