设计模式项目文档

——动物运动会

董恩琦 1853410

崔鑫宇 1753444

章逸飞 1852641

印昱炜 1850170

汪毕强 1852577

侯祖光 1853201

江一舸 1750225

李佳雯 1851351

梁黄思语 1852142

董晓 1853126

完成日期 2020-11-28

目录

[1. 项目简介 5](#_Toc57424118)

[2. Design Pattern 汇总表 6](#_Toc57424119)

[3. 设计模式详述 9](#_Toc57424120)

[3.1 Abstract Factory 9](#_Toc57424121)

[3.1.1 JudgeIngredientFactory实现API 10](#_Toc57424122)

[3.1.1.1 API 描述 10](#_Toc57424123)

[3.1.1.2 类图 11](#_Toc57424124)

[3.1.1.3 相关函数及其作用 11](#_Toc57424125)

[3.2 Adapter 12](#_Toc57424126)

[3.2.1 Pay实现API 12](#_Toc57424127)

[3.2.1.1 API描述 12](#_Toc57424128)

[3.2.1.2 类图 13](#_Toc57424129)

[3.2.1.3 相关函数及其作用 13](#_Toc57424130)

[3.3 Bridge 14](#_Toc57424131)

[3.3.1 EquipmentImplementor实现API 14](#_Toc57424132)

[3.3.1.1 API描述 14](#_Toc57424133)

[3.3.1.2 类图 15](#_Toc57424134)

[3.3.1.3 相关函数及其作用 15](#_Toc57424135)

[3.4 Builder 16](#_Toc57424136)

[3.4.1 GamesBuilder 实现 API 17](#_Toc57424137)

[3.4.1.1 API 描述 17](#_Toc57424138)

[3.4.1.2 类图 18](#_Toc57424139)

[3.4.1.3 相关函数及其作用 18](#_Toc57424140)

[3.5 Chain of Responsibility Pattern 19](#_Toc57424141)

[3.5.1 XXX实现API 19](#_Toc57424142)

[3.5.1.1 API描述 19](#_Toc57424143)

[3.5.1.2 类图（无） 20](#_Toc57424144)

[3.5.1.3 相关函数及其作用 20](#_Toc57424145)

[3.6 Command 20](#_Toc57424146)

[3.6.1 OrderCondiment实现 API 21](#_Toc57424147)

[3.6.1.1 API 描述 21](#_Toc57424148)

[3.6.1.2 类图 22](#_Toc57424149)

[3.6.1.3 相关函数及其作用 22](#_Toc57424150)

[3.7 Composite 23](#_Toc57424151)

[3.7.1 Component 实现API 23](#_Toc57424152)

[3.7.1.1 API描述 23](#_Toc57424153)

[3.7.1.2 类图 24](#_Toc57424154)

[3.7.1.3 相关函数及其作用 24](#_Toc57424155)

[3.8 Decorator / Wrapper 25](#_Toc57424156)

[3.8.1 XXX 实现 API 25](#_Toc57424157)

[3.8.1.1 API 描述 25](#_Toc57424158)

[3.8.1.2 类图 26](#_Toc57424159)

[3.8.1.3 相关函数及其作用 26](#_Toc57424160)

[3.9 Facade 26](#_Toc57424161)

[3.9.1 SignUpFacade 实现 API 27](#_Toc57424162)

[3.9.1.1 API描述 27](#_Toc57424163)

[3.9.1.2 类图 28](#_Toc57424164)

[3.9.1.3 相关函数及其作用 29](#_Toc57424165)

[3.10 Factory Method 29](#_Toc57424166)

[3.10.1 IFactory实现API 29](#_Toc57424167)

[3.10.1.1 API 描述 29](#_Toc57424168)

[3.10.1.2 类图 30](#_Toc57424169)

[3.10.1.3 相关函数及其作用 30](#_Toc57424170)

[3.11 Flyweight Pattern 31](#_Toc57424171)

[3.11.1 ？？？实现API 31](#_Toc57424172)

[3.11.1.1 API 描述 31](#_Toc57424173)

[3.11.1.2 类图 32](#_Toc57424174)

[3.11.1.3 相关函数及其作用 33](#_Toc57424175)

[3.12 Interpreter Pattern 33](#_Toc57424176)

