实验报告

实验设计名称： 状态模式

专 业： 软 件 工技术

班 级： 2018级

姓 名： 林志龙

学 号： 1817700302

2020 年 4 月 22 日

1. 实验内容

在某银行系统定义的账户有三种状态：

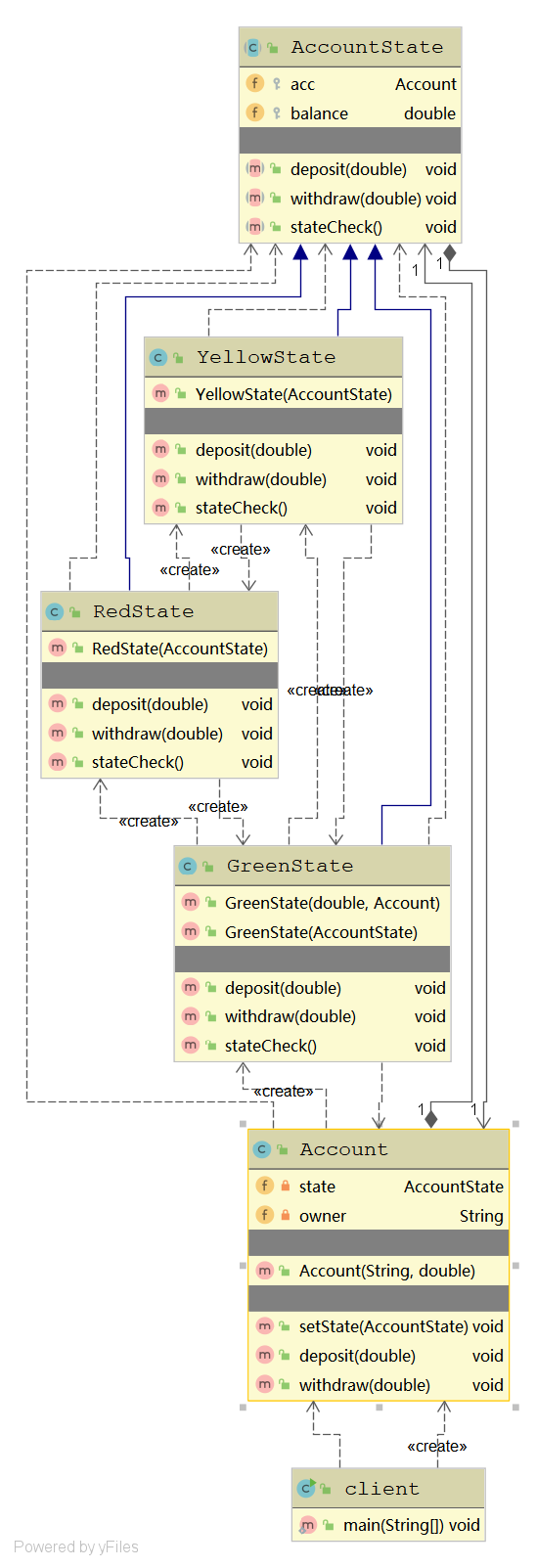
(1) 如果账户(Account)中余额(balance)大于等于0，此时账户的状态为绿色(GreenState)，即正常状态，表示既可以向该账户存款(deposit)也可以从该账户取款(withdraw)；

(2) 如果账户中余额小于0，并且大于等于-1000，则账户的状态为黄色(YellowState)，即欠费状态，此时既可以向该账户存款也可以从该账户取款；

(3) 如果账户中余额小于-1000，那么账户的状态为红色(RedState)，即透支状态，此时用户只能向该账户存款，不能再从中取款。

现用状态模式来实现状态的转化问题，用户只需要执行简单的存款和取款操作，系统根据余额数量自动转换到相应的状态。

1. 实验设计类图(UML)



1. 实验代码

状态模式包含三个角色，分别为：(Context) 环境类、(State) 抽象状态类和(ConcreteState) 具体状态类。

1. 环境类代码实现：

**package** 状态模式;  
*//银行账户：环境类***public class** Account {  
 **private** AccountState **state**;  
 **private** String **owner**;  
 *//构造函数* **public** Account(String owner, **double** init){  
 **this**.**owner** =owner;  
 **this**.**state** = **new** GreenState(init,**this**);  
 System.***out***.println(**this**.**owner** +**"开户，初始金额为："** +init);  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
 }  
   
