Head First设计模式-读书简记

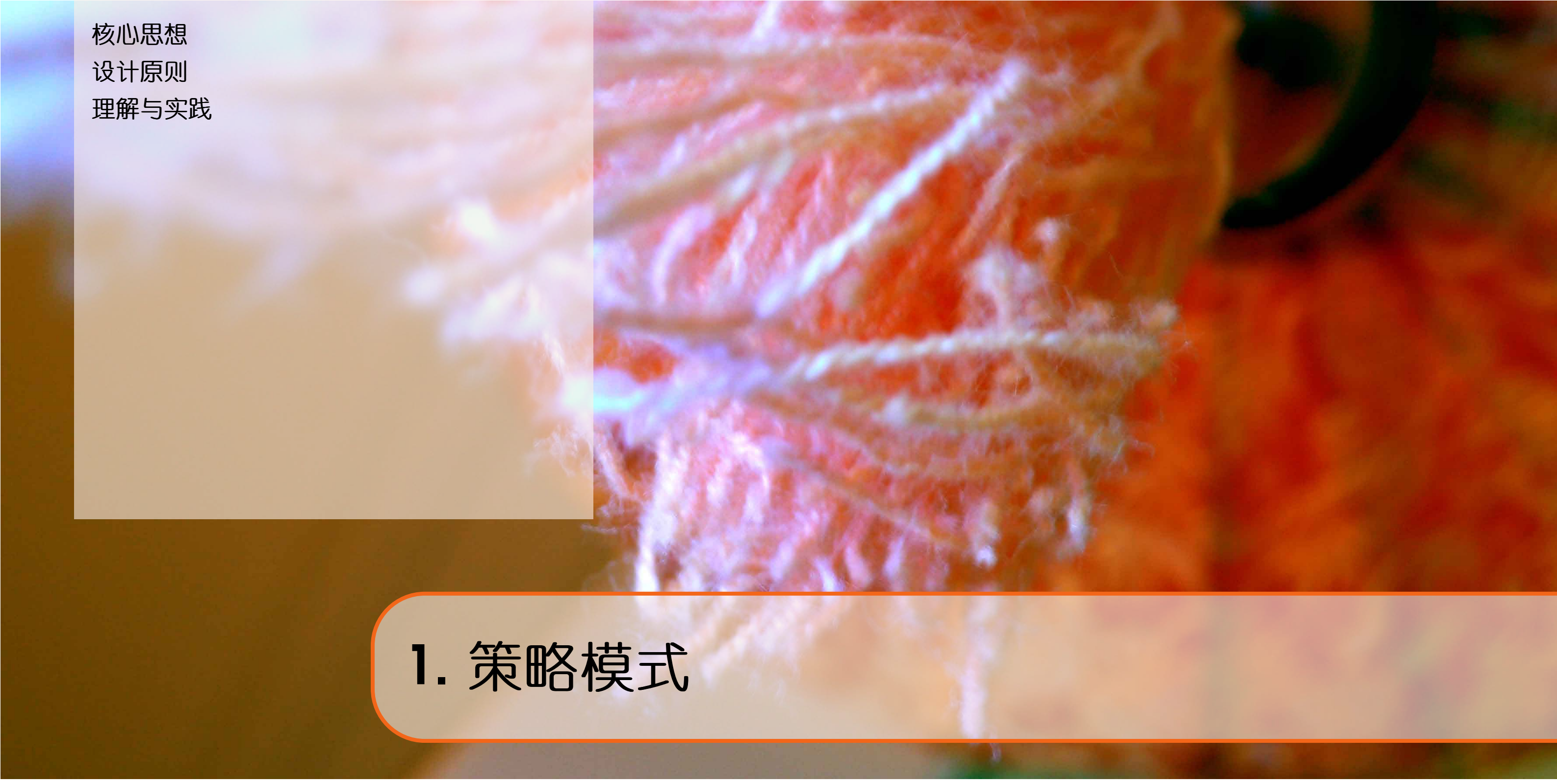
冰蓝

http://lanbing510.info

Copyright © 2014 冰蓝

First printing, December 2014

Contents



1. 策略模式

1.1 核心思想

首先先看策略模式的定义：

设计模式 1 — 策略模式. 定义算法族，分别封装起来，让它们之间可以相互替换，此模式让算法的变化独立于使用算法的客户。

1.2 设计原则

一些设计原则如下：

TIP 1 找出应用中可能需要变化之处，将其独立出来,不要和那些不需要变化的代码混在一起 ■

TIP 2 针对接口编程，而不是针对实现编程 ■

1.3 理解与实践

实战 1 设计一套鸭子模拟游戏，游戏中会出现各种鸭子，鸭子的飞行、游泳、叫声等行为各异

分析：

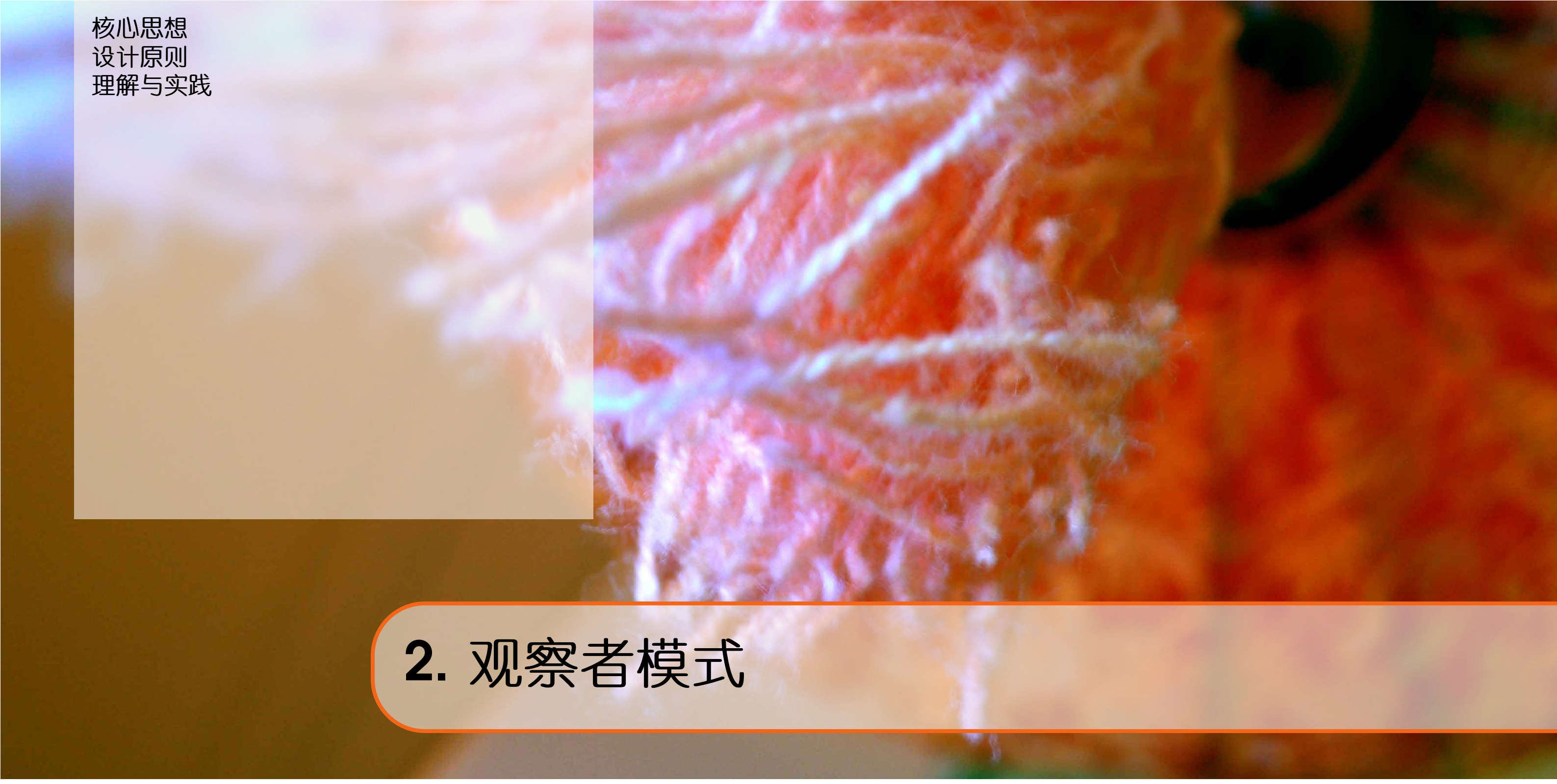
1 如果用继承来实现，例如鸭子类中加入fly(),quack(),很难知道鸭子的全部行为，加入一个新的行为后，也造成了不需要此行为的鸭子的改动；同时会造成代码在多个子类中重复；运行时的行为也不容易改变。违背TIP1,2。

2 如果使用接口，例如使用Flyable,Quackable,每个使用接口的类都要负责实现，无法达到代码复用，如果需要修改某个行为，必须追踪到每个定义此行为的类中修改。违背TIP2。

3 解决方法，使用策略模式，即可以将行为算法从类中独立出来，建立一组新类。鸭子的飞行、叫等行为委托给行为类。这样我们还可以在鸭子类中通过设定方法来动态改变鸭子的行为。

点睛程序

```java
//Duck.java

package headfirst.strategy;
public abstract class Duck {
        FlyBehavior flyBehavior;
        QuackBehavior quackBehavior;

        public Duck() {
        }

        //set the behavior of duck dynamically
        public void setFlyBehavior (FlyBehavior fb) {
                flyBehavior = fb;
        }

        public void setQuackBehavior(QuackBehavior qb) {
                quackBehavior = qb;
        }

        abstract void display();

        public void performFly() {
                flyBehavior.fly();
        }

        public void performQuack() {
                quackBehavior.quack();
        }

        public void swim() {
                System.out.println("all ducks float, even
                    decoys!");
        }
}
```



2. 观察者模式

2.1 核心思想

首先先看观察者模式的定义：

设计模式 2 — 观察者模式. 定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会受到通知并自动更新。

2.2 设计原则

一些设计原则

TIP 3 为了交互对象之间的松耦合设计而努力。对象之间的松耦合使对象之间的互赖性降到最低，能让我们建立有弹性的OO系统，应对变化。 ■

2.3 理解与实践

实战 2 设计一个应用，能讲气象站提供的信息分别更新到状况布告板、气象统计布告板及预报布告板。

错误示范

```
//WeatherData.java

public class WeatherData {

        public void measurementsChangeed() {

                //defination of some variables

                float temp=getTemperature();
                float humidity=getHumidity();
                float pressure=getPressure();
```

```java

                        curentConditionsDisplay.update(temp,
                            humidity,pressure);
                        staticticsDisplay.update(temp,humidity,
                            pressure);
                        forecastDisplay.update(temp,humidity,
                            pressure);

                        //othe methods of WeatherData
            }
}
```

分析:

1 错误示范代码针对具体实现编程，导致以后增删布告板时必须修改程序
2 根据TIP1，将改变的地方，也就是观察者（布告板），封装起来。观察者只需订阅主题（气象站）或者移除订阅既可。正确代码如下

点睛程序

```java
package headfirst.observer.weather;

public interface Subject {
        public void registerObserver(Observer o);
        public void removeObserver(Observer o);
        public void notifyObservers();
}
```

```java
//Observer.java

package headfirst.observer.weather;

public interface Observer {
        public void update(float temp, float humidity,
            float pressure);
}
```

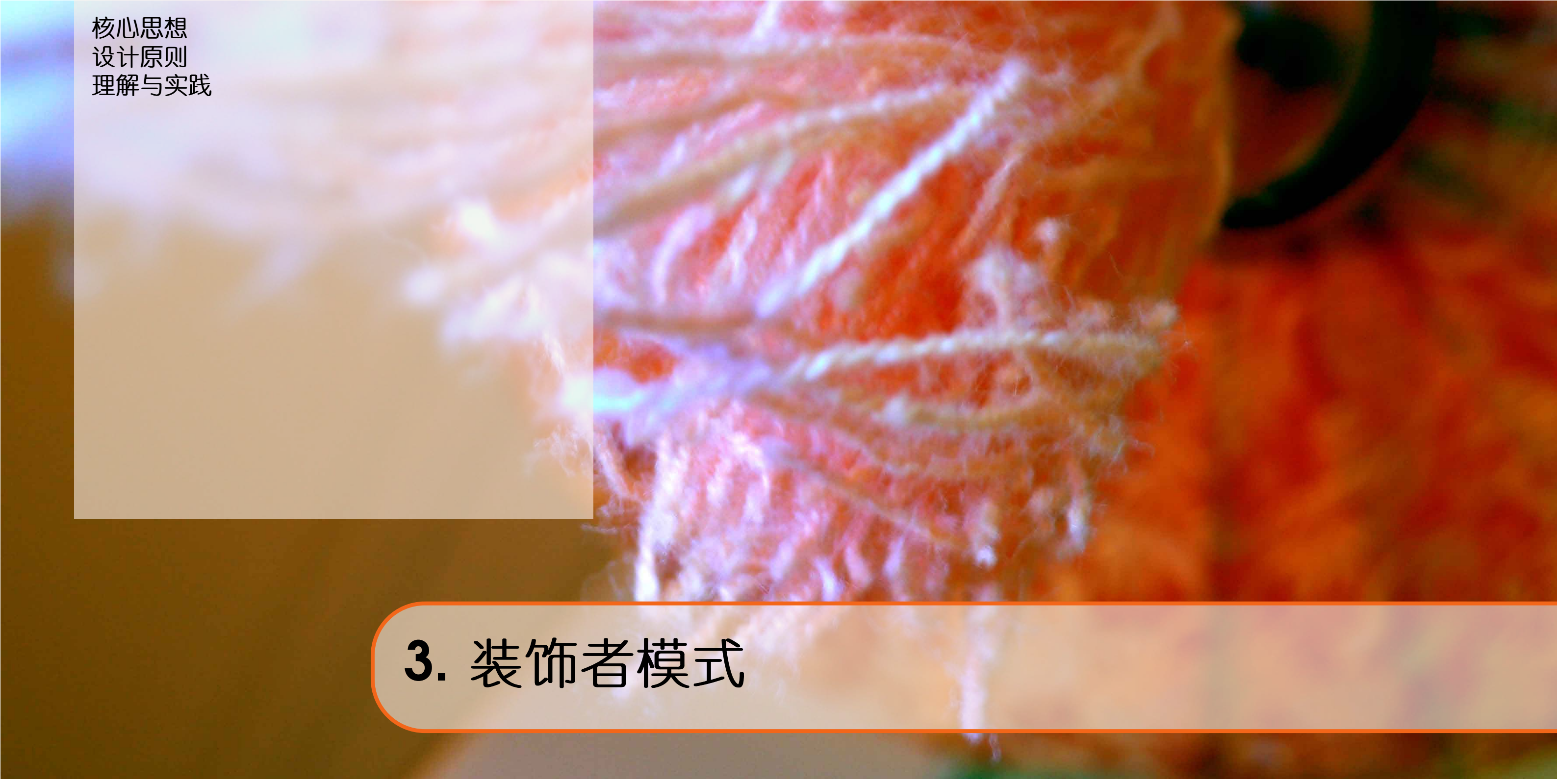
```java
//WeatherData.java

package headfirst.observer.weather;

import java.util.*;

public class WeatherData implements Subject {
        private ArrayList observers;
        private float temperature;
        private float humidity;
        private float pressure;

        public WeatherData() {
                observers = new ArrayList();
        }

        public void registerObserver(Observer o) {
                observers.add(o);
        }

        public void removeObserver(Observer o) {
                int i = observers.indexOf(o);
                if (i >= 0) {
                        observers.remove(i);
                }
        }

        public void notifyObservers() {
                for (int i = 0; i < observers.size(); i++) {
                        Observer observer = (Observer) observers.get(i);
                        observer.update(temperature, humidity, pressure);
                }
        }

        public void measurementsChanged() {
                notifyObservers();
        }

        public void setMeasurements(float temperature, float humidity, float pressure) {
                this.temperature = temperature;
                this.humidity = humidity;
                this.pressure = pressure;
                measurementsChanged();
        }

        public float getTemperature() {
                return temperature;
        }

        public float getHumidity() {
                return humidity;
        }

        public float getPressure() {
                return pressure;
        }
}
```

```java
//CurrentConditionsDisplay.java
```

```java

package headfirst.observer.weather;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
	private float temperature;
	private float humidity;
	private Subject weatherData;

	public CurrentConditionsDisplay(Subject weatherData) {
		this.weatherData = weatherData;
		weatherData.registerObserver(this);
	}

	public void update(float temperature, float humidity, float pressure) {
		this.temperature = temperature;
		this.humidity = humidity;
		display();
	}

	public void display() {
		System.out.println("Current conditions: " + temperature
			+ "F degrees and " + humidity + "% humidity");
	}
}
```

```java
//WeatherStation.java

package headfirst.observer.weather;

import java.util.*;

public class WeatherStation {

	public static void main(String[] args) {
		WeatherData weatherData = new WeatherData();

		CurrentConditionsDisplay currentDisplay =
			new CurrentConditionsDisplay(weatherData);
		StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
		ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

```

```
                weatherData.setMeasurements(80, 65, 30.4f);
                weatherData.setMeasurements(82, 70, 29.2f);
                weatherData.setMeasurements(78, 90, 29.2f);
        }
}
```



3. 装饰者模式

3.1 核心思想

首先先看装饰者模式的定义：

设计模式 3 — **装饰者模式.** 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有弹性的替代方案。

