行为型模式

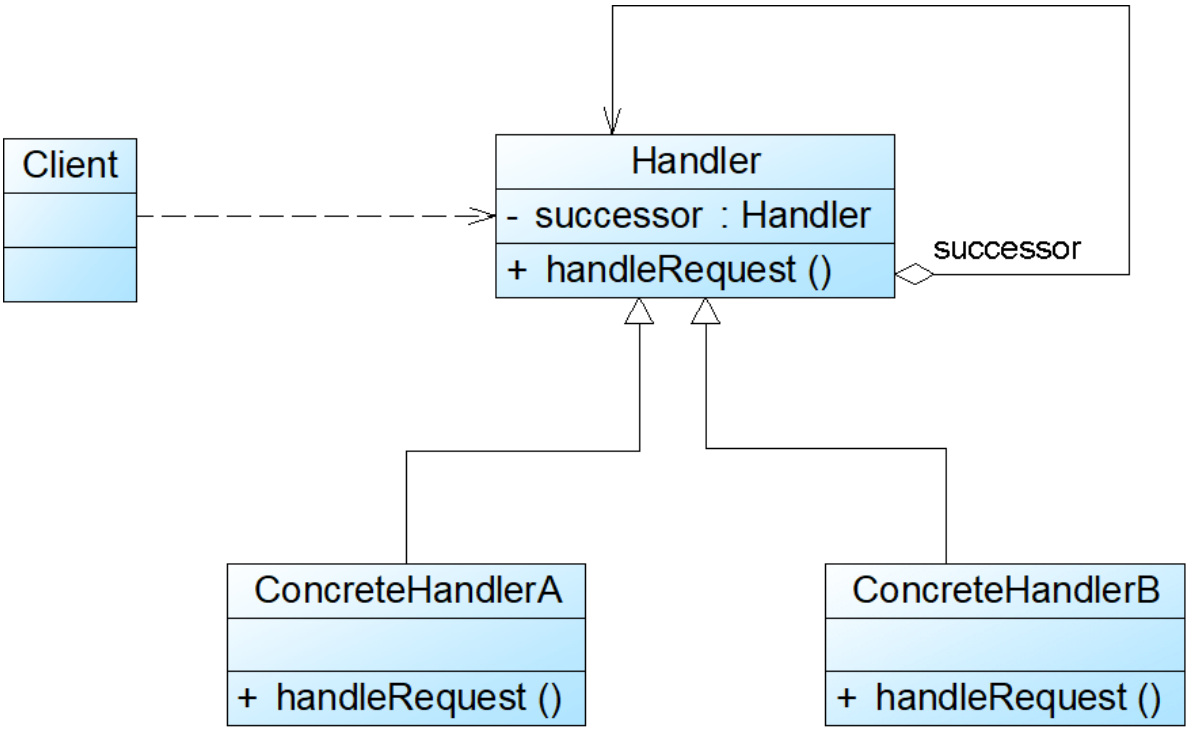
十四、职责链模式

你去政府部门求人办事过吗？有时候你会遇到过官员踢球推责，你的问题在我这里能解决就解决，不能解决就推卸给另外个一个部门（对象）。至于到底谁来解决这个问题呢？政府部门就是为了可以避免屁民的请求与官员之间耦合在一起，让多个（部门）对象都有可能接收请求，将这些（部门）对象连接成一条链，并且沿着这条链传递请求，直到有（部门）对象处理它为止。

例子：js的事件浮升机制

职责链模式：避免将一个请求的发送者与接收者耦合在一起，让多个对象都有机会处理请求。将接收请求的对象连接成一条链，并且沿着这条链传递请求，直到有一个对象能够处理它为止。

1、UML



2、代码

抽象处理者

```
public abstract class Handler {
    //维持对下家的引用
    protected Handler successor;

    public void setSuccessor(Handler successor) {
        this.successor=successor;
    }

    public abstract void handleRequest(String request);
}
```

具体处理者

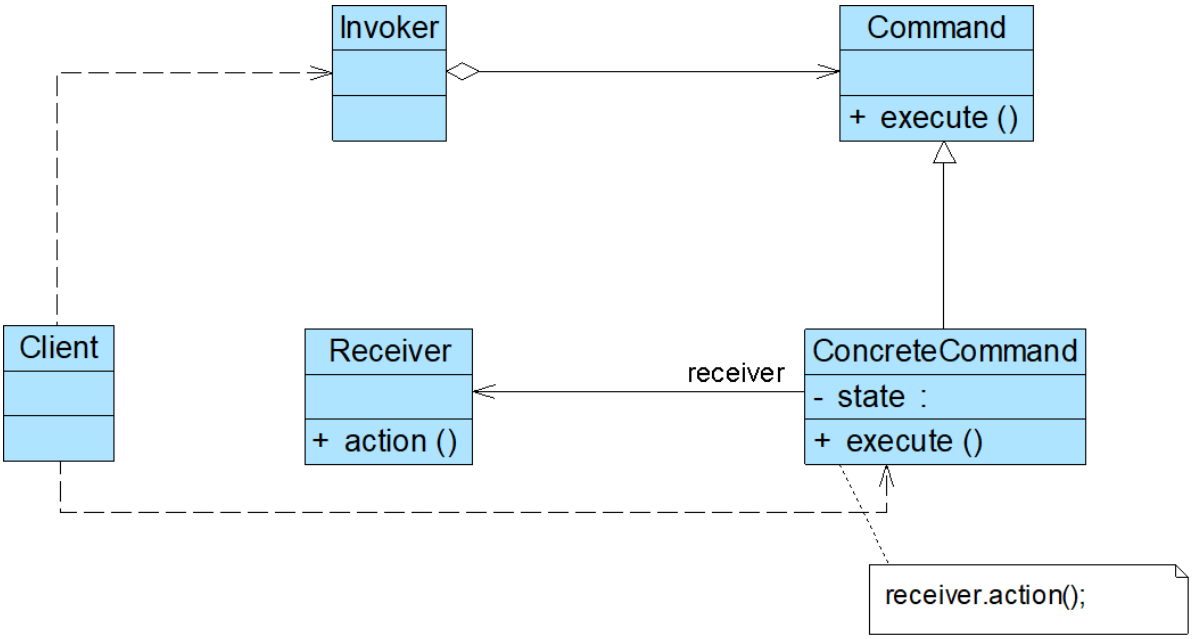
```java
public class ConcreteHandler extends Handler {
    public void handleRequest(String request) {
        if (请求满足条件) {
            //处理请求
        }
        else {
            this.successor.handleRequest(request); //转发请求
        }
    }
}
```

客户端

```
Handler handler1, handler2, handler3;
handler1 = new ConcreteHandlerA();
handler2 = new ConcreteHandlerB();
handler3 = new ConcreteHandlerC();
//创建职责链
handler1.setSuccessor(handler2);
handler2.setSuccessor(handler3);
//发送请求，请求对象通常为自定义类型
handler1.handleRequest("请求对象");
```