[3.12.1 PriceOperation实现API 34](#_Toc57424177)

[3.12.1.1 API 描述 34](#_Toc57424178)

[3.12.1.2 类图 35](#_Toc57424179)

[3.12.1.3 相关函数及其作用 35](#_Toc57424180)

[3.13 Iterator Pattern 35](#_Toc57424181)

[3.13.1 XXX实现API 36](#_Toc57424182)

[3.13.1.1 API 描述 36](#_Toc57424183)

[3.13.1.2 类图 37](#_Toc57424184)

[3.13.1.3 相关函数及其作用 37](#_Toc57424185)

[3.14 Mediator Pattern 38](#_Toc57424186)

[3.14.1 AthleteAndOthersMediator实现API 38](#_Toc57424187)

[3.14.1.1 API 描述 38](#_Toc57424188)

[3.14.1.2 类图 39](#_Toc57424189)

[3.14.1.3 相关函数及其作用 39](#_Toc57424190)

[3.15 Memento Pattern 40](#_Toc57424191)

[3.15.1 OrderCondiment实现API 41](#_Toc57424192)

[3.15.1.1 API 描述 41](#_Toc57424193)

[3.15.1.2 类图 41](#_Toc57424194)

[3.15.1.3 相关函数及其作用 42](#_Toc57424195)

[3.16 Observer Pattern 42](#_Toc57424196)

[3.16.1 Subject实现API 43](#_Toc57424197)

[3.16.1.1 API 描述 43](#_Toc57424198)

[3.16.1.2 类图 44](#_Toc57424199)

[3.16.1.3 相关函数及其作用 44](#_Toc57424200)

[3.17 Prototype Pattern 45](#_Toc57424201)

[3.17.1 ？？？实现API 45](#_Toc57424202)

[3.17.1.1 API 描述 45](#_Toc57424203)

[3.17.1.2 类图 46](#_Toc57424204)

[3.17.1.3 相关函数及其作用 46](#_Toc57424205)

[3.18 Proxy Pattern 47](#_Toc57424206)

[3.18.1 ScoreSheet实现API 47](#_Toc57424207)

[3.18.1.1 API 描述 47](#_Toc57424208)

[3.18.1.2 类图 48](#_Toc57424209)

[3.19 Singleton Pattern 48](#_Toc57424210)

[3.19.1 Animal Sport Meeting实现API 49](#_Toc57424211)

[3.19.1.1 API 描述 49](#_Toc57424212)

[3.19.1.2 类图 49](#_Toc57424213)

[3.19.1.3 相关函数及其作用 49](#_Toc57424214)

[3.19.2 Games Builder实现API 50](#_Toc57424215)

[3.19.2.1 API 描述 50](#_Toc57424216)

[3.19.2.2 类图 50](#_Toc57424217)

[3.19.2.3 相关函数及其作用 50](#_Toc57424218)

[3.19.3 CCommandFn实现API 51](#_Toc57424219)

[3.19.3.1 API 描述 51](#_Toc57424220)

[3.19.3.2 类图 51](#_Toc57424221)

[3.19.3.3 相关函数及其作用 51](#_Toc57424222)

[3.19.4 Athlete Container实现API 52](#_Toc57424223)

[3.19.4.1 API 描述 52](#_Toc57424224)

[3.19.4.2 类图 52](#_Toc57424225)

[3.19.4.3 相关函数及其作用 52](#_Toc57424226)

[3.19.5 Game Container实现API 53](#_Toc57424227)

[3.19.5.1 API 描述 53](#_Toc57424228)

[3.19.5.2 类图 53](#_Toc57424229)

[3.19.5.3 相关函数及其作用 53](#_Toc57424230)

[3.19.6 JudgeContainer实现API 54](#_Toc57424231)

[3.19.6.1 API 描述 54](#_Toc57424232)

[3.19.6.2 类图 54](#_Toc57424233)

[3.19.6.3 相关函数及其作用 54](#_Toc57424234)

[3.20 State Pattern 55](#_Toc57424235)

[3.20.1 Athlete状态实现API 55](#_Toc57424236)

[3.20.1.1 API 描述 55](#_Toc57424237)

[3.20.1.2 类图 56](#_Toc57424238)