3.2 设计原则

一些设计原则

TIP 4 类应该对扩展开放，对修改关闭 ■

3.3 理解与实践

实战 3 为星巴克咖啡店设计一个订单结账系统，饮料有深培咖啡、脱因咖啡、独家调配咖啡等，每种咖啡有不同的价格，消费者还可以加各种不同价格的不同剂量的调料。

分析：

1 如果对每个类别和不同剂量调料的咖啡都设计一个类，造成了“类爆炸”，一旦有新的需要、价钱变化、一些饮料不可以加一些调料的情况就等进行大量代码的更改
2 对此，我们可以以饮料为主题，然后在运行时用各种调料来“装饰”饮料，调用cost()方法并依赖委托（delegate）将调料价格加上去
3 装饰者和被装饰者有共同的超类（通过继承达到类型匹配，而不是获得行为），行为来自装饰者和基础组件或与其他装饰者的组合关系。

点睛程序

```java
package headfirst.decorator.starbuzz;

public abstract class Beverage {
        String description = "Unknown Beverage";

        public String getDescription() {
                return description;
        }

        public abstract double cost();
}
```

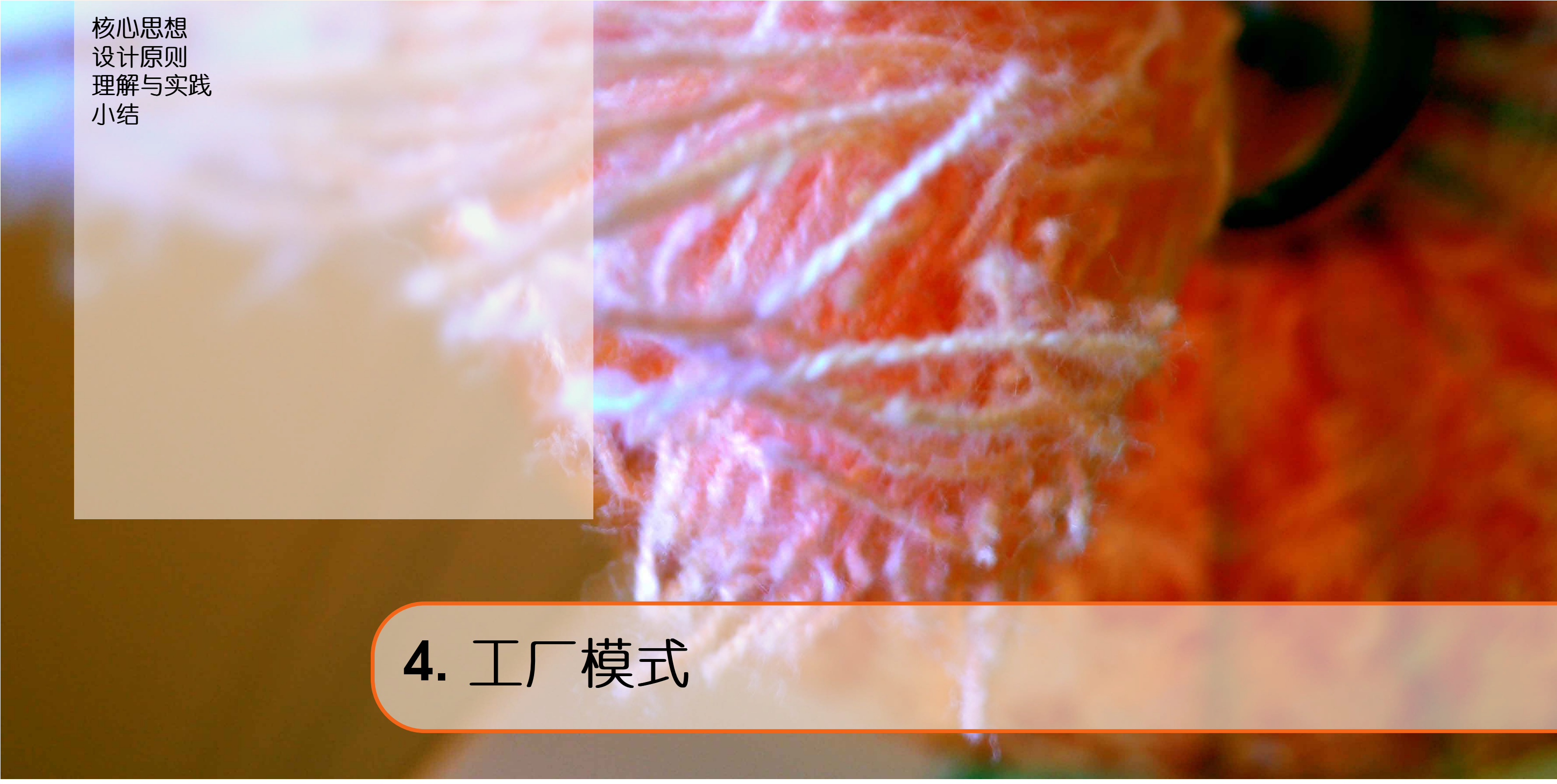
```java
package headfirst.decorator.starbuzz;

public abstract class CondimentDecorator extends Beverage {
        public abstract String getDescription();
}
```

```java
package headfirst.decorator.starbuzz;

public class Milk extends CondimentDecorator {
        Beverage beverage;

        public Milk(Beverage beverage) {
                this.beverage = beverage;
        }

        public String getDescription() {
                return beverage.getDescription() + ", Milk"
                    ;
        }

        public double cost() {
                return .10 + beverage.cost();
        }
}
```

```java
package headfirst.decorator.starbuzz;

public class DarkRoast extends Beverage {
        public DarkRoast() {
                description = "Dark Roast Coffee";
        }

        public double cost() {
                return .99;
        }
}
```

```java
package headfirst.decorator.starbuzz;
```

```

public class StarbuzzCoffee {

	public static void main(String args[]) {
		Beverage beverage = new Espresso();
		System.out.println(beverage.getDescription()
			+ " $" + beverage.cost());

		Beverage beverage2 = new DarkRoast();
		beverage2 = new Mocha(beverage2);
		beverage2 = new Mocha(beverage2);
		beverage2 = new Whip(beverage2);
		System.out.println(beverage2.getDescription()
			+ " $" + beverage2.cost());

		Beverage beverage3 = new HouseBlend();
		beverage3 = new Soy(beverage3);
		beverage3 = new Mocha(beverage3);
		beverage3 = new Whip(beverage3);
		System.out.println(beverage3.getDescription()
			+ " $" + beverage3.cost());
	}
}
```



4. 工厂模式

4.1 核心思想

首先先看工厂模式的定义：

设计模式 4 — 工厂方法模式. 定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。

设计模式 5 — 抽象工厂模式. 提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体的类。

4.2 设计原则

一些设计原则

TIP 5 要依赖抽象，不要依赖具体类 ■

4.3 理解与实践

实战 4 为对象村匹萨店设计应用系统，匹萨有希腊匹萨、素食匹萨，允许加入新的类型的匹萨（比如蛤蜊匹萨），允许加盟店做自己特色的匹萨，但原料和打包等可以统一管理方便监督。

错误示范

```
//PizzaStoreWrong.java

package headfirst.factory.pizzas;

public class PizzaStore {

        public Pizza orderPizza(String type) {
                Pizza pizza;

                if(type.equals("Cheese")){
                        pizza=new CheesePizza();
                }else if(type.equals("greek")){
                        pizza=new GreekPizza();
                }
                //......

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();

                return pizza;
        }

}
```

分析：

1 上例代码没有对修改封闭。如果披萨店改变所提供的比萨风味，就得去orderPizz()里面修改。
2 我们可以封装创建对象的代码，将其搬到另一个对象中，这个对象只负责生产匹萨，称此对象为“工厂”。参加下面代码（简单工厂模式），很常用。

点睛程序

```
//SimplePizzaFactory.java

package headfirst.factory.pizzas;

public class SimplePizzaFactory {

        public Pizza createPizza(String type) {
                Pizza pizza = null;

                if (type.equals("cheese")) {
                        pizza = new CheesePizza();
                } else if (type.equals("pepperoni")) {
                        pizza = new PepperoniPizza();
                } else if (type.equals("clam")) {
                        pizza = new ClamPizza();
                } else if (type.equals("veggie")) {
                        pizza = new VeggiePizza();
                }
                return pizza;
        }
}
```

```java
//PizzaStore.java

package headfirst.factory.pizzas;

public class PizzaStore {
        SimplePizzaFactory factory;

        public PizzaStore(SimplePizzaFactory factory) {
                this.factory = factory;
        }

        public Pizza orderPizza(String type) {
                Pizza pizza;

                pizza = factory.createPizza(type);

                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();

                return pizza;
        }

}
```

分析：

1 现在考虑加入加盟店，在推广SimpleFactory时，每个区域的加盟店需要创建各自的PizzaFactory，加入自己的改良。但其他部分，如切片、烘烤方法、盒子模具，可能会产生不统一。如果想加入多一些的质量控制，把加盟店和创建比萨捆绑到一起同时保持一定的弹性，则需要下面的框架（工厂模式）。

点睛程序

```java
//PizzaStore.java

package headfirst.factory.pizzafm;

public abstract class PizzaStore {

        abstract Pizza createPizza(String item);

        public Pizza orderPizza(String type) {
                Pizza pizza = createPizza(type);
                System.out.println("--- Making a " + pizza.
                    getName() + " ---");
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

```java
//NYPizzaStore.java

package headfirst.factory.pizzafm;

public class NYPizzaStore extends PizzaStore {

        Pizza createPizza(String item) {
                if (item.equals("cheese")) {
                        return new NYStyleCheesePizza();
                } else if (item.equals("veggie")) {
                        return new NYStyleVeggiePizza();
                } else if (item.equals("clam")) {
                        return new NYStyleClamPizza();
                } else if (item.equals("pepperoni")) {
                        return new NYStylePepperoniPizza();
                } else return null;
        }
}
```

分析：

1 现在为了确保没家加盟店使用高质量的原料，我们打算建造一家生产原料的工厂，但加盟店坐落在不同的城市，原料也会有差异。对此需要用到（抽象工厂模式），见下面的框架。

点睛程序

```java
//PizzaStore.java

package headfirst.factory.pizzafm;

public abstract class PizzaStore {

        abstract Pizza createPizza(String item);

        public Pizza orderPizza(String type) {
                Pizza pizza = createPizza(type);
                System.out.println("--- Making a " + pizza.
                    getName() + " ---");
                pizza.prepare();
                pizza.bake();
                pizza.cut();
                pizza.box();
                return pizza;
        }
}
```

```
//PizzaIngredientFactory.java

package headfirst.factory.pizzaaf;

public interface PizzaIngredientFactory {

        public Dough createDough();
        public Sauce createSauce();
        public Cheese createCheese();
        public Veggies[] createVeggies();
        public Pepperoni createPepperoni();
        public Clams createClam();

}
```

```
//NYPizzaStore.java

package headfirst.factory.pizzaaf;

public class NYPizzaStore extends PizzaStore {

        protected Pizza createPizza(String item) {
                Pizza pizza = null;
                PizzaIngredientFactory ingredientFactory =
                        new NYPizzaIngredientFactory();

                if (item.equals("cheese")) {

                        pizza = new CheesePizza(
                           ingredientFactory);
                        pizza.setName("New York Style
                           Cheese Pizza");

                } else if (item.equals("veggie")) {

                        pizza = new VeggiePizza(
                           ingredientFactory);
                        pizza.setName("New York Style
                           Veggie Pizza");

                } else if (item.equals("clam")) {

                        pizza = new ClamPizza(
                           ingredientFactory);
                        pizza.setName("New York Style Clam
                           Pizza");

```

```java
                } else if (item.equals("pepperoni")) {

                        pizza = new PepperoniPizza(
                            ingredientFactory);
                        pizza.setName("New York Style
                            Pepperoni Pizza");

                }
                return pizza;
        }
}
```

```java
//NYPizzaIngredientFactory.java

package headfirst.factory.pizzaaf;

public class NYPizzaIngredientFactory implements
    PizzaIngredientFactory {

        public Dough createDough() {
                return new ThinCrustDough();
        }

        public Sauce createSauce() {
                return new MarinaraSauce();
        }

        public Cheese createCheese() {
                return new ReggianoCheese();
        }

        public Veggies[] createVeggies() {
                Veggies veggies[] = { new Garlic(), new
                    Onion(), new Mushroom(), new RedPepper()
                     };
                return veggies;
        }

        public Pepperoni createPepperoni() {
                return new SlicedPepperoni();
        }

        public Clams createClam() {
                return new FreshClams();
        }
}
```

```java
//Pizza.java

package headfirst.factory.pizzaaf;
```

```java

public abstract class Pizza {
        String name;

        Dough dough;
        Sauce sauce;
        Veggies veggies[];
        Cheese cheese;
        Pepperoni pepperoni;
        Clams clam;

        abstract void prepare();

        void bake() {
                System.out.println("Bake for 25 minutes at 350");
        }

        void cut() {
                System.out.println("Cutting the pizza into diagonal slices");
        }

        void box() {
                System.out.println("Place pizza in official PizzaStore box");
        }

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

        String getName() {
                return name;
        }

        public String toString() {
                StringBuffer result = new StringBuffer();
                result.append("---- " + name + " ----\n");
                if (dough != null) {
                        result.append(dough);
                        result.append("\n");
                }
                if (sauce != null) {
                        result.append(sauce);
                        result.append("\n");
                }
                if (cheese != null) {
                        result.append(cheese);
```

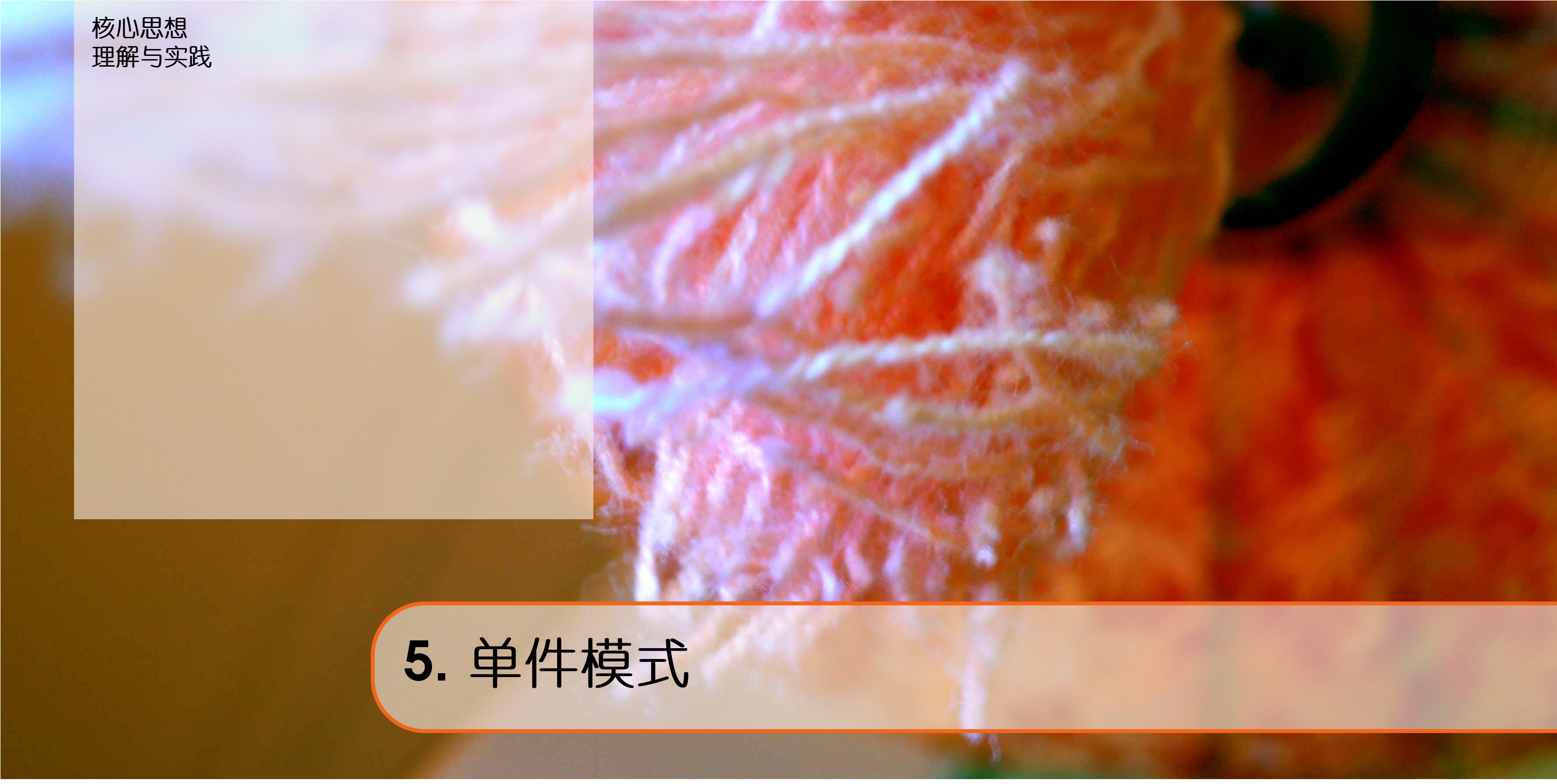
```java
            result.append("\n");
        }
        if (veggies != null) {
            for (int i = 0; i < veggies.length; i++) {
                result.append(veggies[i]);
                if (i < veggies.length-1) {
                    result.append(", ");
                }
            }
            result.append("\n");
        }
        if (clam != null) {
            result.append(clam);
            result.append("\n");
        }
        if (pepperoni != null) {
            result.append(pepperoni);
            result.append("\n");
        }
        return result.toString();
    }
}
```

```java
//CheesePizza.java

package headfirst.factory.pizzaaf;

public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;

    public CheesePizza(PizzaIngredientFactory ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }

    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
        cheese = ingredientFactory.createCheese();
    }
}
```