十五、命令模式

1、UML



- Command（抽象命令类）
- ConcreteCommand（具体命令类）
- Invoker（调用者）
- Receiver（接收者）

2、代码

抽象命令类

```
public abstract class Command {
    public abstract void execute();
}
```

调用者（请求发送者）类

```
public class Invoker {
    private Command command;

    //构造注入
    public Invoker(Command command) {
        this.command = command;
    }

    //设值注入
    public void setCommand(Command command) {
        this.command = command;
    }

    //业务方法，用于调用命令类的execute()方法
    public void call() {
        command.execute();
    }
}
```

具体命令类

```
public class ConcreteCommand extends Command {
    private Receiver receiver; //维持一个对请求接收者对象的引用

    public void execute() {
        receiver.action(); //调用请求接收者的业务处理方法action()
    }
}
```

请求接收者类

```
public class Receiver {
    public void action() {
        //具体操作
    }
}
```

下面就用看电视的人（Watcher),电视机（Television),遥控器（TeleController）来模拟一下这个命令模式,其中Watcher是Client角色，Television是Receiver角色，TeleController是Invoker角色。

1、设计一个抽象的命令类：

在本系统中，命令的接收者对象就是电视机Tevevision了：

```java
public abstract class Command {
    //命令接收者：电视机
    protected Television television;

    public Command(Television television) {
        this.television = television;
    }

    //命令执行
    abstract void execute();
}
```

将播放各个卫视的操作封装成一个一个命令，实现如下：

```java
//播放cctv1的命令
public class CCTV1Command extends Command {
    @Override
    void execute() {
        television.playCctv1();
    }
}

//播放cctv2的命令
public class CCTV6Command extends Command {
    @Override
    void execute() {
        television.playCctv2();
    }
}
。。。。。。。。

//播放cctv6的命令
public class CCTV1Command extends Command {
    @Override
    void execute() {
        television.playCctv6();
    }
}
```

抽象类Command的几个子类实现也很简单，就是将电视机TeleVision对象的playXxTV方法分布于不同的命令对象中，且因为不同的命令对象拥有共同的抽象类，我们很容易将这些名利功能放入一个数据结构（比如数组）中来存储执行过的命令。

命令对象设计好了，那么就引入命令的调用着Invoker对象了，在此例子中电视遥控器TeleController就是扮演的这个角色：

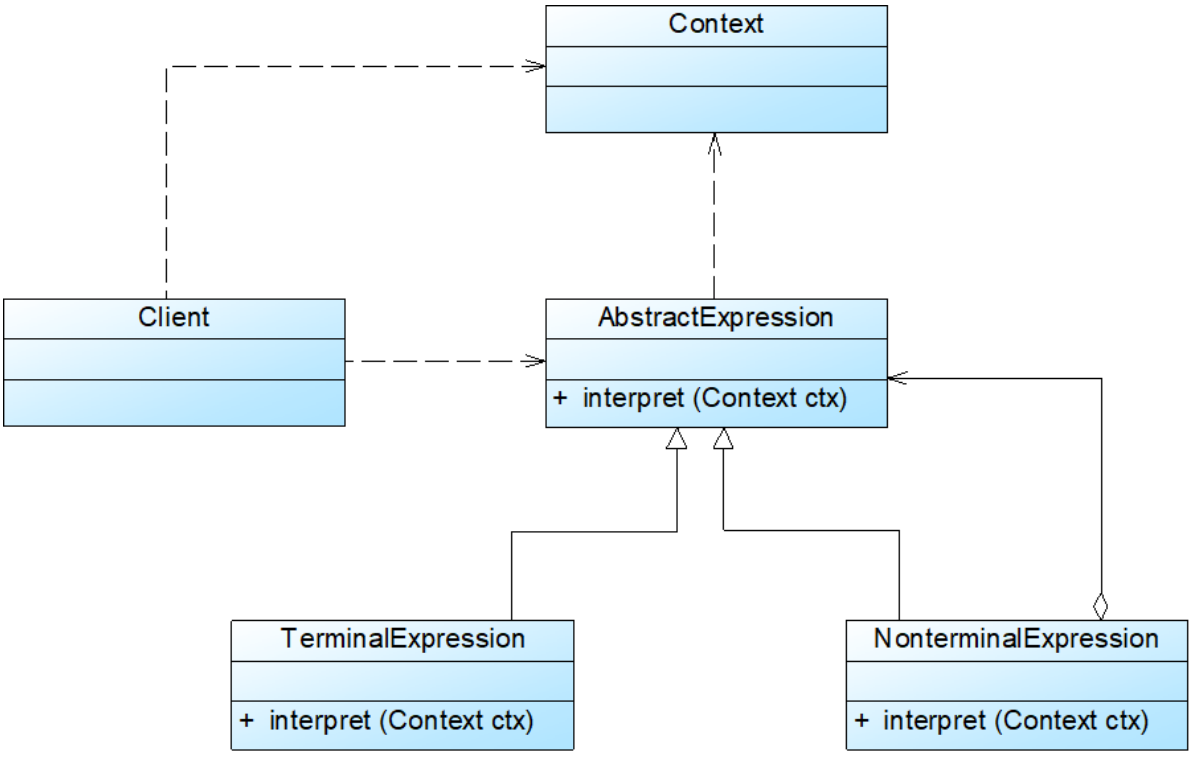
```java
public class TeleController {
    //播放记录
    List<Command> historyCommand = new ArrayList<Command>();

    //切换卫视
    public void switchCommand(Command command) {
        historyCommand.add(command);
        command.execute();
    }

```

```
    //遥控器返回命令
    public void back() {
        if (historyCommand.isEmpty()) {
            return;
        }
        int size = historyCommand.size();
        int preIndex = size-2<=0?0:size-2;

        //获取上一个播放某卫视的命令
        Command preCommand = historyCommand.remove(preIndex);

        preCommand.execute();
    }

}
```