 **public void** setState(AccountState state){  
 **this**.**state** =state;  
 }  
 *//存款* **public void** deposit(**double** amount){  
 System.***out***.println(**this**.**owner** + **"存款"**+amount + **"元"**);  
 **state**.deposit(amount);  
 System.***out***.println(**"现在余额为"**+**state**.**balance**);  
 System.***out***.println(**"现在账户状态为"** + **state**.getClass().getName());  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
  
 }  
 *//取款* **public void** withdraw(**double** amount){  
 System.***out***.println(**this**.**owner** +**"取款"** +amount);  
 **state**.withdraw(amount);  
 System.***out***.println(**"现在余额为"** +**state**.**balance**);  
 System.***out***.println(**"现在账户状态为"** + **state**.getClass().getName());  
 System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  
  
 }  
}

1. 抽象状态类代码实现：

**package** 状态模式;  
*//抽象状态类***public abstract class** AccountState {  
   
 **protected** Account **acc**;  
 **protected double balance**;  
 *//抽象存款方法* **public abstract void** deposit(**double** amount) ;  
 *//抽象取款方法* **public abstract void** withdraw(**double** amount);  
 *//抽象自我检查状态方法* **public abstract void** stateCheck();  
}

1. 具体状态类代码实现：

①绿色状态具体实现类：

**package** 状态模式;  
*//绿色状态 ：具体实现类***public class** GreenState **extends** AccountState {  
  
 *//构造函数* **public** GreenState(**double** balance, Account acc) {  
 **this**.**balance** =balance;  
 **this**.**acc** =acc;  
 }  
  
 **public** GreenState(AccountState state) {  
 **this**.**balance** =state.**balance**;  
 **this**.**acc** =state.**acc**;  
 }  
 *// 计算重新存款后的金额* @Override  
 **public void** deposit(**double** amount) {  
 **this**.**balance**+=amount;  
 stateCheck();  
 }  
 *//计算重新取款后的金额* @Override  
 **public void** withdraw(**double** amount) {  
 **this**.**balance**-=amount;  
 stateCheck();  
 }  
 *//检查金额是否满足绿色状态，不满足则考虑红色和黄色状态* @Override  
 **public void** stateCheck() {  
 **if**(**balance**>=-1000&&**balance**<0){  
 **acc**.setState(**new** YellowState(**this**));  
 }**else if**(**balance**<-1000){  
 **acc**.setState(**new** RedState(**this**));  
 }  
 }  
}