```java
//Cheese.java

package headfirst.factory.pizzaaf;

public interface Cheese {
        public String toString();
}
```

4.4 小结

1 工厂模式使用的是继承的方法；抽象工厂模式使用的方法是对象的组合。
2 当需要把客户代码从需要实例化的具体类中解耦，或者当目前还不知道将来需要实例化那些具体类时，使用工厂方法。使用方式：将其继承成子类，并实现其工厂方法即可。
3 当需要创建产品家族和想让相关产品集合起来时，使用抽象工厂方法。使用组合来实现，对象的创建被实现在工厂所暴露的方法中。
4 简单工厂虽然不是真正的设计模式，但仍不失为一个简单的方法，可以将客户程序从具体类解耦。
5 所有的工厂都是用来封装对象的创建。
6 依赖倒置原则，指导我们避免依赖具体的类型，而要尽量依赖抽象。



5. 单件模式

5.1 核心思想

首先先看单件模式的定义：

设计模式 6 — 单件模式. 确保一个类只有一个实例，并提供一个全局访问点。

点睛程序

```java
package headfirst.singleton.classic;

// NOTE: This is not thread safe!

public class Singleton {
	private static Singleton uniqueInstance;

	// other useful instance variables here

	private Singleton() {}

	public static Singleton getInstance() {
		if (uniqueInstance == null) {
			uniqueInstance = new Singleton();
		}
		return uniqueInstance;
	}

	// other useful methods here
}
```

5.2 理解与实践

实战 5 设计一个巧克力锅炉控制器，锅炉做的事就是把巧克力和牛奶融在一起，然后送到下一个阶段，制成巧克力棒。注意，锅炉满的时候不能填入原料，锅炉排出时必须是煮过的，满的，煮混合物是锅炉必须是满的等条件。

错误示范

```java
//ChocolateBoiler.java

package headfirst.singleton.chocolate;

public class ChocolateBoiler {
        private boolean empty;
        private boolean boiled;
        private static ChocolateBoiler uniqueInstance;

        private ChocolateBoiler() {
                empty = true;
                boiled = false;
        }

        public static ChocolateBoiler getInstance() {
                if (uniqueInstance == null) {
                        System.out.println("Creating unique instance of Chocolate Boiler");
                        uniqueInstance = new ChocolateBoiler();
                }
                System.out.println("Returning instance of Chocolate Boiler");
                return uniqueInstance;
        }

        public void fill() {
                if (isEmpty()) {
                        empty = false;
                        boiled = false;
                        // fill the boiler with a milk/chocolate mixture
                }
        }

        public void drain() {
                if (!isEmpty() && isBoiled()) {
                        // drain the boiled milk and chocolate
                        empty = true;
                }
        }

        public void boil() {
                if (!isEmpty() && !isBoiled()) {
                        // bring the contents to a boil
                        boiled = true;
                }
        }

        public boolean isEmpty() {
                return empty;
        }

        public boolean isBoiled() {
                return boiled;
        }
}
```

```java
//ChocolateController.java

package headfirst.singleton.chocolate;

public class ChocolateController {
        public static void main(String args[]) {
                ChocolateBoiler boiler = ChocolateBoiler.
                    getInstance();
                boiler.fill();
                boiler.boil();
                boiler.drain();

                // will return the existing instance
                ChocolateBoiler boiler2 = ChocolateBoiler.
                    getInstance();
        }
}
```

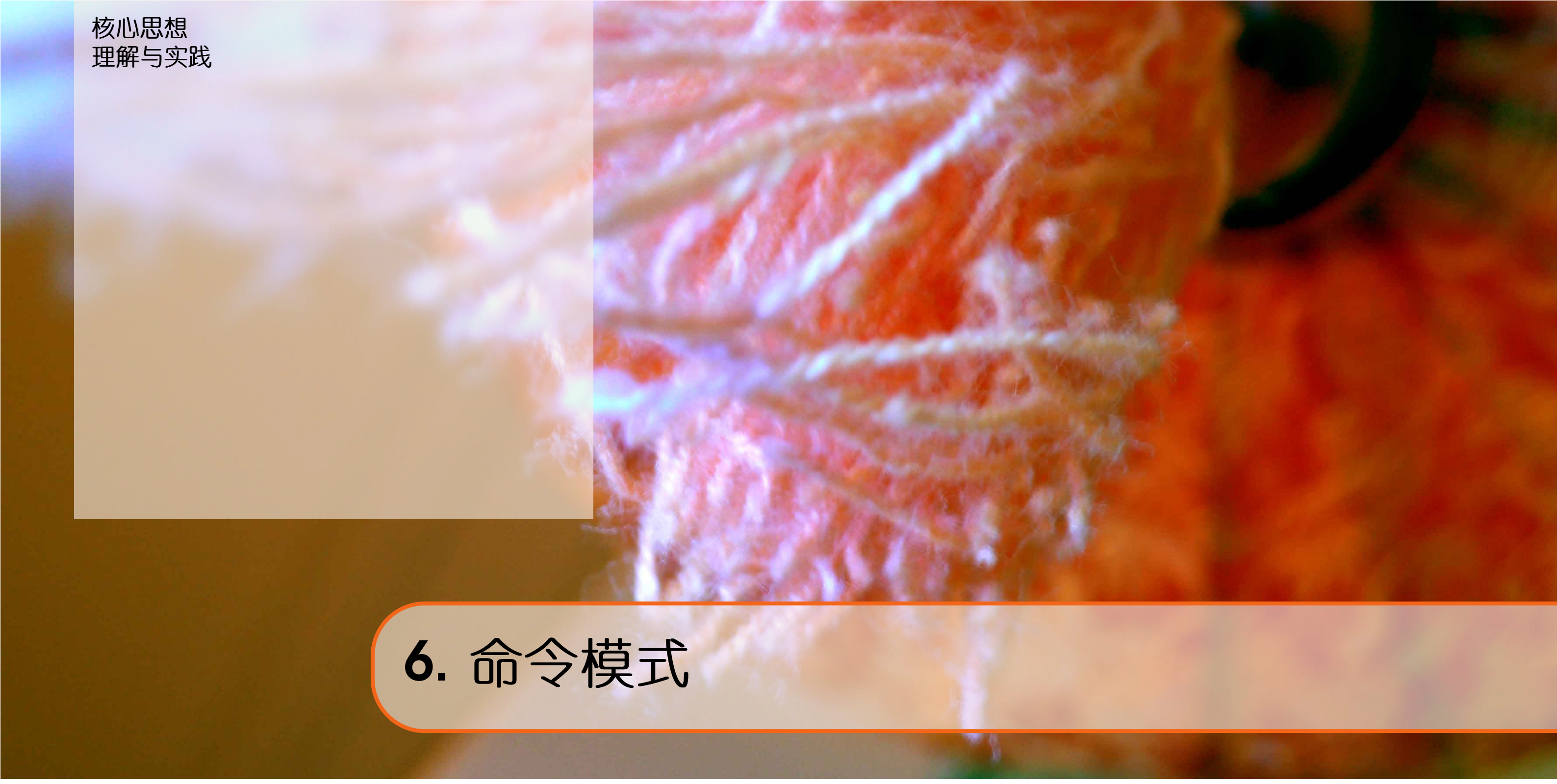
分析：

1 可以想到，如果有多于一个巧克力锅炉的实例存在，就可能发生很糟糕的事情。
2 第一眼看上述程序没问题，但是如果用到多线程，很可能在第一个线程创建完实例的时候，第二个线程也已经进入创建的条件判断。
2 解决方法是，利用双重检查加锁，程序如下。

点睛程序

```java
//Singleton.java

package headfirst.singleton.dcl;

//
// Danger!  This implementation of Singleton not
// guaranteed to work prior to Java 5
```

```java
//

public class Singleton {
    private volatile static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            synchronized (Singleton.class) {
                if (uniqueInstance == null) {
                    uniqueInstance = new Singleton();
                }
            }
        }
        return uniqueInstance;
    }
}
```



6. 命令模式

6.1 核心思想

首先先看命令模式的定义：

设计模式 7 — 命令模式. 将“请求”封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销的操作。

6.2 理解与实践

实战 6 设计一个家电自动化的遥控系统，遥控上有七个插槽，每个插槽旁边有开和关两个按钮用来控制分配到改插槽的家电，可支持撤销操作，party 模式。其中每个家电的接口都不是统一的，比如电灯的on off，门的up down，音效打开后初始的音量等。

分析：

1 封装的一个全新境界：将方法调用封装起来。通过封装方法，调用此运算的对象不需要关心事情是如何进行的，只需要用包装成型的方法来完成它既可。
2 我们可以类比对象村餐厅，顾客不需要知道汉堡需要怎么做，只需要下订单，出示也不需要关心是谁要买汉堡，接到订单只需要去做，中间的传递者–服务员，所做的工作只是接受订单，然后放到订单柜台通知厨师orderup。插槽可以类比为服务员，command相当于订单，receiver–各种家电相当于厨师，取走订单相当于setcommand，execute相当于orderup，顾客相当于client。
3 命令模式，实现了请求调用者和请求接受者之间的解耦。command是接口。
4 命令模式的更多用途：队列请求、日志请求等

点睛程序

```
// Command.java

package headfirst.command.party;

```

```java
public interface Command {
        public void execute();
        public void undo();
}
```

```java
// MacroCommand.java

package headfirst.command.party;

public class MacroCommand implements Command {
        Command[] commands;

        public MacroCommand(Command[] commands) {
                this.commands = commands;
        }

        public void execute() {
                for (int i = 0; i < commands.length; i++) {
                        commands[i].execute();
                }
        }

    /**
     * NOTE:  these commands have to be done backwards to
        ensure proper undo functionality
     */
        public void undo() {
                for (int i = commands.length -1; i >= 0; i
                    --) {
                        commands[i].undo();
                }
        }
}
```

```java
// RemoteControl.java

package headfirst.command.party;

import java.util.*;

//
// This is the invoker
//
public class RemoteControl {
        Command[] onCommands;
        Command[] offCommands;
        Command undoCommand;

        public RemoteControl() {
                onCommands = new Command[7];
```

```java
            offCommands = new Command[7];

            Command noCommand = new NoCommand();
            for(int i=0;i<7;i++) {
                onCommands[i] = noCommand;
                offCommands[i] = noCommand;
            }
            undoCommand = noCommand;
        }

        public void setCommand(int slot, Command onCommand, Command offCommand) {
            onCommands[slot] = onCommand;
            offCommands[slot] = offCommand;
        }

        public void onButtonWasPushed(int slot) {
            onCommands[slot].execute();
            undoCommand = onCommands[slot];
        }

        public void offButtonWasPushed(int slot) {
            offCommands[slot].execute();
            undoCommand = offCommands[slot];
        }

        public void undoButtonWasPushed() {
            undoCommand.undo();
        }

        public String toString() {
            StringBuffer stringBuff = new StringBuffer();
            stringBuff.append("\n——— Remote Control ————\n");
            for (int i = 0; i < onCommands.length; i++) {
                stringBuff.append("[slot " + i + "] " + onCommands[i].getClass().getName()
                    + "     " + offCommands[i].getClass().getName() + "\n");
            }
            stringBuff.append("[undo] " + undoCommand.getClass().getName() + "\n");
            return stringBuff.toString();
        }
}
```

```java
// RemoteLoader.java

package headfirst.command.party;

public class RemoteLoader {

        public static void main(String[] args) {

                RemoteControl remoteControl = new
                    RemoteControl();

                Light light = new Light("Living Room");
                TV tv = new TV("Living Room");
                Stereo stereo = new Stereo("Living Room");
                Hottub hottub = new Hottub();

                LightOnCommand lightOn = new LightOnCommand
                    (light);
                StereoOnCommand stereoOn = new
                    StereoOnCommand(stereo);
                TVOnCommand tvOn = new TVOnCommand(tv);
                HottubOnCommand hottubOn = new
                    HottubOnCommand(hottub);
                LightOffCommand lightOff = new
                    LightOffCommand(light);
                StereoOffCommand stereoOff = new
                    StereoOffCommand(stereo);
                TVOffCommand tvOff = new TVOffCommand(tv);
                HottubOffCommand hottubOff = new
                    HottubOffCommand(hottub);

                Command[] partyOn = { lightOn, stereoOn,
                    tvOn, hottubOn};
                Command[] partyOff = { lightOff, stereoOff,
                     tvOff, hottubOff};

                MacroCommand partyOnMacro = new
                    MacroCommand(partyOn);
                MacroCommand partyOffMacro = new
                    MacroCommand(partyOff);

                remoteControl.setCommand(0, partyOnMacro,
                    partyOffMacro);

                System.out.println(remoteControl);
                System.out.println("—— Pushing Macro On——
                    ");
                remoteControl.onButtonWasPushed(0);
                System.out.println("—— Pushing Macro Off
                    ——");
                remoteControl.offButtonWasPushed(0);
        }
}
```

```
package headfirst.command.party;

public class Stereo {
        String location;

        public Stereo(String location) {
                this.location = location;
        }

        public void on() {
                System.out.println(location + " stereo is
                    on");
        }

        public void off() {
                System.out.println(location + " stereo is
                    off");
        }

        public void setCD() {
                System.out.println(location + " stereo is
                    set for CD input");
        }

        public void setDVD() {
                System.out.println(location + " stereo is
                    set for DVD input");
        }

        public void setRadio() {
                System.out.println(location + " stereo is
                    set for Radio");
        }

        public void setVolume(int volume) {
                // code to set the volume
                // valid range: 1—11 (after all 11 is
                    better than 10, right?)
                System.out.println(location + " Stereo
                    volume set to " + volume);
        }
}
```

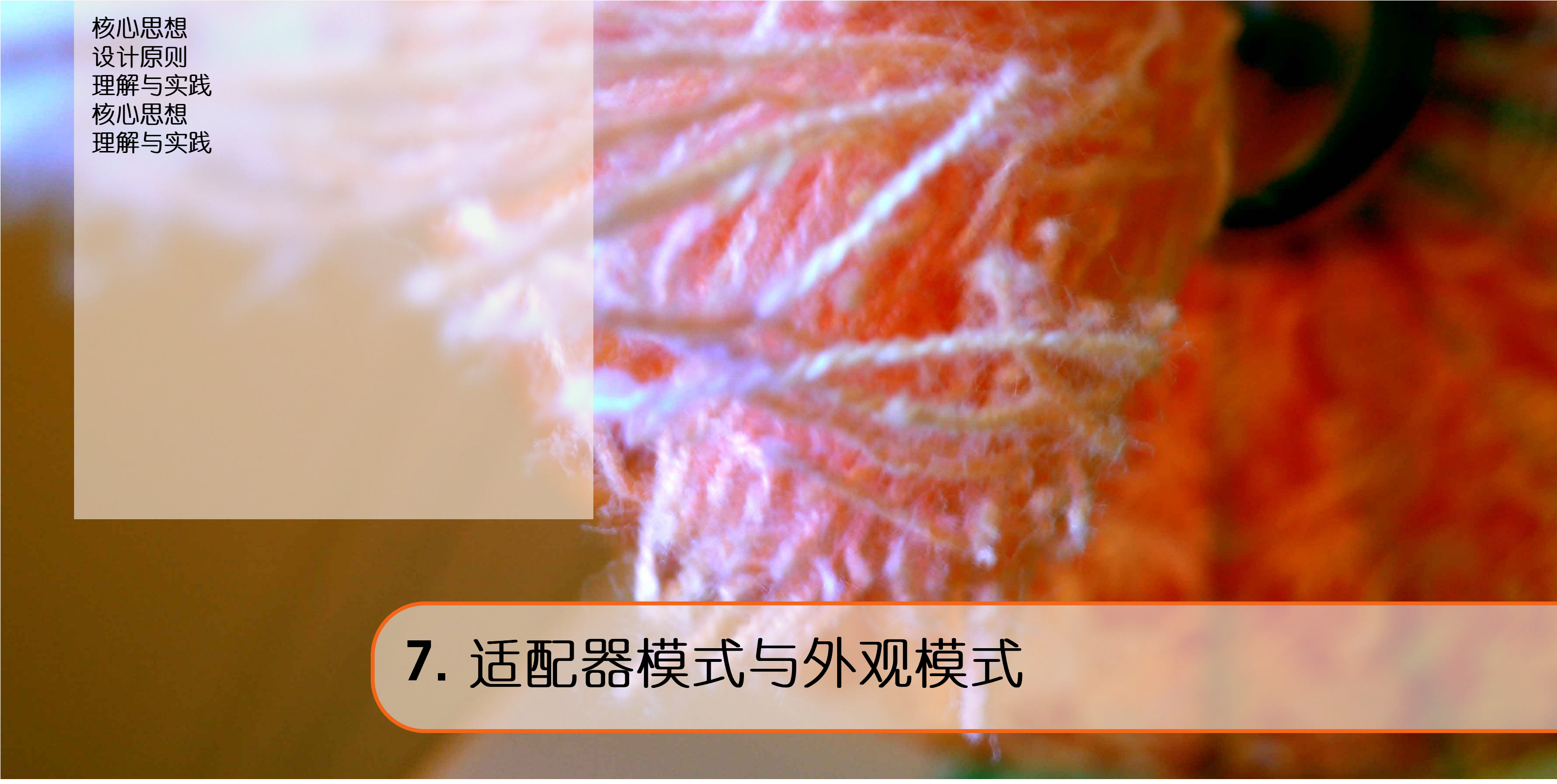
```
// StereoOnCommand.java
```