[3.20.1.3 相关函数及其作用 56](#_Toc57424239)

[3.21 Strategy Pattern 57](#_Toc57424240)

[3.21.1 Games实现API 57](#_Toc57424241)

[3.21.1.1 API 描述 57](#_Toc57424242)

[3.21.1.2 类图 58](#_Toc57424243)

[3.21.1.3 相关函数及其作用 58](#_Toc57424244)

[3.22 Template Method 59](#_Toc57424245)

[3.22.1 ？？？实现API 59](#_Toc57424246)

[3.22.1.1 API描述 59](#_Toc57424247)

[3.22.1.2 类图 60](#_Toc57424248)

[3.22.1.3 相关函数及其作用 60](#_Toc57424249)

[3.23 Visitor 61](#_Toc57424250)

[3.23.1 RankingViewer访问实现API 61](#_Toc57424251)

[3.23.1.1 API描述 61](#_Toc57424252)

[3.23.1.2 类图 62](#_Toc57424253)

[3.23.1.3 相关函数及其作用 62](#_Toc57424254)

[3.24 MVC 63](#_Toc57424255)

[3.24.1 ？？？实现API 64](#_Toc57424256)

[3.24.1.1 API描述 64](#_Toc57424257)

[3.24.1.2 类图 64](#_Toc57424258)

[3.24.1.3 相关函数及其作用 65](#_Toc57424259)

[3.25 Servant Pattern 65](#_Toc57424260)

[3.25.1 ???实现API 66](#_Toc57424261)

[3.25.1.1 API描述 66](#_Toc57424262)

[3.25.1.2 类图 66](#_Toc57424263)

[3.25.1.3 相关函数及其功能 67](#_Toc57424264)

[3.26 Null Object Pattern 67](#_Toc57424265)

[3.26.1 ???实现API 68](#_Toc57424266)

[3.26.1.1 API描述 68](#_Toc57424267)

[3.26.1.2 类图 68](#_Toc57424268)

[3.26.1.3 相关函数及其功能 68](#_Toc57424269)

[3.27 Simple Factory Pattern 68](#_Toc57424270)

[3.27.1 ？？？实现API 69](#_Toc57424271)

[3.27.1.1 API描述 69](#_Toc57424272)

[3.27.1.2 类图 69](#_Toc57424273)

[3.27.1.3 相关函数及其作用 69](#_Toc57424274)

[3.28 Transfer Object Pattern 69](#_Toc57424275)

[3.28.1 XXX实现API 70](#_Toc57424276)

[3.28.1.1 API描述 70](#_Toc57424277)

[3.28.1.2 类图 70](#_Toc57424278)

[3.28.1.3 相关函数及其作用 70](#_Toc57424279)

[3.29 Filter Pattern 70](#_Toc57424280)

[3.29.1 XXX实现API 71](#_Toc57424281)

[3.29.1.1 API描述 71](#_Toc57424282)

[3.29.1.2 类图 71](#_Toc57424283)

[3.29.1.3 相关函数及其作用 71](#_Toc57424284)

[3.30 Intercepting Filter Pattern 71](#_Toc57424285)

[3.30.1 XXX实现API 72](#_Toc57424286)

[3.30.1.1 API描述 72](#_Toc57424287)

[3.30.1.2 类图 72](#_Toc57424288)

[3.30.1.3 相关函数及其作用 72](#_Toc57424289)

[3.31 DAO 72](#_Toc57424290)

[3.31.1 XXX描述API 73](#_Toc57424291)

[3.31.1.1 API描述 73](#_Toc57424292)

[3.31.1.2 类图 73](#_Toc57424293)

[3.31.3 相关函数及其作用 73](#_Toc57424294)

[3.32 Composite Entity Pattern 73](#_Toc57424295)

[3.32.1 XXX实现API 74](#_Toc57424296)

[3.32.1.1 API描述 74](#_Toc57424297)

[3.32.1.2 类图 74](#_Toc57424298)

[3.32.1.3 相关函数及其作用 74](#_Toc57424299)

[3.33 Front Controller Pattern 74](#_Toc57424300)

[3.33.1 XXX实现API 75](#_Toc57424301)

[3.33.1.1 API描述 75](#_Toc57424302)