②黄色状态具体实现类：

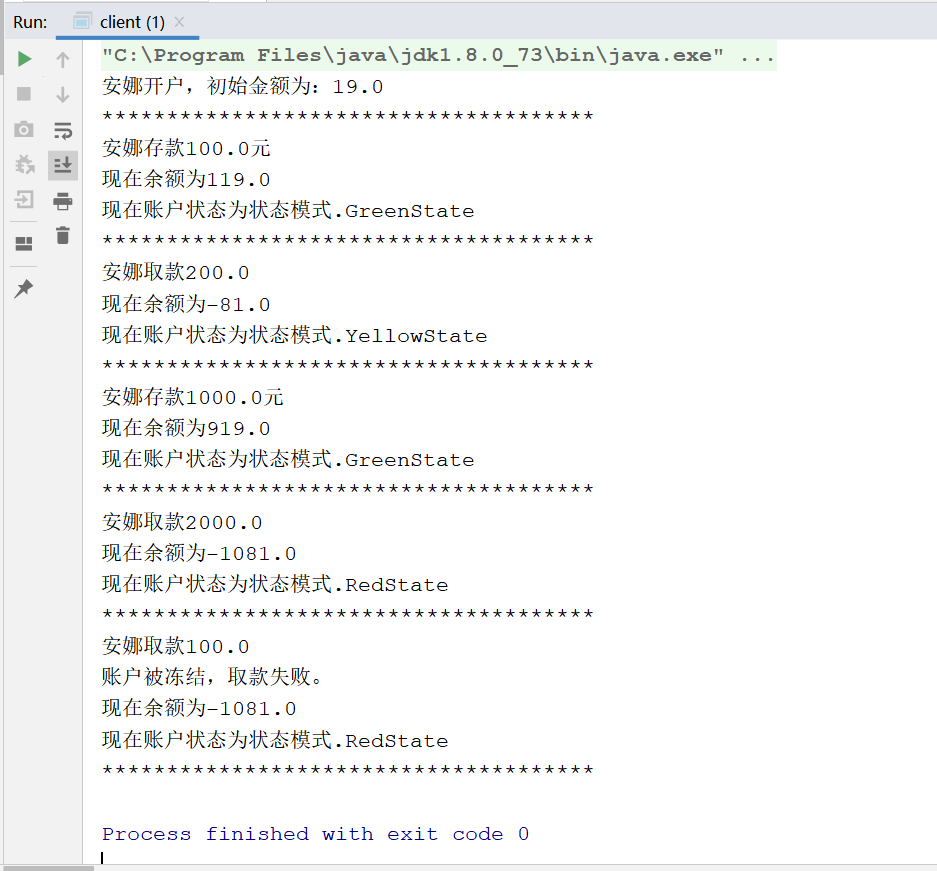
**package** 状态模式;  
*//黄色状态：具体实现类***public class** YellowState **extends** AccountState {  
 *//构造函数* **public** YellowState(AccountState state) {  
 **this**.**balance** =state.**balance**;  
 **this**.**acc** =state.**acc**;  
 }  
 *// 计算重新存款后的金额* @Override  
 **public void** deposit(**double** amount) {  
 **this**.**balance**+=amount;  
 stateCheck();  
 }  
 *//计算重新取款后的金额* @Override  
 **public void** withdraw(**double** amount) {  
 **this**.**balance**-=amount;  
 stateCheck();  
 }  
 *//检查金额是否满足黄色状态，不满足则考虑红色和绿色状态* @Override  
 **public void** stateCheck() {  
 **if**(**balance**>=0){  
 **acc**.setState(**new** GreenState(**this**));  
 }**else if**(**balance**<-100){  
 **acc**.setState(**new** RedState(**this**));  
  
 }  
 }  
}

③红色代码具体实现类：

**package** 状态模式;  
*//红色状态：具体实现类***public class** RedState **extends** AccountState {  
 *//构造函数* **public** RedState(AccountState state) {  
 **this**.**balance** =state.**balance**;  
 **this**.**acc** =state.**acc**;  
 }  
 *// 计算重新存款后的金额* @Override  
 **public void** deposit(**double** amount) {  
 **this**.**balance**+=amount;  
 stateCheck();  
 }  
 *// 若前一状态判定为红色状态，则不允许取款，并冻结账户* @Override  
 **public void** withdraw(**double** amount) {  
 System.***out***.println(**"账户被冻结，取款失败。"**);  
 }  
 *//判断存款后的金额是否满足红色状态，不满足则考虑黄色和绿色状态* @Override  
 **public void** stateCheck() {  
 **if**(**balance**>=0){  
 **acc**.setState(**new** GreenState(**this**));  
 }**else if**(**balance**>=-1000){  
 **acc**.setState(**new** YellowState(**this**));  
 }  
  
 }  
}

1. 实验结果

运行结果截图：



1. 实验分析与总结

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。其别名为状态对象，状态模式是一种对象行为型模式。

状态模式包含三个角色：环境类又称为上下文类，它是拥有状态的对象，在环境类中维护一个抽象状态类State的实例，这个实例定义当前状态，在具体实现时，它是一个State子类的对象，可以定义初始状态；抽象状态类用于定义一个接口以封装与环境类的一个特定状态相关的行为；具体状态类是抽象状态类的子类，每一个子类实现一个与环境类的一个状态相关的行为，每一个具体状态类对应环境的一个具体状态，不同的具体状态类其行为有所不同。

状态模式描述了对象状态的变化以及对象如何在每一种状态下表现出不同的行为。

状态模式的主要优点在于封装了转换规则，并枚举可能的状态，它将所有与某个状态有关的行为放到一个类中，并且可以方便地增加新的状态，只需要改变对象状态即可改变对象的行为，还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；其缺点在于使用状态模式会增加系统类和对象的个数，且状态模式的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的混乱，对于可以切换状态的状态模式不满足“开闭原则”的要求。

状态模式适用情况包括：对象的行为依赖于它的状态（属性）并且可以根据它的状态改变而改变它的相关行为；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，使客户类与类库之间的耦合增强。