十六、解释器模式（非重点）

解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

1、UML

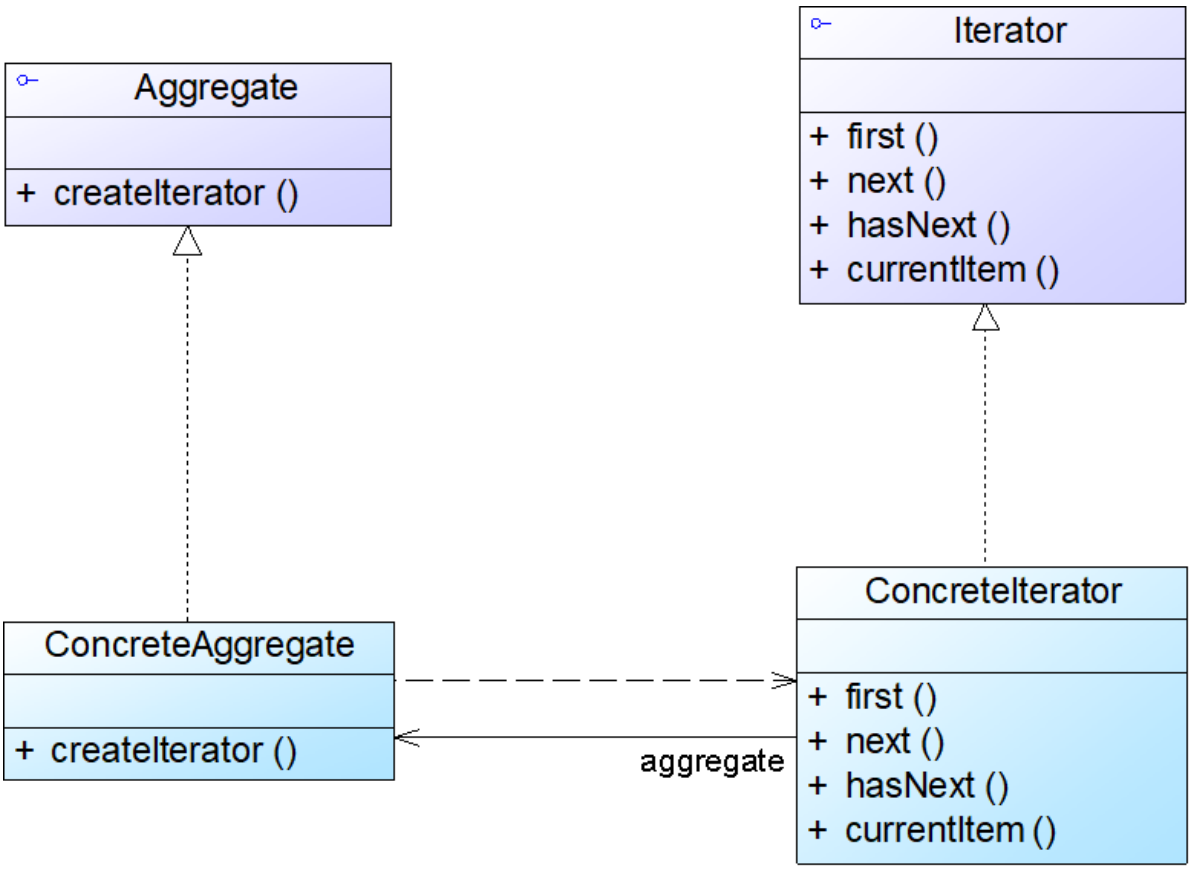


十七、迭代器模式

定义：提供一种方法访问一个容器对象中各个元素，而又不暴露该对象的内部细节。

类型：行为类模式

1、UML



- 抽象容器：一般是一个接口，提供一个iterator()方法，例如java中的Collection接口，List接口，Set接口等。
- 具体容器：就是抽象容器的具体实现类，比如List接口的有序列表实现ArrayList，List接口的链表实现LinkList，Set接口的哈希列表的实现HashSet等。
- 抽象迭代器：定义遍历元素所需要的方法，一般来说会有这么三个方法：取得第一个元素的方法first()，取得下一个元素的方法next()，判断是否遍历结束的方法isDone()（或者叫hasNext()），移出当前对象的方法remove(),
- 迭代器实现：实现迭代器接口中定义的方法，完成集合的迭代。

2、代码

抽象迭代器

```
public interface Iterator {
    public void first();                    //将游标指向第一个元素
    public void next();                     //将游标指向下一个元素
    public boolean hasNext();       //判断是否存在下一个元素
    public Object currentItem();    //获取游标指向的当前元素
}
```

具体迭代器

```
public class ConcreteIterator implements Iterator {
    private ConcreteAggregate objects; //维持一个对具体聚合对象的引用，以便于访问存储在聚合对象中的数据
    private int cursor;  //定义一个游标，用于记录当前访问位置
    public ConcreteIterator(ConcreteAggregate objects) {
        this.objects=objects;
    }

    public void first() {  ......  }
```

```
9
10      public void next() {  ......  }
11
12      public boolean hasNext() {  ......  }
13
14      public Object currentItem() {  ......  }
15  }
```

抽象聚合类

```
1  public interface Aggregate {
2      Iterator createIterator();
3  }
```

具体聚合类

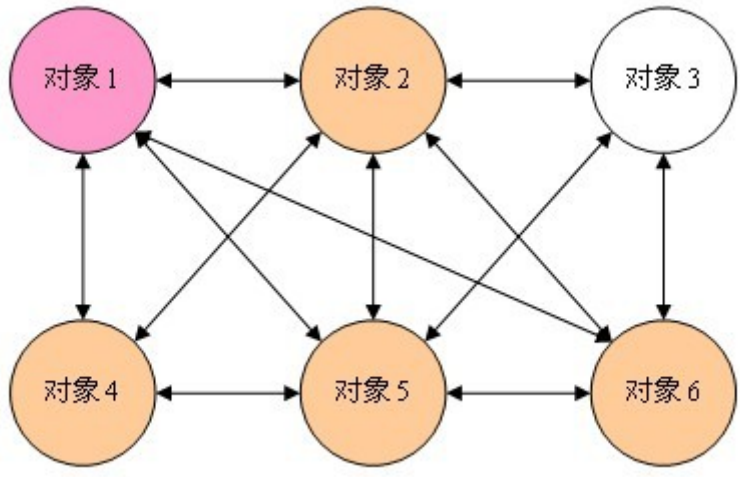
```
1  public class ConcreteAggregate implements Aggregate {
2      ......
3      public Iterator createIterator() {
4          return new ConcreteIterator(this);
5      }
6      ......
7  }
```

十八、中介者模式

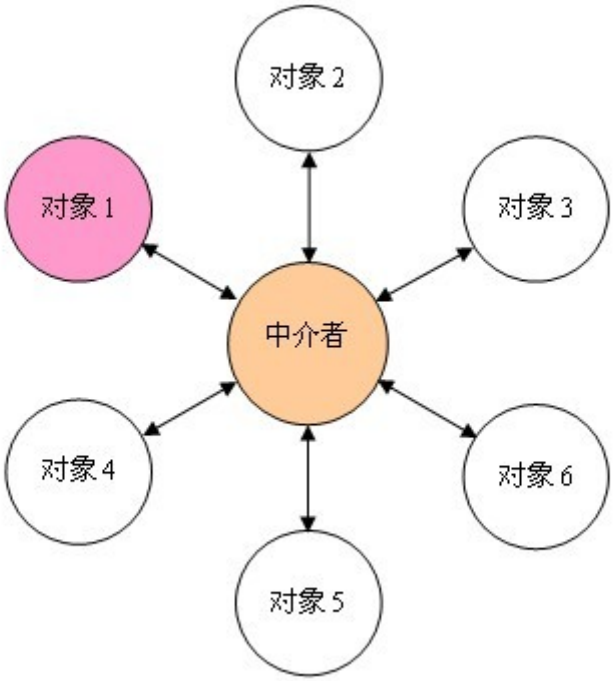
中介者模式：定义一个对象来封装一系列对象的交互。中介者模式使各对象之间不需要显式地相互引用，从而使其耦合松散，而且让你可以独立地改变它们之间的交互。

