**大炉石-魔兽英雄传**

**系统设计说明书**

**版本:** V2.0

**发行日期:2018/1/12**

**作者：周盈坤、王重熙、杨丰远**

**修改情况一览表**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本  Version | 出版日期  Issue Date | 修订章节  Section  Changed | 修订原因  Reason for Issue | 修订者  Modifier |
| 1.0 | 2017/12/14 |  |  |  |
| 2.0 | 2018/1/12 | 第4章 | 新增我们的设计模式 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Catalog

**目录**

[1 Introduction 简介 5](#_Toc503509035)

[1.1 Purpose 目的 5](#_Toc503509036)

[1.2 Scope 范围 5](#_Toc503509037)

[1.2.1 Name 桌游名称 5](#_Toc503509038)

[1.2.2 Functions 桌游功能 5](#_Toc503509039)

[1.2.3 Applications桌游应用 5](#_Toc503509040)

[2 System Design系统总体设计 5](#_Toc503509041)

[2.1 Interaction with users玩家与桌游的交互 5](#_Toc503509042)

[2.2 Design Considerations设计思路 5](#_Toc503509043)

[2.2.1 Design Alternatives 设计可选方案 5](#_Toc503509044)

[2.2.2 Design Constraints 设计约束 5](#_Toc503509045)

[2.2.3 Other Design Considerations 其他设计考虑 6](#_Toc503509046)

[2.3 System Architecture系统结构 6](#_Toc503509047)

[2.3.1 Description of the Architecture系统结构描述 6](#_Toc503509048)

[2.3.2 Representation of the Game Flow游戏流程说明 7](#_Toc503509049)

[2.3.3 Game Part Timing Diagram游戏部分时序图 7](#_Toc503509050)

[2.4 Decomposition Description分解描述 12](#_Toc503509051)

[2.4.1 GamePlayer package Description玩家用户包描述 12](#_Toc503509052)

[2.4.2 Resource Library package Description资料库包描述 13](#_Toc503509053)

[2.4.3 System Manager package Description系统管理包描述 15](#_Toc503509054)

[2.5 Dependency Description依赖性描述 16](#_Toc503509055)

[3 Class Description类详细描述 16](#_Toc503509056)

[3.1 System\_package系统包中的类 16](#_Toc503509057)

[3.1.1 Player\_manager玩家管理器 16](#_Toc503509058)

[3.1.2 Round\_schedular回合调度器 17](#_Toc503509059)

[3.1.3 Fate\_manager命运/机会产生器 17](#_Toc503509060)

[3.1.4 magic\_art\_manager法术/英雄技能施展器 18](#_Toc503509061)

[3.1.5 Diamond\_distributor费用（钻石）分配器 18](#_Toc503509062)

[3.2 gameplayer\_package玩家包中的类 19](#_Toc503509063)

[3.2.1 game\_player玩家 19](#_Toc503509064)

[3.2.2 Card\_heap牌堆 20](#_Toc503509065)

[3.2.3 Attendant随从 21](#_Toc503509066)

[3.3 resource\_package资源库中的类 22](#_Toc503509067)

[3.3.1 Wheel\_of\_Fate命运之轮 22](#_Toc503509068)

[3.3.2 Map地图 22](#_Toc503509069)

[3.3.3 Place地点 22](#_Toc503509070)

[3.3.4 Card卡牌 23](#_Toc503509071)

[4 Design Patterns 设计模式 24](#_Toc503509072)

[4.1 The introduction of design pattern 设计模式的引入 24](#_Toc503509073)

[4.1.1 Problem 1 难题1——系统类的实例化 24](#_Toc503509074)

[4.1.2 Problem 2 难题2——卡牌很多，扩展性要求高 24](#_Toc503509075)

[4.1.3 Problem 3 难题3——手牌中特殊效果达成判断 24](#_Toc503509076)

[4.2 Singleton Pattern 单例模式 24](#_Toc503509077)

[4.2.1 Introduction and characteristics 引入与特性 24](#_Toc503509078)

[4.2.2 Our example 我们的实例 25](#_Toc503509079)

