Lecture 6.
State Pattern (状态模式)
(Behavioral)

- 行为模式关心算法和对象之间的责任分配。
- 它关心的不是仅仅描述对象或类的模式，而是要更加侧重描述它们之间的通信模式。
- 行为模式刻画了很难在运行时跟踪的复杂的控制流。该模式将软件开发者的注意力从控制流转移到对象相互关联的方式方面。

Professor:

Yushan (Michael) Sun

Fall 2020

Contents of this lecture

Example of State Related System

Examples Leading to the State Pattern

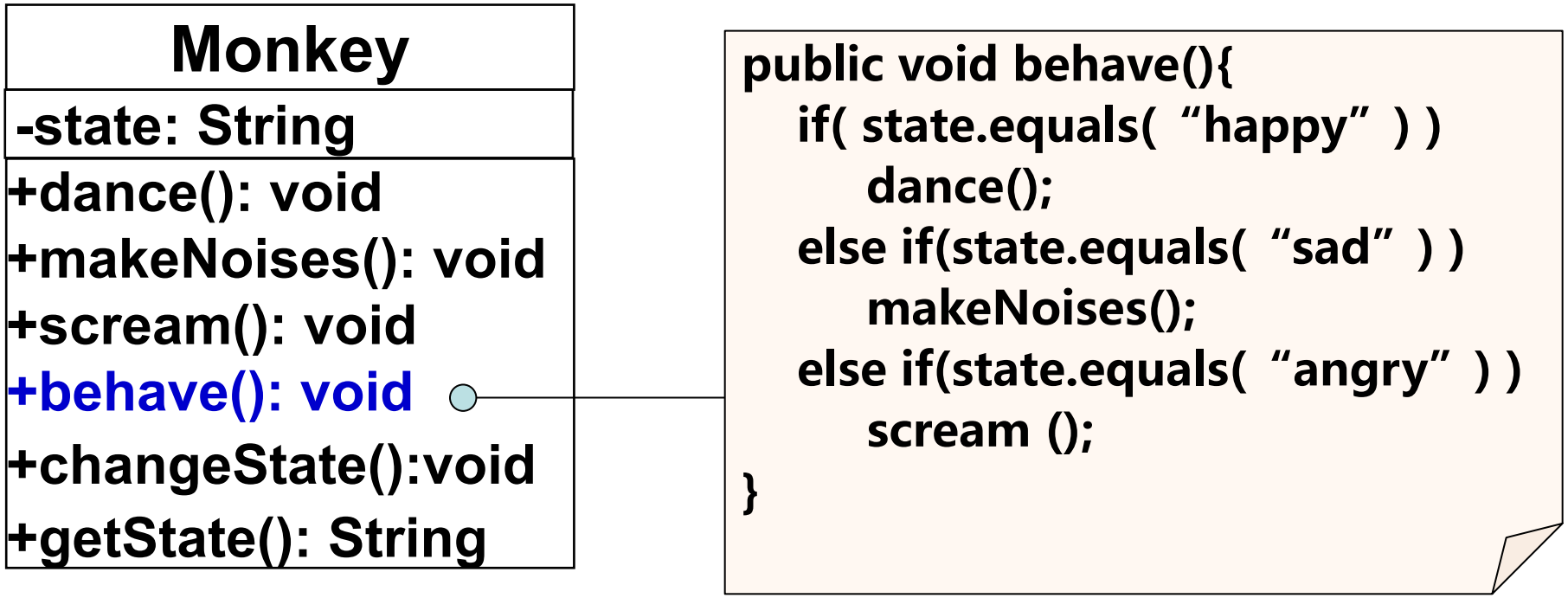
【例1】Monkey example.

A monkey is really a moody animal. Suppose that a monkey has 3 states (moods):

- Happy：dance;
- Mad：make noises
- Angry：scream

Examples Leading to the State Pattern

First design： use a single class to encapsulate behaviors



Original design of class Monkey

Examples Leading to the State Pattern

Drawbacks of the design:

a) 有许多状态相关的条件语句. A lot of conditional statements in the method behave()

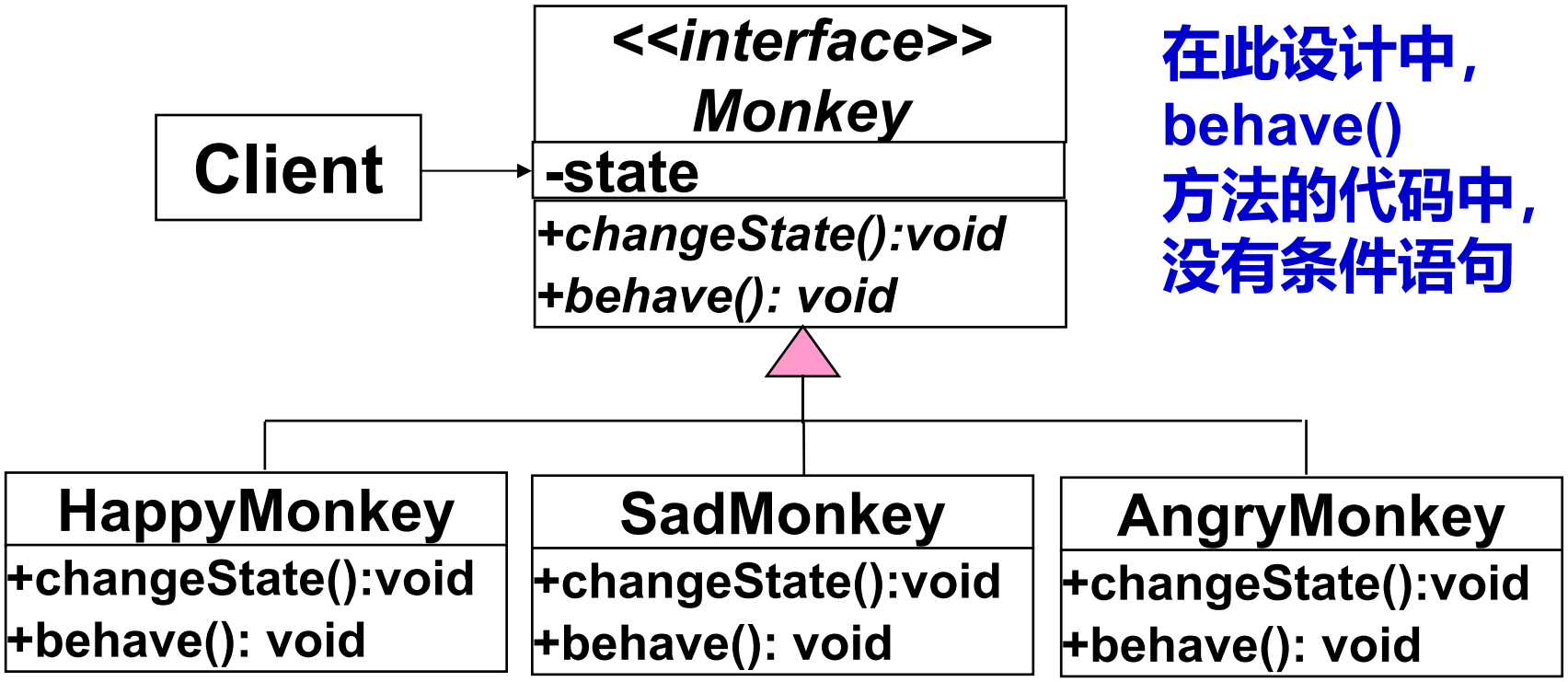
b) 增加新的状态相关的行为比较困难。It is not easy to add new state related behaviors

How to improve the design?