1、为什么要使用中介者模式

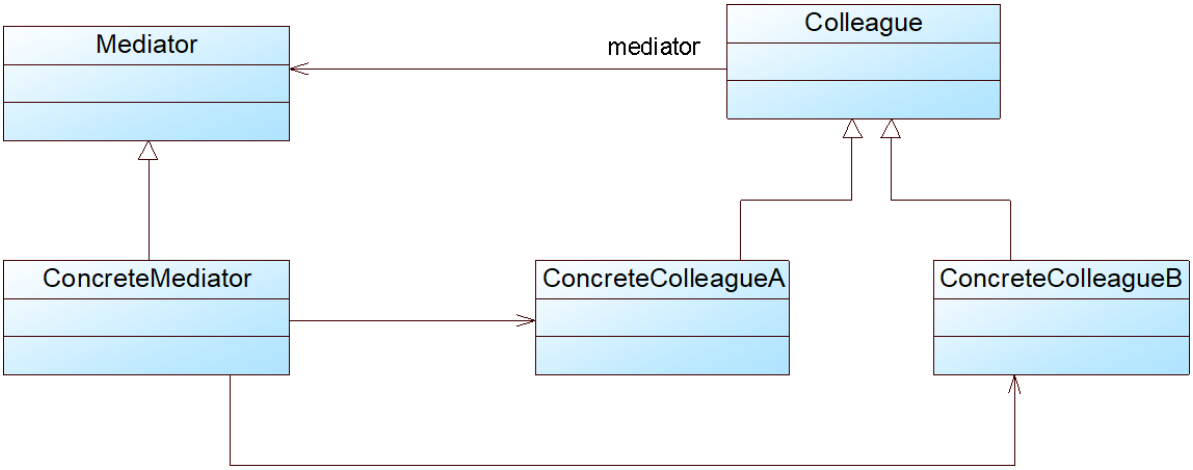
一般来说，同事类之间的关系是比较复杂的，多个同事类之间互相关联时，他们之间的关系会呈现为复杂的网状结构，这是一种过度耦合的架构，即不利于类的复用，也不稳定。例如在下图中，有六个同事类对象，假如对象1发生变化，那么将会有4个对象受到影响。如果对象2发生变化，那么将会有5个对象受到影响。也就是说，同事类之间直接关联的设计是不好的。



如果引入中介者模式，那么同事类之间的关系将变为星型结构，从图中可以看到，任何一个类的变动，只会影响的类本身，以及中介者，这样就减小了系统的耦合。一个好的设计，必定不会把所有的对象关系处理逻辑封装在本类中，而是使用一个专门的类来管理那些不属于自己的行为。



2、UML



- Mediator（抽象中介者）
- ConcreteMediator（具体中介者）
- Colleague（抽象同事类）
- ConcreteColleague（具体同事类）

3、代码

抽象中介者类

```java
public abstract class Mediator {
    protected ArrayList<Colleague> colleagues = new ArrayList<Colleague>();
//用于存储同事对象

    //注册方法，用于增加同事对象
    public void register(Colleague colleague) {
        colleagues.add(colleague);
    }

    //声明抽象的业务方法
    public abstract void operation();
}
```

具体中介者类

```java
public class ConcreteMediator extends Mediator {
    //实现业务方法，封装同事之间的调用
    public void operation() {
        ......
        ((Colleague)(colleagues.get(0))).method1(); //通过中介者调用同事类的方法
        ......
    }
}
```

抽象同事类

```java
public abstract class Colleague {
    protected Mediator mediator; //维持一个抽象中介者的引用

    public Colleague(Mediator mediator) {
        this.mediator=mediator;
    }

    public abstract void method1(); //声明自身方法，处理自己的行为

    //定义依赖方法，与中介者进行通信
    public void method2() {
        mediator.operation();
    }
}
```

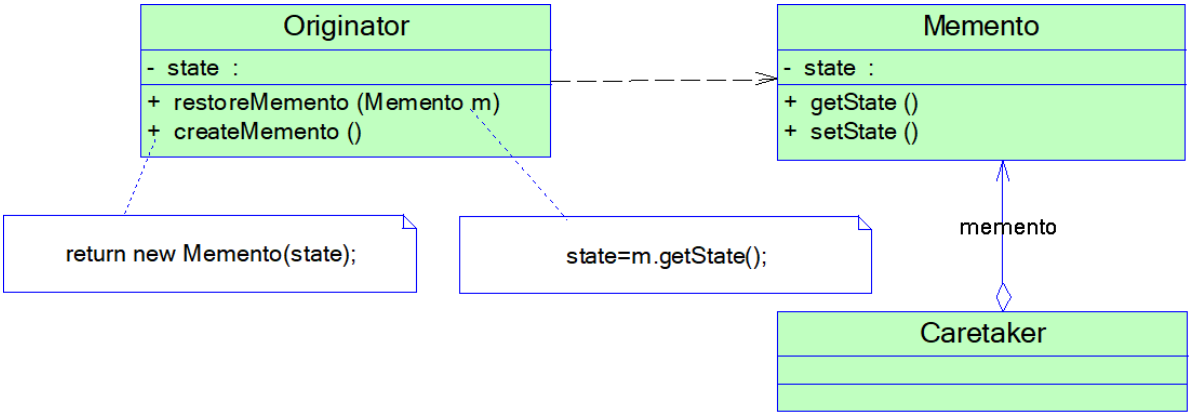
具体同事类

```java
public class ConcreteColleague extends Colleague {
    public ConcreteColleague(Mediator mediator) {
        super(mediator);
    }

    //实现自身方法
    public void method1() {
        ......
    }
}
```

十九、备忘录模式

备忘录模式：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样就可以在以后将对象恢复到原先保存的状态。

1、UML



- Originator（原发器）
- Memento（备忘录）
- Caretaker（负责人）

2、代码

原发器类

```
package designpatterns.memento;
public class Originator {
    private String state;

    public Originator(){}

    //创建一个备忘录对象
    public Memento createMemento() {
        return new Memento(this);
    }

    //根据备忘录对象恢复原发器状态
    public void restoreMemento(Memento m) {
        state = m.state;
    }

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

    public String getState() {
        return this.state;
    }
}
```

备忘录类

```
package designpatterns.memento;
```

```
2
3  //备忘录类，默认可见性，包内可见
4  class Memento {
5      private String state;
6
7      Memento(Originator o) {
8          state = o.getState();
9      }
10
11     void setState(String state) {
12         this.state=state;
13     }
14
15     String getState() {
16         return this.state;
17     }
18 }
```