```java

package headfirst.command.party;

public class StereoOnCommand implements Command {
	Stereo stereo;

	public StereoOnCommand(Stereo stereo) {
		this.stereo = stereo;
	}

	public void execute() {
		stereo.on();
	}

	public void undo() {
		stereo.off();
	}
}
```

```java
//StereoOffCommand.java

package headfirst.command.party;

public class StereoOffCommand implements Command {
	Stereo stereo;

	public StereoOffCommand(Stereo stereo) {
		this.stereo = stereo;
	}

	public void execute() {
		stereo.off();
	}

	public void undo() {
		stereo.on();
	}
}
```

```java
// StereoOnWithCDCommand.java

package headfirst.command.party;

public class StereoOnWithCDCommand implements Command {
	Stereo stereo;

	public StereoOnWithCDCommand(Stereo stereo) {
		this.stereo = stereo;
	}

```

```
        public void execute() {
                stereo.on();
                stereo.setCD();
                stereo.setVolume(11);
        }

        public void undo() {
                stereo.off();
        }
}
```



7. 适配器模式与外观模式

7.1 核心思想

首先先看适配器模式的定义：

设计模式 8 — 适配器模式. 将一个类的接口，转换成客户期望的另一个接口。适配器让原本不兼容的类可以合作无间。

7.2 设计原则

一些设计原则

TIP 6 最少知识原则：只和你的密友谈话。即减少对象之间的交互，只留下几个“密友”。这个原则希望希望我们在设计中不要让太多的类耦合到一起，免得修改系统中的一部分。 ■

7.3 理解与实践

实战 7 Java早期使用的枚举器Enumeration有hasMoreElements()、nextElement()操作，新的集合类开始使用迭代器Iterator接口，具有hasnext()、next()、remove()操作，现在有些遗留代码暴露出枚举器接口，但我们又希望在新的代码中使用迭代器。所以请设计一个适配器，将枚举适配到迭代器。

点睛程序

```java
// EnumerationIterator.java

package headfirst.adapter.iterenum;

import java.util.*;

public class EnumerationIterator implements Iterator {
        Enumeration enumeration;
```

```java

    public EnumerationIterator(Enumeration enumeration)
        {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

```java
// EnumerationIteratorTestDrive.java

package headfirst.adapter.iterenum;

import java.util.*;

public class EnumerationIteratorTestDrive {
    public static void main (String args[]) {
        Vector v = new Vector(Arrays.asList(args));
        Iterator iterator = new EnumerationIterator
            (v.elements());
        while (iterator.hasNext()) {
            System.out.println(iterator.next())
                ;
        }
    }
}
```

分析：

1 适配器就像我们现实生活中的电源转换头。
2 实际上有两种适配器，类的适配器和对象适配器，类的适配器通过继承来实现，后者则通过组合来实现。

7.4 核心思想

接下来看下另一种相近的模式-外观模式的定义：

设计模式 9 — **外观模式.** 提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

7.5 理解与实践

实战 8 假设你组装了一套家庭影院，有爆米花机、DVD播放器、投影机、自动屏幕、立体声等等，现在你要看一部电影，但是不得一个个打开爆米花、打开屏幕、投影机、立体声。外观模式可以通过实现一个提供更合理接口的外观类很好的解决这个问题，

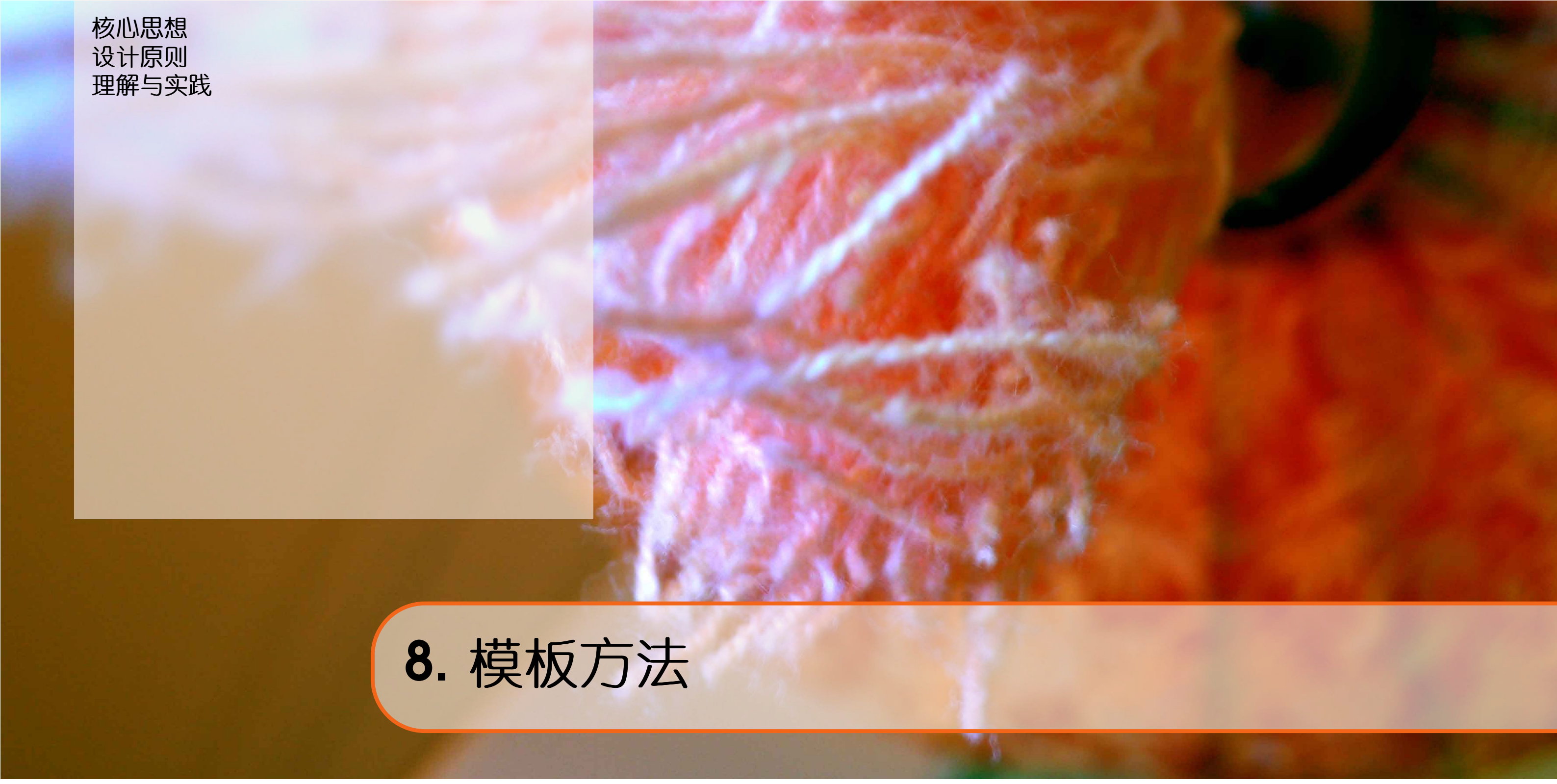
点睛程序

```java
// HomeTheaterFacade.java

package headfirst.facade.hometheater;

public class HomeTheaterFacade {
        Amplifier amp;
        Tuner tuner;
        DvdPlayer dvd;
        CdPlayer cd;
        Projector projector;
        TheaterLights lights;
        Screen screen;
        PopcornPopper popper;

        public HomeTheaterFacade(Amplifier amp,
                                 Tuner tuner,
                                 DvdPlayer dvd,
                                 CdPlayer cd,
                                 Projector projector,
                                 Screen screen,
                                 TheaterLights lights,
                                 PopcornPopper popper) {

                this.amp = amp;
                this.tuner = tuner;
                this.dvd = dvd;
                this.cd = cd;
                this.projector = projector;
                this.screen = screen;
                this.lights = lights;
                this.popper = popper;
        }

        public void watchMovie(String movie) {
                System.out.println("Get ready to watch a
                    movie...");
                popper.on();
                popper.pop();
                lights.dim(10);
                screen.down();
                projector.on();
                projector.wideScreenMode();
                amp.on();
                amp.setDvd(dvd);
                amp.setSurroundSound();
                amp.setVolume(5);
                dvd.on();
                dvd.play(movie);
        }


        public void endMovie() {
                System.out.println("Shutting movie theater down...");
                popper.off();
                lights.on();
                screen.up();
                projector.off();
                amp.off();
                dvd.stop();
                dvd.eject();
                dvd.off();
        }

        public void listenToCd(String cdTitle) {
                System.out.println("Get ready for an audiopile experence...");
                lights.on();
                amp.on();
                amp.setVolume(5);
                amp.setCd(cd);
                amp.setStereoSound();
                cd.on();
                cd.play(cdTitle);
        }

        public void endCd() {
                System.out.println("Shutting down CD...");
                amp.off();
                amp.setCd(cd);
                cd.eject();
                cd.off();
        }

        public void listenToRadio(double frequency) {
                System.out.println("Tuning in the airwaves...");
                tuner.on();
                tuner.setFrequency(frequency);
                amp.on();
                amp.setVolume(5);
                amp.setTuner(tuner);
        }

        public void endRadio() {
                System.out.println("Shutting down the tuner
                    ...");
                tuner.off();
                amp.off();
        }
}
```

```
// HomeTheaterTestDrive.java

package headfirst.facade.hometheater;

public class HomeTheaterTestDrive {
        public static void main(String[] args) {
                Amplifier amp = new Amplifier("Top-O-Line
                    Amplifier");
                Tuner tuner = new Tuner("Top-O-Line AM/FM
                    Tuner", amp);
                DvdPlayer dvd = new DvdPlayer("Top-O-Line
                    DVD Player", amp);
                CdPlayer cd = new CdPlayer("Top-O-Line CD
                    Player", amp);
                Projector projector = new Projector("Top-O-
                    Line Projector", dvd);
                TheaterLights lights = new TheaterLights("
                    Theater Ceiling Lights");
                Screen screen = new Screen("Theater Screen"
                    );
                PopcornPopper popper = new PopcornPopper("
                    Popcorn Popper");

                HomeTheaterFacade homeTheater =
                                new HomeTheaterFacade(amp,
                                    tuner, dvd, cd,
                                                projector,
                                                    screen,
                                                    lights,
                                                    popper);

                homeTheater.watchMovie("Raiders of the Lost
                     Ark");
                homeTheater.endMovie();
        }
}
```

分析：

1 装饰者模式意图是将一个接口转成两一个接口；适配器模式的意图是不改变接口，但加入责任；外观模式意图是让接口更简单。
2 外观不只是简单的简化接口，也将客户从组件的子系统中解耦。
3 当需要使用一个现有的类而其接口并不符合你的需要时，就使用适配器；当需要简化并统一一个很大的接口或者一群复杂的接口时，使用外观。



8. 模板方法

8.1 核心思想

首先先看模板方法的定义：

设计模式 10 — **模板方法.** 在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

8.2 设计原则

一些设计原则

TIP 7 好莱坞原则：别调用（打电话给）给我们，我们会调用（打电话给）你。和依赖导致原则不同，依赖导致原则教我们尽量避免使用具体类，而多用抽象。而好莱坞则是用在创建框架或组件上的一种技巧，好让底层组件能够被挂钩进计算中，而且又不会让高层组件依赖底层组件，两者目标都是在解耦。 ■

8.3 理解与实践

实战 9 咖啡和茶的冲泡方式非常相似，咖啡：将水煮沸、用沸水冲泡咖啡、把咖啡倒进被子、加糖和牛奶；茶：把水煮沸、用沸水侵泡茶叶、将茶倒入被子、加柠檬。设计代码，使其有少的代码重复。

分析：

1 “模板方法”定义了算法的步骤，把步骤的实现延迟到了子类。
2 可以在抽象类咖啡因饮料中使用一个模板方法，避免子类改变这个算法的顺序（模板方法）；对于共同的操作，比如煮沸、倒进杯，可以在抽象类中实现，也允许子类对其覆盖（钩子）；对于不同的操作，声明为抽象的方法，要求子类进行实现（抽象方法）。
3 钩子是一种被声明在抽象类中的方法，但只有空的或者默认的实现。投资的存在可以让子类有能力对算法的不同点进行挂钩。要不要挂钩有子类自行决定。下面

程序中，customerWantsCondiments()是其中的一种。钩子可以让子类实现算法中可选的部分，其另一种用法是让子类能够用机会对模板方法中某些即将发生的或刚刚发生的步骤作出发应。

4 当你的子类“必须”提供算法中某个方法或步骤的实现时，用抽象方法。如果算法的这个部分是可选的，就用钩子。

5 策略模式是封装可交换的行为，然后使用委托来决定要采用哪一个行为；工厂方法是由子类决定实例化哪个具体类；模板方法则是，有子类决定如何实现算法中的步骤。策略模式和模板方法模式都封装算法，前者用组合，后者用继承；工厂方法是模板方法的一个特殊版本。

点睛程序

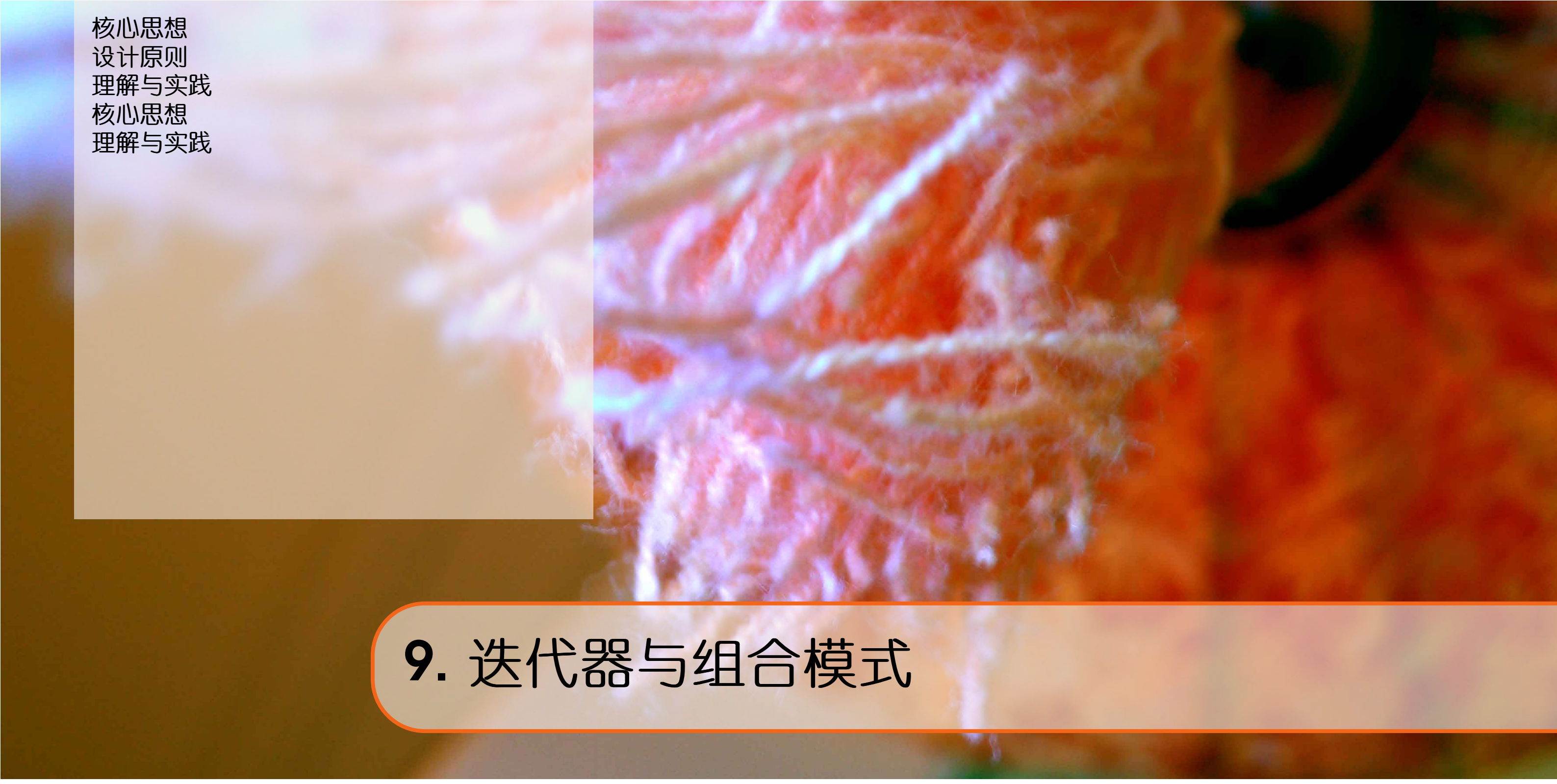
```java
// CaffeineBeverageWithHook.java

package headfirst.templatemethod.barista;

public abstract class CaffeineBeverageWithHook {

        void prepareRecipe() {
                boilWater();
                brew();
                pourInCup();
                if (customerWantsCondiments()) {
                        addCondiments();
                }
        }

        abstract void brew();

        abstract void addCondiments();

        void boilWater() {
                System.out.println("Boiling water");
        }

        void pourInCup() {
                System.out.println("Pouring into cup");
        }

        boolean customerWantsCondiments() {
                return true;
        }
}
```

```java
// CoffeeWithHook.java

package headfirst.templatemethod.barista;

import java.io.*;