[4.2.3 Should be conservatively used 应该保守地使用 25](#_Toc503509080)

[4.2.4 Remedy 补救措施——引入新的设计模式 26](#_Toc503509081)

[4.3 Type Object Pattern 类型对象模式 26](#_Toc503509082)

[4.3.1 Introduction and characteristics 引入与特性 26](#_Toc503509083)

[4.3.2 Our example我们的实例 27](#_Toc503509084)

[4.3.3 A class that produces the class 一个产生类的类 28](#_Toc503509085)

[4.3.4 When to use 何时用 28](#_Toc503509086)

[4.4 Observer Pattern 观察者模式 28](#_Toc503509087)

[4.4.1 Introduction and characteristics 引入与特性 28](#_Toc503509088)

[4.4.2 Our example如何工作 29](#_Toc503509089)

[4.5 Other possible patterns 其他可用的设计模式 29](#_Toc503509090)

# Introduction 简介

## Purpose 目的

本文档是规定大炉石魔兽英雄传桌游系统项目的概要设计说明书，描述了系统的总体设计、总体结构模块和接口。目的是为该系统的设计、开发和测试提供指导。

读者：设计人员、开发人员

## Scope 范围

### Name 桌游名称

大炉石魔兽英雄传

### Functions 桌游功能

融合炉石和大富翁中被广大玩家所喜欢接受的元素，是玩家休闲娱乐的上佳选择。

具体参考大炉石魔兽英雄传-需求规格说明书v1.0

### Applications桌游应用

老少皆宜，适用于四个玩家组成的group在闲暇时光里消遣娱乐

# System Design系统总体设计

## Interaction with users玩家与桌游的交互

本款游戏采用桌游常见的轮和回合制，四个玩家在每一轮中都有自己的回合，也即四个

玩家的回合组成了一轮。玩家首先点击确认进入游戏，系统会提示玩家进行相应的初始化，如选择英雄，组件牌库。正式游戏时，每一轮玩家在自己的回合做出相应的操作。最后当桌上仅剩下一名玩家的时候，该名玩家成为胜者，同时游戏结束，具体的游戏流程以及玩家操作详见2.3.2游戏流程说明

## Design Considerations设计思路

### Design Alternatives 设计可选方案

这里我们就采用轮和回合制+server端、client端模式

### Design Constraints 设计约束

#### Standards compliance 遵循标准

桌游最重要的就是讲求不同玩家的制衡关系，事实上炉石和大富翁之所以风靡全球就是

因为其不同玩家之间的平衡关系设计的好，所以本款游戏也将依次为约束来设计自己的游戏。

#### Technology Limitations 技术限制

桌游的类和client server端均使用Java来实现，为此还要遵循java编程规范及其开发标准来设计该款桌游。

### Other Design Considerations 其他设计考虑

每一个好的设计都是要求高内聚低耦合的。在我们的这款游戏中也是如此，设计应该考虑到类之间的易于重构，为此我们需要一个自顶向下的设计思路，逐层封装。详见2.3，2.4，2.5以及整个第三部分类的描述。

## System Architecture系统结构

### Description of the Architecture系统结构描述



### Representation of the Game Flow游戏流程说明



### Game Part Timing Diagram游戏部分时序图



游戏阶段前的初始交互

玩家、英雄、卡牌、费用的交互



玩家、命运之轮、回合调度的交互



玩家英雄与地狱的交互



玩家英雄与机会的交互



玩家英雄与命运的交互



玩家英雄与空地的交互



玩家英雄与自领地的交互

入侵英雄与敌方领地以及敌方驻地随从的交互

## Decomposition Description分解描述

### GamePlayer package Description玩家用户包描述



以上是玩家的整个package，其中包含了与玩家有关的一些类——玩家类、牌堆类（抽象父类）、玩家牌库类和玩家手牌类（子类）、随从类（父类）、贴身随从类和驻地随从类（子类）以及驻地随从地点相关方法（接口）。