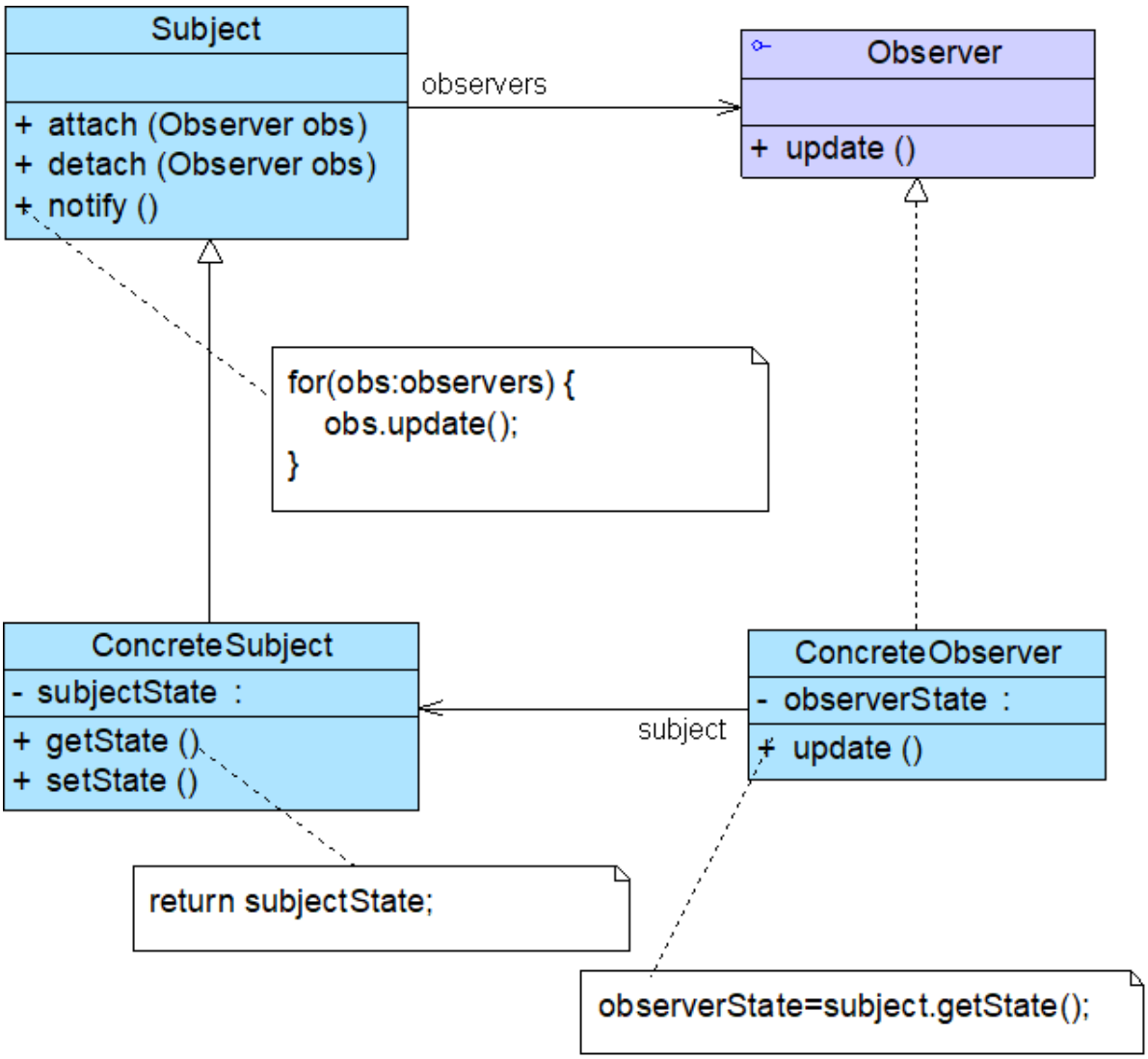
负责人类

```
1  package designpatterns.memento;
2
3  public class Caretaker {
4      private Memento memento;
5
6      public Memento getMemento() {
7          return memento;
8      }
9
10     public void setMemento(Memento memento) {
11         this.memento=memento;
12     }
13 }
```

二十、观察者模式

观察者模式：定义对象之间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象都得到通知并被自动更新。

1、UML



- Subject（目标）
- ConcreteSubject（具体目标）
- Observer（观察者）
- ConcreteObserver（具体观察者）

2、代码

抽象目标类

```
import java.util.*;
public abstract class Subject {
    //定义一个观察者集合用于存储所有观察者对象
    protected ArrayList observers<Observer> = new ArrayList();

    //注册方法，用于向观察者集合中增加一个观察者
    public void attach(Observer observer) {
        observers.add(observer);
    }

    //注销方法，用于在观察者集合中删除一个观察者
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    //声明抽象通知方法
    public abstract void notify();
```

```
}
```

具体目标类

```
public class ConcreteSubject extends Subject {
    //实现通知方法
    public void notify() {
        //遍历观察者集合，调用每一个观察者的响应方法
        for(Object obs:observers) {
            ((Observer)obs).update();
        }
    }
}
```

抽象观察者

```
public interface Observer {
    //声明响应方法
    public void update();
}
```

具体观察者

```
public class ConcreteObserver implements Observer {
    //实现响应方法
    public void update() {
        //具体响应代码
    }
}
```

观察者模式这种发布-订阅的形式我们可以拿微信公众号来举例，假设微信用户就是观察者，微信公众号是被观察者，有多个的微信用户关注了程序猿这个公众号，当这个公众号更新时就会通知这些订阅的微信用户。好了我们来看看用代码如何实现：

抽象观察者（Observer）

```
//抽象观察者接口
public interface Observer {
    public void update(String message);
}
```

具体观察者（ConcrereObserver）

```java
public class WeixinUser implements Observer{
    private String name;

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

    @Override
    public void update(String message) {
        System.out.println(name + "-" + message);
    }
}
```

抽象被观察者（Subject）

```java
public interface Subject {
    //增加订阅者
    public void attach(Observer observer);

    //删除订阅者
    public void detach(Observer observer);

    //通知订阅者更新信息
    public void notify(String message);

}
```

具体被观察者（ConcreteSubject）

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

public class SubscriptionSubject implements Subject {

    //储存订阅公众号的微信用户
    private List<Observer> weixinUserlist = new ArrayList<Observer>();

    @Override
    public void attach(Observer observer) {
        weixinUserlist.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        weixinUserlist.remove(observer);
    }