```

```java
public class CoffeeWithHook extends CaffeineBeverageWithHook {

        public void brew() {
                System.out.println("Dripping Coffee through filter");
        }

        public void addCondiments() {
                System.out.println("Adding Sugar and Milk");
        }

        public boolean customerWantsCondiments() {

                String answer = getUserInput();

                if (answer.toLowerCase().startsWith("y")) {
                        return true;
                } else {
                        return false;
                }
        }

        private String getUserInput() {
                String answer = null;

                System.out.print("Would you like milk and sugar with your coffee (y/n)? ");

                BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
                try {
                        answer = in.readLine();
                } catch (IOException ioe) {
                        System.err.println("IO error trying to read your answer");
                }
                if (answer == null) {
                        return "no";
                }
                return answer;
        }
}
```

```java
// TeaWithHook.java

package headfirst.templatemethod.barista;

import java.io.*;
```

```java

public class TeaWithHook extends CaffeineBeverageWithHook {

        public void brew() {
                System.out.println("Steeping the tea");
        }

        public void addCondiments() {
                System.out.println("Adding Lemon");
        }

        public boolean customerWantsCondiments() {

                String answer = getUserInput();

                if (answer.toLowerCase().startsWith("y")) {
                        return true;
                } else {
                        return false;
                }
        }

        private String getUserInput() {

                String answer = null;

                System.out.print("Would you like lemon with
                    your tea (y/n)? ");

                BufferedReader in = new BufferedReader(new
                    InputStreamReader(System.in));
                try {
                        answer = in.readLine();
                } catch (IOException ioe) {
                        System.err.println("IO error trying
                            to read your answer");
                }
                if (answer == null) {
                        return "no";
                }
                return answer;
        }
}
```



9. 迭代器与组合模式

9.1 核心思想

首先先看迭代器模式的定义：

设计模式 11 — **迭代器模式.** 提供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。

9.2 设计原则

一些设计原则

TIP 8 一个类应该只有一个引起变化的原因。类的每个责任都有改变的潜在区域。超过一个责任，意味着超过一个改变的区域。这个原则告诉我们，尽量让每个类保持单一责任。当一个模块或一个类被设计成支持一组相关的功能时，我们说其有高内聚。 ■

9.3 理解与实践

实战 10 对象村餐厅（午餐）和对象村煎饼屋（早餐）进行了合并，但两者的菜单实现不一致，一种使用的是真正的数组MenuItem[] menuItems，一种使用的是ArrayList menuItems;现在的问题在于，女招待需要应对顾客的需要打印定制的菜单，甚至告诉顾客某个菜单项是素食的。

分析：

---

1 如果不封装由不同集合类型造成的遍历（变化的部分），一违反的封装，女招待需要知道每个菜单如何表达内部的菜单项集合（ArrayList用get和size方法，数组用字段和中括号）；二十会有重复的代码，N个循环来遍历N个不同的菜单；三是这种方法是针对的具体实现进行的编码，而不是针对的接口。

点睛程序

```java
// MenuItem.java

package headfirst.iterator.dinermergeri;

public class MenuItem {
        String name;
        String description;
        boolean vegetarian;
        double price;

        public MenuItem(String name,
                        String description,
                        boolean vegetarian,
                        double price)
        {
                this.name = name;
                this.description = description;
                this.vegetarian = vegetarian;
                this.price = price;
        }

        public String getName() {
                return name;
        }

        public String getDescription() {
                return description;
        }

        public double getPrice() {
                return price;
        }

        public boolean isVegetarian() {
                return vegetarian;
        }
}
```

```java
// Menu.java

package headfirst.iterator.dinermergeri;

import java.util.Iterator;

public interface Menu {
        public Iterator createIterator();
}
```

```java
// PancakeHouseMenu.java
```

```java

package headfirst.iterator.dinermergeri;

import java.util.ArrayList;
import java.util.Iterator;

public class PancakeHouseMenu implements Menu {
        ArrayList menuItems;

        public PancakeHouseMenu() {
                menuItems = new ArrayList();

                addItem("K&B s Pancake Breakfast", "
                   Pancakes with scrambled eggs, and toast"
                   ,
                        true,
                        2.99);

                addItem("Regular Pancake Breakfast", "
                   Pancakes with fried eggs, sausage",
                        false,
                        2.99);

                addItem("Blueberry Pancakes", "Pancakes
                   made with fresh blueberries, and
                   blueberry syrup",
                        true,
                        3.49);

                addItem("Waffles", "Waffles, with your
                   choice of blueberries or strawberries",
                        true,
                        3.59);
        }

        public void addItem(String name, String description
            ,
                             boolean vegetarian, double
                                price)
        {
                MenuItem menuItem = new MenuItem(name,
                   description, vegetarian, price);
                menuItems.add(menuItem);
        }

        public ArrayList getMenuItems() {
                return menuItems;
        }

        public Iterator createIterator() {
                return menuItems.iterator();
        }

}
```

```
// DinerMenuIterator.java

package headfirst.iterator.dinermergeri;

import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
        MenuItem[] list;
        int position = 0;

        public DinerMenuIterator(MenuItem[] list) {
                this.list = list;
        }

        public Object next() {
                MenuItem menuItem = list[position];
                position = position + 1;
                return menuItem;
        }

        public boolean hasNext() {
                if (position >= list.length || list[
                   position] == null) {
                        return false;
                } else {
                        return true;
                }
        }

        public void remove() {
                if (position <= 0) {
                        throw new IllegalStateException
                                ("You cant remove an item until
                                   youve done at least one
                                   next()");
                }
                if (list[position-1] != null) {
                        for (int i = position-1; i < (list.
                           length-1); i++) {
                                list[i] = list[i+1];
                        }
                        list[list.length-1] = null;
                }
        }
```

```java
}
```

```java
// DinerMenu.java

package headfirst.iterator.dinermergeri;

import java.util.Iterator;

public class DinerMenu implements Menu {
        static final int MAX_ITEMS = 6;
        int numberOfItems = 0;
        MenuItem[] menuItems;

        public DinerMenu() {
                menuItems = new MenuItem[MAX_ITEMS];

                addItem("Vegetarian BLT", "Fakin Bacon with
                    lettuce & tomato on whole wheat",
                        true, 2.99);
                addItem("BLT", "Bacon with lettuce & tomato
                    on whole wheat",
                        false, 2.99);
                addItem("Soup of the day", "Soup of the day
                    , with a side of potato salad",
                        false, 3.29);
                addItem("Hotdog", "A hot dog, with
                    saurkraut, relish, onions, topped with
                    cheese",
                        false, 3.05);
                addItem("Steamed Veggies and Brown Rice", "
                    Steamed vegetables over brown rice",
                        true, 3.99);
                addItem("Pasta", "Spaghetti with Marinara
                    Sauce, and a slice of sourdough bread",
                        true, 3.89);
        }

        public void addItem(String name, String description
            , boolean vegetarian, double price)
        {
                MenuItem menuItem = new MenuItem(name,
                    description, vegetarian, price);
                if (numberOfItems >= MAX_ITEMS) {
                        System.err.println("Sorry, menu is
                            full! Can not add item to menu")
                            ;
                } else {
                        menuItems[numberOfItems] = menuItem
                            ;
                        numberOfItems = numberOfItems + 1;
                }
        }

        public MenuItem[] getMenuItems() {
                return menuItems;
        }

        public Iterator createIterator() {
                return new DinerMenuIterator(menuItems);
        }

}
```

```
// Waitress.java

package headfirst.iterator.dinermergeri;

import java.util.Iterator;


public class Waitress {
        Menu pancakeHouseMenu;
        Menu dinerMenu;

        public Waitress(Menu pancakeHouseMenu, Menu
            dinerMenu) {
                this.pancakeHouseMenu = pancakeHouseMenu;
                this.dinerMenu = dinerMenu;
        }

        public void printMenu() {
                Iterator pancakeIterator = pancakeHouseMenu
                    .createIterator();
                Iterator dinerIterator = dinerMenu.
                    createIterator();

                System.out.println("MENU\n----\nBREAKFAST")
                    ;
                printMenu(pancakeIterator);
                System.out.println("\nLUNCH");
                printMenu(dinerIterator);
        }

        private void printMenu(Iterator iterator) {
                while (iterator.hasNext()) {
                        MenuItem menuItem = (MenuItem)
                            iterator.next();
                        System.out.print(menuItem.getName()
                             + ", ");
```

```java
                System.out.print(menuItem.getPrice() + " -- ");
                System.out.println(menuItem.getDescription());
            }
        }

    public void printVegetarianMenu() {
        System.out.println("\nVEGETARIAN MENU\n----\nBREAKFAST");
        printVegetarianMenu(pancakeHouseMenu.createIterator());
        System.out.println("\nLUNCH");
        printVegetarianMenu(dinerMenu.createIterator());
    }

    public boolean isItemVegetarian(String name) {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        if (isVegetarian(name, pancakeIterator)) {
            return true;
        }
        Iterator dinerIterator = dinerMenu.createIterator();
        if (isVegetarian(name, dinerIterator)) {
            return true;
        }
        return false;
    }


    private void printVegetarianMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem) iterator.next();
            if (menuItem.isVegetarian()) {
                System.out.print(menuItem.getName());
                System.out.println("\t\t" + menuItem.getPrice());
                System.out.println("\t" + menuItem.getDescription());
            }
        }
    }

```

```
        private boolean isVegetarian(String name, Iterator
            iterator) {
                while (iterator.hasNext()) {
                        MenuItem menuItem = (MenuItem)
                            iterator.next();
                        if (menuItem.getName().equals(name)
                            ) {
                                if (menuItem.isVegetarian()
                                    ) {
                                        return true;
                                }
                        }
                }
                return false;
        }
}
```

```
// MenuTestDrive.java

package headfirst.iterator.dinermergeri;

import java.util.*;

public class MenuTestDrive {
        public static void main(String args[]) {
                PancakeHouseMenu pancakeHouseMenu = new
                    PancakeHouseMenu();
                DinerMenu dinerMenu = new DinerMenu();
                Waitress waitress = new Waitress(
                    pancakeHouseMenu, dinerMenu);
                waitress.printMenu();
                waitress.printVegetarianMenu();

                System.out.println("\nCustomer asks, is the
                    Hotdog vegetarian?");
                System.out.print("Waitress says: ");
                if (waitress.isItemVegetarian("Hotdog")) {
                        System.out.println("Yes");
                } else {
                        System.out.println("No");
                }
                System.out.println("\nCustomer asks, are
                    the Waffles vegetarian?");
                System.out.print("Waitress says: ");
                if (waitress.isItemVegetarian("Waffles")) {
                        System.out.println("Yes");
                } else {
                        System.out.println("No");
                }

```

```
            }
}
```

分析：

1 迭代器模式把在元素间游走的责任交给了迭代器，而不是聚合对象。这不仅让聚合聚合的接口和实现变得简洁，也可以让聚合更专注在它所应该专注的事情上面（即管理对象集合），而不必理会遍历的事情。

9.4 核心思想

首先先看组合模式的定义：

设计模式 12 — 组合模式. 允许你讲对象组合成树形结构来表现“整体/部分”层次结构。组合能让客户以一致的方式处理个别对象及对象组合。

9.5 理解与实践

实战 11 现在考虑到顾客需要，对象村的餐厅菜单里希望能够加上一份餐后甜点的“子菜单”。

分析：

1 组合模式能让我们用树形方式创建对象的结构，树里面包含了组合Menu以及个别的对象MenuItem。我们需要一个菜单组件MenuComponent 来为叶节点MenuItem和组合结点Menu提供一个共同的接口。
2 使用组合结构，我们能把相同的操作应用在组合和个别对象上。即，在大多数情况下，我们可以忽略对象组合和个别对象之间的差别。

点睛程序

```java
// MenuComponent.java

package headfirst.composite.menu;

import java.util.*;

public abstract class MenuComponent {

            public void add(MenuComponent menuComponent) {
                        throw new UnsupportedOperationException();
            }
            public void remove(MenuComponent menuComponent) {
                        throw new UnsupportedOperationException();
            }
            public MenuComponent getChild(int i) {
                        throw new UnsupportedOperationException();
            }

```

```java
        public String getName() {
                throw new UnsupportedOperationException();
        }
        public String getDescription() {
                throw new UnsupportedOperationException();
        }
        public double getPrice() {
                throw new UnsupportedOperationException();
        }
        public boolean isVegetarian() {
                throw new UnsupportedOperationException();
        }

        public void print() {
                throw new UnsupportedOperationException();
        }
}
```

```java
// MenuItem.java

package headfirst.composite.menu;

import java.util.Iterator;
import java.util.ArrayList;

public class MenuItem extends MenuComponent {
        String name;
        String description;
        boolean vegetarian;
        double price;

        public MenuItem(String name,
                        String description,
                        boolean vegetarian,
                        double price)
        {
                this.name = name;
                this.description = description;
                this.vegetarian = vegetarian;
                this.price = price;
        }

        public String getName() {
                return name;
        }

        public String getDescription() {
                return description;
        }

        public double getPrice() {
                return price;
        }

        public boolean isVegetarian() {
                return vegetarian;
        }

        public void print() {
                System.out.print("  " + getName());
                if (isVegetarian()) {
                        System.out.print("(v)");
                }
                System.out.println(", " + getPrice());
                System.out.println("     -- " +
                    getDescription());
        }
}
```

```java
// Menu.java

package headfirst.composite.menu;

import java.util.Iterator;
import java.util.ArrayList;

public class Menu extends MenuComponent {
        ArrayList menuComponents = new ArrayList();
        String name;
        String description;

        public Menu(String name, String description) {
                this.name = name;
                this.description = description;
        }

        public void add(MenuComponent menuComponent) {
                menuComponents.add(menuComponent);
        }

        public void remove(MenuComponent menuComponent) {
                menuComponents.remove(menuComponent);
        }

        public MenuComponent getChild(int i) {
                return (MenuComponent)menuComponents.get(i)
                    ;
        }

        public String getName() {
```

```java
        return name;
    }

    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

```java
// Waitress.java

package headfirst.composite.menu;

import java.util.Iterator;

public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

```java
// MenuTestDrive.java

package headfirst.composite.menu;

import java.util.*;

public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");
        MenuComponent coffeeMenu = new Menu("COFFEE MENU", "Stuff to go with your afternoon coffee");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem("KB s Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99));
        pancakeHouseMenu.add(new MenuItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99));
        pancakeHouseMenu.add(new MenuItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries, and blueberry syrup",
            true,
            3.49));
        pancakeHouseMenu.add(new MenuItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59));

        dinerMenu.add(new MenuItem("Vegetarian BLT"
            ,
                "Fakin Bacon with lettuce & tomato
                    on whole wheat",
                true,
                2.99));
        dinerMenu.add(new MenuItem("BLT",
                "Bacon with lettuce & tomato on
                    whole wheat",
                false,
                2.99));
        dinerMenu.add(new MenuItem("Soup of the day
            ",
                "A bowl of the soup of the day,
                    with a side of potato salad",
                false,
                3.29));
        dinerMenu.add(new MenuItem("Hotdog",
                "A hot dog, with saurkraut, relish,
                    onions, topped with cheese",
                false,
                3.05));
        dinerMenu.add(new MenuItem("Steamed Veggies
            and Brown Rice",
                "Steamed vegetables over brown rice
                    ",
                true,
                3.99));

        dinerMenu.add(new MenuItem("Pasta",
                "Spaghetti with Marinara Sauce, and
                    a slice of sourdough bread",
                true,
                3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem("Apple Pie",
                "Apple pie with a flakey crust,
                    topped with vanilla icecream",
                true,
                1.59));

        dessertMenu.add(new MenuItem("Cheesecake",
                "Creamy New York cheesecake, with a
                    chocolate graham crust",
                true,
                1.99));
        dessertMenu.add(new MenuItem("Sorbet",
```

```java
                                "A scoop of raspberry and a scoop of lime",
                                true,
                                1.89));

                cafeMenu.add(new MenuItem("Veggie Burger and Air Fries",
                                "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
                                true,
                                3.99));
                cafeMenu.add(new MenuItem("Soup of the day",
                                "A cup of the soup of the day, with a side salad",
                                false,
                                3.69));
                cafeMenu.add(new MenuItem("Burrito",
                                "A large burrito, with whole pinto beans, salsa, guacamole",
                                true,
                                4.29));

                cafeMenu.add(coffeeMenu);

                coffeeMenu.add(new MenuItem("Coffee Cake",
                                "Crumbly cake topped with cinnamon and walnuts",
                                true,
                                1.59));
                coffeeMenu.add(new MenuItem("Bagel",
                                "Flavors include sesame, poppyseed, cinnamon raisin, pumpkin",
                                false,
                                0.69));
                coffeeMenu.add(new MenuItem("Biscotti",
                                "Three almond or hazelnut biscotti cookies",
                                true,
                                0.89));

                Waitress waitress = new Waitress(allMenus);

                waitress.printMenu();
        }
}
```

分析：

1 组合结构内的任意对象称为组件，组件可以是组合，也可以是叶节点。叶节点和组合结点的角色不同，所以有些方法可能不适合某些结点。面对这种情况，有时候你最好使抛出运行时异常。在实现组合模式时，有许多设计上的折中，你要根据需要平衡透明性和安全性。

2 上述方法是在print方法内部实用了迭代器，但是如果女招待需要遍历整个组合来挑出素食项，上述实现灵活度就不高了。我们需要使用下面的组合迭代器。

点睛程序

```java
// We just need to add the following code in the
    corresponding file

// Menu.java
public Iterator createIterator() {
        return new CompositeIterator(menuComponents.
            iterator());
}

// MenuItem.java
public Iterator createIterator() {
        return new NullIterator();
}
```

```java
// CompositeIterator.java

package headfirst.composite.menuiterator;


import java.util.*;

public class CompositeIterator implements Iterator {
        Stack stack = new Stack();

        public CompositeIterator(Iterator iterator) {
                stack.push(iterator);
        }

        public Object next() {
                if (hasNext()) {
                        Iterator iterator = (Iterator)
                            stack.peek();
                        MenuComponent component = (
                            MenuComponent) iterator.next();
                        if (component instanceof Menu) {
                                stack.push(component.
                                    createIterator());
                        }
                        return component;
                } else {
                        return null;
                }
        }

        public boolean hasNext() {
                if (stack.empty()) {
                        return false;
                } else {
                        Iterator iterator = (Iterator)
                            stack.peek();
                        if (!iterator.hasNext()) {
                                stack.pop();
                                return hasNext();
                        } else {
                                return true;
                        }
                }
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

