



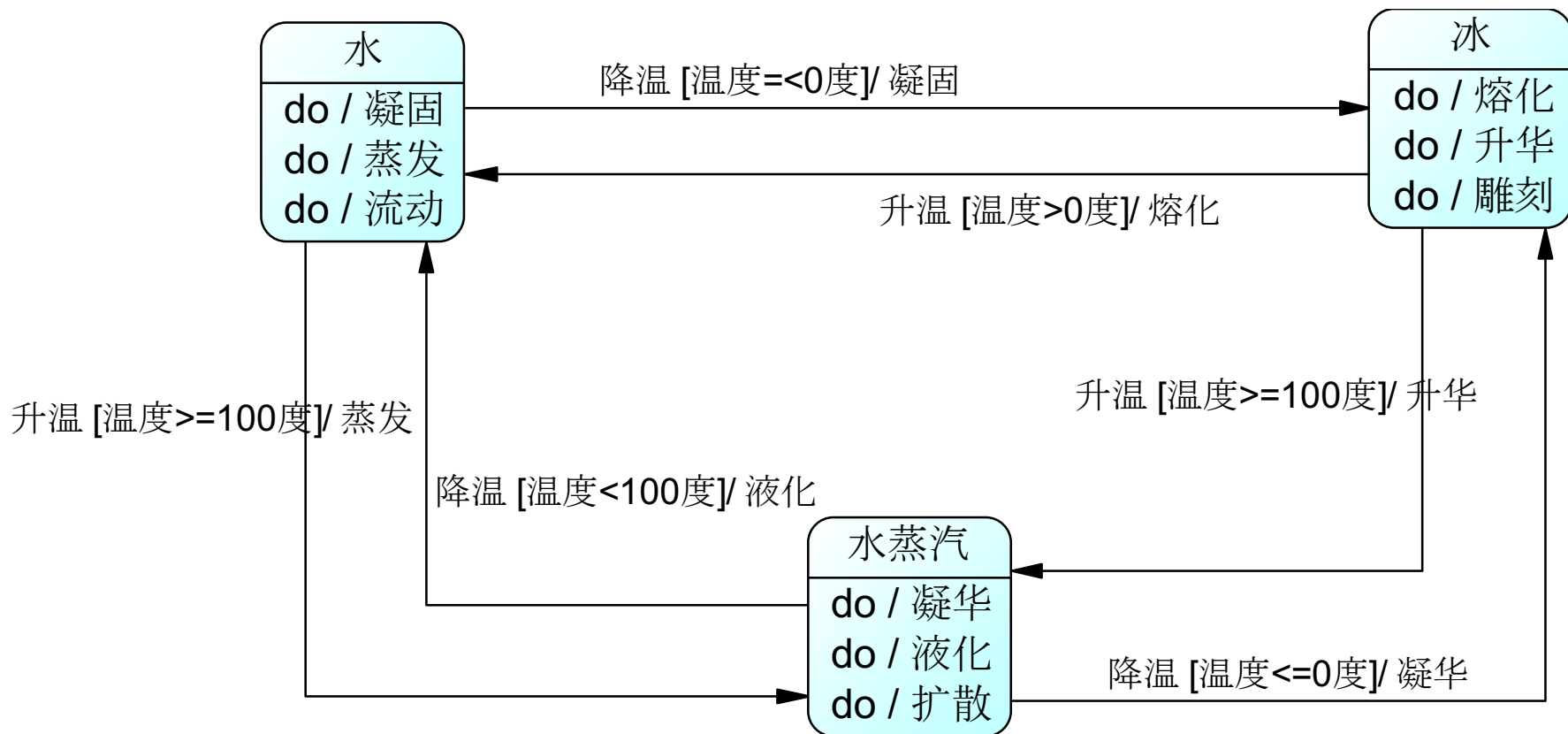
南京大学

设计模式-行为型模式(三)

Design Pattern-Behavioral Pattern (3)

状态模式概述

- H₂O的三种状态（未考虑临界点）



状态模式概述

- 分析

```
public class TestXYZ {  
    int behaviour;  
    //省略Getter和Setter方法  
    .....  
    public void handleAll() {  
        if (behaviour == 0) {  
            //do something }  
        else if (behaviour == 1) {  
            //do something }  
        else if (behaviour == 2) {  
            //do something }  
        else if (behaviour == 3) {  
            //do something }  
        ... some more else if ...  
    }  
}
```

状态模式概述

- 状态模式的定义

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。

State Pattern: Allow an object to **alter its behavior when its internal state changes. The object will appear to change its class.**

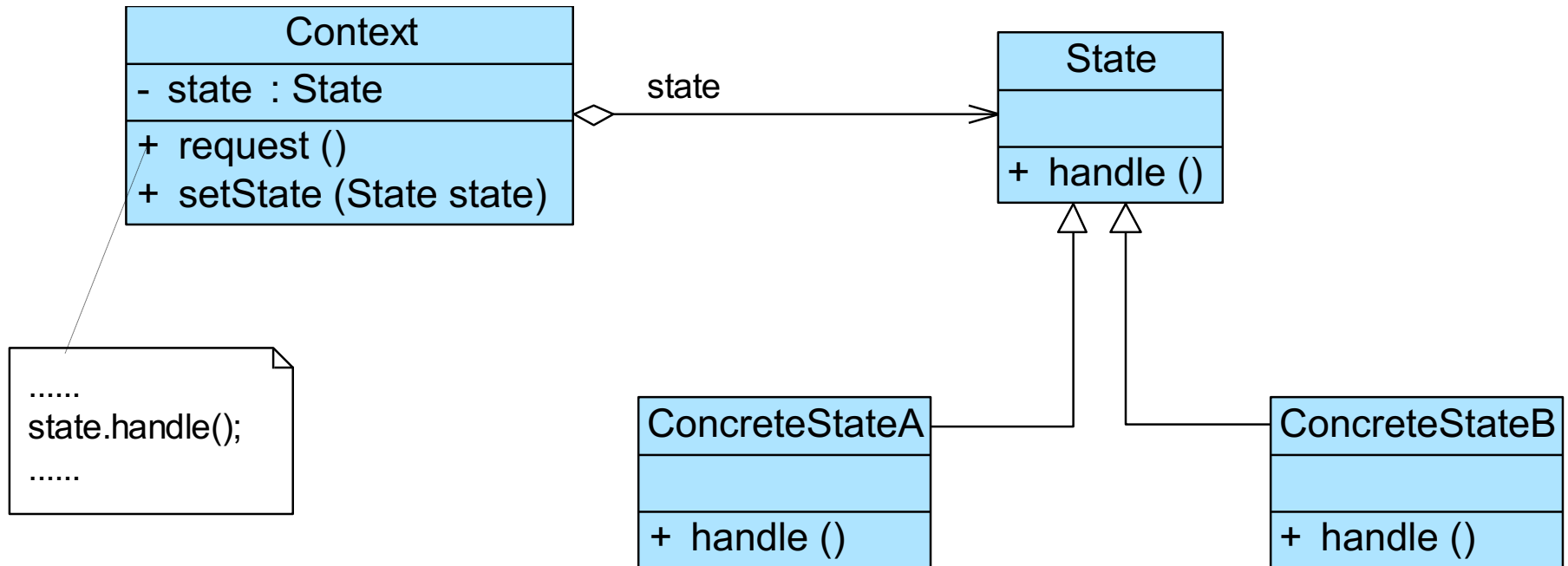
- 对象行为型模式

状态模式概述

- 状态模式的定义
 - 又名状态对象(Objects for States)
 - 用于解决系统中复杂对象的状态转换以及不同状态下行为的封装问题
 - 将一个对象的状态从该对象中分离出来，封装到专门的状态类中，使得对象状态可以灵活变化
 - 对于客户端而言，无须关心对象状态的转换以及对象所处的当前状态，无论对于何种状态的对象，客户端都可以一致处理

状态模式的结构与实现

- 状态模式的结构



状态模式的结构与实现

- 状态模式的结构
 - 状态模式包含以下3个角色：
 - Context（环境类）
 - State（抽象状态类）
 - ConcreteState（具体状态类）

状态模式的结构与实现

- 状态模式的实现
 - 典型的抽象状态类代码：

```
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();  
}
```


状态模式的结构与实现

- 状态模式的实现
 - 典型的**具体状态类**代码：

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

状态模式的结构与实现

- 状态模式的实现

- 曲形的环境米代码 •

```
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  
    }  
}
```

状态模式的结构与实现

- 状态模式的实现
 - 状态转换的实现：

```
.....  
public void changeState()  
{  
    //判断属性值，根据属性值进行状态转换  
    if (value == 0)  
    {  
        this.setState(new ConcreteStateA());  
    }  
    else if (value == 1)  
    {  
        this.setState(new ConcreteStateB());  
    }  
    .....  
}  
.....
```

状态模式的结构与实现

- 状态模式的实现

- 状态转换的实现：

- (2) 由具体状态类来负责状态之间的转换，可以在

具体状态类中，通过判断环境对象的属性值来实现状态转换。

```
.....  
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());  
    }  
    .....  
}  
.....
```

状态模式的应用实例

• 实例说明

某软件公司要为一银行开发一套信用卡业务系统，银行账户(Account)是该系统的核心类之一，通过分析，该软件公司开发人员发现在系统中账户存在3种状态，且在不同状态下账户存在不同的行为，具体说明如下：

(1) 如果账户中余额大于等于0，则账户的状态为正常状态(Normal State)，此时用户既可以向该账户存款也可以从该账户取款；

(2) 如果账户中余额小于0，并且大于-2000，则账户的状态为透支状态(Overdraft State)，此时用户既可以向该账户存款也可以从该账户取款，但需要按天计算利息；