[3.33.1.2 类图 75](#_Toc57424303)

[3.33.1.3 相关函数及其作用 75](#_Toc57424304)

[3.34 Delegate Pattern 75](#_Toc57424305)

[3.34.1 XXX实现API 76](#_Toc57424306)

[3.34.1.1 API描述 76](#_Toc57424307)

[3.34.1.2 类图 76](#_Toc57424308)

[3.34.1.3 相关函数及其作用 76](#_Toc57424309)

[3.35 Lazy initialization 76](#_Toc57424310)

[3.35.1 XXX实现API 77](#_Toc57424311)

[3.35.1.1 API描述 77](#_Toc57424312)

[3.35.1.2 类图 77](#_Toc57424313)

[3.35.1.3 相关函数及其作用 77](#_Toc57424314)

## 1. 项目简介

《动物运动会》是由我们小组设计的一套动物运动会题材的体验类动物运动会API，用户可以扮演一个动物运动会的参赛小动物，在设定的大厅和比赛里进行游戏。

玩家可以选择所使用动物的种族和颜色等特征，随后即可参加动物运动会；

动物可以在大厅选择将要进行的动作，包括参加比赛，进入饮品台，打印成绩单，购买装备，询问其他动物成绩；

参加比赛，系统会根据你所选择的动物运动员种族，获取相应的能力参数，根据你的动物运动员的状态获取状态参数，通过成绩计算函数获取相应的比赛成绩，并与其他7名系统生成的小动物竞赛，得到你在本场比赛中的成绩和名次。

进入饮品台，可以选择想要的饮品，包括：coffee，cola，mineral water，对于每个饮品可以选择添加配料，包括：milk，lemon，sugar，在添加配料的过程中可以进行redo操作，也可以进行undo操作，结束饮品选择后要进行付款操作，在付款过程中要对不同种族的货币按一定比率进行兑换，从而成功完成付款操作。不同的饮品可以为该动物带来不同的属性加成和状态恢复，帮助动物在运动会中取得更好的成绩；

选择打印成绩单，可以查看你所选择的动物在当前已完成的所有比赛项目的成绩，并为你选择的动物打印出相应的成绩单，以获得动物在本次运动会上的具体表现；

进入装备购买界面中…

选择查询其他动物成绩，可以选择你要询问的动物对象，并选择你想要知道的该动物具体某一项运动的成绩，该动物将如实告知你他的成绩，以便你判断你的动物在本次运动会的项目中与其他小动物的差距；

总而言之，《动物运动会》具有很强的可操作性与参与度，是一套经典的赛事体验类游戏API。

## 2. Design Pattern 汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | Design Pattern Name | 实现个（套）数 | sample programs 个数 | 备注 |
| 1 | Abstract Factory | 1 | 1 | 完成 |
| 2 | Adapter | 1 | 1 | 董晓 |
| 3 | Bridge | 1 | 1 | 董恩琦 |
| 4 | Builder | 1 | 1 | 完成 |
| 5 | Chain of Responsibility | 1 | 1 | 江一舸 |
| 6 | Command | 1 | 1 |  |
| 7 | Composite | 1 | 1 | 侯祖光 |
| 8 | Decorator | 1 | 1 |  |
| 9 | Facade | 1 | 1 | 李佳雯 |
| 10 | Factory Method | 2 | 2 | 董恩琦 |
| 11 | Flyweight | 1 | 1 |  |
| 12 | Interpreter | 1 | 1 | 侯祖光 |
| 13 | Iterator | 1 | 1 |  |
| 14 | Mediator | 1 | 1 | 李佳雯 |
| 15 | Memento | 1 | 1 | 江一舸 |
| 16 | Observer | 1 | 1 | 梁黄思语 |
| 17 | Prototype | 1 | 1 |  |
| 18 | Proxy | 1 | 1 | 完成 |
| 19 | Singleton | 6 | 6 | 完成 |
| 20 | State | 1 | 1 | 完成 |
| 21 | Strategy | 1 | 1 | 完成 |
| 22 | Template Method | 3 | 3 |  |
| 23 | Visitor | 1 | 1 | 梁黄思语 |
| 24 | MVC | 1 | 1 |  |
| 25 | Servant | 1 | 1 |  |
| 26 | Null Object | 1 | 1 |  |
| 27 | Simple Factory | 1 | 1 |  |
| 28 | Transfer Object | 1 | 1 |  |
| 29 | Filter | 1 | 1 |  |
| 30 | Intercepting Filter | 1 | 1 |  |
| 31 | DAO | 1 | 1 |  |
| 32 | Composite Entity | 1 | 1 |  |
| 33 | Front Controller | 1 | 1 |  |
| 34 | Composite Entity | 1 | 1 |  |
| 35 | Lazy initialization | 1 | 1 |  |
|  |  |  |  |  |
| 总计 | | 31 | 31 |  |

