状态模式

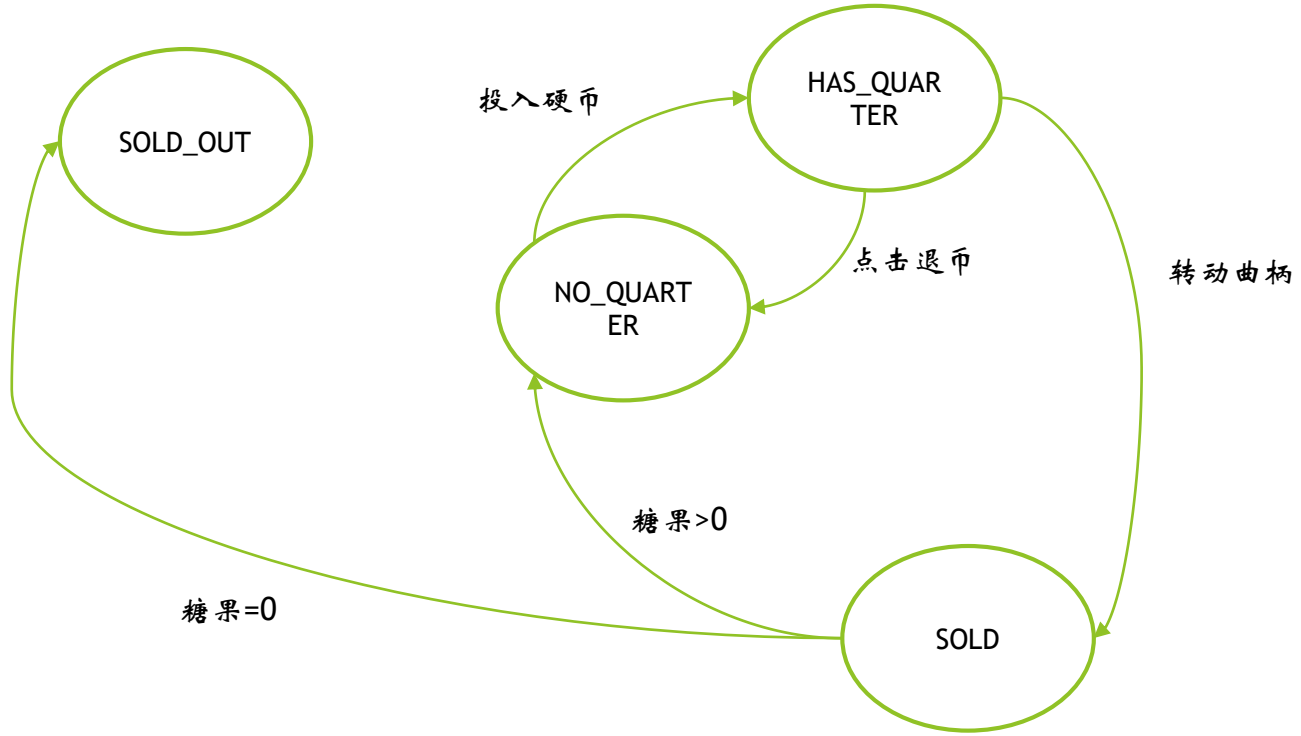
钟业弘

场景

- 一个糖果机程序，它接受硬币，转动曲柄以吐出糖果，这样的一个糖果机有若干状态：
  - 售罄状态（SOLD_OUT）：糖果机里没有糖果了
  - 无硬币状态（NO_QUARTER）：糖果机目前没有接受硬币
  - 有硬币状态（HAS_QUARTER）：糖果机接受了硬币
  - 销售中（SOLD）：糖果机在吐出糖果，这是一个过程而不是一个瞬间，所以需要为其定义一个状态

状态图

- 转动曲柄



传统做法

- 定义一个糖果机类
- 糖果机类里有一个成员变量，标识糖果机的状态
- 为糖果积类定义若干成员函数，以表示用户操作，用户操作有投币，退币，转动曲柄三个
- 不同状态下的糖果机对用户操作的反应是不一样的，所以在每个函数里，需要对糖果机的状态进行判定，即每个函数里都会有很多的if-else语句（或是switch-case）去对状态进行判定以进行相应的操作（要对不同的状态作出不同反应以给用户提示，因为用户可能在任何状态下执行这3个操作）
- 在该例子里，除了3个用户操作，还定义了一个内部函数dispense，用来封装发放糖果的功能，由转动曲柄函数调用。

代码示例

```
public void insertQuarter() {
        if (state == HAS_QUATER) {
                System.out.println(“You can’t insert another quarter”);
        } else if (state == NO_QUATER) {
                state = HAS_QUATER;
                System.out.println(“You inserted a quarter”);
        } else if (state == SOLD_OUT) {
                System.out.println(“You can’t insert a quarter, the machine is sold out”);
        } else if (state == SOLD) {
                System.out.println(“Please wait, we’re already giving you a gumball”);
        }
}
```