```
// NullIterator.java

package headfirst.composite.menuiterator;

import java.util.Iterator;

public class NullIterator implements Iterator {

        public Object next() {
                return null;
        }

        public boolean hasNext() {
                return false;
        }

        public void remove() {
                throw new UnsupportedOperationException();
        }
}
```

```
// Waitress.java

package headfirst.composite.menuiterator;

import java.util.Iterator;

```

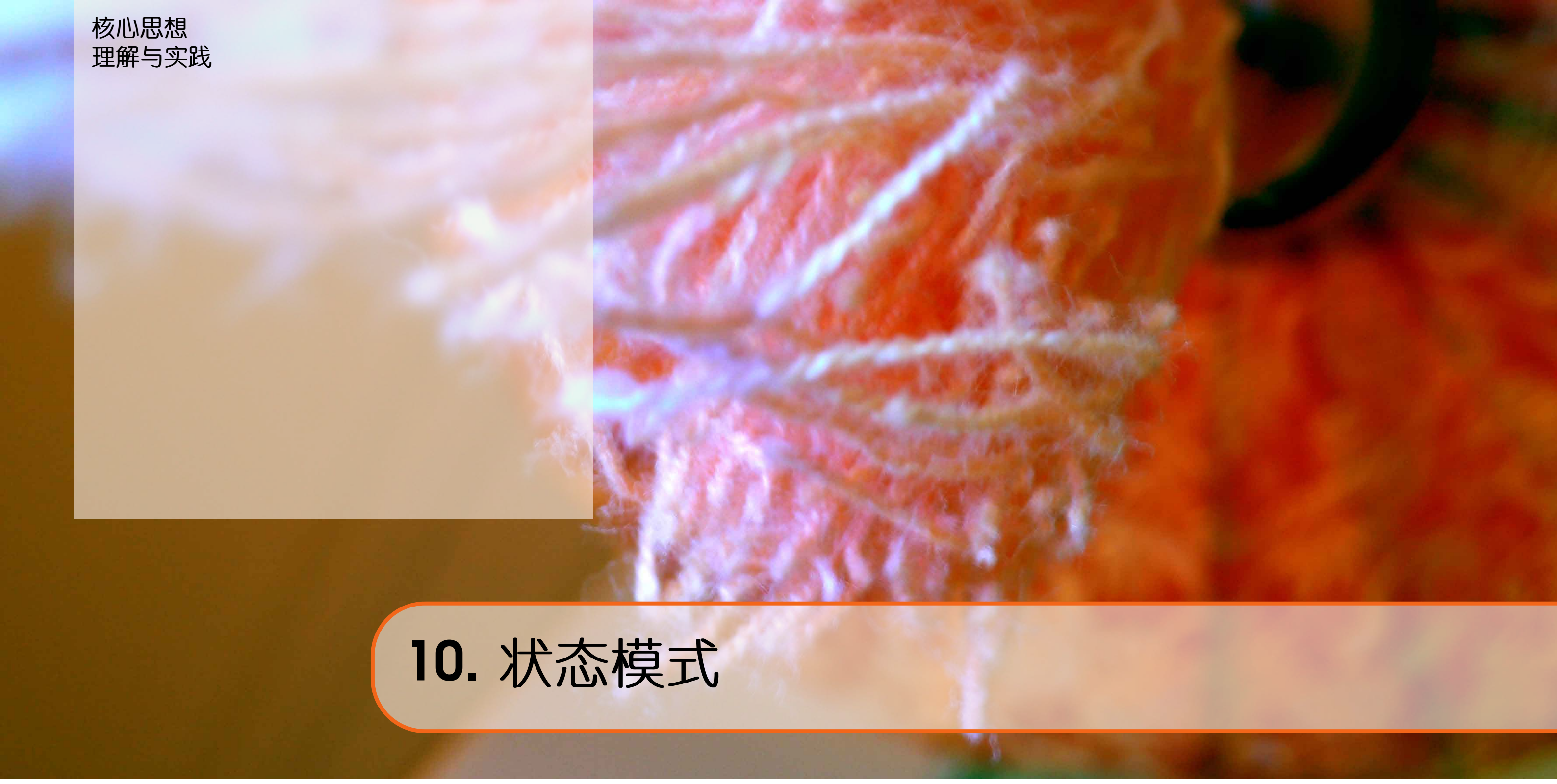
```java
public class Waitress {
        MenuComponent allMenus;

        public Waitress(MenuComponent allMenus) {
                this.allMenus = allMenus;
        }

        public void printMenu() {
                allMenus.print();
        }

        public void printVegetarianMenu() {
                Iterator iterator = allMenus.createIterator();

                System.out.println("\nVEGETARIAN MENU\n----");
                while (iterator.hasNext()) {
                        MenuComponent menuComponent =
                                        (MenuComponent)iterator.next();
                        try {
                                if (menuComponent.isVegetarian()) {
                                        menuComponent.print();
                                }
                        } catch (UnsupportedOperationException e) {}
                }
        }
}
```

```java
// MenuTestDrive.java

package headfirst.composite.menuiterator;

import java.util.*;

public class MenuTestDrive {
        public static void main(String args[]) {

                MenuComponent pancakeHouseMenu =
                        new Menu("PANCAKE HOUSE MENU", "Breakfast");
                MenuComponent dinerMenu =
                        new Menu("DINER MENU", "Lunch");
                MenuComponent cafeMenu =
                        new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem("KB Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99));
        pancakeHouseMenu.add(new MenuItem("Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99));
        pancakeHouseMenu.add(new MenuItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries, and blueberry syrup",
            true,
            3.49));
        pancakeHouseMenu.add(new MenuItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries",
            true,
            3.59));

        dinerMenu.add(new MenuItem("Vegetarian BLT",
            "Fakin Bacon with lettuce & tomato on whole wheat",
            true,
            2.99));
        dinerMenu.add(new MenuItem("BLT",
            "Bacon with lettuce & tomato on whole wheat",
            false,
            2.99));
        dinerMenu.add(new MenuItem("Soup of the day",
                "A bowl of the soup of the day, with a side of potato salad",
                false,
                3.29));
        dinerMenu.add(new MenuItem("Hotdog",
                "A hot dog, with saurkraut, relish, onions, topped with cheese",
                false,
                3.05));
        dinerMenu.add(new MenuItem("Steamed Veggies and Brown Rice",
                "A medly of steamed vegetables over brown rice",
                true,
                3.99));

        dinerMenu.add(new MenuItem("Pasta",
                "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
                true,
                3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem("Apple Pie",
                "Apple pie with a flakey crust, topped with vanilla icecream",
                true,
                1.59));
        dessertMenu.add(new MenuItem("Cheesecake",
                "Creamy New York cheesecake, with a chocolate graham crust",
                true,
                1.99));
        dessertMenu.add(new MenuItem("Sorbet",
                "A scoop of raspberry and a scoop of lime",
                true,
                1.89));

        cafeMenu.add(new MenuItem("Veggie Burger and Air Fries",
                "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
                true,
                3.99));
```

```
                    cafeMenu.add(new MenuItem("Soup of the day"
                        ,
                              "A cup of the soup of the day, with
                                   a side salad",
                              false,
                              3.69));
                    cafeMenu.add(new MenuItem("Burrito",
                              "A large burrito, with whole pinto
                                  beans, salsa, guacamole",
                              true,
                              4.29));

                    Waitress waitress = new Waitress(allMenus);

                    waitress.printVegetarianMenu();

          }
}
```

分析：

1 策略模式是封装可互换的行为，并使用委托决定使用哪一个；观察者模式是当某个状态改变时，允许一群对象被通知到；适配器模式是改变一个或多个类的接口；外观模式是简化一群类的接口；迭代器模式是提供一个方式来遍历集合，而无须暴露集合的实现；组合模式是客户可以讲对象的集合以及个别对象一视同仁。



10. 状态模式

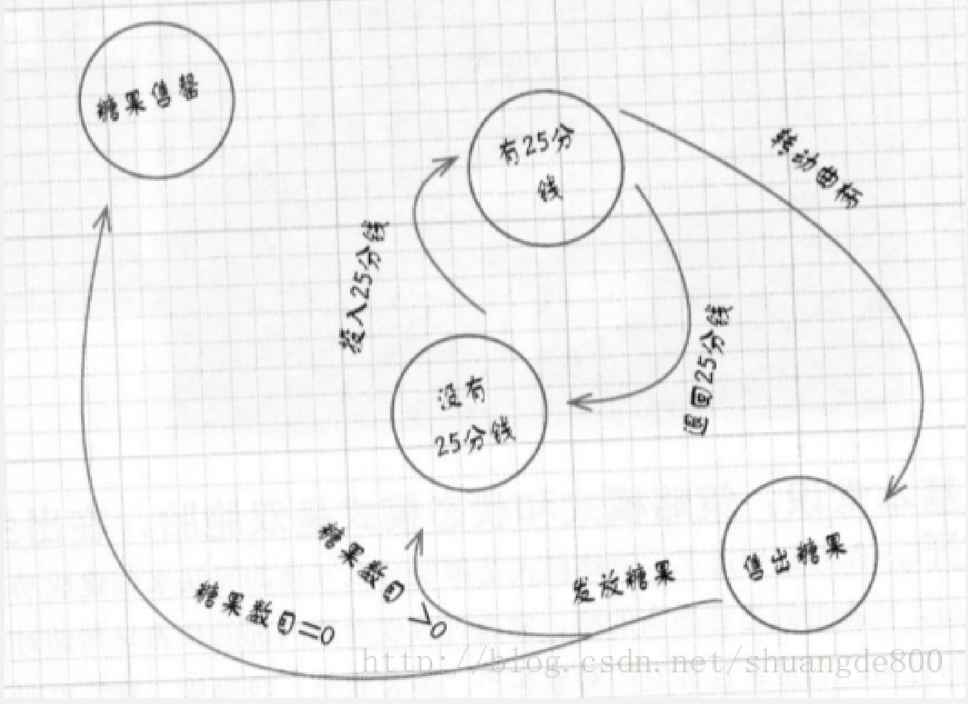
10.1 核心思想

首先先看状态模式的定义：

设计模式 13 — 状态模式. 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

10.2 理解与实践

实 战 12 万 能 糖 果 公 司 需 要 设 计 一 个 糖 果 机 的 控 制 器， 其 工 作 状 态 如 图， 需 要 让 设 计 能 够 尽 量 有 弹 性 而 且 好 维 护， 将 来 可 能 扩 展 更 多 的 行 为。



http://blog.csdn.net/shuangde800

错误示范

```java
// GumballMachine.java

package headfirst.state.gumball;

public class GumballMachine {

        final static int SOLD_OUT = 0;
        final static int NO_QUARTER = 1;
        final static int HAS_QUARTER = 2;
        final static int SOLD = 3;

        int state = SOLD_OUT;
        int count = 0;

        public GumballMachine(int count) {
                this.count = count;
                if (count > 0) {
                        state = NO_QUARTER;
                }
        }

        public void insertQuarter() {
                if (state == HAS_QUARTER) {
                        System.out.println("You can not
                            insert another quarter");
                } else if (state == NO_QUARTER) {
                        state = HAS_QUARTER;
                        System.out.println("You inserted a
                            quarter");
                } else if (state == SOLD_OUT) {
                        System.out.println("You can not
                            insert a quarter, the machine is
                             sold out");
                } else if (state == SOLD) {
                System.out.println("Please wait, we are
                    already giving you a gumball");
                }
        }

        public void ejectQuarter() {
                if (state == HAS_QUARTER) {
                        System.out.println("Quarter
                            returned");
                        state = NO_QUARTER;
                } else if (state == NO_QUARTER) {
                        System.out.println("You have not
                            inserted a quarter");
        } else if (state == SOLD) {
            System.out.println("Sorry, you already turned the crank");
        } else if (state == SOLD_OUT) {
        System.out.println("You can not eject, you have not inserted a quarter yet");
        }
    }




    public void turnCrank() {
        if (state == SOLD) {
            System.out.println("Turning twice does not get you another gumball!");
        } else if (state == NO_QUARTER) {
            System.out.println("You turned but there is no quarter");
        } else if (state == SOLD_OUT) {
            System.out.println("You turned, but there are no gumballs");
        } else if (state == HAS_QUARTER) {
            System.out.println("You turned...");
            state = SOLD;
            dispense();
        }
    }

    public void dispense() {
        if (state == SOLD) {
            System.out.println("A gumball comes rolling out the slot");
            count = count - 1;
            if (count == 0) {
                System.out.println("Oops, out of gumballs!");
                state = SOLD_OUT;
            } else {
                state = NO_QUARTER;
            }
        } else if (state == NO_QUARTER) {
            System.out.println("You need to pay first");
        } else if (state == SOLD_OUT) {
            System.out.println("No gumball dispensed");
```

```java
                    } else if (state == HAS_QUARTER) {
                            System.out.println("No gumball
                                dispensed");
                    }
            }

            public void refill(int numGumBalls) {
                    this.count = numGumBalls;
                    state = NO_QUARTER;
            }

            public String toString() {
                    StringBuffer result = new StringBuffer();
                    result.append("\nMighty Gumball, Inc.");
                    result.append("\nJava-enabled Standing
                        Gumball Model #2004\n");
                    result.append("Inventory: " + count + "
                        gumball");
                    if (count != 1) {
                            result.append("s");
                    }
                    result.append("\nMachine is ");
                    if (state == SOLD_OUT) {
                            result.append("sold out");
                    } else if (state == NO_QUARTER) {
                            result.append("waiting for quarter"
                                );
                    } else if (state == HAS_QUARTER) {
                            result.append("waiting for turn of
                                crank");
                    } else if (state == SOLD) {
                            result.append("delivering a gumball
                                ");
                    }
                    result.append("\n");
                    return result.toString();
            }
}
```

```java
// GumballMachineTestDrive.java

package headfirst.state.gumball;

public class GumballMachineTestDrive {

        public static void main(String[] args) {
                GumballMachine gumballMachine = new
                    GumballMachine(5);

                System.out.println(gumballMachine);

                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();

                System.out.println(gumballMachine);

                gumballMachine.insertQuarter();
                gumballMachine.ejectQuarter();
                gumballMachine.turnCrank();

                System.out.println(gumballMachine);

                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();
                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();
                gumballMachine.ejectQuarter();

                System.out.println(gumballMachine);

                gumballMachine.insertQuarter();
                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();
                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();
                gumballMachine.insertQuarter();
                gumballMachine.turnCrank();

                System.out.println(gumballMachine);
        }
}
```

分析：

1 上述程序考虑周详，但很不容易进行扩展，加入一个新的状态的话会大篇幅修改程序。
2 遵循“封装变化”的原则，我们应该试着局部化每个状态的行为，这样一来，如果我们针对某个状态做个改变，就不会把其他的代码给搞乱了。
3 我们可以这样做，首先定义一个State接口。接口内糖果机每个动作都有一个对应的方法；然后为状态机中的每个状态实现状态类，这些类负责在对应的状态下进行机器的行为；最后，摆脱旧的条件代码，将动作委托到状态类。

点睛程序

```java
// GumballMachine.java

package headfirst.state.gumballstatewinner;

public class GumballMachine {

    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;

    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        winnerState = new WinnerState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
    }

    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }

    int getCount() {
```

```
                return count;
        }

        void refill(int count) {
                this.count = count;
                state = noQuarterState;
        }

    public State getState() {
        return state;
    }

    public State getSoldOutState() {
        return soldOutState;
    }

    public State getNoQuarterState() {
        return noQuarterState;
    }

    public State getHasQuarterState() {
        return hasQuarterState;
    }

    public State getSoldState() {
        return soldState;
    }

    public State getWinnerState() {
        return winnerState;
    }

        public String toString() {
                StringBuffer result = new StringBuffer();
                result.append("\nMighty Gumball, Inc.");
                result.append("\nJava-enabled Standing
                   Gumball Model #2004");
                result.append("\nInventory: " + count + "
                   gumball");
                if (count != 1) {
                        result.append("s");
                }
                result.append("\n");
                result.append("Machine is " + state + "\n")
                   ;
                return result.toString();
        }
}
```

```
// State.java
```

```

package headfirst.state.gumballstatewinner;

public interface State {

        public void insertQuarter();
        public void ejectQuarter();
        public void turnCrank();
        public void dispense();
}
```

```
// HasQuarterState.java

package headfirst.state.gumballstatewinner;

import java.util.Random;

public class HasQuarterState implements State {
        Random randomWinner = new Random(System.
            currentTimeMillis());
        GumballMachine gumballMachine;

        public HasQuarterState(GumballMachine
            gumballMachine) {
                this.gumballMachine = gumballMachine;
        }

        public void insertQuarter() {
                System.out.println("You can not insert
                    another quarter");
        }

        public void ejectQuarter() {
                System.out.println("Quarter returned");
                gumballMachine.setState(gumballMachine.
                    getNoQuarterState());
        }

        public void turnCrank() {
                System.out.println("You turned...");
                int winner = randomWinner.nextInt(10);
                if ((winner == 0) && (gumballMachine.
                    getCount() > 1)) {
                        gumballMachine.setState(
                            gumballMachine.getWinnerState())
                            ;
                } else {
                        gumballMachine.setState(
                            gumballMachine.getSoldState());
                }
```

```java
        }

    public void dispense() {
        System.out.println("No gumball dispensed");
    }

        public String toString() {
                return "waiting for turn of crank";
        }
}
```

```java
// NoQuarterState.java

package headfirst.state.gumballstatewinner;

public class NoQuarterState implements State {
    GumballMachine gumballMachine;

    public NoQuarterState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

        public void insertQuarter() {
                System.out.println("You inserted a quarter");
                gumballMachine.setState(gumballMachine.getHasQuarterState());
        }

        public void ejectQuarter() {
                System.out.println("You haven not inserted a quarter");
        }

        public void turnCrank() {
                System.out.println("You turned, but there is no quarter");
         }

        public void dispense() {
                System.out.println("You need to pay first");
        }

        public String toString() {
                return "waiting for quarter";
        }
}
```