其中每一个类的具体属性方法描述见第3节，在这里主要描写该包中的类之间的依赖关系，如下：

玩家类（game\_player）是玩家这个包中最重要的一个类，也是主要与其他包交互过程中的轴心点。在玩家包中玩家类（game\_player）和牌堆类（Card\_heap）的关系是组合（复合）关系，因为玩家这个类中会拥有两个牌堆，分别是玩家牌库和玩家手牌，并且玩家生命周期结束后牌堆也不复存在，故他们之间是较强的组合关系。而玩家类（game\_player）和随从类（Attendant）之间的关系是聚合关系，因为在我们的设计中玩家和随从是可以单独存在，没有较强的依赖关系。

牌堆类（Card\_heap）其实是个抽象类，它的方法由斜体标示，表示方法的具体实现是有子类实现的。它的功能由它的两个子类玩家牌库（Card\_base）和玩家手牌（hand\_cards）继承，他们之间的关系是继承关系。

随从类（attendant）是一个父类，它的功能也由两个子类继承，分别是贴身随从和驻地随从，其中驻地随从多出了驻地地点的信息，所以加了一个叫做驻地随从地点相关方法的接口，而驻地随从中的读取和设置驻地随从位置的方法实现该接口。故他们之间是实现关系。

### Resource Library package Description资料库包描述

资源库包我们分了两个相对独立的部分：



以上是资源库的第一部分，它包含了与卡牌有关的一些类——卡牌类（最顶层的父类）、随从牌类和法术牌类（父类）、随从牌i号（子类）、法术牌i号（子类）、随从牌信息接口（接口）。

卡牌类是个抽象类，方法由斜体标示，表示需由子类具体实现。它由两个次级的父类继承，一个是随从牌类一个是法术牌类，其中随从牌类中实现了随从牌信息接口（接口）。故随从牌类跟接口的关系自然是实现，而随从牌和法术牌和卡牌的关系是继承关系。

我们的设计中简便期间具体设计了4个随从牌类型，分别为随从牌i号（i=1,2,3,4），然后随从牌i号这些类便是子类了，和随从牌是继承关系。法术牌也一样，我们目前设计中也是4个子类，法术牌i号（i=1,2,3,4）。



以上是资源库包的第二部分，主要包含英雄职业类、命运之轮类、地图类、地点类（父类）、地狱/命运/机会/空地/领地类（子类）、领地增减随从接口（接口）。

命运之轮类（Wheel\_of\_Fate）主要是归玩家调用，产生随机数，用于决定玩家次序，决定玩家行走几步等。

地图类（Map）保存当前地图上的各个地点状态，并且提供一些方法给系统包中的回合调度器，如getplace\_type就用于在到达某个地点时访问地图的该地点，看是什么状态。地图类（map）和地点类（place）是组合关系（也称复合关系），因为地点不能单独于地图存在。

地点类（place）是父类，其子类有五个，其中比较特殊的就是领地这个子类，因为一个地点成为领地需要记录该领地上有哪些驻地随从，所以我们设计了这样一个接口——领地增减随从接口，故领地类和该接口是实现关系。

### System Manager package Description系统管理包描述



以上是系统包，主要包含了系统调度有关的类——玩家管理器、回合调度器、机会/命运产生器、法术/英雄技能施展器、钻石（费用）分配器。

该包中的核心是回合调度器（Round\_schedular），回合调度器中实现回合调度，和用户进行交互，包括给用户提供现在玩家的位置，地图信息，转到的命运之轮的点数，目前手牌信息，然后实现包括玩家要使用哪张卡牌，决策哪个角色承受伤害等。这些操作都是由回合调度器去调用玩家、英雄、地图的相关方法实现的。然后机会/命运产生器、法术/英雄技能施展器、钻石（费用）分配器就辅助完成这些功能。

玩家管理器类（Player\_manager）也是在其中和用户进行交互，然后调用玩家类（game\_player）的方法完成对玩家的初始化等操作。

## Dependency Description依赖性描述

因为我们设计准则其中一条就是尽量保证高内聚低耦合，故我们尽量保证了关联关系都出现在包内。当然包与包之间肯定是有关联的，那么这些关联就在于——

首先系统包和玩家包之间：这里有必有提一下，就是玩家包中的类不应该知道系统类的存在才对，所以我们的设计就是统一由系统类调用玩家类中的方法。回到包与包之间上的依赖性描述，主要是玩家管理器和玩家的直接关联关系，玩家管理器中会记录有多少个玩家，并且调用玩家中的方法对玩家进行初始化；另外回合调度器中很多地方都会引用玩家、英雄等类，他们之间有直接关联关系。