Idea: build a state class hierarchy to encapsulate each state into a separate class

Examples Leading to the State Pattern

New design：



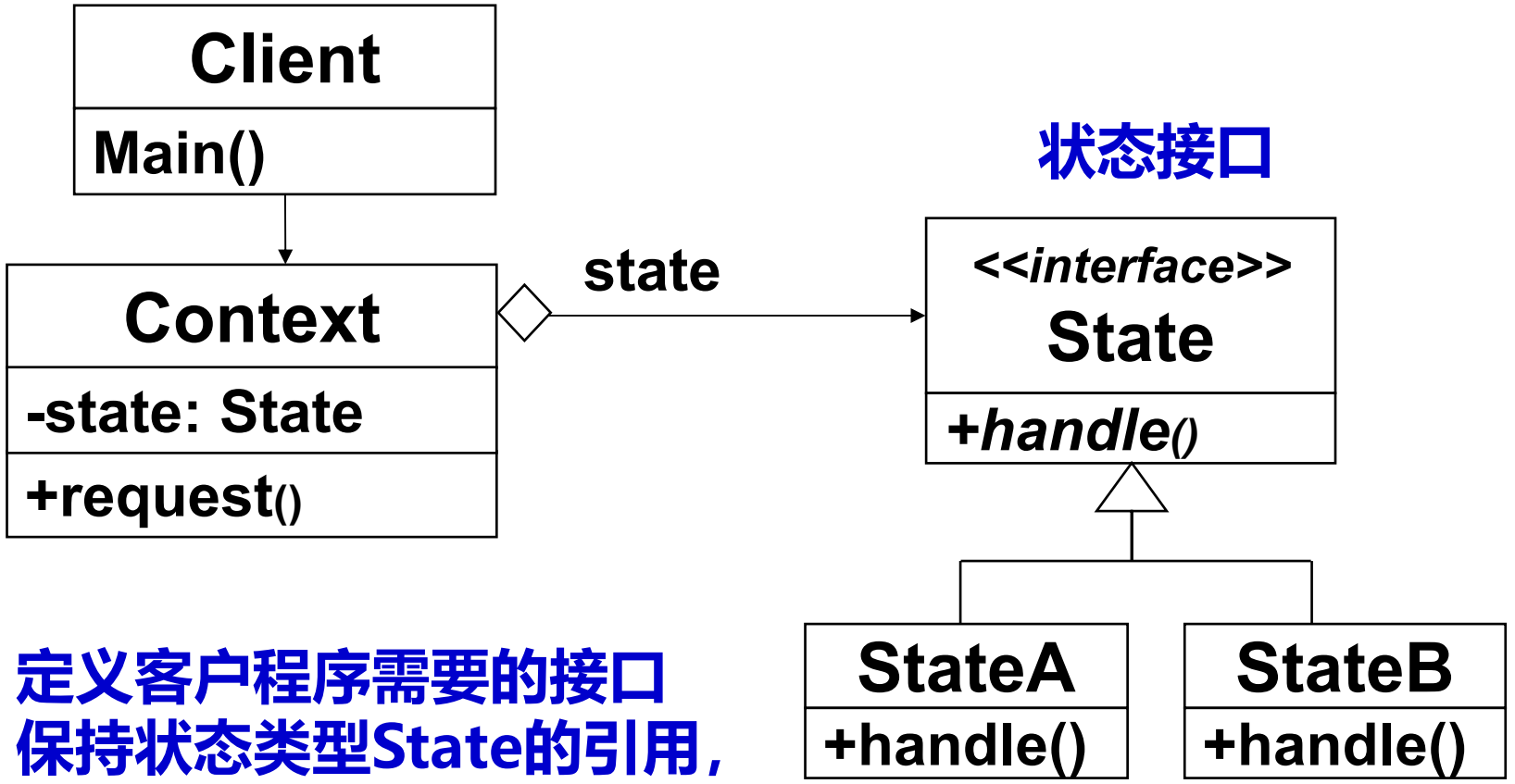
在此设计中，
behave()
方法的代码中，
没有条件语句

Design by encapsulating each state into a separate class

Back

The State Design Pattern

The State Design Pattern



1. 定义客户程序需要的接口
2. 保持状态类型State的引用，程序运行时，Context对象包含当前状态子类对象
3. 可以包含部分业务逻辑

封装了不同状态下的行为

Structure of State Pattern

The State Design Pattern

Participants

- Context
  - 定义客户程序需要的接口。Defines the interface of interest to clients.
  - 保持当前状态子类对象。Maintains an instance of a State subclass that defines the current state.
  - 可以包含部分业务逻辑

The State Design Pattern

- **State**

  Defines an interface for encapsulating the behavior associated with a particular state of the Context.

- **State subclasses**

  Each subclass implements a behavior associated with a state of the Context.

The State Design Pattern

When to use state pattern?

a) 当对象的行为依赖于状态，而该对象必须根据其状态（在运行时)改变其行为

When an object’ s behavior depends on its state, and it must change its behavior (at run-time) depending on that state.

Tiger
-age: int
+hunt()

运行时改变状态与行为

The State Design Pattern

b) 当操作带有大量状态相关的、多部分的条件语句

When operations have large, multipart conditional statements that depend on the object's state.

- **The State pattern puts each branch of the conditional in a separate class.**
- **This lets you treat the object‘s state as an object in its own right that can vary independently from other objects. (使得软件工程师可以独立改变一种状态的代码)**
- **见Monkey的例子**

The State Design Pattern

协作关系 (Collaborations)

a) **Context代表状态相关的请求。**
   Context delegates state-specific requests to the current Concrete State object.

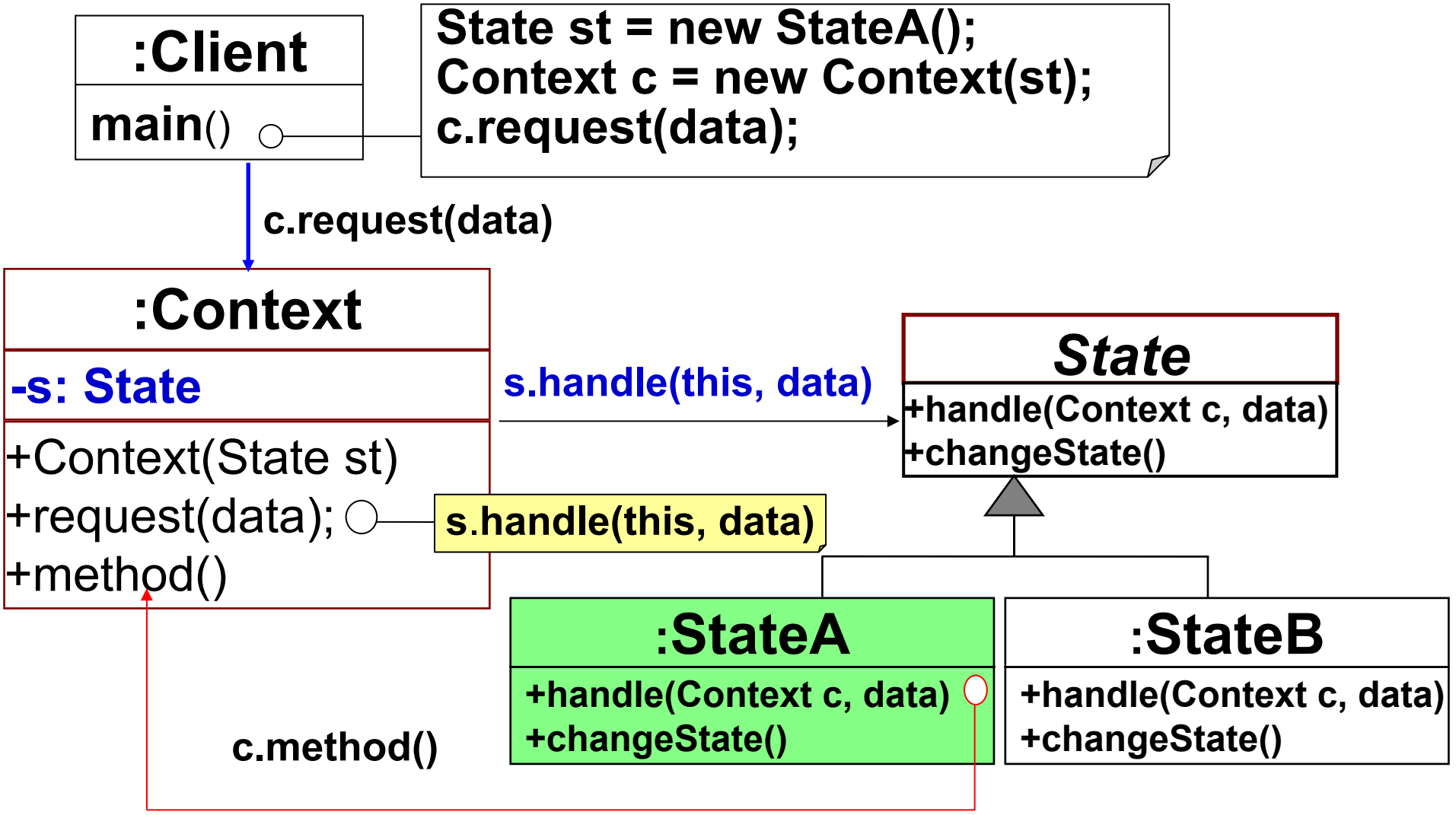
b) **Context可将自己作为一个对象通过参数传递给状态对象，从而可让状态对象访问Context类的方法**
   A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.