    @Override
    public void notify(String message) {
        for(Observer observer : weixinUserlist) {
            observer.update(message);
        }
    }
}
```

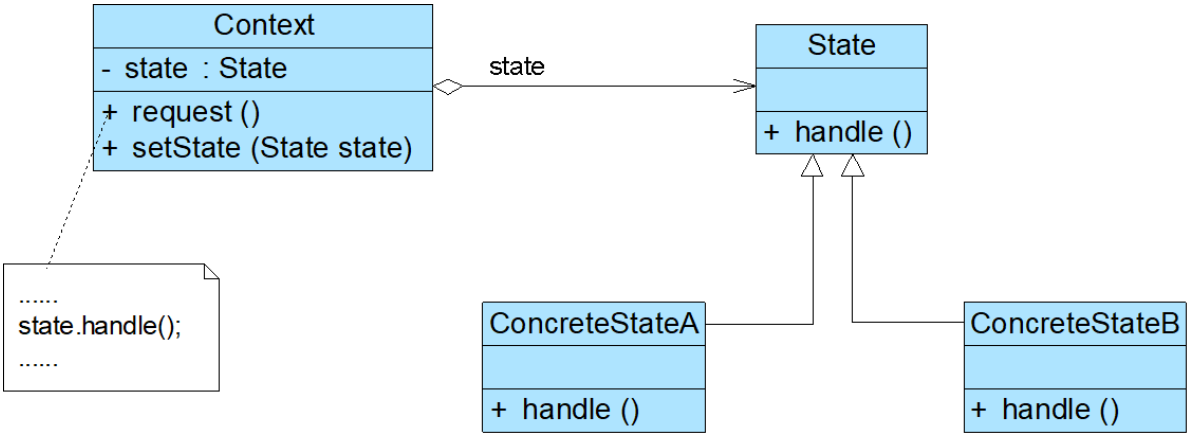
客户端调用

```
public class Client {
    public static void main(String[] args) {
        SubscriptionSubject mSubscriptionSubject = new
SubscriptionSubject();
        //创建微信用户
        WeixinUser user1 = new WeixinUser("aaa");
        WeixinUser user2 = new WeixinUser("bbb");
        WeixinUser user3 = new WeixinUser("ccc");

        //订阅公众号
        mSubscriptionSubject.attach(user1);
        mSubscriptionSubject.attach(user2);
        mSubscriptionSubject.attach(user3);

        //公众号更新发出消息给订阅的微信用户
        mSubscriptionSubject.notify("专栏更新了");
    }
}
```

二十一、状态模式

状态模式：允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

1、UML



- Context（环境类）
- State（抽象状态类）
- ConcreteState（具体状态类）

2、代码

抽象状态类

```
public abstract class State {
    //声明抽象业务方法，不同的具体状态类可以有不同的实现
    public abstract void handle();
}
```

具体状态类

```
public class ConcreteState extends State {
    public void handle() {
        //方法具体实现代码
    }
}
```

环境类

```
public class Context {
    private State state; //维持一个对抽象状态对象的引用
    private int value;  //其他属性值，该属性值的变化可能会导致对象的状态发生变化

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

    public void request() {
        //其他代码
        state.handle(); //调用状态对象的业务方法
        //其他代码
    }
}
```

状态转换的实现

```
public void changeState(Context ctx) {
        //根据环境对象中的属性值进行状态转换
        if (ctx.getValue() == 1) {
        ctx.setState(new ConcreteStateB());
        }
        else if (ctx.getValue() == 2) {
        ctx.setState(new ConcreteStateC());
        }
        ......
        }
```

二十二、策略模式

1. 介绍

1.1 定义

定义一系列算法，将每个算法封装到具有公共接口的一系列策略类中，从而使它们可以相互替换 & 让算法可在不影响客户端的情况下发生变化

> 简单来说：准备一组算法 & 将每一个算法封装起来，让外部按需调用 & 使得互换

1.2 作用（解决的问题）

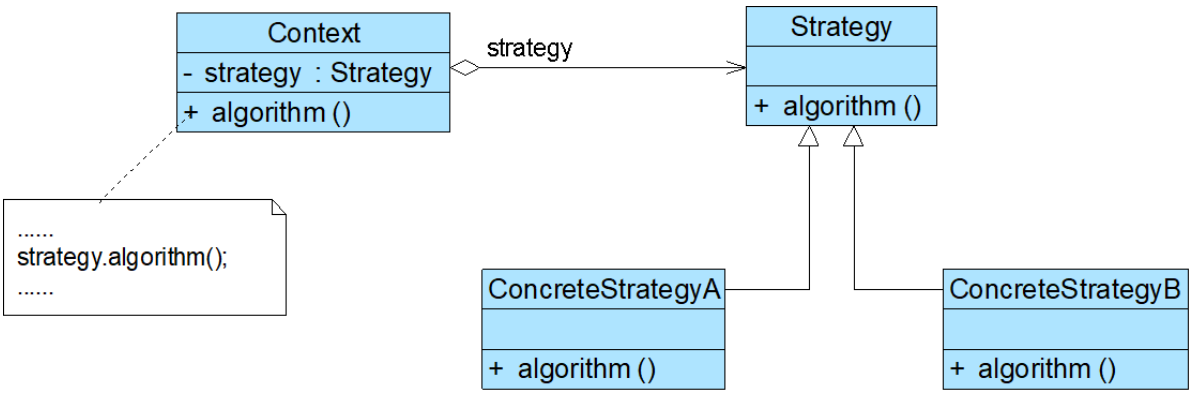
将算法的责任和本身进行解耦，使得：

1. 算法可独立于使用外部而变化
2. 客户端方便根据外部条件选择不同策略来解决不同问题

> 策略模式仅仅封装算法（包括添加 & 删除），但策略模式并不决定在何时使用何种算法，算法的选择由客户端来决定

2. 模式原理

2.1 UML类图



- Context（环境类）
- Strategy（抽象策略类）
- ConcreteStrategy（具体策略类）

2.2 代码

抽象策略类

```
public abstract class Strategy {
    public abstract void algorithm();  //声明抽象算法
}
```

具体策略类

```
public class ConcreteStrategyA extends Strategy {
    //算法的具体实现
    public void algorithm() {
        //算法A
    }
}
```

环境类

```
public class Context {
    private Strategy strategy; //维持一个对抽象策略类的引用

    //注入策略对象
    public void setStrategy(Strategy strategy) {
        this.strategy= strategy;
    }

    //调用策略类中的算法
    public void algorithm() {
        strategy.algorithm();
    }
}
```

客户端

```
Context context = new Context();
Strategy strategy;
strategy = new ConcreteStrategyA(); //可在运行时指定类型，通过配置文件和反射机制实现
context.setStrategy(strategy);
context.algorithm();
```

实例讲解

接下来我用一个实例来对策略模式进行更深一步的介绍。

a. 实例概况

- 背景：小成有一家百货公司，最近在定年度的促销活动
- 冲突：每个节日用同一个促销活动太枯燥，没吸引力
- 解决方案：针对不同节目使用不同促销活动进行促销

b. 使用步骤

步骤1： 定义抽象策略角色（Strategy）：百货公司所有促销活动的共同接口

```
public abstract class Strategy {

    public abstract void Show();
}
```

步骤2：定义具体策略角色（Concrete Strategy）：每个节日具体的促销活动

```
//为春节准备的促销活动A
class StrategyA extends Strategy{

    @Override
    public void show() {
        System.out.println("为春节准备的促销活动A");
    }
}

//为中秋节准备的促销活动B
class StrategyB extends Strategy{

    @Override
    public void show() {
        System.out.println("为中秋节准备的促销活动B");
    }
}

//为圣诞节准备的促销活动C
class StrategyC extends Strategy{

    @Override
    public void show() {
        System.out.println("为圣诞节准备的促销活动C");
    }
}
```

步骤3：定义环境角色（Context）：用于连接上下文，即把促销活动推销给客户，这里可以理解为销售员