然后是系统包和资源库包之间：依然是刚刚的准则，资源包中的类不应知道系统类的存在，故他们之间的关系也是主要体现在回合调度器和地图的直接关联关系上。

最后是玩家包和资源库包之间：玩家类要转动资源库类中的命运之轮，是关联关系；然后卡牌和牌堆的关系显而易见是聚合关系，随从牌和随从之间有关联关系。主要就以上几种。

# Class Description类详细描述

## System\_package系统包中的类

### Player\_manager玩家管理器



属性：

player\_num：现在已经就位的玩家数量

方法：

getuser\_num()：获得已经就位的玩家数量

userEnter(String player\_name): 玩家进入游戏

player\_prepare()：和用户交互实现玩家信息初始化

### Round\_schedular回合调度器



属性：

order\_array:int[]：4个玩家的回合进行先后次序

方法：

determine\_order()：决定玩家回合进行的先后次序

start\_round()：正式开始游戏，4个玩家按顺序轮流执行自己的回合

go\_to\_hell(player cur\_player)：让玩家进地狱

call\_fate(player)：对抽到命运牌的玩家执行相应命运效果

call\_chance(player) ：对抽到机会牌的玩家执行相应机会效果

### Fate\_manager命运/机会产生器



属性：无

方法：

random\_do\_a\_chance(player)：随机产生一个机会或命运，并对玩家释放。

### magic\_art\_manager法术/英雄技能施展器



属性：无

方法：

do\_magic(player,type)：根据传入的法术牌类型对玩家释放法术

do\_heropower(hero\_t,type)：根据传入的英雄类型释放英雄技能

### Diamond\_distributor费用（钻石）分配器



属性：无

方法：

distribute\_diamond(player)：由回合调度器调用该方法给指定玩家分配响应的费用。

## gameplayer\_package玩家包中的类

### game\_player玩家



属性：

player\_name:String：玩家昵称

hero：玩家的英雄类型

position：玩家目前的位置

circle\_num：玩家已经走过的圈数

diamond：玩家现在拥有的钻石数

card\_base：玩家的牌库

hand\_card：玩家的手牌

Attendant\_array：玩家拥有的随从们

方法：

setname(String name)：设置玩家姓名

getname()：获取玩家姓名

setcard\_base(Card\_heap)：设置玩家牌库

sethand\_card(Card\_heap)：设置玩家手牌

sethero(read\_hero)：设置玩家英雄

gethero()：获取玩家英雄类型

setcircle(int cir\_num)：设置圈数

getposition()：获取玩家位置

setdiamond(int diamond)：设置玩家当前费用值

build\_cardbase()：组建玩家牌库

deal\_one\_card()：从牌库抽一张牌

use\_one\_card(int index)：使用手牌中index对应的牌

rotate\_wheel\_of\_fate()：转动命运之轮

### Card\_heap牌堆



属性：

card\_heap:Card[]：牌堆中有哪些牌

valid[]：表示数组某个位置上的牌是否有效

方法：

add(Card card)：向牌堆中增加一张牌

delete(int index)：从牌堆中删除一张牌

### Attendant随从



属性：

type：随从类型

blood：随从血量

ATK：攻击力

方法：

getblood()：获取生命值

setblood(int)：设置生命值

getATK()：获取攻击力

setATK(int)：设置攻击力

接口：（又驻地随从类实现）

getposition()：获取驻地随从的位置

setposition(int position)：设置驻地随从的位置

## resource\_package资源库中的类

### Wheel\_of\_Fate命运之轮



属性：