代码示例

```
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 ····
·····
```

·····（在代码的逻辑里，这里别的状态不会发生，但在本例子中依然对其进行判断并打印错误信息）

需求变更

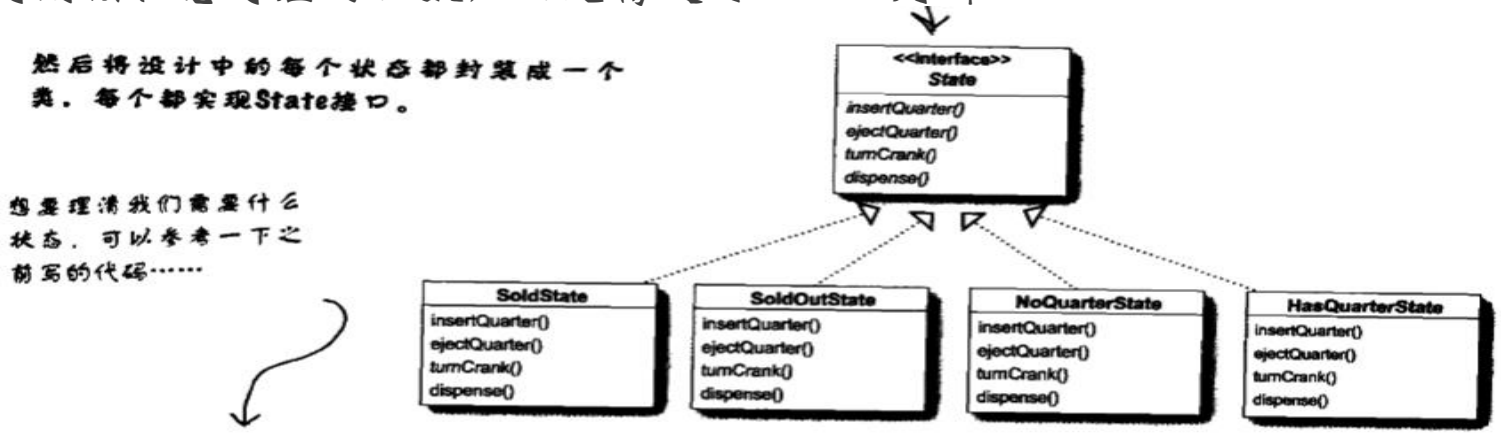
- 售出糖果时，有10%的概率中奖，如果糖果机里有2个以上的糖果，就吐出2个糖果。
- 办法一：修改dispense，在其中加入判断中奖的逻辑。这么做的缺点是没有体现中奖这一状态，这不OO。
- 办法二：增加一个中奖状态，在转动曲柄动作发生时判断是否中奖决定进入普通售出状态（SOLD）或是中奖售出状态（WINNER），在dispense的时候对状态进行判断。这么做的缺点是要为其他函数也增加对WINNER状态的判断，正如前面所说，对不同的状态要有不同反应，为了维持整体设计，应当对其进行判断（哪怕其行为与SOLD状态一致）

设计的缺陷

- 正如上面需求变更所暴露的问题，当需求发生变动时，如增加一个新的状态，则需要对现有的这几个函数都进行修改，加入对新状态的判断及相应逻辑。
- 状态转换被隐藏在条件语句中，使得这不清晰
- 一个实现好的函数在新的需求加入后需要频繁的修改，不遵守开放封闭原则（对扩展开放，对修改封闭）

新的设计

- 定义一个状态接口，接口定义了糖果机所有的操作
- 对每一个状态，实现一个状态类，这个状态类实现状态接口，并加入该状态对应这些操作的逻辑
- 在糖果机类里，定义一个实例变量来存储当前状态，在调用糖果机的函数时，会调用该状态对应的函数，而无需进行if-else判断。