```
class Context_SalesMan{
//持有抽象策略角色的引用
    private Strategy strategy;

    //生成销售员实例时告诉销售员什么节日（构造方法）
    //使得让销售员根据传入的参数（节日）选择促销活动（这里使用一个简单的工厂模式）
    public SalesMan(String festival) {
        switch ( festival) {
            //春节就使用春节促销活动
            case "A":
                strategy = new StrategyA();
                break;
            //中秋节就使用中秋节促销活动
            case "B":
                strategy = new StrategyB();
                break;
            //圣诞节就使用圣诞节促销活动
            case "C":
                strategy = new StrategyC();
                break;
        }

    }

    //向客户展示促销活动
    public void SalesManShow(){
        strategy.show();
    }

}
```

步骤4：客户端调用-让销售员进行促销活动的落地

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

        Context_SalesMan mSalesMan ;

        //春节来了，使用春节促销活动
        System.out.println("对于春节：");
        mSalesMan =  Context_SalesMan SalesMan("A");
        mSalesMan.SalesManShow();


        //中秋节来了，使用中秋节促销活动
        System.out.println("对于中秋节：");
        mSalesMan =  Context_SalesMan SalesMan("B");
        mSalesMan.SalesManShow();

        //圣诞节来了，使用圣诞节促销活动
        System.out.println("对于圣诞节：");
        mSalesMan =  Context_SalesMan SalesMan("C");
        mSalesMan.SalesManShow();
  }
}
```

结果输出

```
对于春节：
为春节准备的促销活动A
对于中秋节：
为中秋节准备的促销活动B
对于圣诞节：
为圣诞节准备的促销活动B
```

二十三、模板方法模式

模板方法模式：定义一个操作中算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类不改变一个算法的结构即可重定义该算法的某些特定步骤。

1. 介绍

1.1 定义

定义一个模板结构，将具体内容延迟到子类去实现。

1.2 主要作用

在不改变模板结构的前提下在子类中重新定义模板中的内容。

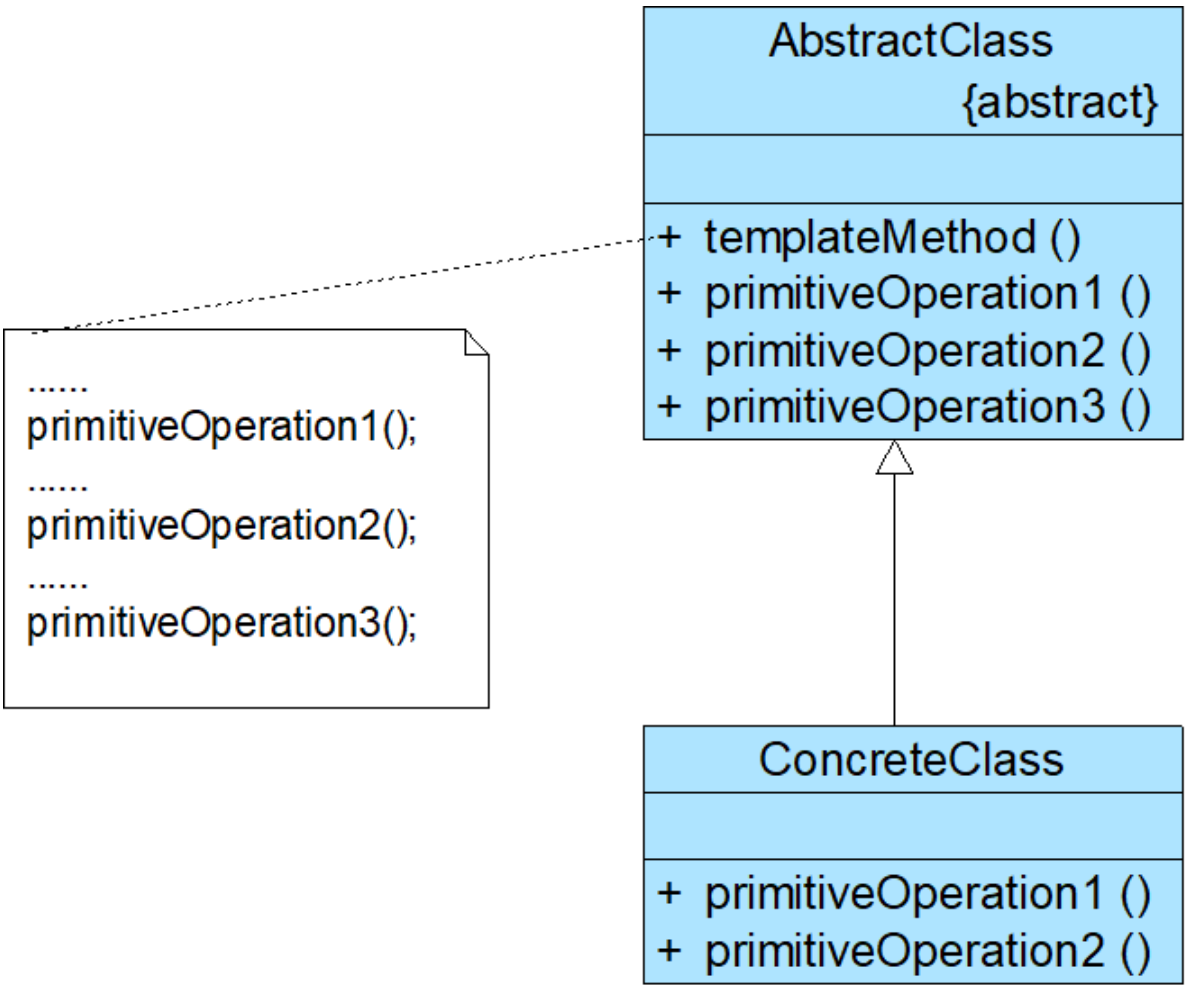
> 模板方法模式是基于"继承"的；

1.3 解决的问题

- 提高代码复用性 将相同部分的代码放在抽象的父类中，而将不同的代码放入不同的子类中
- 实现了反向控制 通过一个父类调用其子类的操作，通过对子类的具体实现扩展不同的行为，实现了反向控制 & 符合"开闭原则"