## 3. 设计模式详述

### 3.1 Abstract Factory

设计模式简述

抽象工厂模式即以同一界面创建一整族相关或相依的objects，不需点名各对象真正所属的具体类。为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。

使用抽象工厂的条件：

* 系统中有多个产品族，每个具体工厂创建同一族但属于不同等级结构的产品。
* 系统一次只可能消费其中某一族产品，即同族的产品一起使用。

优点：

* 可以在类的内部对产品族中相关联的多等级产品共同管理，而不必专门引入多个新的类来进行管理。
* 当增加一个新的产品族时不需要修改原代码，满足开闭原则。

#### 3.1.1 JudgeIngredientFactory实现API

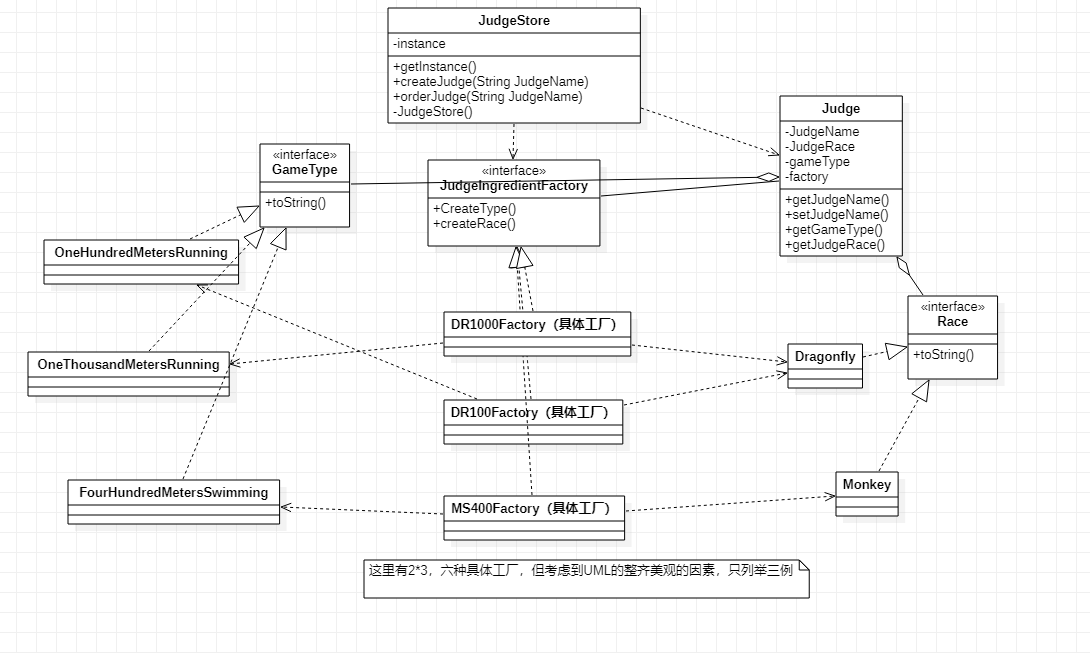
##### 3.1.1.1 API 描述

从逻辑上来讲，裁判的种族和裁判所管理的比赛类型都属于裁判这个角色本身的属性，只是二者所表示的属性内容和其作用不同，将他们共同的特点——裁判属性抽象成父类，并在父类中将裁判的种族和所管理的比赛类型也抽象化。如此一来，可以在具体的裁判工厂中指定裁判的种族以及所管理的比赛类型，同时用户可以方便的切换工厂，以生成不同类型的裁判满足不同比赛的需求。