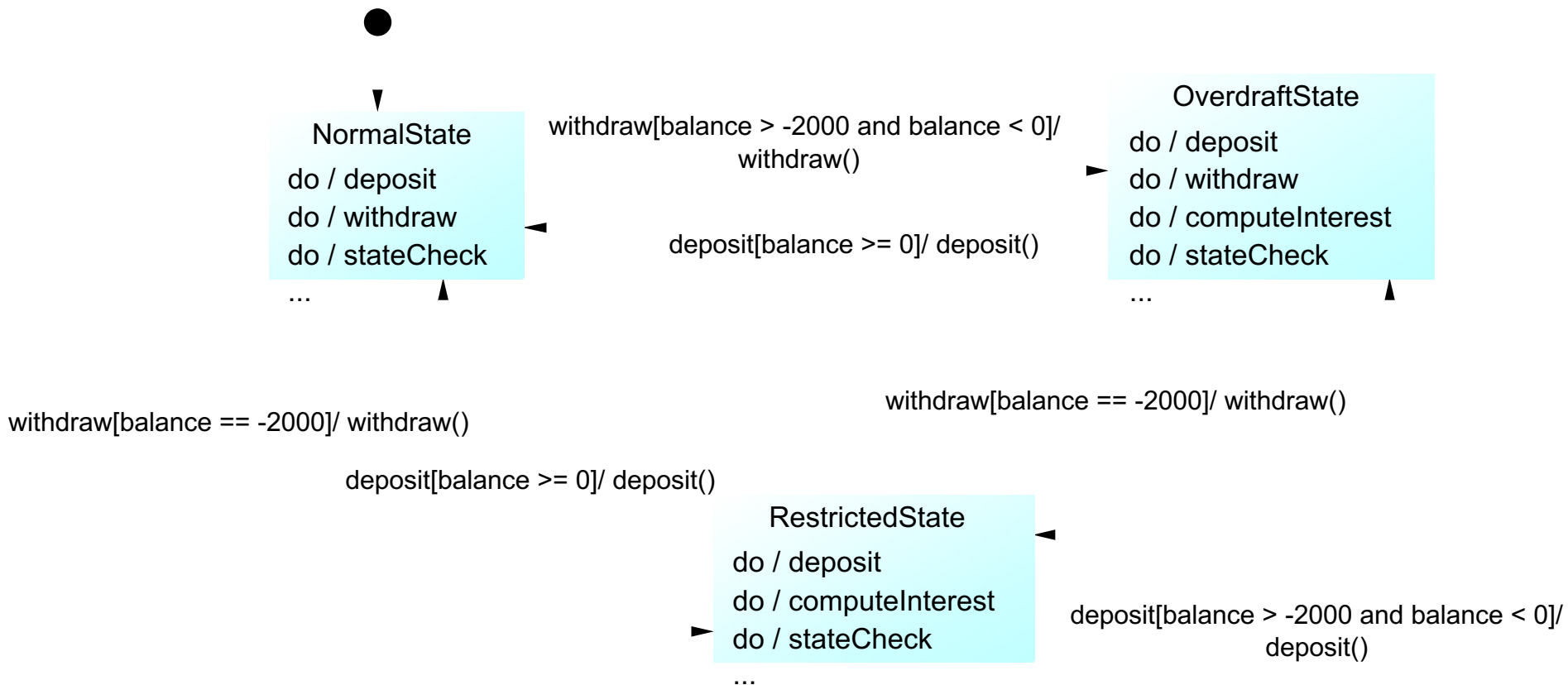
(3) 如果账户中余额等于-2000，那么账户的状态为受限状态(Restricted State)，此时用户只能向该账户存款，不能再从中取款，同时也将按天计算利息；

(4) 根据余额的不同，以上3种状态可发生相互转换。

现使用状态模式设计并实现银行账户状态的转换。

状态模式的应用实例

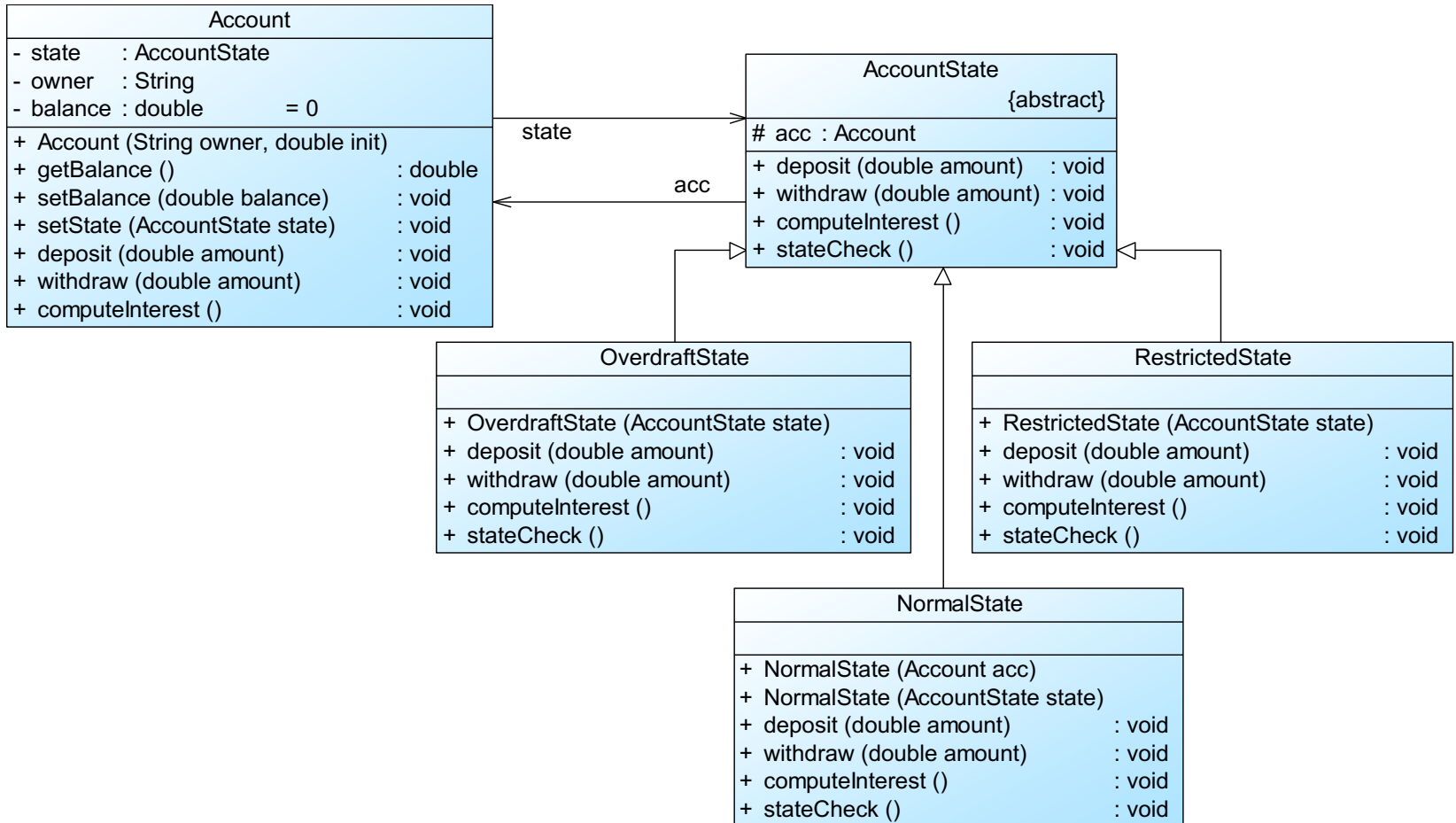
- 实例分析与类图



银行账户状态图

状态模式的应用实例

• 实例分析与类图



银行账户结构图

状态模式的应用实例

- 实例代码

- (1) Account : 银行账户, 充当环境类
- (2) AccountState : 账户状态类, 充当抽象状态类
- (3) NormalState : 正常状态类, 充当具体状态类
- (4) OverdraftState : 透支状态类, 充当具体状态类
- (5) RestrictedState : 受限状态类, 充当具体状态类
- (6) Client : 客户端测试类

演示.....

Code (designpatterns.state)