The State Design Pattern

c) 客户对象先创建一个具体的状态子类的对象s，然后，在创建Context对象时，通过参数将s传递给Context对象。此后，客户程序就不必与该状态对象直接交互。Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don’t have to deal with the State objects directly.

d) 状态转换： Context类或具体的状态子类负责
Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

The State Design Pattern



Typical Collaborations of the state design pattern

The State Design Pattern

关于状态的转换的建议：

- 客户类Client不负责状态转换
- 客户类只创建一个“种子对象”，然后，传递给Context对象；
- 再由Context对象根据状态变化的情况，负责创建所有的其它的对象
- 或由State层次类决定状态的转换

The State Design Pattern

状态模式的优点：Advantages

a) **Easy in adding a new state class**

Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.

b) **It makes state transitions explicit**

Introducing separate objects for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal states.

Back

Design Example Using the State Pattern

Design Example Using the State Pattern

【例2】交通信号控制软件的例子

a) 信号灯有红，黄，绿三个状态。使用状态模式设计交通信号灯控制软件。

b) 设计一个LightState抽象类，和3个具体的状态子类

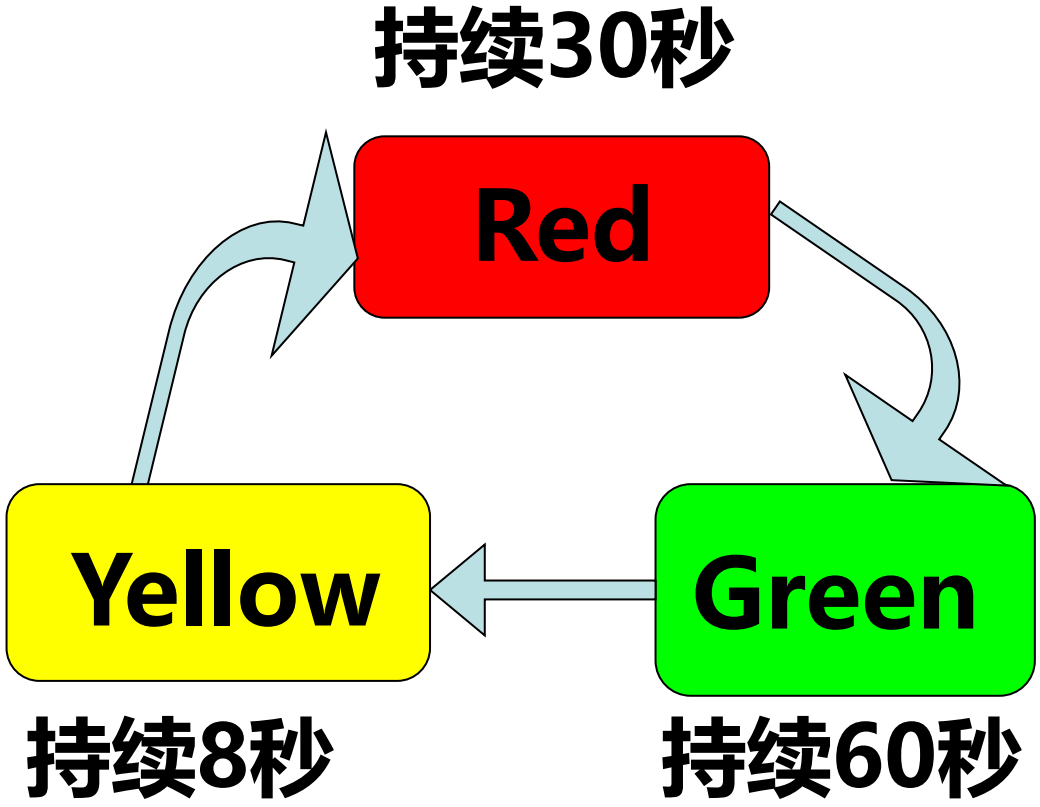
- Red
- Yellow
- Green

c) 状态超类或者状态子类的任务：

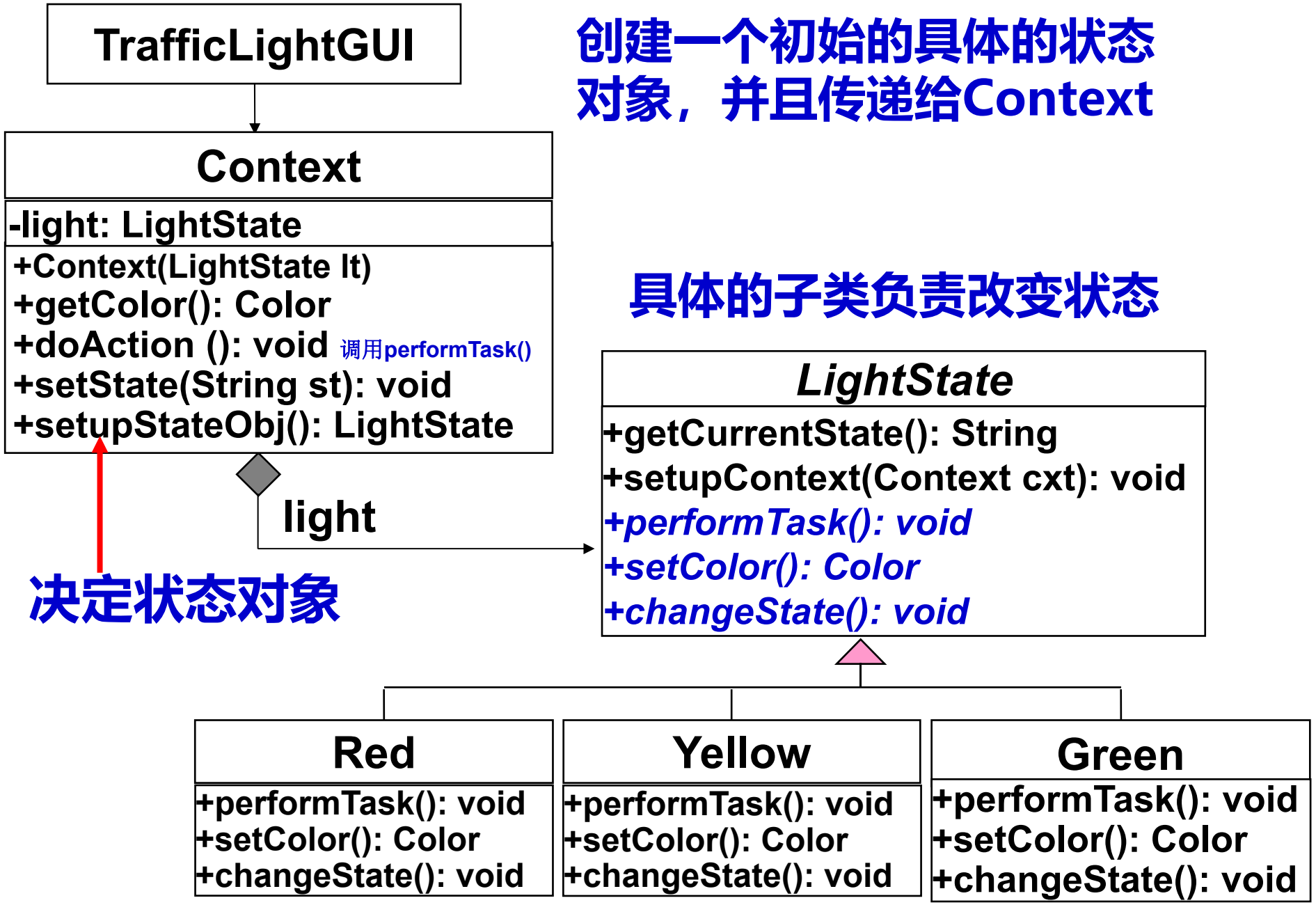
- 拍照，统计车辆个数，记录违章车辆
- 负责改变状态，红→绿→黄→红

d) Context对象提供交通灯所需要的颜色。

Design Example Using the State Pattern



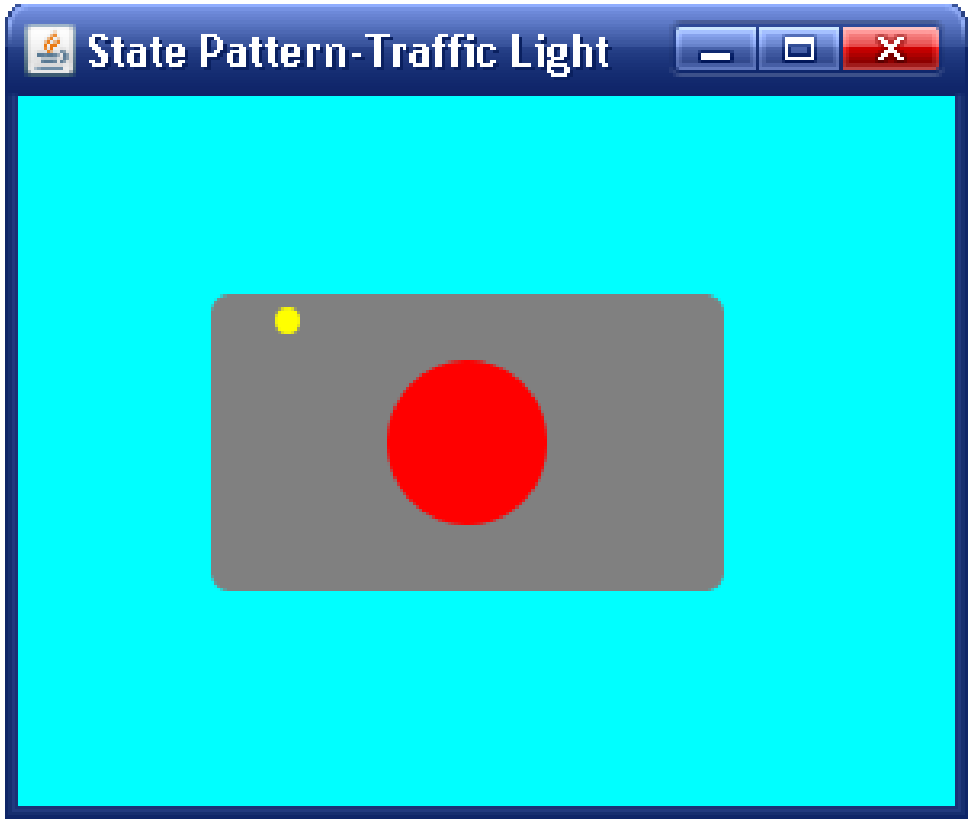
交通灯控制软件状态图


TrafficLightGUI
创建一个初始的具体的状态对象，并且传递给Context
Context
-light: LightState
+Context(LightState lt)
+getColor(): Color
+doAction (): void 调用performTask()
+setState(String st): void
+setupStateObj(): LightState
具体的子类负责改变状态
light
决定状态对象
LightState
+getCurrentState(): String
+setupContext(Context cxt): void
+performTask(): void
+setColor(): Color
+changeState(): void
Red
+performTask(): void
+setColor(): Color
+changeState(): void
Yellow
+performTask(): void
+setColor(): Color
+changeState(): void
Green
+performTask(): void
+setColor(): Color
+changeState(): void