在该API中，可以为不同比赛生成出不同类型的裁判，并将裁判以所管理的比赛类型加入到对应裁判队列中，方便管理，在需要比赛需要调用裁判时，则将相应的裁判取出使用。从用户角度来看，我们直接为已经确定的比赛生成对应的裁判，符合真实场景中远动员不可以自由选择裁判的场景，也有利于管理和用户操作体验。

该API满足开放闭守原则，如果动物运动会打算为一百米短跑额外添加一个小熊猫的100米熊猫裁判工厂，只需构建PR100Factory，在工厂中生成一百米短跑的裁判员小熊猫，并在生成裁判的过程中，由用户自由调用100米熊猫裁判工厂，不需要修改工厂的内部实现也不需要增加额外的接口，并且一切选择权交由用户自己决定。

##### 3.1.1.2 类图



##### 3.1.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public abstract Race CreateRace() | 作为抽象工厂的接口函数。JudgeIngredientFactory基类中的方法，开放给用户的接口，内部首先生成裁判的种族，之后加入裁判种族队列中，具体的裁判种族在具体的产品工厂中提供。 |
| public abstract GameType CreateType(); | 作为抽象工厂的接口函数。JudgeIngredientFactory基类中的方法，开放给用户的接口，内部首先生成裁判所管理的比赛类型，之后加入裁判所管理比赛队列中，具体的裁判所管理的比赛在具体的产品工厂中提供。 |

### 3.2 Adapter

设计模式简述

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。

在现实生活中，经常出现两个对象因接口不兼容而不能在一起工作的实例，这时需要第三者进行适配。在软件设计中也可能出现：需要开发的具有某种业务功能的组件在现有的组件库中已经存在，但它们与当前系统的接口规范不兼容，如果重新开发这些组件成本又很高，这时用适配器模式能很好地解决 这些问题。

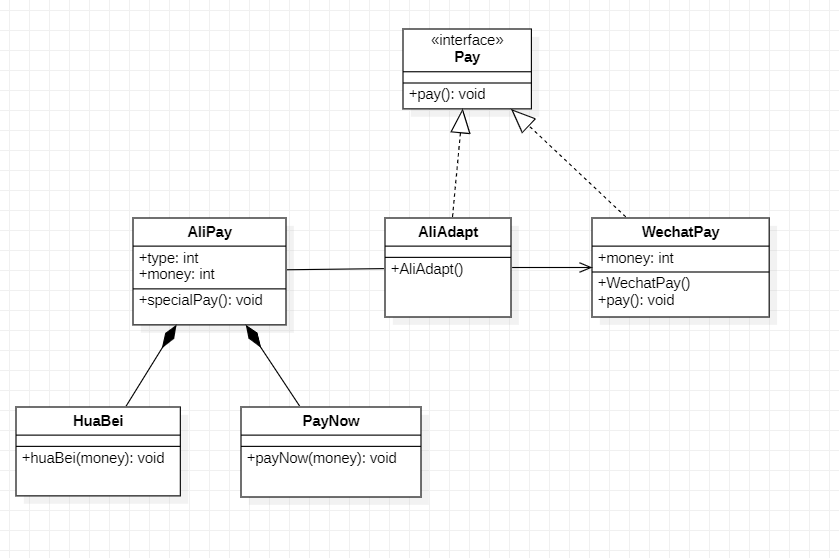
#### 3.2.1 Pay实现API

##### 3.2.1.1 API描述

Wechat支付和Ali支付都是支付方式，在只提供WeChat支付方式的时候，其他的支付方式并不适用与WeChat支付，因此使用一个适配器来对接其他的支付方式，以实现不同的支付方式。

运动员在饮品台购买饮品后需要支付一定的钱款，该场景下提供了两种付款方式：微信付款和支付宝付款，同时为支付宝设置了两种不同的支付方式：余额支付和花呗支付。对于一般支付，只需要调用Pay接口下的pay()函数即可完成支付，如微信支付，只需要执行WechatPay下的同名函数pay()，即可通过微信完成付款。但是对于支付宝形式的支付，由于提供了两种更细化的支付方式，仅仅靠pay()函数无法成功完成付款，需要一个适配器Adapter连接普通的支付方式pay和支付宝特殊形式的花呗与余额才能顺利完成支付。该场景下设置了适配器AliAdapter来继承AliPay并且实现Pay接口，在pay()函数里调用AliPay独有的函数specialPay()，完成本无法适配的支付。