max\_point：最大点数

min\_point：最小点数

方法：

rotate()：转动

### Map地图



属性：

Place\_array：记录每个地点信息的数组

方法：

getplace\_type(int position)：获取指定地点的类型

add\_fixattendant(int pos,Attendant att)：为某个地点添加驻地随从

dele\_fixattendant(int pos,Attendant att) ：为某个地点删除驻地随从

### Place地点



属性：

place\_type：地点类别

hero\_here：现在哪个英雄在这里

方法：无

接口：（由领地这个子类实现）

add\_attendant(Attendant att)：为领地添加驻地随从

dele\_attendant(Attendant att)：为领地删除驻地随从

### Card卡牌



属性：

type：卡牌类型

name：卡牌名

cost：卡牌费用

方法：

gettype()：获取该卡牌的类型

getname()：获取该开拍名称

getcost()：获取卡牌所需费用

接口：（由随从牌类实现）

getattendantINFO()：获取该随从牌对应的随从信息

# Design Patterns 设计模式

## The introduction of design pattern 设计模式的引入

### Problem 1 难题1——系统类的实例化

关于系统类的实例化。像系统类这种应当是有且仅有一次实例化的类应该如何实例化，怎样才能保证它只实例化一次。顺着这个思路我们发现可以选用设计模式中的单例模式来保证正确性。

### Problem 2 难题2——卡牌很多，扩展性要求高

我们的设计延续炉石多卡牌的形式，需要设计的卡牌很多，而且还想像炉石那样，不定期更新加入新的扩展包，因此对卡牌的扩展性要求极高。我们不想每一个卡牌单独都要修改，然后重新编译，故我们选择引入类型对象模式来解决这个问题。

### Problem 3 难题3——手牌中特殊效果达成判断

我们的游戏中常有一个类需要和另外的类交互，当另外一个类达成某个条件时需要同时触发该类的一些状态更新，就比如手牌中特殊效果达成判断（成就达成）。这类问题很明显可以引入观察者模式来解决。

## Singleton Pattern 单例模式

### Introduction and characteristics 引入与特性

我们会有这样的时候，当一个类有多个实例的时候，就会出错。在上一个阶段也就是系统设计阶段，我们曾着手用java实现我们的游戏，当时我们就遇到了这样一个问题——我们的系统类的实例化必须有且只有一次才对，不然肯定有问题，就比如这里提到的玩家管理器，我们进行一场游戏不能有两个玩家管理器。我们当时就了解了一下单例模式，只要将系统类按单例模式设计，就能限制其只能实例化一次。单例模式还有另一个特性，那就是提供一个全局的访问指针，我们的理解下比如我们设计中的命运之轮，系统类也可以去调用来决定玩家的先后次序，然后每个玩家实例都会去调用命运之轮以决定我下面要走几步。

特性：

1、限制一个类只有一个实例

2、提供一个全局的访问指针

### Our example 我们的实例



这就是我们的玩家管理器类，我们使用饿汉式单例类来实现的，即在类初始化时已经自行初始化好了，对比懒汉式单例是在第一次调用的时候实例化自己。毕竟本游戏肯定得有玩家管理器，早晚也得实例化，我们就用的饿汉式单例类。

### Should be conservatively used 应该保守地使用

由以上我们确实能总结出为什么使用单例模式的种种理由：

* 我们的系统包中的类可以“方便”用在我们想用的地方
* 不用的时候可以不去创建（懒汉式单例）
* 它可以是在运行时初始化的，一直等到它所需要的信息在运行时才准备好。

但我们这里也想提一句，其应该保守地使用。除了崇高的理想，GOF在书中对Singleton模式的描述，弊大于利。为什么这么说呢？

第一，它是全局变量，不好调试，想一下如果我们在一个别人写的函数里找一个bug。如果这个函数没有用全局变量，我们可以通过理解函数体和传进的参数来推测问题的来源。但是，若里边有全局变量，为了弄明白发生了什么，我们必须翻遍代码库去找什么与这个全局变量相关。当你在凌晨三点，在上百万代码中找一个导致静态变量值错误的调用时，你会恨死全局变量。