状态模式的应用实例

段誉开户，初始金额为0.0

- 段誉存款1000.0
现在余额为1000.0
现在帐户状态为designpatterns.state.NormalState
-

段誉取款2000.0
现在余额为-1000.0
现在帐户状态为designpatterns.state.OverdraftState

段誉存款3000.0
现在余额为2000.0
现在帐户状态为designpatterns.state.NormalState

段誉取款4000.0
现在余额为-2000.0
现在帐户状态为designpatterns.state.RestrictedState

段誉取款1000.0
帐号受限，取款失败
现在余额为-2000.0
现在帐户状态为designpatterns.state.RestrictedState

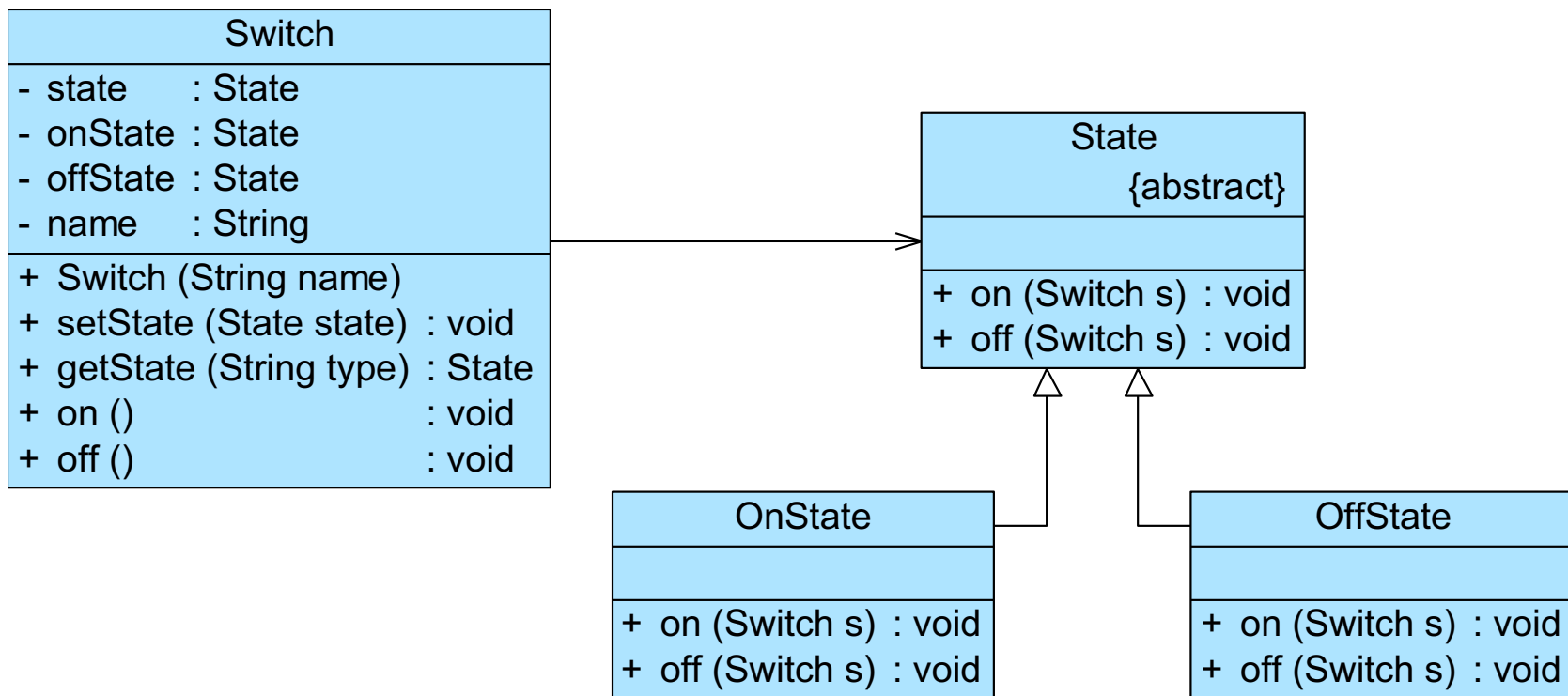
计算利息！

共享状态

- 动机
 - 在有些情况下，多个环境对象可能需要共享同一个状态
 - 如果希望在系统中实现多个环境对象共享一个或多个状态对象，那么需要将这些状态对象定义为环境类的静态成员对象

共享状态

- 结构



开关及其状态设计结构图

共享状态

- 实现
 - 开关类：Switch（环境类）
 - 开关状态类：SwitchState（抽象状态类）
 - 打开状态类：OnState（具体状态类）
 - 关闭状态类：OffState（具体状态类）
 - 客户端测试类：Client

演示.....

Code (designpatterns.state.switchstate)

使用环境类实现状态转换

- 动机
 - 对于客户端而言，无须关心状态类，可以为环境类设置默认的状态类，将状态的转换工作交给环境类（或具体状态类）来完成，具体的转换细节对于客户端而言是透明的
 - 可以通过环境类来实现状态转换，环境类作为一个状态管理器，统一实现各种状态之间的转换操作

使用环境类实现状态转换

- 实例

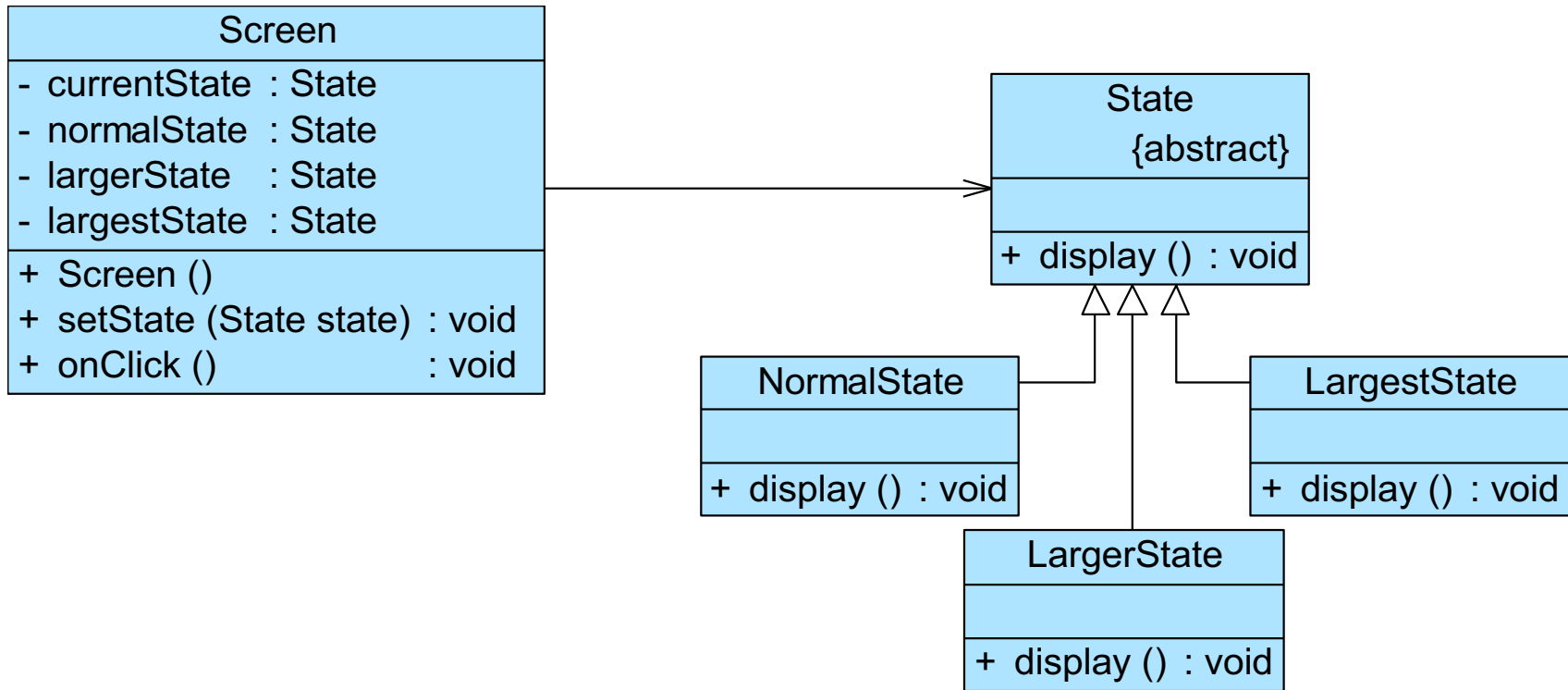
现欲开发一个屏幕放大镜工具，其具体功能描述如下：

用户单击“放大镜”按钮之后屏幕将放大一倍，再单击一次“放大镜”按钮屏幕再放大一倍，第三次单击该按钮后屏幕将还原到默认大小。

现使用状态模式来设计该屏幕放大镜工具。

使用环境类实现状态转换

- 结构



屏幕放大镜工具结构图

使用环境类实现状态转换

- 实现
 - 屏幕类：Screen（环境类）
 - 抽象状态类：State
 - 正常状态类：NormalState（具体状态类）
 - 二倍状态类：LargerState（具体状态类）
 - 四倍状态类：LargestState（具体状态类）
 - 客户端测试类：Client

演示.....

Code (designpatterns.state.screen)

状态模式的优缺点与适用环境

- 模式优点

- 封装了状态的转换规则，可以对状态转换代码进行集中管理，而不是分散在一个个业务方法中
- 将所有与某个状态有关的行为放到一个类中，只需要注入一个不同的状态对象即可使环境对象拥有不同的行为
- 允许状态转换逻辑与状态对象合成一体，而不是提供一个巨大的条件语句块，可以避免使用庞大的条件语句来将业务方法和状态转换代码交织在一起
- 可以让多个环境对象共享一个状态对象，从而减少系统中对象的个数