2. 模式原理

2.1 UML



- AbstractClass（抽象类）
- ConcreteClass（具体子类）

2.2 实例讲解

接下来我用一个实例来对模板方法模式进行更深一步的介绍。

a. 实例概况

- 背景：小成希望学炒菜：手撕包菜 & 蒜蓉炒菜心
- 冲突：两道菜的炒菜步骤有的重复有的却差异很大，记不住
- 解决方案：利用代码记录下来

b. 使用步骤

步骤1：创建抽象模板结构（Abstract Class）：炒菜的步骤

```
public  abstract class Abstract Class {
//模板方法，用来控制炒菜的流程 （炒菜的流程是一样的-复用）
//申明为final，不希望子类覆盖这个方法，防止更改流程的执行顺序
        final void cookProcess(){
        //第一步：倒油
        this.pourOil();
        //第二步：热油
         this.HeatOil();
        //第三步：倒蔬菜
         this.pourVegetable();
        //第四步：倒调味料
         this.pourSauce（）；
        //第五步：翻炒
```

```
          this.fry();
    }

//定义结构里哪些方法是所有过程都是一样的可复用的，哪些是需要子类进行实现的

//第一步：倒油是一样的，所以直接实现
void pourOil(){
        System.out.println("倒油");
    }

//第二步：热油是一样的，所以直接实现
    void  HeatOil(){
        System.out.println("热油");
    }

//第三步：倒蔬菜是不一样的（一个下包菜，一个是下菜心）
//所以声明为抽象方法，具体由子类实现
    abstract void  pourVegetable();

//第四步：倒调味料是不一样的（一个下辣椒，一个是下蒜蓉）
//所以声明为抽象方法，具体由子类实现
    abstract void  pourSauce();

//第五步：翻炒是一样的，所以直接实现
    void fry();{
        System.out.println("炒啊炒啊炒到熟啊");
    }
}
```

步骤2：创建具体模板（Concrete Class）,即”手撕包菜“和”蒜蓉炒菜心“的具体步骤

```
//炒手撕包菜的类
  public class ConcreteClass_BaoCai extend  Abstract Class{
    @Override
    public void  pourVegetable(){
        System.out.println(”下锅的蔬菜是包菜“);
    }
    @Override
    public void  pourSauce() {
        System.out.println(”下锅的酱料是辣椒“);
    }
}
//炒蒜蓉菜心的类
  public class ConcreteClass_CaiXin extend  Abstract Class{
    @Override
    public void  pourVegetable(){
        System.out.println(”下锅的蔬菜是菜心“);
    }
    @Override
    public void  pourSauce() {
        System.out.println(”下锅的酱料是蒜蓉“);
    }
}
```

步骤3：客户端调用-炒菜了

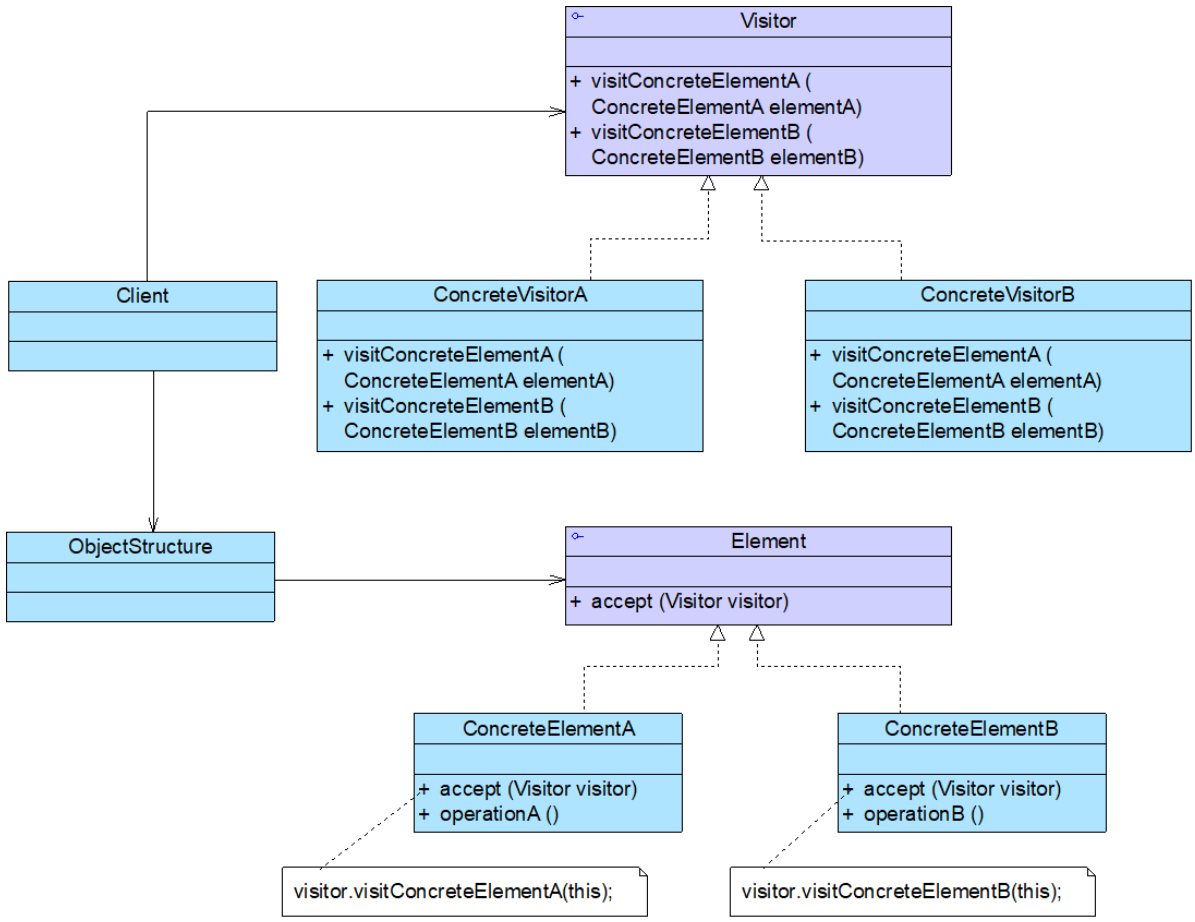
```
public class Template Method{
  public static void main(String[] args){

//炒 - 手撕包菜
    ConcreteClass_BaoCai BaoCai = new ConcreteClass_BaoCai ();
    BaoCai.cookProcess();

//炒 - 蒜蓉菜心
  ConcreteClass_ CaiXin = new ConcreteClass_CaiXin ();
    CaiXin.cookProcess();
    }

}
```

结果输出

```
倒油
热油
下锅的蔬菜是包菜
下锅的酱料是辣椒
炒啊炒啊炒到熟

倒油
热油
下锅的蔬菜是菜心
下锅的酱料是蒜蓉
炒啊炒啊炒到熟
```

二十四、访问者模式

访问者模式：表示一个作用于某对象结构中的各个元素的操作。访问者模式让你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。


Visitor
+ visitConcreteElementA (ConcreteElementA elementA)
+ visitConcreteElementB (ConcreteElementB elementB)
Client
ConcreteVisitorA
+ visitConcreteElementA (ConcreteElementA elementA)
+ visitConcreteElementB (ConcreteElementB elementB)
ConcreteVisitorB
+ visitConcreteElementA (ConcreteElementA elementA)
+ visitConcreteElementB (ConcreteElementB elementB)
ObjectStructure
Element
+ accept (Visitor visitor)
ConcreteElementA
+ accept (Visitor visitor)
+ operationA ()
ConcreteElementB
+ accept (Visitor visitor)
+ operationB ()
visitor.visitConcreteElementA(this);
visitor.visitConcreteElementB(this);