第二，它会导致紧耦合。我们面向对象追求的是高内聚低耦合，而全局变量的引入，似乎就意味着耦合度的提高？

第三，对并发性不友好。游戏运行于单核设备的年代差不多结束了。今天的代码至少要运行在多线程下，即时它并没有得到并发的所有优势。当我们让一些代码全局可见，我们就创建了一些所有线程都可以看到并且能修改的内存，无论他们是否知道有别的线程也在做同样的事情。这会导致死锁，竞争，和其他一些很难修复的线程同步bug。

### Remedy 补救措施——引入新的设计模式

在上一点中我们明确提到我们是会后悔使用单例模式的，那么我们用接着引入什么样的设计模式来避免这些单例模式所带来的问题呢？

总所周知，不使用全局变量架构的游戏？很难实现。一个通用的准则是我们希望变量的作用域越小越好。对象的作用域越小，我们处理它的时候，需要关注的地方就越少。

我们需要引入一些新的设计模式：

* Service Loctor模式为一些对象提供全局访问入口， 使得对象全局可见，但是它让人更灵活的配置这个对象。
* Subclass Sandbox模式提供了访问一个类实例的属性，而又不需要把他做成全局变量。

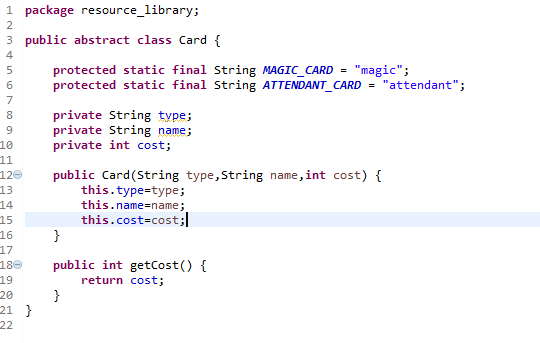
## Type Object Pattern 类型对象模式

### Introduction and characteristics 引入与特性

有一点问题也是我们设计中头大的部分，就卡牌而言，如果我们的游戏也想做出像炉石传说那样数以百计的卡牌，并且还要不定时上线新的卡牌包以刺激消费。设计中若没有很强的扩展性必然是不行的。故我们这里就引入一种新的设计模式，名叫类型对象模式。

它的特性就是允许用一个类去创建一些新“类”，每一个新“类”产生的实例都代表了一种对象。或者说我们不需要重新编译代码就可以定义新的卡牌类型。

### Our example我们的实例





这是我们卡牌父类的类设计初稿。

接着刚刚说的，大家可能会问，要实现许多不同种类的卡牌直接使用继承的方式不就好了么。诚然，继承方式确实很好用，定义一个父类叫做Card，然后不同子类继承Card，并且覆盖掉父类的比如说GetCost方法。这样看起来一切顺利，继承出十几个，几十个不同的卡牌都没问题，但是如果设计师最后说想要几百种卡牌，然后我们就会发现，我们所有的时间都花在了写那个只有几行代码的子类了。更糟糕的是我们的设计师会频繁得修改那些我们写好的类型。整天要为修改这些小数据忙来忙去，同时不停地重新编译整个工程，真的会很头疼。

### A class that produces the class 一个产生类的类

根据以上的问题，如果能让设计师不通过程序员也能创建和修改怪物种类，那就更好了。于是就到这个设计模式大显身手的时候了，我们可以不去继承Card类，而是把Card和类型分开，变成Card和Breed.



这样就只有两个类，一点继承结构也没有。在这样的系统下，每一个卡牌都是Card类的一个实例，相同种类的卡牌共享一个Breed（类型）实例，Breed维护了这个卡牌的共同属性：所需费用值，攻击力，初始生命值。

为了把卡牌分类，我们让每一个卡牌都维护一个Breed对象的引用，这个对象存储了这种怪物的属性。如果要获取所需费用值，卡牌类只需要调用它的Breed中的方法即可。这个Breed类本质上就代表了卡牌的类型。每一个Breed对象都代表了一种不同的概念类型，因此，这个模式被叫做：Type Object。