状态模式的优缺点与适用环境

- 模式缺点

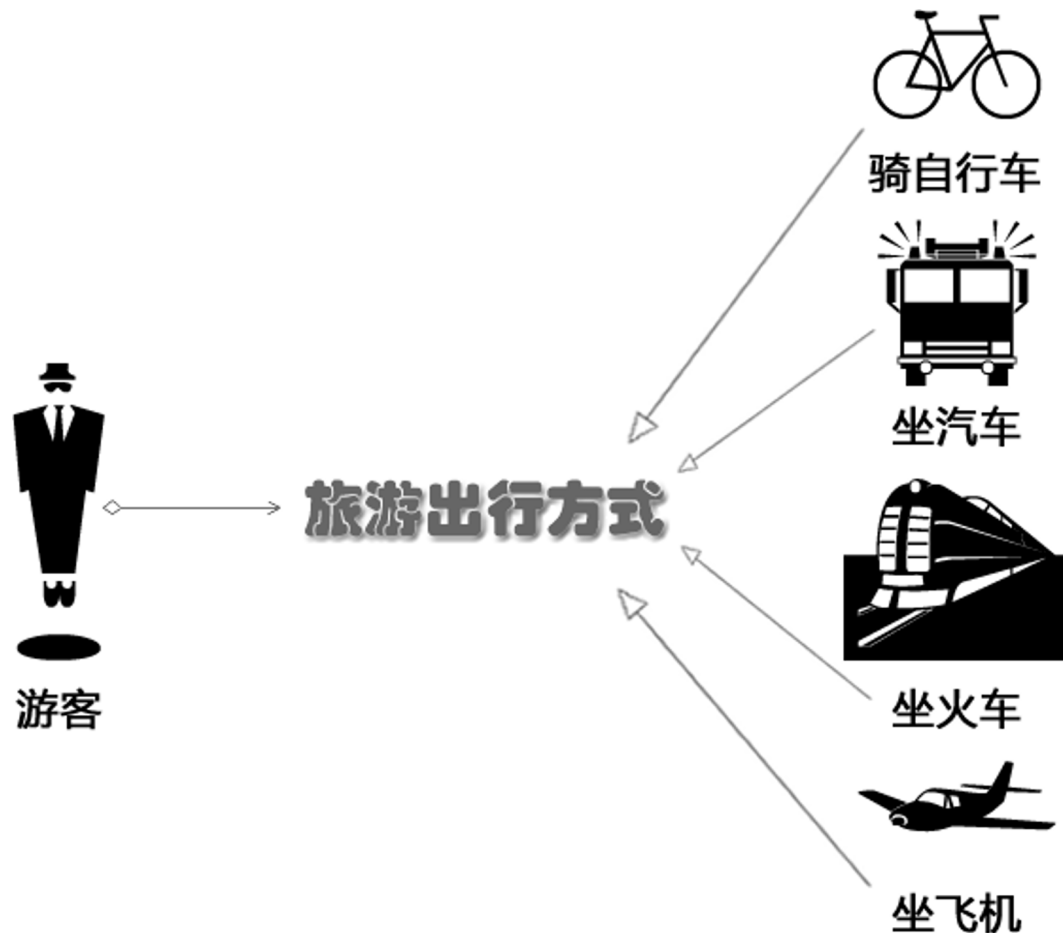
- 会增加系统中类和对象的个数，导致系统运行开销增大
- 结构与实现都较为复杂，如果使用不当将导致程序结构和代码混乱，增加系统设计的难度
- 对开闭原则的支持并不太好，增加新的状态类需要修改负责状态转换的源代码，否则无法转换到新增状态；而且修改某个状态类的行为也需要修改对应类的源代码

状态模式的优缺点与适用环境

- 模式适用环境
 - 对象的行为依赖于它的状态（例如某些属性值），状态的改变将导致行为的变化
 - 在代码中包含大量与对象状态有关的条件语句，这些条件语句的出现会导致代码的可维护性和灵活性变差，不能方便地增加和删除状态，并且导致客户类与类库之间的耦合增强

策略模式概述

- 旅游出行方式示意图



策略模式概述

- 分析
 - 实现某个目标的途径不止一条，可根据实际情况选择一条合适的途径
- 软件开发：
 - 多种算法，例如排序、查找、打折等
 - 使用硬编码(Hard Coding)实现将导致系统违背开闭原则，扩展性差，且维护困难
 - 可以定义一些独立的类来封装不同的算法，每一个类封装一种具体的算法→策略类

策略模式概述

- 策略模式的定义

策略模式：定义一系列算法，将每一个算法封装起来，并让它们可以相互替换。策略模式让算法可以独立于使用它的客户变化。

Strategy Pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

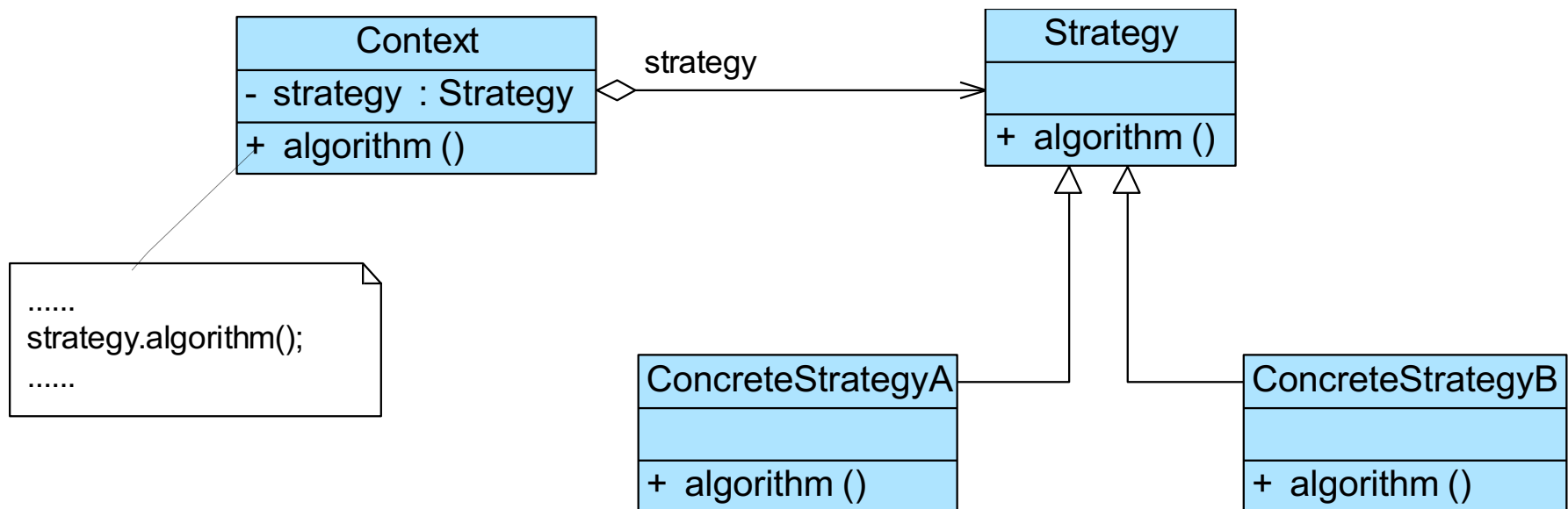
- 对象行为型模式

策略模式概述

- 策略模式的定义
 - 又称为政策(Policy)模式
 - 每一个封装算法的类称之为策略(Strategy)类
 - 策略模式提供了一种可插入式(Pluggable)算法的实现方案

策略模式的结构与实现

- 策略模式的结构



策略模式的结构与实现

- 策略模式的结构
 - 策略模式包含以下3个角色：
 - Context（环境类）
 - Strategy（抽象策略类）
 - ConcreteStrategy（具体策略类）

策略模式的结构与实现

- 策略模式的实现
 - 典型的抽象策略类代码：

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

策略模式的结构与实现

- 策略模式的实现
 - 典型的**具体策略类**代码：

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

策略模式的结构与实现

- 策略模式的实现
 - 典型的环境类代码：

```
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();  
    }  
}
```

策略模式的结构与实现

- 策略模式的实现
 - 典型的客户端代码片段：

```
.....  
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();  
.....
```

策略模式的应用实例

- 实例说明

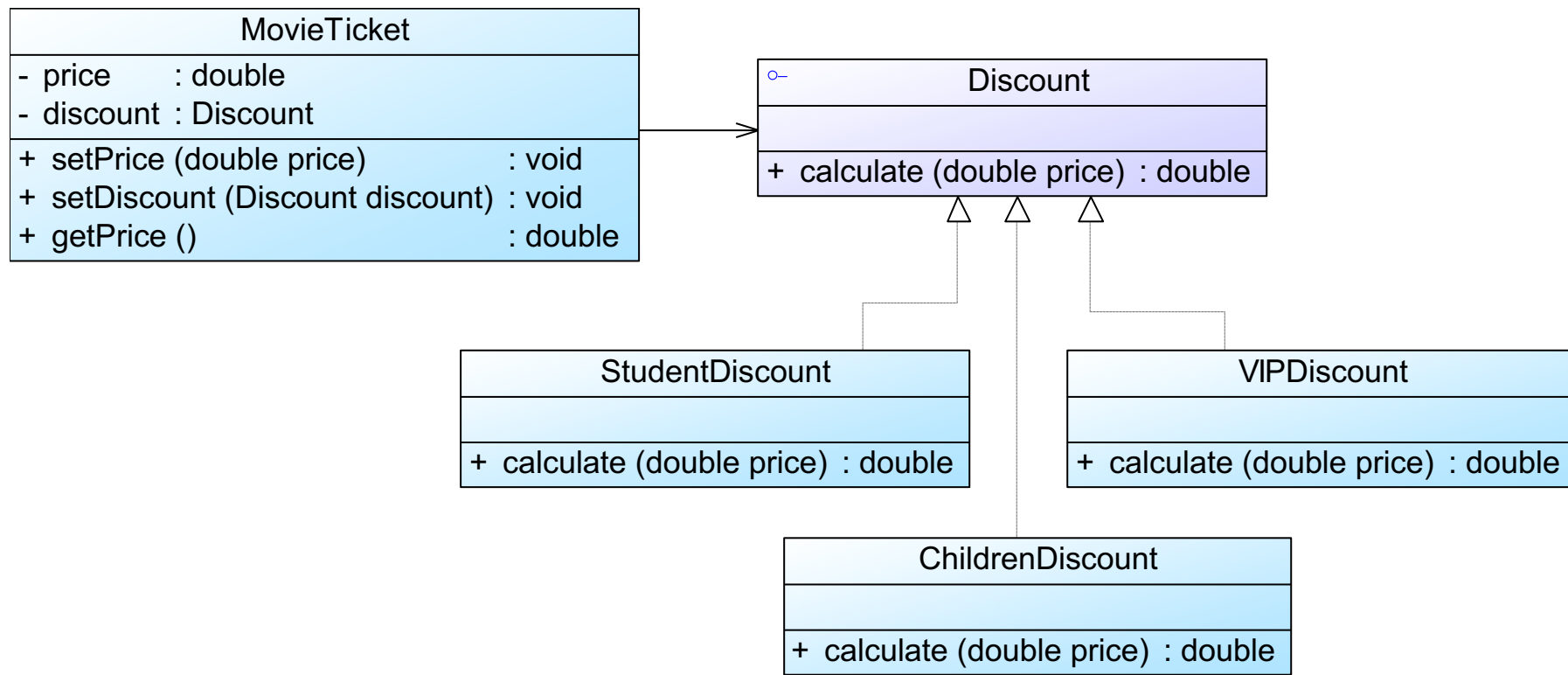
某软件公司为某电影院开发了一套影院售票系统，在该系统中需要为不同类型的用户提供不同的电影票打折方式，具体打折方案如下：