```java
// SoldState.java
```

```java

package headfirst.state.gumballstatewinner;

public class SoldState implements State {
    GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

        public void insertQuarter() {
                System.out.println("Please wait, we are already giving you a gumball");
        }

        public void ejectQuarter() {
                System.out.println("Sorry, you already turned the crank");
        }

        public void turnCrank() {
                System.out.println("Turning twice doesn not get you another gumball!");
        }

        public void dispense() {
                gumballMachine.releaseBall();
                if (gumballMachine.getCount() > 0) {
                        gumballMachine.setState(gumballMachine.getNoQuarterState());
                } else {
                        System.out.println("Oops, out of gumballs!");
                        gumballMachine.setState(gumballMachine.getSoldOutState());
                }
        }

        public String toString() {
                return "dispensing a gumball";
        }
}
```

```java
// SoldOutState.java

package headfirst.state.gumballstatewinner;

public class SoldOutState implements State {
    GumballMachine gumballMachine;

    public SoldOutState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

        public void insertQuarter() {
                System.out.println("You can not insert a
                    quarter, the machine is sold out");
        }

        public void ejectQuarter() {
                System.out.println("You can not eject, you
                    haven not inserted a quarter yet");
        }

        public void turnCrank() {
                System.out.println("You turned, but there
                    are no gumballs");
        }

        public void dispense() {
                System.out.println("No gumball dispensed");
        }

        public String toString() {
                return "sold out";
        }
}
```

```java
// WinnerState.java

package headfirst.state.gumballstatewinner;

public class WinnerState implements State {
    GumballMachine gumballMachine;

    public WinnerState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

        public void insertQuarter() {
                System.out.println("Please wait, we are
                    already giving you a Gumball");
        }

        public void ejectQuarter() {
                System.out.println("Please wait, we are
                    already giving you a Gumball");
        }

        public void turnCrank() {
                System.out.println("Turning again doesn not
                    get you another gumball!");
        }

        public void dispense() {
                System.out.println("YOU are A WINNER! You
                    get two gumballs for your quarter");
                gumballMachine.releaseBall();
                if (gumballMachine.getCount() == 0) {
                        gumballMachine.setState(
                            gumballMachine.getSoldOutState()
                            );
                } else {
                        gumballMachine.releaseBall();
                        if (gumballMachine.getCount() > 0)
                            {
                                gumballMachine.setState(
                                    gumballMachine.
                                    getNoQuarterState());
                        } else {
                System.out.println("Oops, out of gumballs!"
                    );
                                gumballMachine.setState(
                                    gumballMachine.
                                    getSoldOutState());
                        }
                }
        }

        public String toString() {
                return "despensing two gumballs for your
                    quarter, because YOU are A WINNER!";
        }
}
```

```java
// GumballMachineTestDrive.java

package headfirst.state.gumballstatewinner;

public class GumballMachineTestDrive {

        public static void main(String[] args) {
                GumballMachine gumballMachine =
                        new GumballMachine(10);

                System.out.println(gumballMachine);

                gumballMachine.insertQuarter();
```

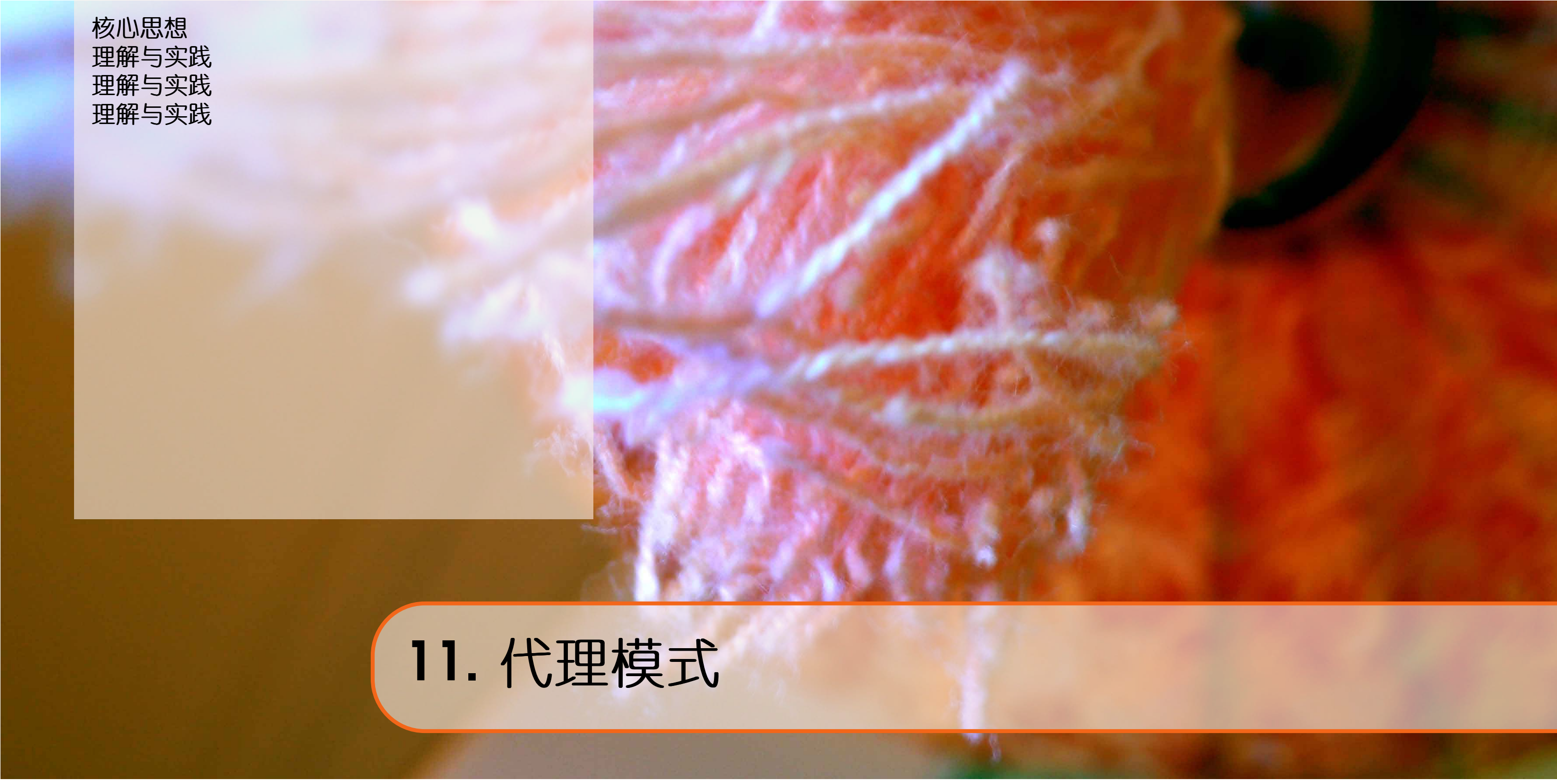
```
                    gumballMachine.turnCrank();
                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();

                    System.out.println(gumballMachine);

                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();
                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();

                    System.out.println(gumballMachine);

                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();
                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();

                    System.out.println(gumballMachine);

                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();
                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();

                    System.out.println(gumballMachine);

                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();
                    gumballMachine.insertQuarter();
                    gumballMachine.turnCrank();

                    System.out.println(gumballMachine);
          }
}
```

分析：

1 状态模式是讲一群行为封装在状态对象中，context的行为随时可以委托到那些状态对象中的一个。随着时间的流逝，当前状态在状态对象中游走改变。而以策略模式而言，客户通常主动指定context所要组合的策略对象是哪一个。固然策略模式能让我们具有弹性，能够在运行时改变策略，但对于某个context对象来说，通常只有一个最适当的策略对象。

2 状态模式允许一个对象基于内部状态而拥有不同的行为。状态模式和策略模式有相同的类图，但是他们的意图不同；策略模式通常会用行为或算法来配置context类。

3 策略模式是将可以互换的行为封装起来，然后使用委托的方法，决定使用哪一个行为；模板方法是由子类决定如何实现算法中的某些步骤；策略模式是将可以互换的行为封装起来，然后使用委托的方法，决定使用哪一个行为。



11. 代理模式

11.1 核心思想

首先先看代理模式的定义：

设计模式 14 — **代理模式.** 为另一个对象提供一个替身或占位符以控制对这个对象的访问

11.2 理解与实践

实战 13 对之前我们实现的糖果机添加新的功能，使得总裁可以远程查看不同地域糖果机的不同状态

分析：

1 对此问题我们可以用远程代理，就好比“远程对象的本地代表”，你的客户所做的就像是在做远程方法调用
2 看下面的例子，使用了Java的RMI

点睛程序

```
// GumballMachine.java

package headfirst.proxy.gumball;

import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
                extends UnicastRemoteObject implements
                    GumballMachineRemote
{
        State soldOutState;
```

```java
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State winnerState;

    State state = soldOutState;
    int count = 0;
    String location;

    public GumballMachine(String location, int numberGumballs) throws RemoteException {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        winnerState = new WinnerState(this);

        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        }
        this.location = location;
    }


    public void insertQuarter() {
        state.insertQuarter();
    }

    public void ejectQuarter() {
        state.ejectQuarter();
    }

    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }

    void setState(State state) {
        this.state = state;
    }

    void releaseBall() {
        System.out.println("A gumball comes rolling out the slot...");
        if (count != 0) {
            count = count - 1;
        }
        }

        public void refill(int count) {
                this.count = count;
                state = noQuarterState;
        }

        public int getCount() {
                return count;
        }

    public State getState() {
        return state;
    }

    public String getLocation() {
        return location;
    }

    public State getSoldOutState() {
        return soldOutState;
    }

    public State getNoQuarterState() {
        return noQuarterState;
    }

    public State getHasQuarterState() {
        return hasQuarterState;
    }

    public State getSoldState() {
        return soldState;
    }

    public State getWinnerState() {
        return winnerState;
    }

        public String toString() {
                StringBuffer result = new StringBuffer();
                result.append("\nMighty Gumball, Inc.");
                result.append("\nJava-enabled Standing Gumball Model #2004");
                result.append("\nInventory: " + count + " gumball");
                if (count != 1) {
                        result.append("s");
                }
                result.append("\n");
                result.append("Machine is " + state + "\n");
                return result.toString();
        }
}
```

```java
// State.java

package headfirst.proxy.gumball;

import java.io.*;

public interface State extends Serializable {
        public void insertQuarter();
        public void ejectQuarter();
        public void turnCrank();
        public void dispense();
}
```

```java
// SoldState.java

package headfirst.proxy.gumball;

public class SoldState implements State {
    transient GumballMachine gumballMachine;

    public SoldState(GumballMachine gumballMachine) {
        this.gumballMachine = gumballMachine;
    }

        public void insertQuarter() {
                System.out.println("Please wait, we are already giving you a gumball");
        }

        public void ejectQuarter() {
                System.out.println("Sorry, you already turned the crank");
        }

        public void turnCrank() {
                System.out.println("Turning twice does not get you another gumball!");
        }

        public void dispense() {
                gumballMachine.releaseBall();
                if (gumballMachine.getCount() > 0) {
```

```java
            gumballMachine.setState(gumballMachine.getNoQuarterState());
        } else {
            System.out.println("Oops, out of gumballs!");
            gumballMachine.setState(gumballMachine.getSoldOutState());
        }
    }

    public String toString() {
        return "dispensing a gumball";
    }
}
```

```java
// GumballMachineRemote.java

package headfirst.proxy.gumball;

import java.rmi.*;

public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```

```java
// GumballMonitor.java

package headfirst.proxy.gumball;

import java.rmi.*;

public class GumballMonitor {
    GumballMachineRemote machine;

    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }

    public void report() {
        try {
            System.out.println("Gumball Machine: " + machine.getLocation());
            System.out.println("Current inventory: " + machine.getCount() + " gumballs");
```

```
                    System.out.println("Current state: " + machine.getState());
            } catch (RemoteException e) {
                    e.printStackTrace();
            }
        }
}
```

```
package headfirst.proxy.gumball;
import java.rmi.*;

public class GumballMachineTestDrive {

        public static void main(String[] args) {
                GumballMachineRemote gumballMachine = null;
                int count;

                if (args.length < 2) {
                        System.out.println("GumballMachine <name> <inventory>");
                        System.exit(1);
                }

                try {
                        count = Integer.parseInt(args[1]);

                        gumballMachine =
                                new GumballMachine(args[0], count);
                        Naming.rebind("//" + args[0] + "/gumballmachine", gumballMachine);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

```
package headfirst.proxy.gumball;

// GumballMonitorTestDrive.java

import java.rmi.*;

public class GumballMonitorTestDrive {

        public static void main(String[] args) {
                String[] location = {"rmi://santafe.mightygumball.com/gumballmachine",
                                                "rmi://boulder.
                                                    mightygumball.com/
                                                    gumballmachine",
                                                "rmi://seattle.
                                                    mightygumball.com/
                                                    gumballmachine"};

                        if (args.length >= 0)
            {
                location = new String[1];
                location[0] = "rmi://" + args[0] + "/
                    gumballmachine";
            }

                        GumballMonitor[] monitor = new
                            GumballMonitor[location.length];



                        for (int i=0;i < location.length; i++) {
                                    try {
                                    GumballMachineRemote machine =
                                                (GumballMachineRemote)
                                                    Naming.lookup(location[i
                                                    ]);
                                    monitor[i] = new GumballMonitor(
                                        machine);
                                                System.out.println(monitor[
                                                    i]);
                        } catch (Exception e) {
                        e.printStackTrace();
                        }
                        }

                        for(int i=0; i < monitor.length; i++) {
                                    monitor[i].report();
                        }
            }
}
```

分析：

1 远程代理可以作为另一个JVM上对象的本地代表。调用代理的方法，会被代理李永网络转发到远程执行，并且结果会通过网络返给代理，再由代理将结果转给客户。

2 上述代码首先为GumballMachine创建一个远程接口GumballMachineRemote，改口提供了一组和口译远程调用的方法，并确定接口的所有返回类型都是可序列化的。

3 服务完成后，在RMI registry中注册，好让客户可以找到他，参考GumballMachineTestDrive1中代码；然后GumbalMonitor就可以代理调用。

4 使用带领模式创建代表对象，让代表对象控制某对象的访问，被代理的对象处了像上述的远程对象外，还可以是开销大的对象或需要安全控制的对象，下面的例子分别是虚拟代理和保护代理

11.3 理解与实践

实战 14 建立一个应用程序来展现你最喜欢的CD封面，有事限于连接带宽和网络负载，下载可能需要一段时间，我们想实现在等待图像加载的时候来显示一些东西，程序也不被挂起，一旦图像加载完成，就用刚下载的图像来替代显示。

点睛程序

```java
// ImageComponent.java

package headfirst.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
        private Icon icon;

        public ImageComponent(Icon icon) {
                this.icon = icon;
        }

        public void setIcon(Icon icon) {
                this.icon = icon;
        }

        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                int w = icon.getIconWidth();
                int h = icon.getIconHeight();
                int x = (800 - w)/2;
                int y = (600 - h)/2;
                icon.paintIcon(this, g, x, y);
        }
}
```

```java
// ImageProxy.java

package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
class ImageProxy implements Icon {
        ImageIcon imageIcon;
        URL imageURL;
        Thread retrievalThread;
        boolean retrieving = false;

        public ImageProxy(URL url) { imageURL = url; }

        public int getIconWidth() {
                if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
                        return 800;
                }
        }

        public int getIconHeight() {
                if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
                        return 600;
                }
        }

        public void paintIcon(final Component c, Graphics
           g, int x,  int y) {
                if (imageIcon != null) {
                        imageIcon.paintIcon(c, g, x, y);
                } else {
                        g.drawString("Loading CD cover,
                           please wait...", x+300, y+190);
                        if (!retrieving) {
                                retrieving = true;

                                retrievalThread = new
                                   Thread(new Runnable() {
                                        public void run() {
                                        try {
                                        imageIcon = new
                                        ImageIcon(imageURL,
                                           "CD Cover");
                                        c.repaint();
                                        } catch (Exception
                                           e)
                                        {
                                           e.
                                              printStackTrace
                                              ();
                                        }
```

```
                                                        }
                                                    });
                                                    retrievalThread.start();
                                        }
                            }
                }
}
```

```
// mageProxyTestDrive.java

package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class ImageProxyTestDrive {
        ImageComponent imageComponent;
        JFrame frame = new JFrame("CD Cover Viewer");
    JMenuBar menuBar;
    JMenu menu;
        Hashtable cds = new Hashtable();

        public static void main (String[] args) throws
            Exception {
                ImageProxyTestDrive testDrive = new
                    ImageProxyTestDrive();
        }

        public ImageProxyTestDrive() throws Exception{
                cds.put("Ambient: Music for Airports","http
                    ://images.amazon.com/images/P/B000003S2K
                    .01.LZZZZZZZ.jpg");
        cds.put("Buddha Bar","http://images.amazon.com/
            images/P/B00009XBYK.01.LZZZZZZZ.jpg");
        cds.put("Ima","http://images.amazon.com/images/P/
            B000005IRM.01.LZZZZZZZ.jpg");
        cds.put("Karma","http://images.amazon.com/images/P/
            B000005DCB.01.LZZZZZZZ.gif");
        cds.put("MCMXC A.D.","http://images.amazon.com/
            images/P/B000002URV.01.LZZZZZZZ.jpg");
        cds.put("Northern Exposure","http://images.amazon.
            com/images/P/B000003SFN.01.LZZZZZZZ.jpg");
        cds.put("Selected Ambient Works, Vol. 2","http://
            images.amazon.com/images/P/B000002MNZ.01.
            LZZZZZZZ.jpg");

```