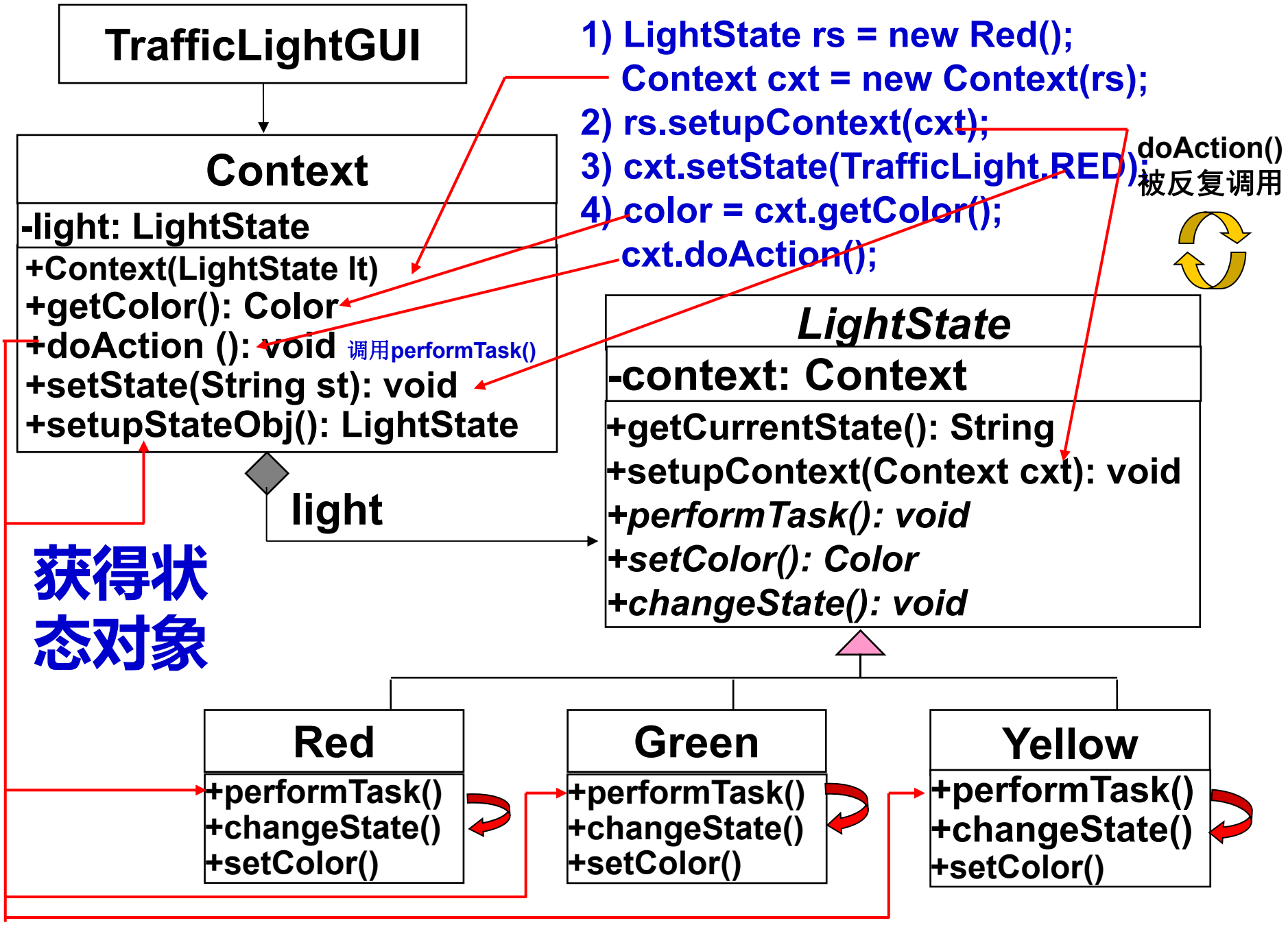
Design Example Using the State Pattern

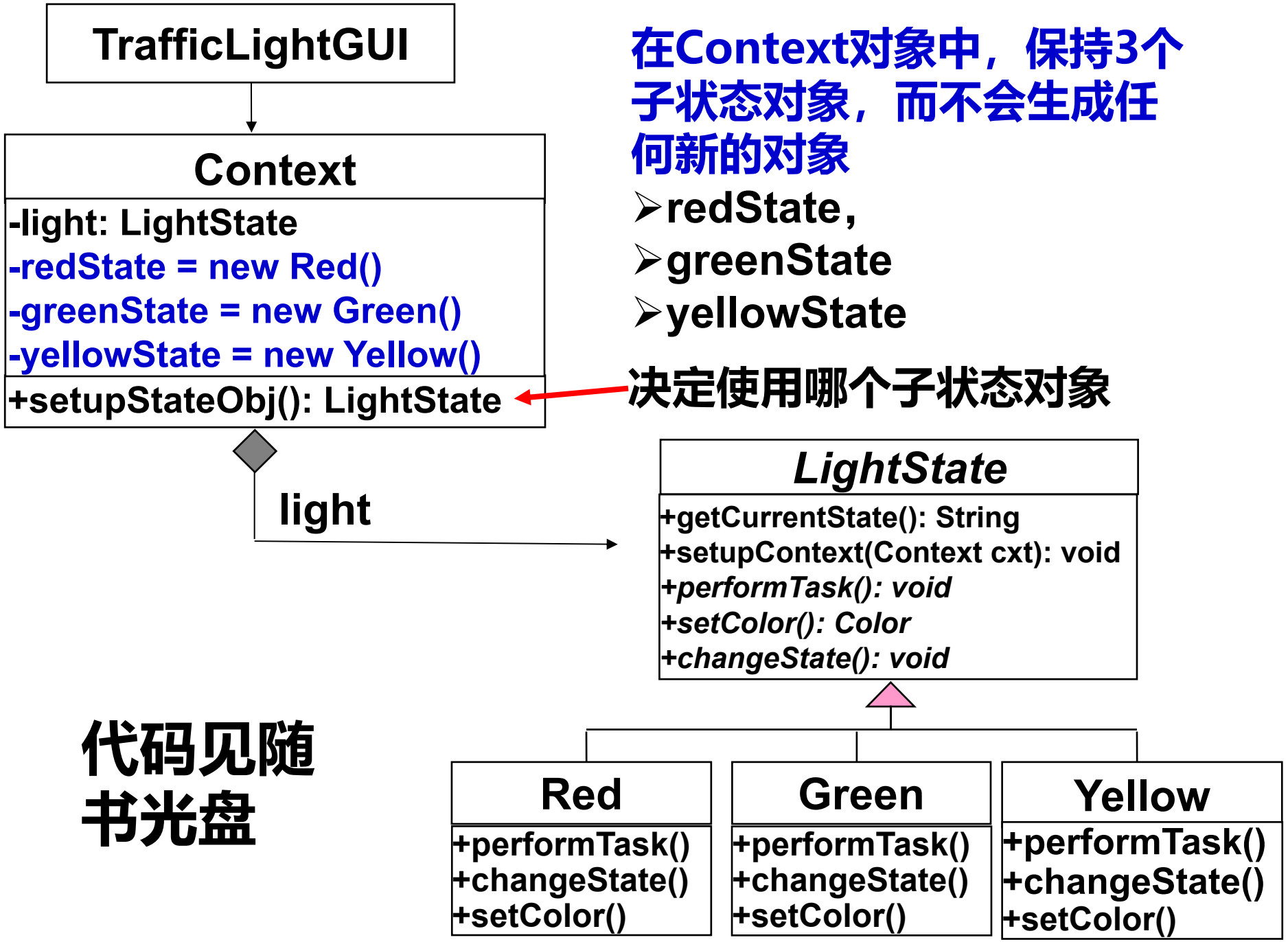
- **关于主用户界面TrafficLightGUI**
- **Extends JFrame**
- **自动更新图形界面**:
  - 利用JFrame类的paint()方法，该方法在TrafficLightGUI生成的时候，将会自动被调用。
  - 在paint()方法中，使用了repaint()方法，该方法将自动调用将用paint()。
- **停顿效果**:
  - 使用了Thread类的wait()方法。产生停顿效果，从而模拟了交通信号灯。

Design Example Using the State Pattern



交通灯控制软件图形界面


TrafficLightGUI
1) LightState rs = new Red();
Context cxt = new Context(rs);
2) rs.setupContext(cxt);
3) cxt.setState(TrafficLight.RED);
4) color = cxt.getColor();
cxt.doAction();
doAction()
被反复调用
Context
-light: LightState
+Context(LightState lt)
+getColor(): Color
+doAction (): void 调用performTask()
+setState(String st): void
+setupStateObj(): LightState
light
LightState
-context: Context
+getCurrentState(): String
+setupContext(Context cxt): void
+performTask(): void
+setColor(): Color
+changeState(): void
获得状态对象
Red
+performTask()
+changeState()
+setColor()
Green
+performTask()
+changeState()
+setColor()
Yellow
+performTask()
+changeState()
+setColor()


TrafficLightGUI
Context
-light: LightState
-redState = new Red()
-greenState = new Green()
-yellowState = new Yellow()
+setupStateObj(): LightState
light
在Context对象中，保持3个子状态对象，而不会生成任何新的对象
redState,
greenState
yellowState
决定使用哪个子状态对象
LightState
+getCurrentState(): String
+setupContext(Context cxt): void
+performTask(): void
+setColor(): Color
+changeState(): void
Red
+performTask()
+changeState()
+setColor()
Green
+performTask()
+changeState()
+setColor()
Yellow
+performTask()
+changeState()
+setColor()
代码见随书光盘

Design Example Using the State Pattern

本程序的交互情况

1) 【在客户类中】客户类首先创建Red类的对象，在创建Context类的对象的时候，将Red类的对象以参数的形式传给Context类的对象cxt。

```
rs = new Red();
cxt = new Context(rs);
```

2) 【在客户类中】在状态类中还需要保持Context类的对象，因此需要调用

```
rs.setupContext(cxt);
```

将刚刚创建的Context对象cxt传递给状态类

3) 【在客户类中】还需要给Context类的状态变量state赋予初始值

```
cxt.setState(TrafficLight.RED);
```

4) 【在客户类中循环调用】在方法runTrafficLights()中，使用语句

```
color = cxt.getColor();
cxt.doAction();
```

反复调用Context类的方法，以便产生循环显示灯的颜色的效果。

Design Example Using the State Pattern

第4步的进一步解释

- 【在Context类中】每当调用cxt.doAction()的时候，相应地状态子类的doAction()将会被调用，changeState()也将会被调用
- 【在状态类中】 changeState()负责反复更新Context类中的state变量
- 【在Context类中】 Context类的setupStateObj()根据新的状态，决定使用某个状态子类对象light

light.performTask()

Design Example Using the State Pattern

本设计的特点

- 在Context类中保持String类型的状态变量state
- 在TrafficLight类中保持Context类的对象cxt
- 在Context类中，调用状态类的方法

  setupContext(Context)