- (1) 学生凭学生证可享受票价8折优惠。
- (2) 年龄在10周岁及以下的儿童可享受每张票减免10元的优惠（原始票价需大于等于20元）。
- (3) 影院VIP用户除享受票价半价优惠外还可进行积分，积分累计到一定额度可换取电影院赠送的奖品。

该系统在将来可能还要根据需要引入新的打折方式。现使用策略模式设计该影院售票系统的打折方案。

策略模式的应用实例

- 实例类图



电影票打折方案结构图

策略模式的应用实例

- 实例代码

- (1) MovieTicket : 电影票类 , 充当环境类
- (2) Discount : 折扣类 , 充当抽象策略类
- (3) StudentDiscount : 学生票折扣类 , 充当具体策略类
- (4) ChildrenDiscount : 儿童票折扣类 , 充当具体策略类
- (5) VIPDiscount : VIP会员票折扣类 , 充当具体策略类
- (6) Client : 客户端测试类

演示.....

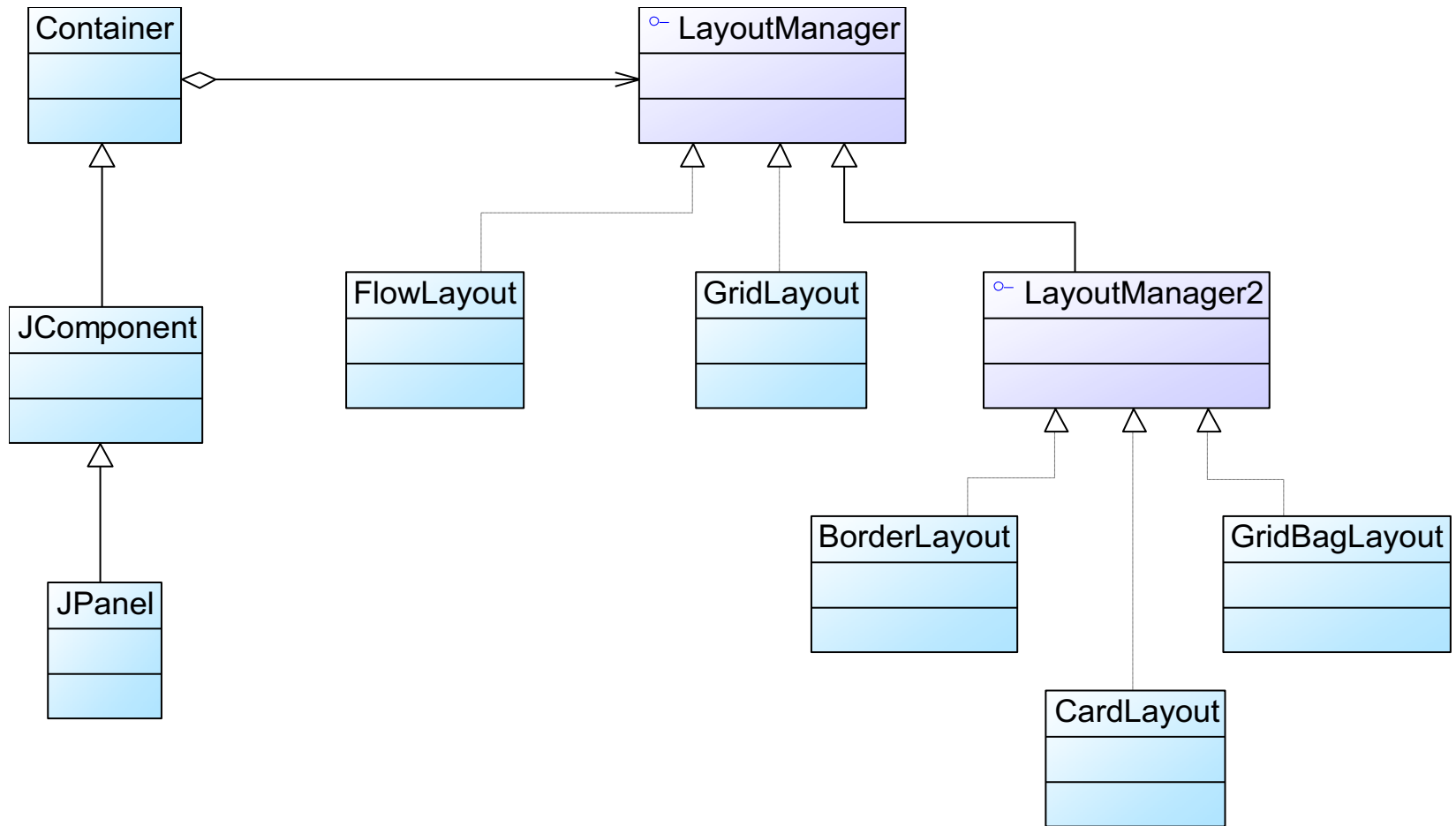
Code (designpatterns.strategy)

策略模式的应用实例

- 结果及分析
 - 如果需要更换具体策略类，无须修改源代码，只需修改**配置文件**即可，完全符合**开闭原则**

```
<?xml version="1.0"?>
<config>
  <className>designpatterns.strategy.StudentDiscount</className>
</config>
```

Java SE中的布局管理



策略模式的优缺点与适用环境

- 模式优点

- 提供了对开闭原则的完美支持，用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为
- 提供了管理相关的算法族的办法
- 提供了一种可以替换继承关系的办法
- 可以避免多重条件选择语句
- 提供了一种算法的复用机制，不同的环境类可以方便地复用策略类

策略模式的优缺点与适用环境

- 模式缺点
 - 客户端必须知道所有的策略类，并自行决定使用哪一个策略类
 - 将造成系统产生很多具体策略类
 - 无法同时在客户端使用多个策略类

策略模式的优缺点与适用环境

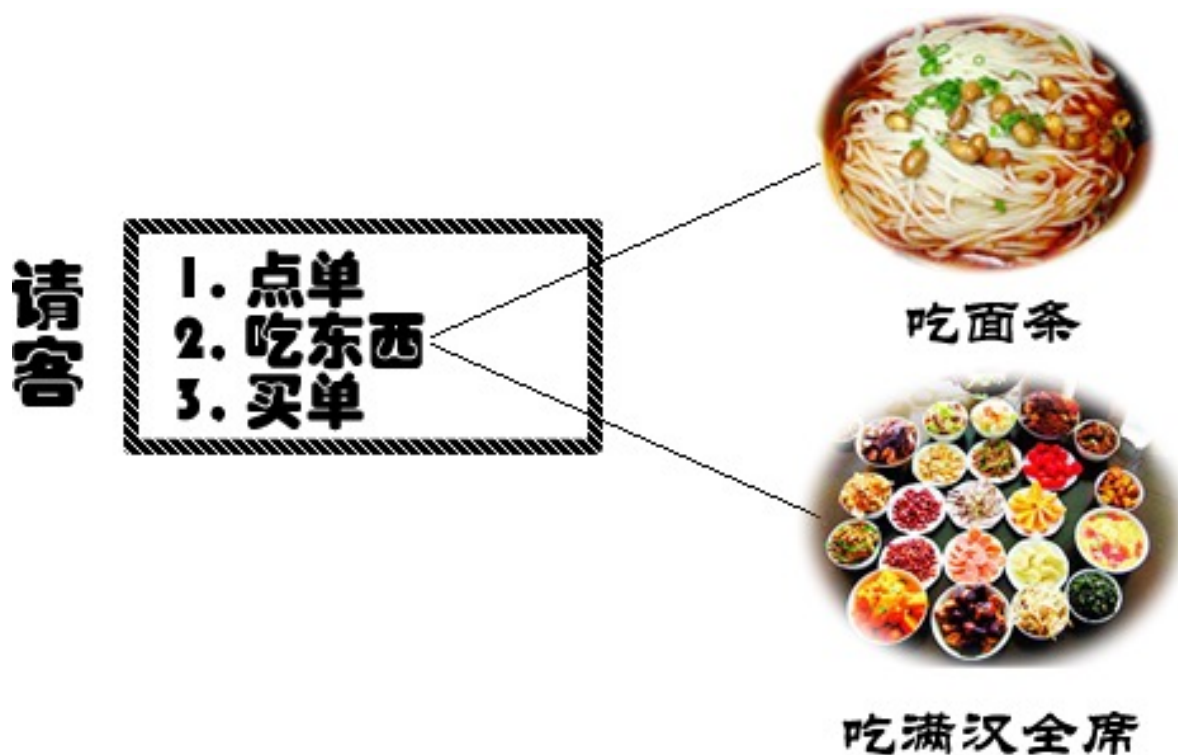
- 模式适用环境
 - 一个系统需要动态地在几种算法中选择一种
 - 避免使用难以维护的多重条件选择语句
 - 不希望客户端知道复杂的、与算法相关的数据结构，提高算法的保密性与安全性

思考

- 在策略模式中，一个环境类Context能否对应多个不同的策略等级结构？如何设计？

模板方法模式概述

- 请客吃饭示意图



模板方法模式概述

- 分析

- 请客吃饭：(1) 点单 → (2) 吃东西 → (3) 买单
- 软件开发：某个方法的实现需要多个步骤（类似“请客”），其中有些步骤是固定的（类似“买单”），有些步骤并不固定，存在可变性（类似“吃东西”）
- 模板方法模式：基本方法（点单、买东西和买单） ↔ 模板方法（“请客”）

具体方法

抽象方法

模板方法模式概述

- 模板方法模式的定义

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**。

Template Method Pattern: Define **the skeleton of an algorithm** in an operation, **deferring some steps to subclasses**. Template Method lets subclasses **redefine certain steps** of an algorithm without changing the algorithm's structure.

