UCAS-大富翁

2019K8009937003

> 面向对象程序设计课程大作业——第三阶段报告

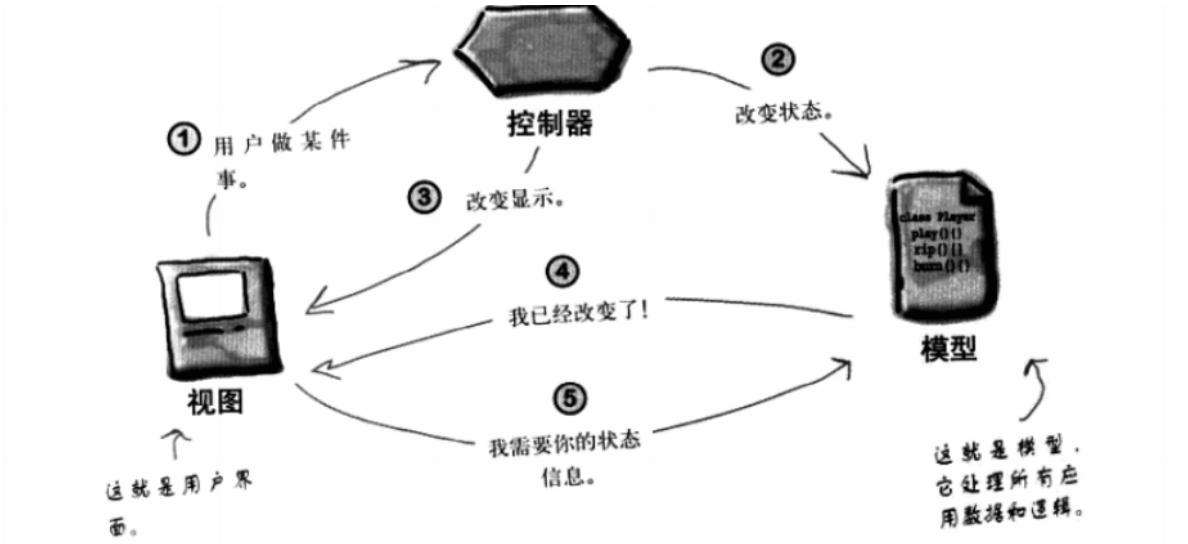
序言

在上一次报告中，笔者已经对整个游戏大致的框架以及拟实现的各个类做了简单说明。在过去一段时间内，笔者主要将前两次报告中介绍的内容转化为了具体的代码，故第三次也是最后一次报告将会针对代码中运用到的设计模式以及体现的面向对象设计原则展开。

整体框架

笔者最终决定采用模型-视图-控制器（MVC）框架模式。之所以在设计中采用该框架，是因为该框架将系统分割为了三个职责明确且不同的子系统：用户界面子系统（视图）、游戏对象子系统（模型）以及游戏功能和管理子系统（控制器）。这种分解的目的是降低系统的复杂性。通过这种分解，可以更容易地划分工作、理解问题所在并修改所需的子系统。与未分解的系统相比，将系统划分为三个子系统增加了系统的一致性并降低了系统的耦合性。

MVC 三部分的功能与关系如下图所示（图源网络）



设计模式

在这一部分中，笔者将针对代码中运用到的设计模式进行简要的分析

1. 观察者模式

观察者模式（又被称为发布-订阅（Publish/Subscribe）模式）属于行为型模式的一种，它定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态变化时，会通知所有的观察者对象，使他们能够自动更新自己。

在MVC框架中，模型大多都使用到了观察者模式，在笔者的设计中，所有模型类都实现了Observer接口，并且在控制器中有一个链表来存放这些所谓的“观察者”们

```java
/**
 * 接口
 * Observer
 */
public interface Observer
{
    void update(long tick);
    void initGame();
}
```

```java
private List<Observer> models = new ArrayList<>();
```