```java
        URL initialURL = new URL((String)cds.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Favorite CDs");
    menuBar.add(menu);
    frame.setJMenuBar(menuBar);

        for(Enumeration e = cds.keys(); e.hasMoreElements();) {
            String name = (String)e.nextElement();
        JMenuItem menuItem = new JMenuItem(name);
        menu.add(menuItem);
        menuItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent event) {
                    imageComponent.setIcon(new ImageProxy(getCDUrl(event.getActionCommand())));
                        frame.repaint();
            }
        });
        }

        // set up frame and menus

        Icon icon = new ImageProxy(initialURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }

    URL getCDUrl(String name) {
        try {
            return new URL((String)cds.get(name));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

分析：

1 虚拟代理作为创建开销大的对象的代表。虚拟代理经常直到我们真正需要一个对象的时候才创建它。但对象在创建前和创建中时，由虚拟代理来扮演对象的替身，对象创建后，代理就会讲请求直接委托给对象。
2 上例中ImageProxy是控制ImageIcon的访问，代理将客户从ImageIcon解耦了
3 下面是个保护代理的例子，使用了Java的内置功能

11.4 理解与实践

实战 15 对象村的约会配对服务，顾客不可以改变自己的HotOrNot评分，也不可以改变其他顾客的信息。

点睛程序

```java
// PersonBean.java

package headfirst.proxy.javaproxy;

public interface PersonBean {

        String getName();
        String getGender();
        String getInterests();
        int getHotOrNotRating();

    void setName(String name);
    void setGender(String gender);
    void setInterests(String interests);
    void setHotOrNotRating(int rating);

}
```

```java
// PersonBeanImpl.java

package headfirst.proxy.javaproxy;

public class PersonBeanImpl implements PersonBean {
        String name;
        String gender;
        String interests;
        int rating;
        int ratingCount = 0;

        public String getName() {
                return name;
        }

        public String getGender() {
                return gender;
        }

        public String getInterests() {
                return interests;
        }

        public int getHotOrNotRating() {
                if (ratingCount == 0) return 0;
                return (rating/ratingCount);
        }


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

        public void setGender(String gender) {
                this.gender = gender;
        }

        public void setInterests(String interests) {
                this.interests = interests;
        }

        public void setHotOrNotRating(int rating) {
                this.rating += rating;
                ratingCount++;
        }
}
```

```java
// OwnerInvocationHandler.java

package headfirst.proxy.javaproxy;

import java.lang.reflect.*;

public class OwnerInvocationHandler implements
    InvocationHandler {
        PersonBean person;

        public OwnerInvocationHandler(PersonBean person) {
                this.person = person;
        }

        public Object invoke(Object proxy, Method method,
            Object[] args)
                        throws IllegalAccessException {

                try {
                        if (method.getName().startsWith("
                            get")) {
```

```java
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                throw new IllegalAccessException();
            } else if (method.getName().startsWith("set")) {
                return method.invoke(person, args);
            }
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
        return null;
    }
}
```

```java
// NonOwnerInvocationHandler.java

package headfirst.proxy.javaproxy;

import java.lang.reflect.*;

public class NonOwnerInvocationHandler implements InvocationHandler {
    PersonBean person;

    public NonOwnerInvocationHandler(PersonBean person) {
        this.person = person;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
            throws IllegalAccessException {

        try {
            if (method.getName().startsWith("get")) {
                return method.invoke(person, args);
            } else if (method.getName().equals("setHotOrNotRating")) {
                return method.invoke(person, args);
            } else if (method.getName().startsWith("set")) {
```

```
                                        throw new
                                            IllegalAccessException()
                                            ;
                              }
          } catch (InvocationTargetException e) {
               e.printStackTrace();
          }
                    return null;
          }
}
```

```
// MatchMakingTestDrive.java

package headfirst.proxy.javaproxy;

import java.lang.reflect.*;
import java.util.*;

public class MatchMakingTestDrive {
          Hashtable datingDB = new Hashtable();

          public static void main(String[] args) {
                    MatchMakingTestDrive test = new
                        MatchMakingTestDrive();
                    test.drive();
          }

          public MatchMakingTestDrive() {
                    initializeDatabase();
          }

          public void drive() {
                    PersonBean joe = getPersonFromDatabase("Joe
                         Javabean");
                    PersonBean ownerProxy = getOwnerProxy(joe);
                    System.out.println("Name is " + ownerProxy.
                        getName());
                    ownerProxy.setInterests("bowling, Go");
                    System.out.println("Interests set from
                        owner proxy");
                    try {
                              ownerProxy.setHotOrNotRating(10);
                    } catch (Exception e) {
                              System.out.println("Can not set
                                  rating from owner proxy");
                    }
                    System.out.println("Rating is " +
                        ownerProxy.getHotOrNotRating());

```

```java
        PersonBean nonOwnerProxy = getNonOwnerProxy(joe);
        System.out.println("Name is " + nonOwnerProxy.getName());
        try {
            nonOwnerProxy.setInterests("bowling, Go");
        } catch (Exception e) {
            System.out.println("Can not set interests from non owner proxy");
        }
        nonOwnerProxy.setHotOrNotRating(3);
        System.out.println("Rating set from non owner proxy");
        System.out.println("Rating is " + nonOwnerProxy.getHotOrNotRating());
    }

    PersonBean getOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
        person.getClass().getClassLoader(),
        person.getClass().getInterfaces(),
        new OwnerInvocationHandler(person));
    }

    PersonBean getNonOwnerProxy(PersonBean person) {

    return (PersonBean) Proxy.newProxyInstance(
        person.getClass().getClassLoader(),
        person.getClass().getInterfaces(),
        new NonOwnerInvocationHandler(person));
    }

    PersonBean getPersonFromDatabase(String name) {
        return (PersonBean)datingDB.get(name);
    }

    void initializeDatabase() {
        PersonBean joe = new PersonBeanImpl();
        joe.setName("Joe Javabean");
        joe.setInterests("cars, computers, music");
        joe.setHotOrNotRating(7);
        datingDB.put(joe.getName(), joe);

        PersonBean kelly = new PersonBeanImpl();
        kelly.setName("Kelly Klosure");
        kelly.setInterests("ebay, movies, music");
```

```
                    kelly.setHotOrNotRating(6);
                    datingDB.put(kelly.getName(), kelly);
          }
}
```

分析：

1 远程代理管理客户和远程对象之间的交互
2 虚拟代理控制访问实例化开销大的对象
3 保护代理基于调用者控制对象方法的访问
4 还有很多其他代理，如缓存代理、同步代理、防火墙代理、写入时复制代理

12. 复合模式

12.1 核心思想

首先先看复合模式的定义：

设计模式 15 — 复合模式. 复合模式结合两个或以上的模式，组成一个解决方案，解决以再发生的一般性问题。

12.2 理解与实践

实战 16 有一群会叫的鸭子，要实现如下功能：想要在使用鸭子的地方使用鹅、统计呱呱叫的次数、控制生产各种不同类型的鸭子、作为一个整体来管理鸭子、观察个别鸭子的行为

分析：

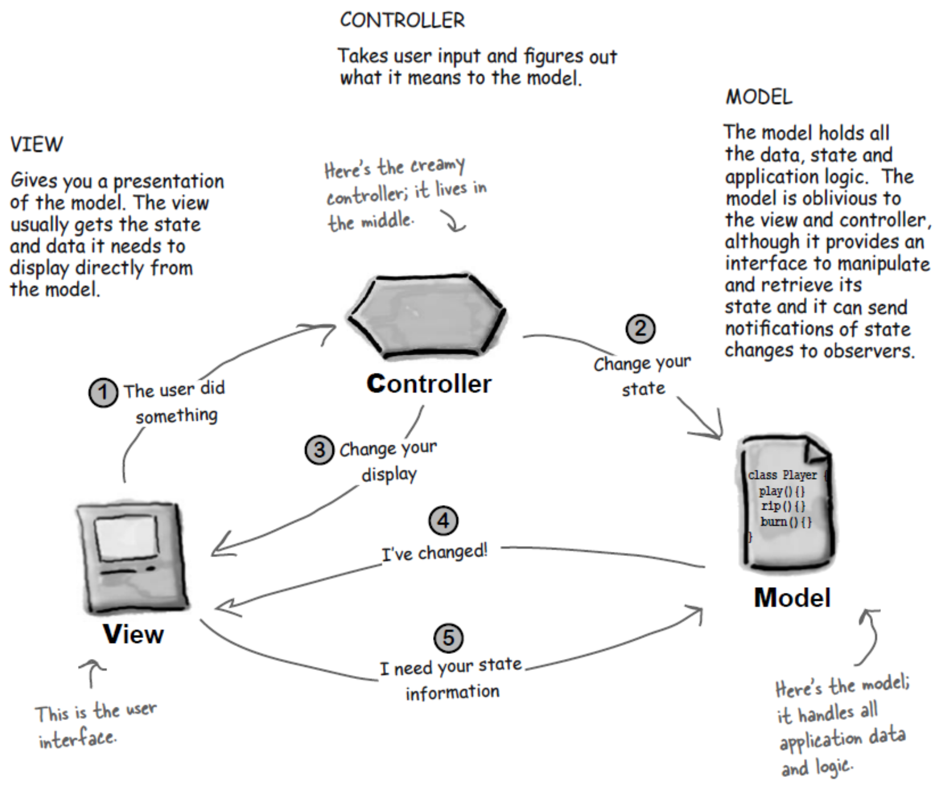
1 在使用鸭子的地方使用鹅，采用适配器模式
2 统计呱呱叫的次数，采用装饰者模式
3 控制生产各种不同类型的鸭子，采用工厂模式，使用抽象工厂创建鸭子，不会取得没有经过装饰的鸭子
4 作为一个整理来管理鸭子，采用组合模式和迭代器模式
5 观察个别鸭子行为，采用观察者模式，将呱呱叫学家注册为观察者，将要观察的鸭子注册为主题对象，当呱呱叫时他就会得到通知
6 具体实现参加程序代码

12.3 理解与实践

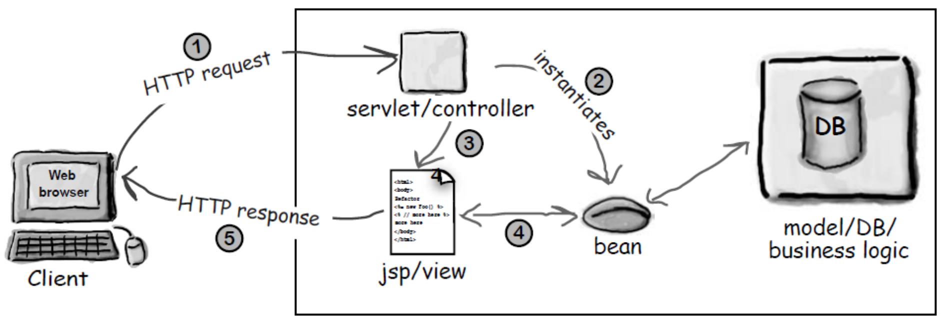
实战 17 设计一个MP3播放器

分析：

1 我们使用MVC来实现，MVC，被称为复合模式之王，Model-View-Controller，即模型-视图-控制器
2 如下图，模型持有所有的数据、状态和程序逻辑，模型没有注意到视图和控制器，虽然它提供了操纵和检索状态的接口并发送状态改变通知给观察者；视图用来呈现模型，视图通知直接从模型中取得它需要显示的状态与数据；控制器取得用户的输入并解读其对模型的意思，控制器把控制逻辑从视图中分离，让模型和视图之间解耦。
3 模型利用观察者模式然控制器和视图可以随最新的状态而更新。使用观察者模式让模型完全独立于视图和控制器，同一个模型可以使用不同的视图，甚至可以同时使用多个视图；视图和控制器实现了经典的策略模式，视图是一个对象，可以被调整使用不同的策略，而控制器提供了策略，如果希望有不同的行为，可以直接换一个控制器；视图内部使用组合模式来管理窗口、按钮以及其他组件。


CONTROLLER
Takes user input and figures out what it means to the model.
MODEL
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.
VIEW
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
Here's the creamy controller; it lives in the middle.
Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
class Player play() {} rip() {} burn() {}
Model
View
This is the user interface.
Here's the model; it handles all application data and logic.

4 Web开发人员也都在适配MVC，使它符合浏览器/服务器模型。我们称这样的适配为Model2。Model2工作模型见下图。



13. 与设计模式相处

13.1 核心思想

首先先看模式的定义：

设计模式 16 — **模式.** 是在某情景下，针对某问题的某种解决方案。

分析：

1 情境就是应用某个模式的情况；问题就是你想在某情境下达到的目标；解决方案就是你追求的一个通用的设计。
2 模式是在特定的问题和情境下，解决重复出现的问题。

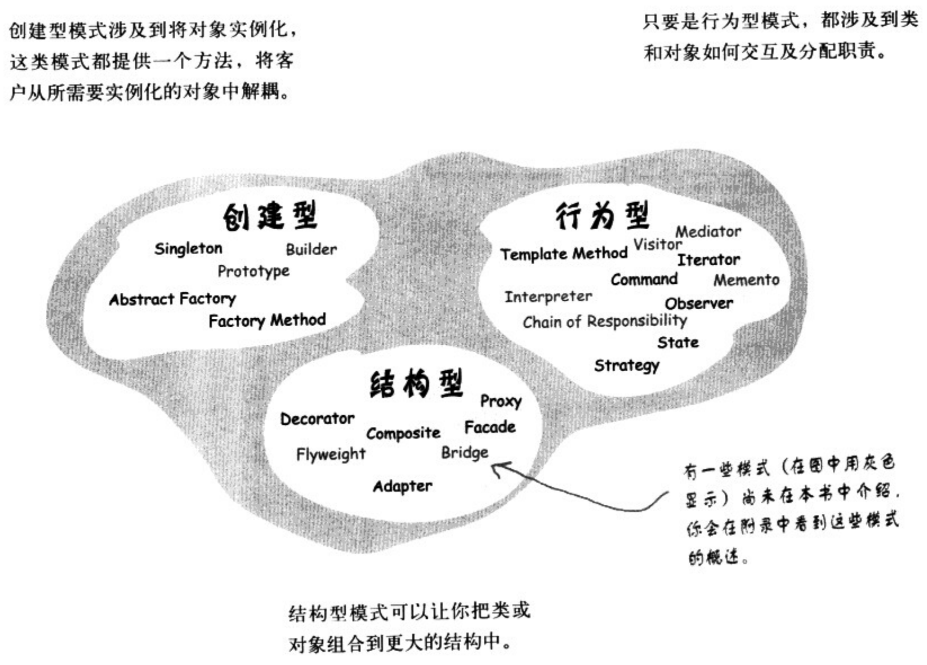
本书十四种设计模式：

1 装饰者模式：包装一个对象，以提供新的行为。
2 状态模式：封装了基于状态的行为，并使用委托在行为之间切换。
3 迭代器模式：在对象的集合之中游走，而不暴露集合的实现。
4 外观模式：简化一群类的接口。
5 策略模式：封装可以互换的行为，并使用委托在行为之间切换。
6 代理模式：包装对象，以控制对此对象的访问。
7 工厂方法模式：有子类决定要创建的具体类是哪一个。
8 适配器模式：封装对象，并提供不同的接口。
9 观察者模式：让对象能够在状态改变时被通知。
10 模板方法模式：由子类决定如何实现一个算法中的步骤。
11 组合模式：客户用一致的方式处理对象集合和单个对象。
12 单件模式：确保有且只有一个对象被创建。
13 抽象工厂模式：允许客户创建对象的家族，而无需指定它们的具体类。
14 命令模式：封装请求成为对象。

分析：

1 设计模式通常被归为三类：创建型、行为型和结构型。

2 创建型模式涉及到将对象实例化，这类模式都提供一个方法，将客户从所需要的实例化对象中解耦。
3 行为型模式都涉及到类和对象如何交互及分配职责。
4 结构型模式可以让你把类或对象组合到更大的结构中。
5 分类见下图。


创建型模式涉及到将对象实例化，这类模式都提供一个方法，将客户从所需要实例化的对象中解耦。
只要是行为型模式，都涉及到类和对象如何交互及分配职责。
创建型
Singleton
Builder
Prototype
Abstract Factory
Factory Method
行为型
Template Method
Visitor
Mediator
Iterator
Command
Memento
Interpreter
Observer
Chain of Responsibility
State
Strategy
结构型
Proxy
Decorator
Composite
Facade
Flyweight
Bridge
Adapter
有一些模式（在图中用灰色显示）尚未在本书中介绍，你会在附录中看到这些模式的概述。
结构型模式可以让你把类或对象组合到更大的结构中。

13.2 设计原则

用模式思考

TIP 9 保持简单，你的目标是解决问题，而不是使用模式，让设计模式自热而然的出现在你的设计中，而不是为了使用而使用。 ■

剩下的设计模式：

1 桥接模式：通过将实现和抽象放在两个不同的类层次中而使它们可以独立改变。
2 生成器模式：封装一个产品的构造过程，并允许按步骤构造。
3 责任链模式：让一个以上的对象有机会能够处理某个请求，将请求的发送者和接受者解耦。
4 蝇量模式：让某个类的一个实例能够提供很多虚拟实例，集中管理，减少运行时对象实例个数，接受内存。
5 解释器模式：为语言创建解释器。
6 中介者模式：集中相关对象之间负责的沟通和控制，通过将对象彼此解耦，增加对象的复用性。
7 备忘录模式：让对象返回之前的状态
8 原型模式：当创建给定类的实例的过程很昂贵或复杂时，使用此模式。
9 访问者模式：当你想要为一个对象的组合增加新的能力，且封装并不重要时，用此模式。