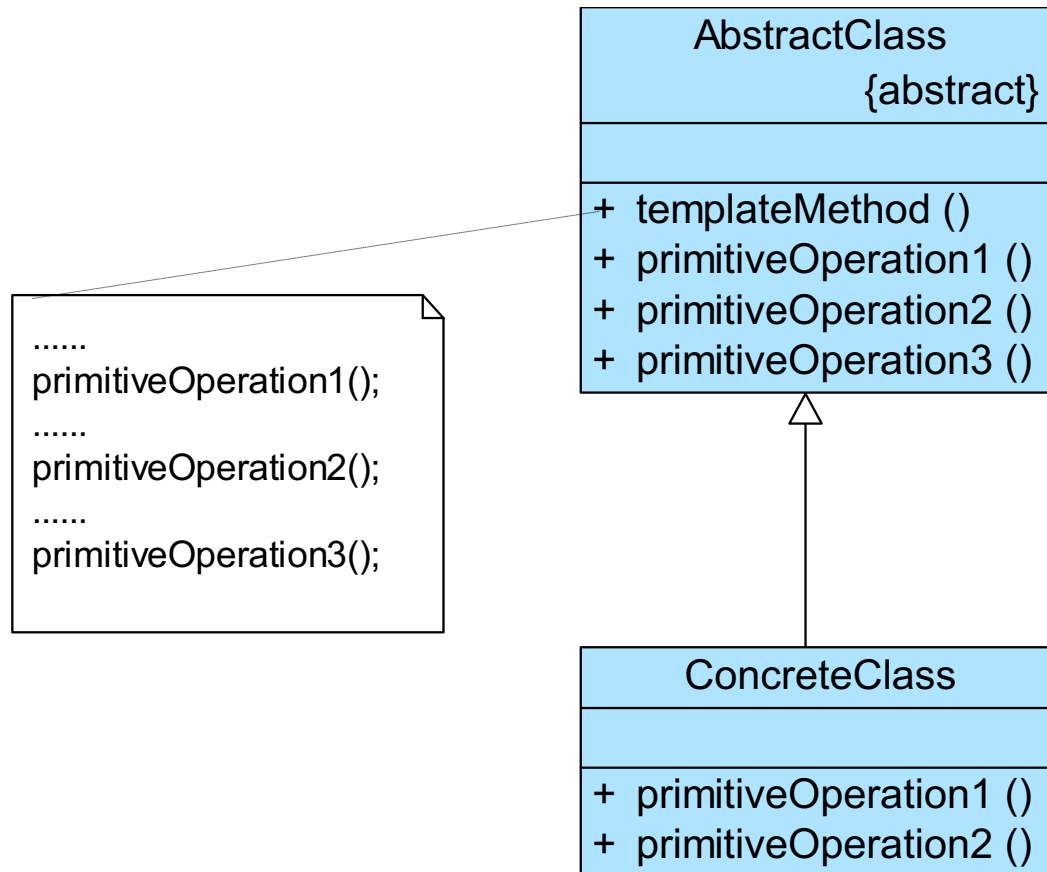
- 类行为型模式

模板方法模式概述

- 模板方法模式的定义
 - 是一种基于继承的代码复用技术
 - 将一些复杂流程的实现步骤封装在一系列基本方法中
 - 在抽象父类中提供一个称之为模板方法的方法来定义这些基本方法的执行次序，而通过其子类来覆盖某些步骤，从而使得相同的算法框架可以有不同的执行结果

模板方法模式的结构与实现

- 模板方法模式的结构



模板方法模式的结构与实现

- 模板方法模式的结构
 - 模板方法模式包含以下两个角色：
 - AbstractClass（抽象类）
 - ConcreteClass（具体子类）

模板方法模式的结构与实现

- 模板方法模式的实现
 - 模板方法 (Template Method)
 - 基本方法 (Primitive Method)
 - 抽象方法 (Abstract Method)
 - 具体方法 (Concrete Method)
 - 钩子方法 (Hook Method)

模板方法模式的结构与实现

- 模板方法模式的实现

.....

//模板方法

```
public void template() {
```

```
    open();
```

```
    display();
```

//通过钩子方法来确定某一步骤是否执行

```
    if(isPrint()) {
```

```
        print();
```

```
    }
```

```
}
```

//钩子方法

```
public boolean isPrint() {
```

```
    return true;
```

```
}
```

.....

模板方法模式的结构与实现

```
public abstract class AbstractClass {
```

//模板方法

```
public void templateMethod() {
```

```
    primitiveOperation1();
```

```
    primitiveOperation2();
```

```
    primitiveOperation3();
```

```
}
```

//基本方法—具体方法

```
public void primitiveOperation1() {
```

```
    //实现代码
```

```
}
```

//基本方法—抽象方法

```
public abstract void primitiveOperation2();
```

//基本方法—钩子方法

```
public void primitiveOperation3()
```

```
{ }
```

```
}
```

模板方法模式的结构与实现

- 模板方法模式的实现
 - 具体子类典型代码：

```
public class ConcreteClass extends AbstractClass {  
    public void primitiveOperation2() {  
        //实现代码  
    }  
  
    public void primitiveOperation3() {  
        //实现代码  
    }  
}
```


模板方法模式的应用实例

- 实例说明

某软件公司要为某银行的业务支撑系统开发一个利息计算模块，利息的计算流程如下：

(1) 系统根据账号和密码验证用户信息，如果用户信息错误，则系统显示出错提示。

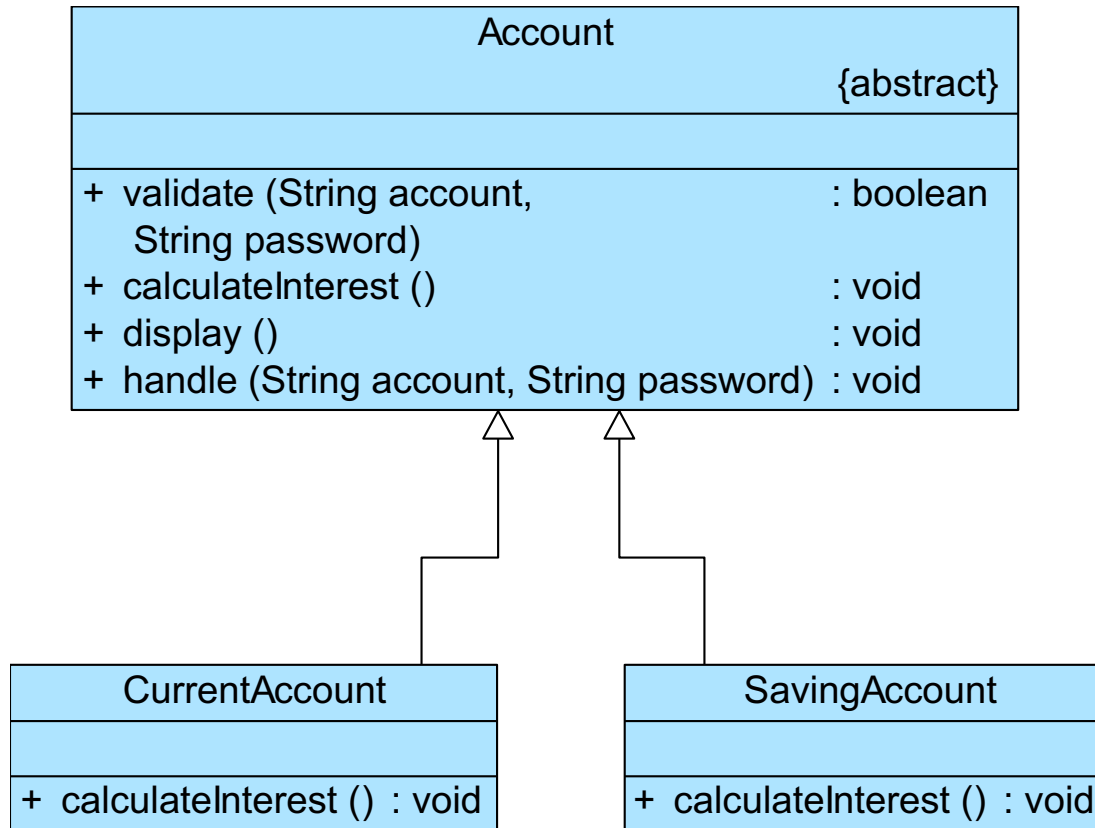
(2) 如果用户信息正确，则根据用户类型的不同使用不同的利息计算公式计算利息（如活期账户和定期账户具有不同的利息计算公式）。

(3) 系统显示利息。

现使用模板方法模式设计该利息计算模块。

模板方法模式的应用实例

- 实例类图



银行利息计算模块结构图

模板方法模式的应用实例

- 实例代码

- (1) Account：账户类，充当抽象类
- (2) CurrentAccount：活期账户类，充当具体子类
- (3) SavingAccount：定期账户类，充当具体子类
- (4) Client：客户端测试类

演示.....