新的设计——状态类

首先我们需要实现State接口。

我们通过构造器得到糖果机的引用，然后将它记录在实例变量中。

```
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't inserted a quarter");
    }

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

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

如果有人投入了25分钱，我们就打印出一条消息，说我们接受了25分钱，然后改变机器的状态到HasQuarterState。

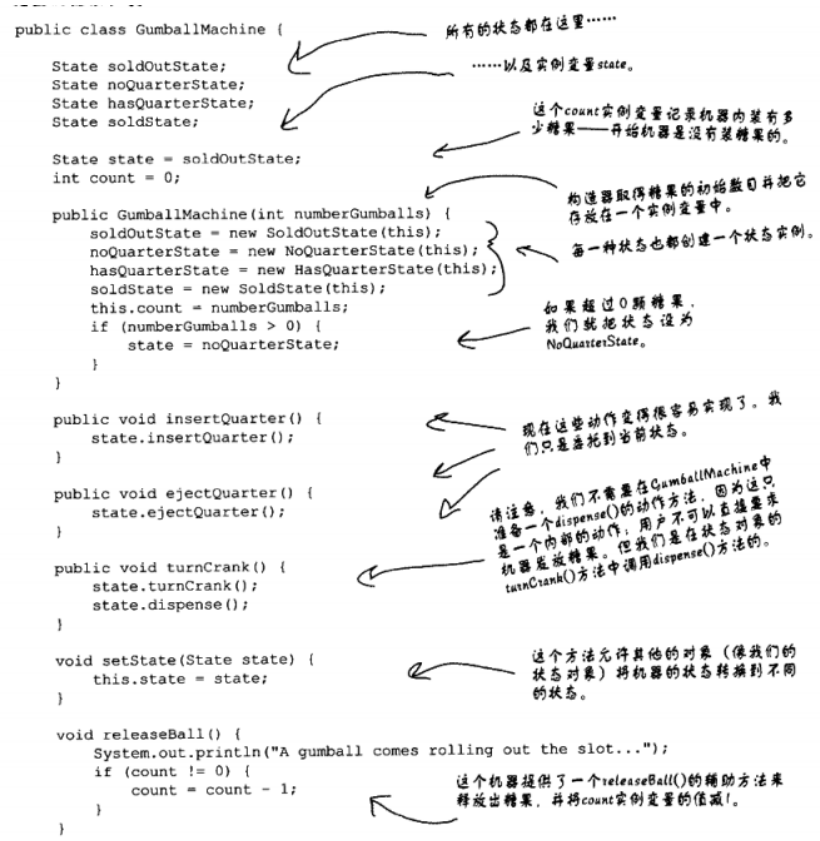
你马上就会看到这是如何工作的。

如果没给钱，就不能要求退钱。

如果没给钱，就不能要求糖果。

如果没得到钱，我们就不能发放糖果。

新的设计——糖果类



```
public class GumballMachine {

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

    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);
        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;
        }
    }
```

所有的状态都在这里……

……以及实例变量state。

这个count实例变量记录机器内装有多少糖果——开始机器是没有装糖果的。

构造器取得糖果的初始数目并把它存放在一个实例变量中。

每一种状态也都创建一个状态实例。

如果超过0颗糖果，我们就把状态设为NoQuarterState。

现在这些动作变得很容易实现了。我们只是委托到当前状态。

请注意，我们不需要在GumballMachine中准备一个dispense()的动作方法，因为这只是一个内部的动作；用户不可以直接要求机器发放糖果。但我们是在状态对象的turnCrank()方法中调用dispense()方法的。

这个方法允许其他的对象（像我们的状态对象）将机器的状态转换到不同的状态。

这个机器提供了一个releaseBall()的辅助方法来释放出糖果，并将count实例变量的值减1。

新的设计——售出状态

```
public class SoldState implements State {
    // 构造器和实例变量在这里

    public void insertQuarter() {
        System.out.println("Please wait, we're 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't 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());
        }
    }
}
```

的动作。

真正的工作在这里。

我们现在是在SoldState状态，也就是说顾客已经付钱了。所以我们首先需要机器发放糖果。

我们问机器糖果的剩余数目是多少，然后将状态转换到NoQuarterState或者SoldOutState。

设计的优点

- 将每个状态的行为局部化到自己的类里
- 将容易产生问题的if语句删除，方便日后维护
- 让每一个状态“对修改关闭”，让糖果机“对拓展开放”
- 创建一个新的类结构，更能映射糖果机的模型

应对需求变更

```
public class GumballMachine {

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

    State state = soldOutState;
    int count = 0;

    // 这里有一些方法
}
```

你需要在这里加进一个新的WinnerState状态，然后在构造器中将它初始化。

别忘了在这里提供一个WinnerState的getter方法。

现在让我们实现WinnerState类本身，其实它很像SoldState类：

```
public class WinnerState implements State {

    // 实例变量和构造器

    // insertQuarter错误信息

    // ejectQuarter错误信息

    // turnCrank错误信息

public void dispense() {
        System.out.println("YOU'RE 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());
            }
        }
```

就跟SoldState一样。

我们在这里释放出两颗糖果，然后进入NoQuarterState 或SoldOutState。

如果还有第二颗糖果的话，我们就把它释放出来。

总结

- 状态模式的应用场景是：当一个对象的内在状态改变时允许改变其行为，这个对象看起来像是改变了其类。
- 状态模式的解决方案是：封装基于状态的行为，并将行为委托于当前状态。