  将Context对象传入到TrafficLight类中

- 在状态类中，调用context类的setState(state)方法，对Context类的state变量进行更新
- 在Context类中，首先统一创建三个状态子类的对象。然后，在运行时，根据state变量的情况，决定使用哪个状态子类对象。

Design Example Using the State Pattern

本设计的优点：

1. 在修改状态子类的代码的时候，不需要修改Context类的代码。
2. 在增加新的状态子类的情况下，需少许修改状态类的changeState()方法与Context类中的相应代码。
3. 在对状态层次类添加一个新状态子类或修改状态子类时，都不需要修改客户类TrafficLightGUI。

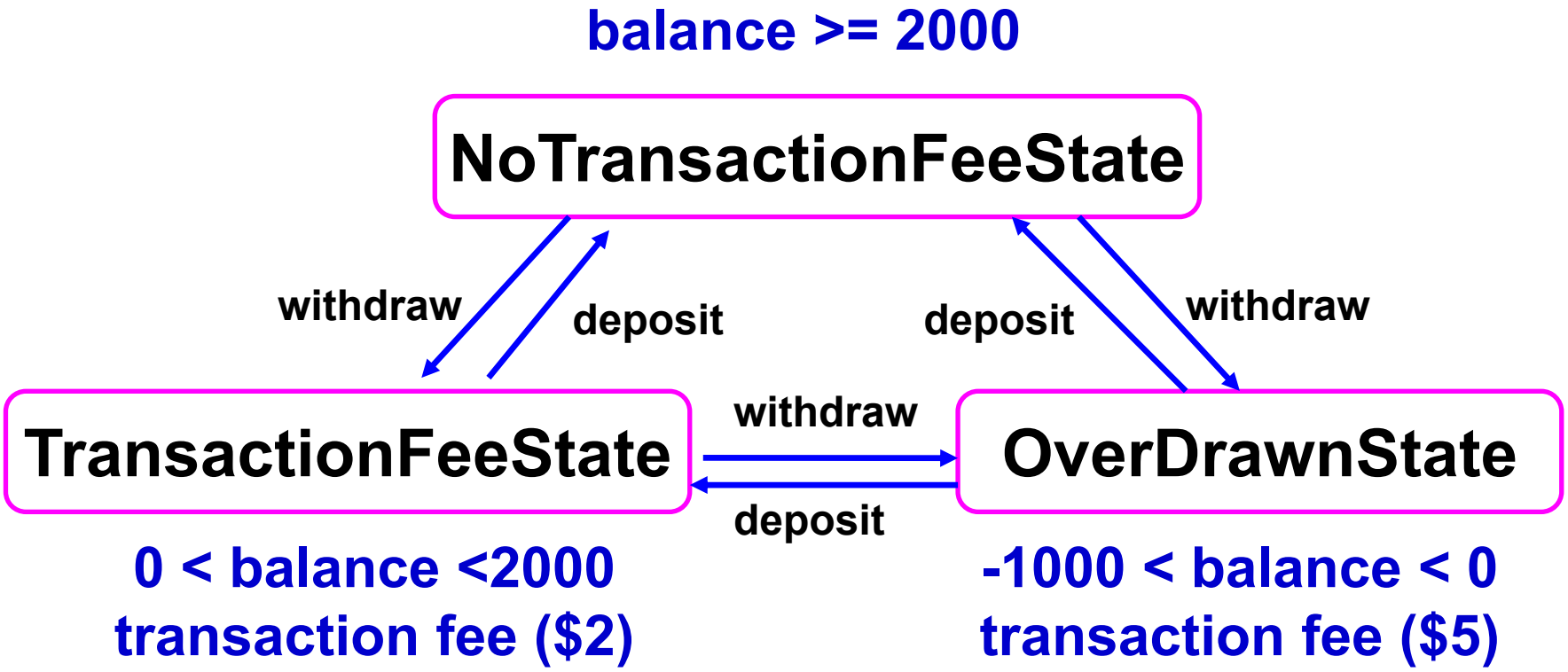
Design Example Using the State Pattern

【例3】银行业务问题。

Consider a business account at a bank. Such an account can exist in any one of the following three states at any given point of time:

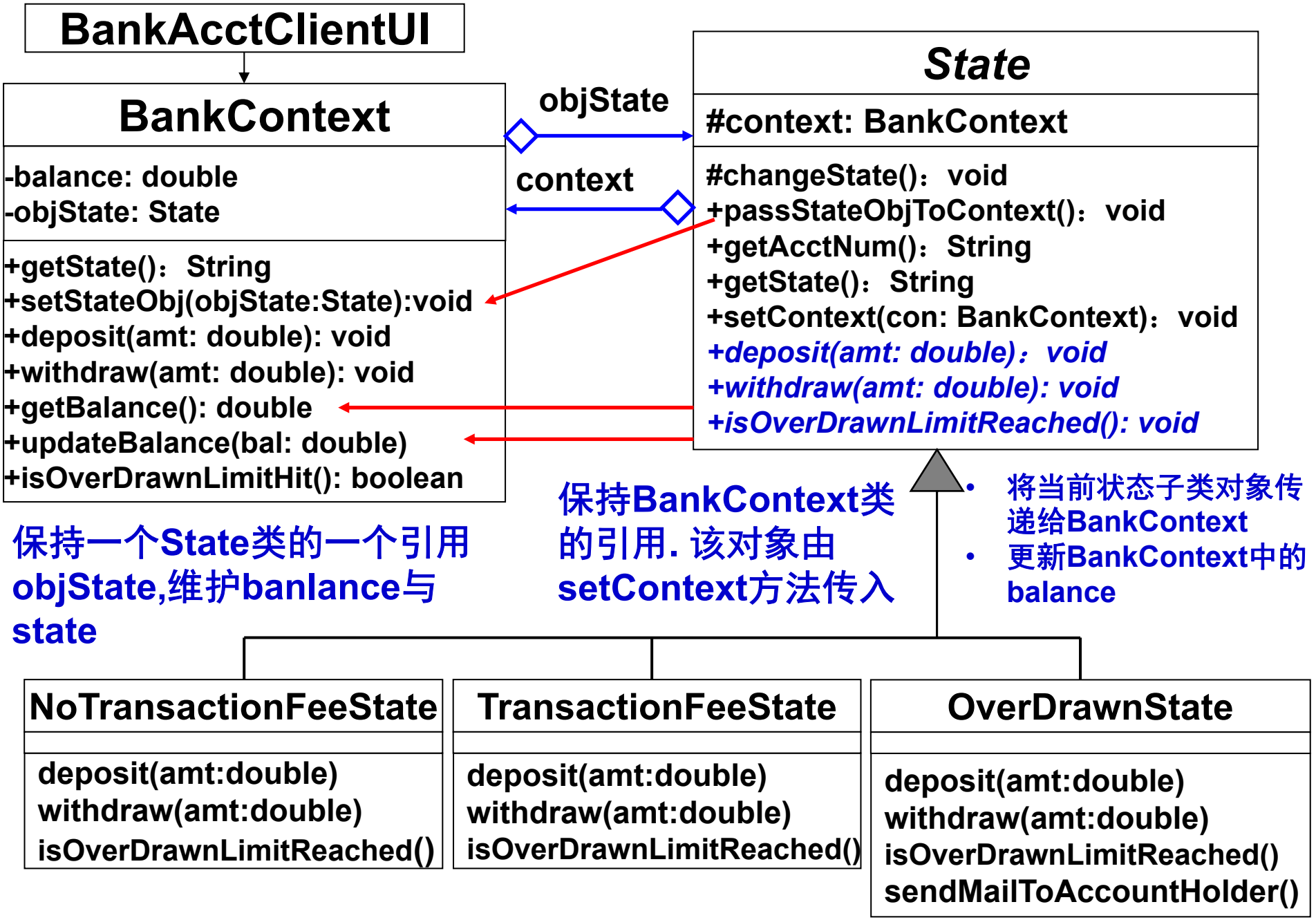
1. ***No transaction fee state。***
2. ***Transaction fee state***
3. ***Overdrawn state***