正如刚刚所说这个模式的强大之处就是现在我们不需要重新编译代码就可以定义新的怪物类型。本质上，就是把写死的class继承结构变成了运行时的数据结构。

我们可以用不同的数据实例化Breed类，从而创建成百上千的怪物类型。如果我们用配置文件种的数据初始化这些实例，那就是完全通过数据来完成定义工作了，这很简单，设计师足可搞定。

### When to use 何时用

关于这种模式何时用？

如果你需要定义很多种类的东西，但是手动用你的语言一个个写出来很恶心，就可以用这种模式了。在实际应用中，满足一下的任意一个条件就可以：

1、你不知道最终会有多少种类型。（例如炉石卡牌）

2、你不想频繁重新编译或者修改代码，就能修改或者添加种类。

## Observer Pattern 观察者模式

### Introduction and characteristics 引入与特性

关于观察者模式，在游戏中最著名的应用就是成就解锁，在我们的游戏设计中当然也不例外。

我们的游戏每个玩家会有手牌，然后每回合可以花费“费用”使用手牌，而卡牌是有特殊属性的，比如有些牌就有“连击”这样的属性，那么这个“连击”的属性何时触发呢？在玩家同一回合中，若先使用了一张其他的卡牌后再使用具有连击的卡就会触发特殊效果。于是这就涉及到“连击”属性手牌对玩家的一个“监听”。换句话说，就是在玩家使用了一张卡牌之后，立即广播一下，那些具有连击属性的手牌就会自动更新为有效状态了。

由此可见其特性就是：

1、成就达成问题

2、使得观察者和被观察者解耦，如刚刚的例子中，玩家使用一张手牌之后只需要广播一下，而不需要没抽一张手牌，若有“连击”属性马上和玩家使用卡牌的行为解耦。

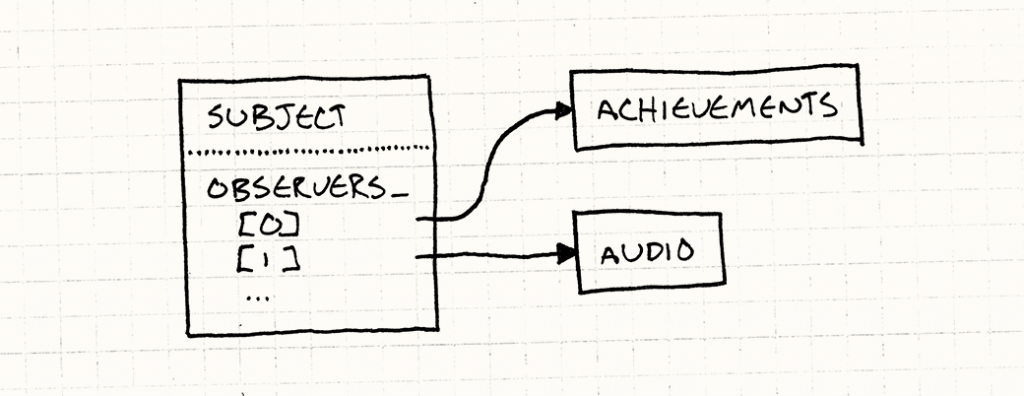
### Our example如何工作

在观察者模式中，观察者就等待主体的通知。就是我们的例子中手牌里那些带“连击”的卡牌等待玩家使用卡牌后的广播。

被观察者维护一个观察者列表，并且有一些暴露在外的公有API，可以去修改这个列表，从而可以控制谁接收通知。对应到我们的例子中就是，由于手牌中牌的更替，当摸到一张新的手牌时若该手牌是“连击”类型的就需要加入到观察者列表中去。

Subject跟Observer沟通但是它不与他们耦合在一起，这一点注意，因为我们的目的本身就是解耦。

每当物理引擎发生了一些重要的事件，都可以像前面的例子那样调用notify()。遍历观察者列表，通知他们。



## Other possible patterns 其他可用的设计模式

还有一些好玩的设计模式也值得借鉴，但是迫于时间的原因，只能在报告的最后一笔带过：

**Flyweight 模式——可以用于我们地图的场景的实现**

**Game Loop模式——可以借鉴其用于我们游戏 循环轮和回合制**