Code (designpatterns.templateMethod)

模板方法模式的应用实例

- 结果及分析
 - 如果需要更换或增加**具体子类**，无须修改源代码，只需修改配置文件即可，**符合开闭原则**

```
<?xml version="1.0"?>
<config>
  <className>designpatterns.templatemethod.CurrentAccount</className>
</config>
```

钩子方法的使用

- 实例

某软件公司要为销售管理系统提供一个数据图表显示功能，该功能的实现包括以下几个步骤：

- (1) 从数据源获取数据。
- (2) 将数据转换为XML格式。
- (3) 以某种图表方式显示XML格式的数据。

该功能支持多种数据源和多种图表显示方式，但所有的图表显示操作都基于XML格式的数据，因此**可能需要对数据进行转换，如果从数据源获取的数据已经是XML数据，则无须转换。**

钩子方法的使用

- 结构



数据图表显示功能结构图

```
//designpatterns.templateMethod.hookMethod.Data Viewer.java
package designpatterns.templateMethod.hookMethod;
```

```
public abstract class DataView {
    //抽象方法： 获取数据
    public abstract void getData();
}
```

```
//designpatterns.templateMethod.hookMethod.XMLData Viewer.java
package designpatterns.templateMethod.hookMethod;
```

```
public class XMLData Viewer extends DataView {
```

```
    //实现父类方法： 获取数据
```

```
    public void getData() {
        System.out.println("从XML文件中获取数据。");
    }
}
```

```
    //实现父类方法： 显示数据，默认以柱状图方式显示，可结合桥接模式来改进
```

```
    public void displayData() {
        System.out.println("以柱状图显示数据。");
    }
}
```

```
    //覆盖父类的钩子方法
```

```
    public boolean isNotXMLData() {
        return false;
    }
}
```

```
}
```

```
}
```

```
}
```

模板方法模式的优缺点与适用环境

- 模式优点

- 在父类中形式化地定义一个算法，而由它的子类来实现细节的处理，在子类实现详细的处理算法时并不会改变算法中步骤的执行次序
- 提取了类库中的公共行为，将公共行为放在父类中，而通过其子类来实现不同的行为
- 可实现一种反向控制结构，通过子类覆盖父类的钩子方法来决定某一特定步骤是否需要执行
- 更换和增加新的子类很方便，符合单一职责原则和开闭原则

模板方法模式的优缺点与适用环境

- 模式缺点
 - 需要为每一个基本方法的不同实现提供一个子类，如果父类中可变的基本方法太多，将会导致类的个数增加，系统会更加庞大，设计也更加抽象（可结合桥接模式）

模板方法模式的优缺点与适用环境

- 模式适用环境
 - 一次性实现一个算法的不变部分，并将可变的行为留给子类来实现
 - 各子类中公共的行为应被提取出来，并集中到一个公共父类中，以避免代码重复
 - 需要通过子类来决定父类算法中某个步骤是否执行，实现子类对父类的反向控制

思考

- 在模板方法模式中，钩子方法如何实现子类控制父类的行为？

访问者模式概述

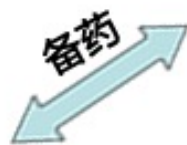
- 医院处方单处理示意图



划价人员



药房工作人员



人民医院	
处方笺	
费别: 现金	医疗证 / 医保卡号: _____
姓名: 李四	性别: 男
门诊 / 住院病历号: 475858	年龄: 26岁
诊断: 肺炎	科室: 药房
电话地址: 广州大道中路129号308	开具日期: 2010年09月10日
Rp	
收费名称	规格 单位 数量 用法 用量
1 青霉素	80万u 瓶 2 口服 成人一次二片, 一日三次, 儿童一日一至三片
2 黄连素片	50ml 支 2 口服 成人一次二片, 一日三次, 儿童一日一至三片
3 葡萄糖盐水	250ml 瓶 1 口服 250ml
医师: 00	
审核药师: _____	调配药师: _____ 核对、发药药师: _____
	收费员: _____




访问者模式概述

- 分析

- 处方单：

- 药品信息的集合，包含一种或多种不同类型的药品信息
 - 不同类型的工作人员（例如划价人员和药房工作人员）在操作同一个药品信息集合时将提供不同的处理方式
 - 可能会增加新类型的工作人员来操作处方单

- 软件开发：

- 处方单  对象结构
 - 药品信息  元素
 - 工作人员  访问者

访问者模式概述

- 分析
 - 对象结构中存储了多种不同类型的对象信息
 - 对同一对象结构中的元素的操作方式并不唯一，可能需要提供多种不同的处理方式
 - 还有可能需要增加新的处理方式

以不同的方式操作复杂对象结构

访问者模式概述

- 访问者模式的定义

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 Pattern: Represent an operation to be performed on the elements of an **object structure**. Visitor lets you **define a new operation without changing the classes of the elements** on which it operates.

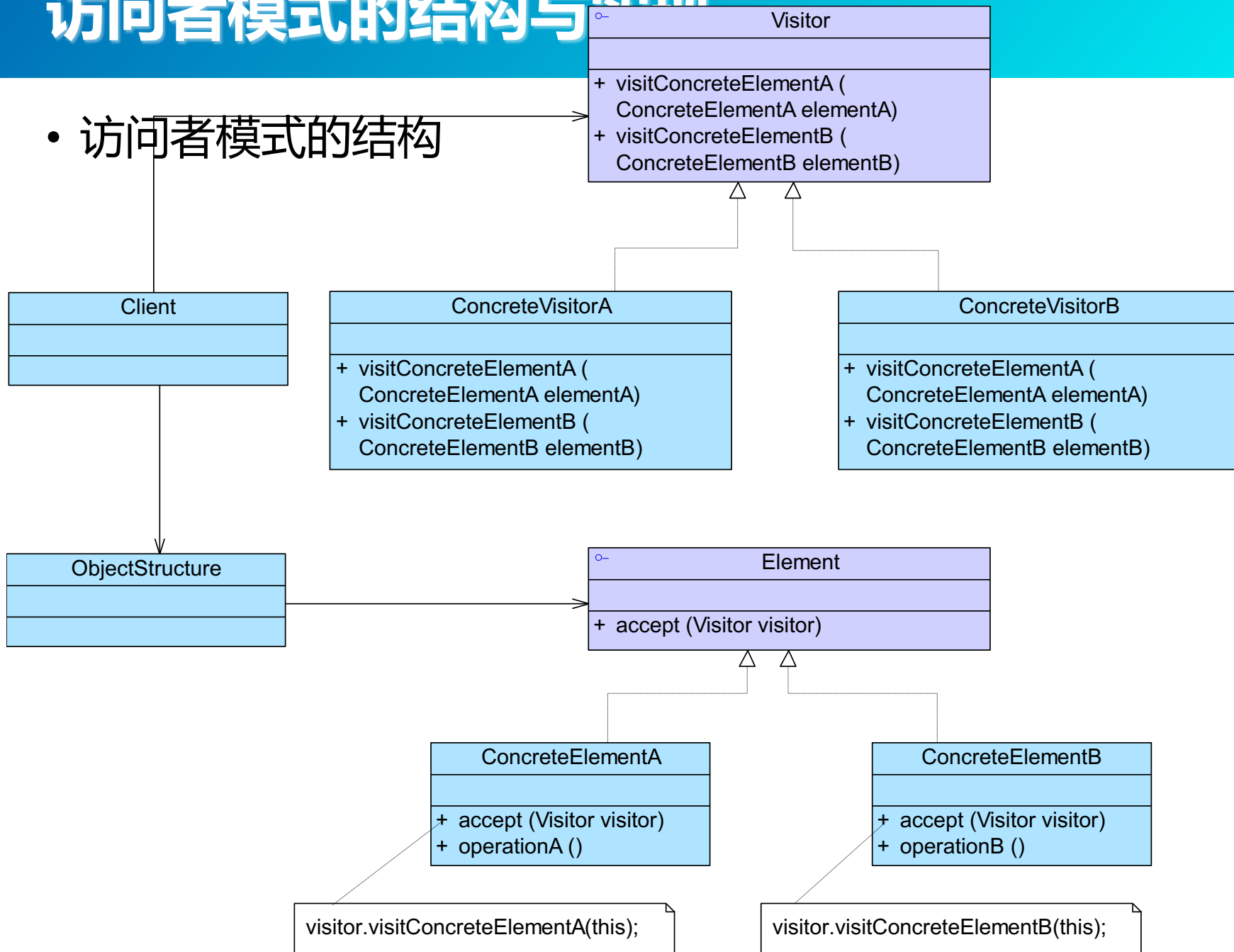
- 对象行为型模式

访问者模式概述

- 访问者模式的定义
 - 它为操作存储不同类型元素的对象结构提供了一种解决方案
 - 用户可以对不同类型的元素施加不同的操作

访问者模式的结构与实现

• 访问者模式的结构



访问者模式的结构与实现

- 访问者模式的结构
 - 访问者模式包含以下5个角色：
 - Visitor（抽象访问者）
 - ConcreteVisitor（具体访问者）
 - Element（抽象元素）
 - ConcreteElement（具体元素）
 - ObjectStructure（对象结构）

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的抽象访问者类代码：

```
public abstract class Visitor {  
    public abstract void visit(ConcreteElementA elementA);  
    public abstract void visit(ConcreteElementB elementB);  
  
    public void visit(ConcreteElementC elementC) {  
        //元素ConcreteElementC操作代码  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的**具体访问者类**代码：

```
public class ConcreteVisitor extends Visitor {  
    public void visit(ConcreteElementA elementA) {  
        //元素ConcreteElementA操作代码  
    }  
  
    public void visit(ConcreteElementB elementB) {  
        //元素ConcreteElementB操作代码  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的抽象元素类代码：

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

访问者模式的结构与实现

- 访问者模式的实现
 - 典型的**具体元素类**代码：

```
public class ConcreteElementA implements Element {  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
  
    public void operationA() {  
        //业务方法  
    }  
}
```

访问者模式的结构与实现

- 访问者模式的实现

- 双重分派

ConcreteElementA.accept(Visitor visitor)