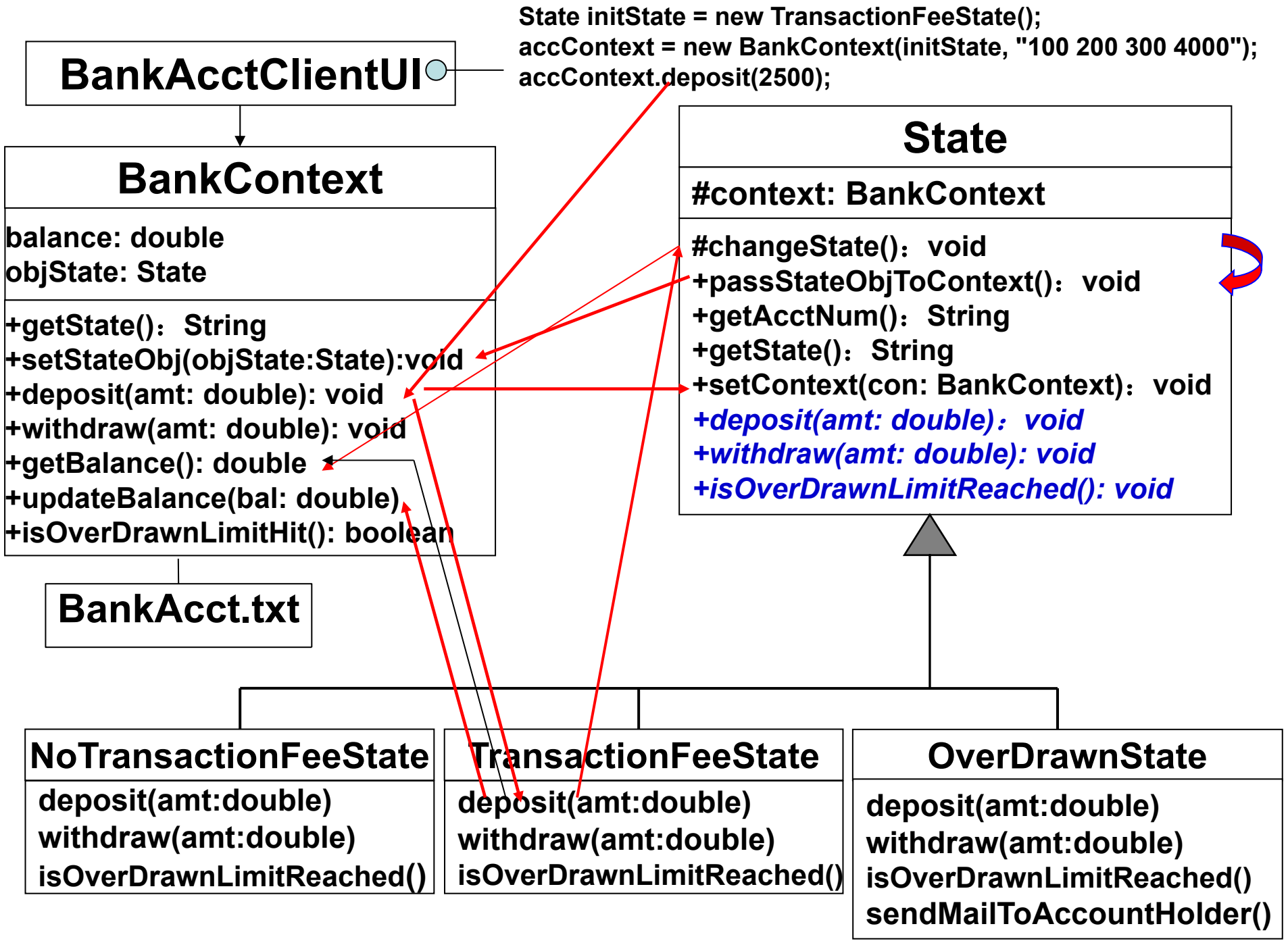
Design Example Using the State Pattern



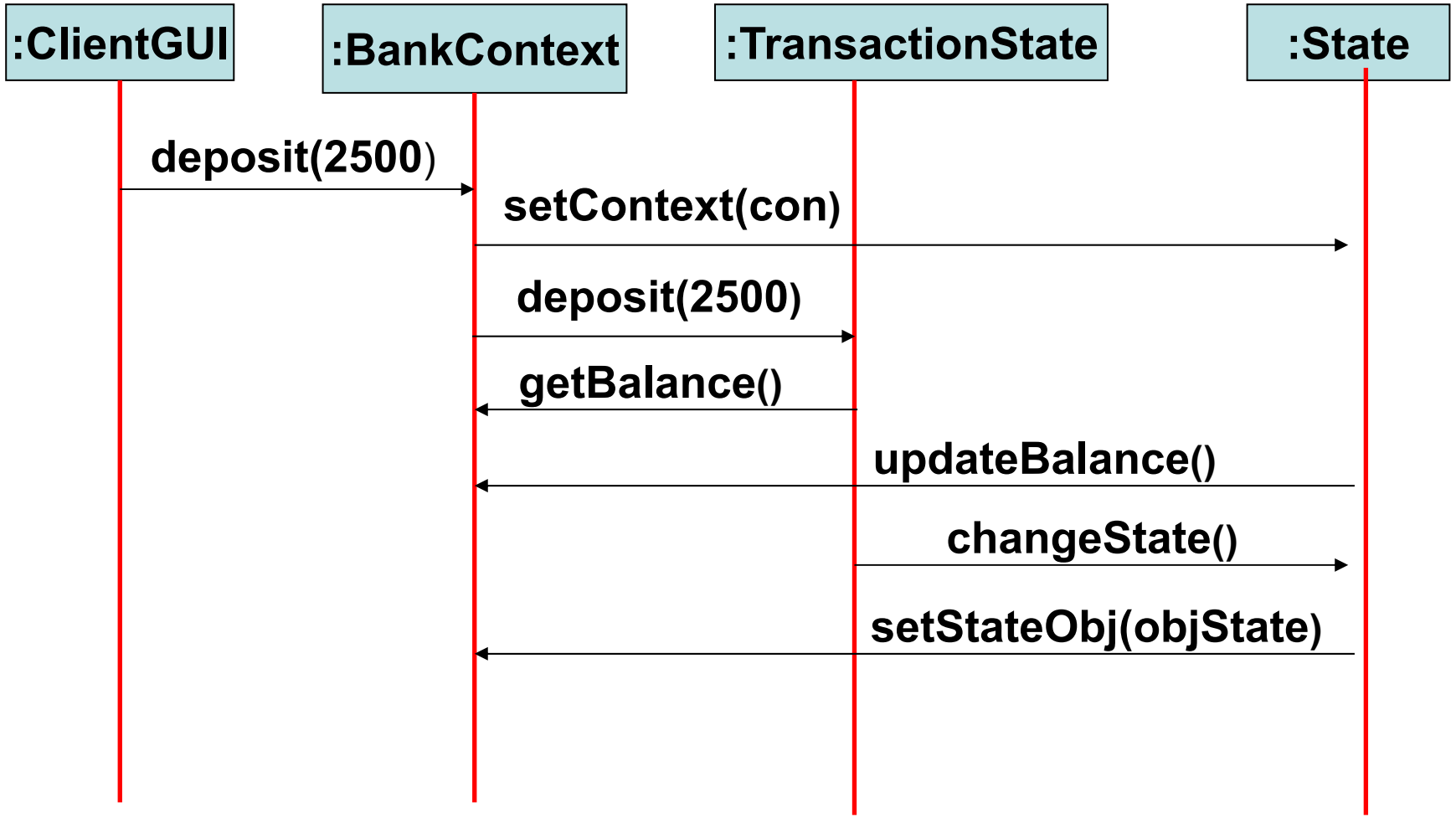
限制：在以上任何状态下，都不允许超过透支上限(-1000)的交易发生。

State transaction diagram


BankAcctClientUI
BankContext
-balance: double
-objState: State
+getState(): String
+setStateObj(objState:State):void
+deposit(amt: double): void
+withdraw(amt: double): void
+getBalance(): double
+updateBalance(bal: double)
+isOverDrawnLimitHit(): boolean
objState
context
State
#context: BankContext
#changeState(): void
+passStateObjToContext(): void
+getAcctNum(): String
+getState(): String
+setContext(con: BankContext): void
+deposit(amt: double): void
+withdraw(amt: double): void
+isOverDrawnLimitReached(): void
保持一个State类的一个引用objState,维护banlance与state
保持BankContext类的引用. 该对象由setContext方法传入
将当前状态子类对象传递给BankContext
更新BankContext中的balance
NoTransactionFeeState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
TransactionFeeState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
OverDrawnState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
sendMailToAccountHolder()