而通知这些“观察者”更新的方法是在游戏内部实现的一个简单计时器，每隔一段时间就通知所有的观察者进行 update 操作，所以这个计时器在其中充当的是“被观察者”的角色

```java
/**
* 游戏计时器
*/
Timer gameTimer = new Timer();
gameTimer.schedule(new TimerTask()
{
    public void run()
    {
        tick++;
        // 通知各对象更新
        for (Observer temp : models)
        {
            temp.update(tick);
        }
        // 更新视图
        mainPanel.repaint();
        }
    }, 0, (1000 / rate)); // 一秒除以帧率即为刷新间隔
}
```

“观察者”其中之一——PlayModel 更新方法

```java
public void update(long tick)
{
    this.curTick = tick;
    if (this.startTick < this.curTick && this.nextTick >= this.curTick)
    // 玩家处于移动状态，需要移动玩家图标且根据是否移动完毕触发经过或停留建筑事件
    {
        this.controller.movePlayer();
        if (this.nextTick != this.curTick)
        {
            this.controller.passBuilding();
        }
        if (this.nextTick == this.curTick)
        {
            this.controller.playerStopJudge();
        }
    }
}
```

2. 组合模式

组合模式（又被称为部分整体模式），是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。

由于笔者的用户图形界面主要是采用 java swing 和 awt 库实现的，而这两个库是组合模式在 java 类库中的一个典型实际应用，所以设计中主要在视图端运用了组合模式。下面以游戏初始数据设置界面为例简单分析一下组合模式的运用。

首先定义四个构件，分别用来获取玩家一的角色确认信息以及名字确认信息

```
private final JButton JBPlayer1 = new JButton("玩家1确认角色");
private final JLabel JLPlayerName1 = new JLabel("名字:");
private final JTextField JTFPlayerName1 = new JTextField(12);
private final JButton JBPlayerName1 = new JButton("玩家1确认名字");
```

然后将它们添加到一个 JPanel 中

```
JPanel jp;
jp.add(JBPlayer1);
jp.add(JLPlayerName1);
jp.add(JTFPlayerName1);
jp.add(JBPlayerName1);
```

这里调用的是 awt.Container 中的 add 方法，JPanel 作为一个容器，他是 awt.Container 的一个间接子类，可以容纳其他组件，

容器父类 Container的部分代码如下

```
public class Container extends Component
{
    private java.util.List<Component> component = new ArrayList<>();

    public Component add(Component comp)
    {
        addImpl(comp, null, -1);
        return comp;
    }
}
```

可以看到， Container 内部又定义了一个集合用于存储 Component 对象，容器组件 Container 和组件如 JButton、JLabel、JTextField（swing）等都是 Component 的子类，而 Component 类中封装了组件通用的方法和属性，如图形的组件对象、大小、显示位置、边界、可见性等，这些子类都继承了它们，所以可以很清楚的看到这里应用了组合模式。

设计原则

（简要提一下两个比较好理解也比较好遵循的原则）

1. 单一职责原则

提到单一职责原则，其实很多人不知不觉都会在设计中遵循这一原则，因为这是一个常识，在软件编程中，谁也不希望因为修改了一个功能导致其他的功能发生故障，而避免出现这一问题的方法就是遵循单一职责原则。当然，我们也可以从不同的维度来看待这一原则。

从小的方面来看，笔者的设计中针对所有类型的建筑和卡牌都分别单独设计了一个类，即使是像监狱和医院这两种功能相近但又存在些许不同的建筑，它们分别记录各自的数据信息并且向外提供访问它们的方法；而从大的层次上来说，在设计中采用的 MVC 框架其实也很好地遵循了这一原则，因为它实现了游戏当中的逻辑和界面也就是模型和视图的分离。

2. 开闭原则

开闭原则是面向对象设计中最基础的设计原则，它指导我们如何建立稳定灵活的系统。但这个原则的定义比较模糊，它只告诉我们对扩展开放，对修改关闭，可是到底如何才能做到对扩展开放，对修改关闭，并没有明确的告诉我们。

笔者在设计卡牌系统的过程中，每添加一张新的卡片就测试一次，而且添加新的卡片只需要增加一个新的卡片类并且增加其相应的处理方法即可，不需要修改原有的模块，这里很大一部分的原因是因为笔者将卡牌类的共性抽象了出来，新增加的类只需要继承这一抽象类，并实现其抽象方法即可，这在一定程度上也体现了设计遵循的开闭原则。