- (1) 调用具体元素类的 **accept(Visitor visitor)** 方法，并将 Visitor 子类对象作为其参数

- (2) 在具体元素类 **accept(Visitor visitor)** 方法内部调用传入的

ConcreteVisitorA.visit(ConcreteElementA elementA)
<ConcreteVisitorA.visit(this)>

ConcreteElementA)，将当前具体元素类对象(this)作为参数，例如
visitor.visit(this)

- (3) 执行 Visitor 对象的 **visit()** 方法，在其中还可以调用具体元素对象的业务方法

ConcreteElementA.operationA()

```
import java.util.*;
```

```
public class ObjectStructure
```

```
{
```

```
    private ArrayList<Element> list = new ArrayList<Element>(); //定义一个集合用于存储元素对象
```

```
    //接受访问者的访问操作
```

```
    public void accept(Visitor visitor) {
```

```
        Iterator i=list.iterator();
```

```
        while(i.hasNext()) {
```

```
            ((Element)i.next()).accept(visitor); //遍历访问集合中的每一个元素
```

```
        }
```

```
    }
```

```
    public void addElement(Element element) {
```

```
        list.add(element);
```

```
    }
```

```
    public void removeElement(Element element) {
```

```
        list.remove(element);
```

```
    }
```

```
}
```


访问者模式的应用实例

• 实例说明

某公司OA系统中包含一个员工信息管理子系统，该公司员工包括正式员工和临时工，每周人力资源部和财务部等部门需要对员工数据进行汇总，汇总数据包括员工工作时间、员工工资等。该公司基本制度如下：

(1) 正式员工每周工作时间为40小时，不同级别、不同部门的员工每周基本工资不同；如果超过40小时，超出部分按照100元/小时作为加班费；如果少于40小时，所缺时间按照请假处理，请假所扣工资以80元/小时计算，直到基本工资扣除到零为止。除了记录实际工作时间外，人力资源部需记录加班时长或请假时长，作为员工平时表现的一项依据。

(2) 临时工每周工作时间不固定，基本工资按小时计算，不同岗位的临时工小时工资不同。人力资源部只需记录实际工作时间。

人力资源部和财务部工作人员可以根据各自的需要对员工数据进行汇总处理，人力资源部负责汇总每周员工工作时间，而财务部负责计算每周员工工资。

现使用访问者模式设计该系统，绘制类图并使用Java语言编码实现。

访问者模式的

Client

Department

{abstract}

+ visit (FulltimeEmployee employee) : void
+ visit (ParttimeEmployee employee) : void

• 实例类图

FADepartment

+ visit (FulltimeEmployee employee) : void
+ visit (ParttimeEmployee employee) : void

HRDepartment

+ visit (FulltimeEmployee employee) : void
+ visit (ParttimeEmployee employee) : void

EmployeeList

- list : ArrayList = new ArrayList()
+ addEmployee (Employee employee) : void
+ accept (Department handler) : void

Employee

+ accept (Department handler) : void

FulltimeEmployee

- name : String
- weeklyWage : double
- workTime : int
+ FulltimeEmployee (String name, double weeklyWage, int workTime)
+ setName (String name) : void
+ getName () : String
+ setWeeklyWage (double weeklyWage) : void
+ getWeeklyWage () : double
+ setWorkTime (int workTime) : void
+ getWorkTime () : int
+ accept (Department handler) : void

ParttimeEmployee

- name : String
- hourWage : double
- workTime : int
+ ParttimeEmployee (String name, double hourWage, int workTime)
+ setName (String name) : void
+ getName () : String
+ setHourWage (double hourWage) : void
+ getHourlyWage () : double
+ setWorkTime (int workTime) : void
+ getWorkTime () : int
+ accept (Department handler) : void

员工数据汇总模块结构图

访问者模式的应用实例

- 实例代码

- (1) Employee : 员工类, 充当抽象元素类
- (2) FulltimeEmployee : 全职员工类, 充当具体元素类
- (3) ParttimeEmployee : 兼职员工类, 充当具体元素类
- (4) Department : 部门类, 充当抽象访问者类
- (5) FADepartment : 财务部类, 充当具体访问者类
- (6) HRDepartment : 人力资源部类, 充当具体访问者类
- (7) EmployeeList : 员工列表类, 充当对象结构
- (8) Client : 客户端测试类

演示.....

Code (designpatterns.visitor)

访问者模式的应用实例

- 结果及分析

- 如果需要增加或更换具体访问者类，无须修改源代码，只需修改配置文件，从增加新的访问者的角度来看，

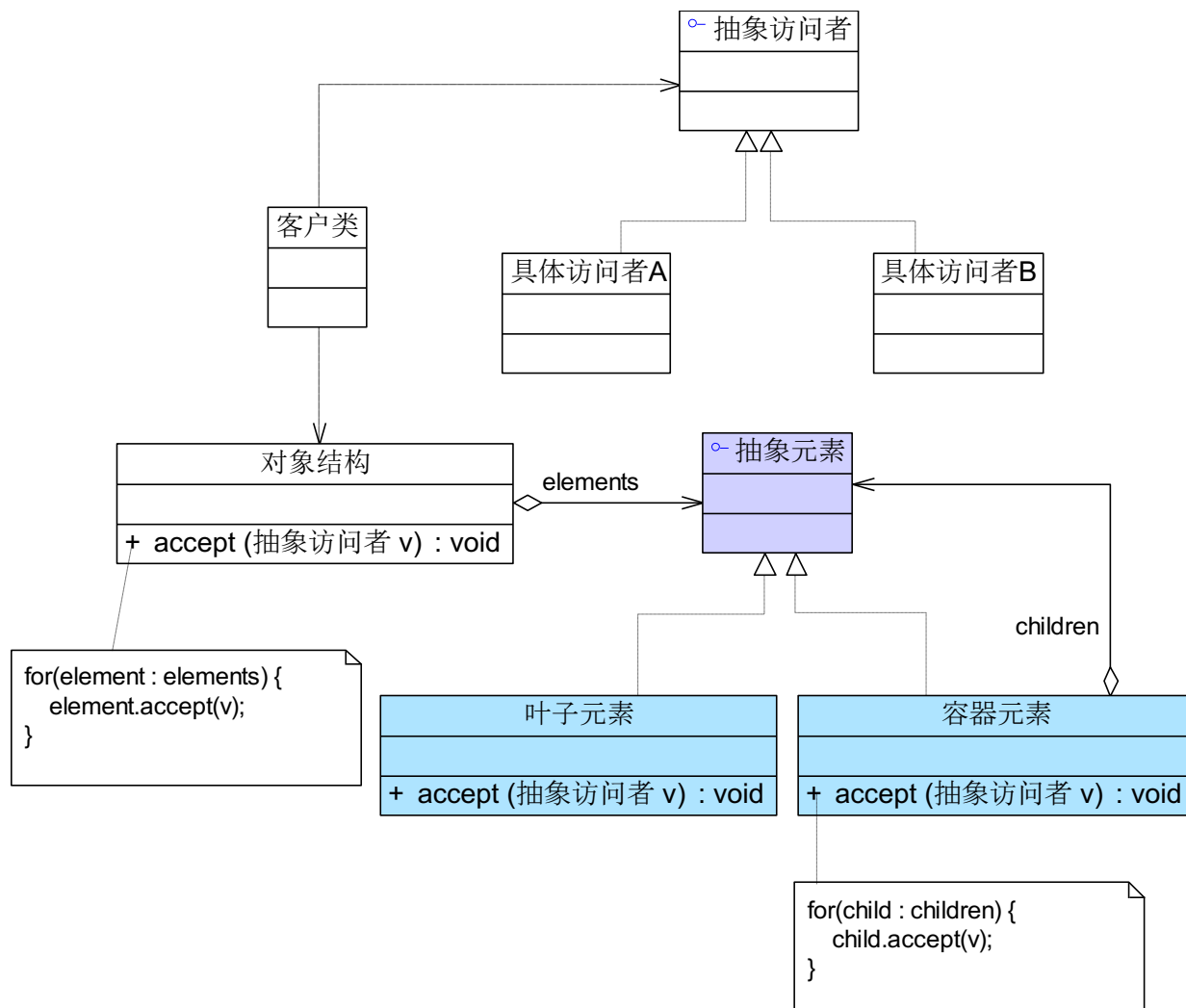
```
<?xml version="1.0"?>
<config>
  <className>designpatterns.visitor.FADepartment</className>
</config>
```

类中增加相应的访问方法，从增加新的元素的角度来

- 看
- 正式员工张无忌实际工资为：3700.0元。
 - 正式员工杨过实际工资为：2000.0元。
 - 正式员工段誉实际工资为：2240.0元。
 - 临时工洪七公实际工资为：1600.0元。
 - 临时工郭靖实际工资为：1080.0元。

访问者模式与组合模式联用

- 结构



访问者模式的优缺点与适用环境

- 模式优点
 - 增加新的访问操作很方便
 - 将有关元素对象的访问行为集中到一个访问者对象中，而不是分散在一个个的元素类中，类的职责更加清晰
 - 让用户能够在不修改现有元素类层次结构的情况下，定义作用于该层次结构的操作

访问者模式的优缺点与适用环境

- 模式缺点
 - 增加新的元素类很困难
 - 破坏了对象的封装性

访问者模式的优缺点与适用环境

- 模式适用环境
 - 一个对象结构包含多个类型的对象，希望对这些对象实施一些依赖其具体类型的操作
 - 需要对一个对象结构中的对象进行很多不同的且无关的操作，并需要避免让这些操作“污染”这些对象的类，也不希望在增加新操作时修改这些类
 - 对象结构中对象对应的类很少改变，但经常需要在此对象结构上定义新的操作

思考

- 什么是双重分派机制？如何用代码实现？