State initState = new TransactionFeeState();
accContext = new BankContext(initState, "100 200 300 4000");
accContext.deposit(2500);
BankAcctClientUI
BankContext
balance: double
objState: State
+getState(): String
+setStateObj(objState:State):void
+deposit(amt: double): void
+withdraw(amt: double): void
+getBalance(): double
+updateBalance(bal: double)
+isOverDrawnLimitHit(): boolean
BankAcct.txt
State
#context: BankContext
#changeState(): void
+passStateObjToContext(): void
+getAcctNum(): String
+getState(): String
+setContext(con: BankContext): void
+deposit(amt: double): void
+withdraw(amt: double): void
+isOverDrawnLimitReached(): void
NoTransactionFeeState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
TransactionFeeState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
OverDrawnState
deposit(amt:double)
withdraw(amt:double)
isOverDrawnLimitReached()
sendMailToAccountHolder()

Design Example Using the State Pattern



Sequence diagram for depositing 2500 dollars

Design Example Using the State Pattern

BankContext class

Context类
提供较高层
业务逻辑

1. 保持数据 (Maintains data):
   - balance,
   - account number
   - State object, and
   - the transaction limits
2. 给客户类提供基本的存款与取款方法。Offers the basic methods
   deposit(double amt) and
   withdraw(double amt)

   inside which the same methods inside a state subclass will be called.

Design Example Using the State Pattern

State层次类
提供较低层
业务逻辑

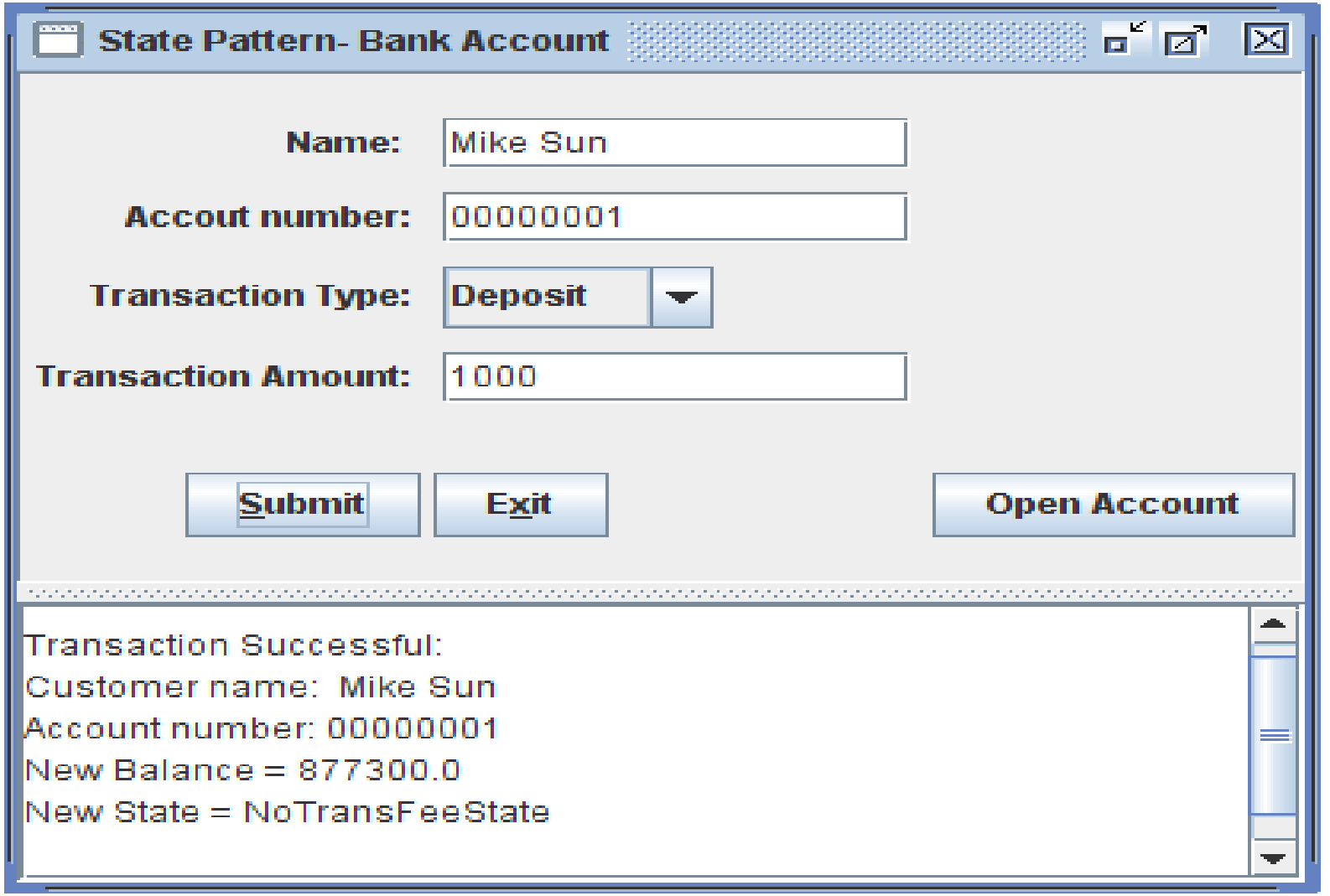
State 类:

- 负责转换状态
- 负责将当前具体状态传递给BankContext类

State 子类:

- 负责存款、取款
- 负责更新BankContext类中的balance

Design Example Using the State Pattern



用户图像界面

Design Example Using the State Pattern

本设计的优点：可扩展性超好

- 因为在客户类与Context类中，都不包含关于与状态有关的条件语句，因此
  - 当要修改某个状态子类的时候，不需要修改客户类与Context类；
  - 当要添加一个新的状态子类的时候，不需要修改客户类与Context类，只须少许修改状态子类的changeState方法

Back

Further discussion of the state design pattern

Comparison of the state design pattern and the strategy pattern

State pattern	Strategy pattern
状态相关的行为 The behavior contained in each State object is specific to a given state of the associated object.	不是状态相关的行为 The behavior contained in each Strategy object is a different algorithm to provide a given functionality.
状态对象可以负责更新配置给Context类的状态对象。 A given State object itself can put the context into a new state. This makes a new State object as the current State object of the context, changing the behavior of the Context object.	由客户类创建策略子类对象，并且传递给Context对象。 A client application using the context needs to explicitly assign a strategy to the context. A Strategy object cannot cause the context to be configured with a different Strategy object.

Comparison of the state design pattern and the strategy pattern

State Pattern	*Strategy Pattern*
状态的选择（转换）依赖于Context或者State对象 The choice of a State object is dependent on the state of the Context object.	由应用类选择策略子类对象 The choice of a Strategy object is based on the application need. Not on the state of the Context object.
有状态转换：可以由Context类或者State层次类进行状态转换	没有状态转换
可以将一些业务逻辑包含在Context类中(包括状态转换)	可以将一些业务逻辑包含在Context类中(没有包括状态转换)

Back