##### 3.2.1.2 类图



##### 3.2.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| pay() | 接口函数，完成支付功能 |
| AliPay() | AliPay构造函数，通过控制台输出相关提示并实现用户输入 |
| WechatPay() | WechatPay构造函数，通过控制台输出相关提示并实现用户输入 |
| specialPay() | 完成Alipay的特殊支付，提供余额和花呗两种选择，并通过控制台输入付款金额 |

### 3.3 Bridge

设计模式简述

桥接模式（Bridge Pattern）是用于把抽象化与实现化解耦，使得二者可以独立变化。这种类型的设计模式属于结构型模式，它通过提供抽象化和实现化之间的桥接结构，来实现二者的解耦。

这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。

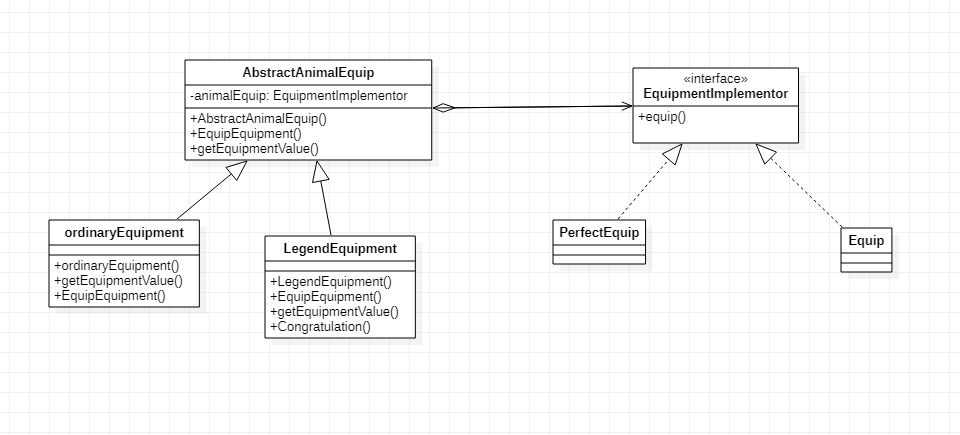
在现实生活中，某些类具有两个或多个维度的变化，如果用继承方式，不但对应的子类很多，而且扩展困难。选择桥接（Bridge）模式，将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。

#### 3.3.1 EquipmentImplementor实现API

##### 3.3.1.1 API描述

在该API中，可以为不同的装备标识上不同的品质，不同品质对应的值不一样，功能也会有不同。并且还可以选择对同一件装备选择不同的安装方式。不同的安装方式对装备的性能也会有影响。

##### 3.3.1.2 类图



##### 3.3.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public int getEquipmentValue() | 作为桥接模式的接口函数。AbstractAnimalEquip基类中的抽象方法，开放给用户的接口，用来获取装备的品质（普通装备或传说装备）。 |
| public void EquipEquipment(Athlete animal, Equipment equipment); | 作为桥接的接口函数。AbstractAnimalEquip基类中的方法，开放给用户的接口，实现具体将某件装备绑定到角色身上。 |
| public AbstractAnimalEquip(EquipmentImplementor animalEquip) | 作为桥接的接口函数。AbstractAnimalEquip基类中的方法，选择装备的安装完成度（糟糕的安装或者完美的安装）。 |

### 3.4 Builder

设计模式简述

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象而避免处理多个类之间过度复杂的继承关系或者使用单个巨大的构造函数。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的。

与此同时，建造者模式允许用户使用同一组构造接口来创造多种不同的对象，从而方便用户的使用。

该模式的主要优点如下：

1. 封装性好，构建和表示分离。
2. 扩展性好，各个具体的建造者相互独立，有利于系统的解耦。
3. 客户端不必知道产品内部组成的细节，建造者可以对创建过程逐步细化，而不对其它模块产生任何影响，便于控制细节风险。

其缺点如下：

1. 产品的组成部分必须相同，这限制了其使用范围。
2. 如果产品的内部变化复杂，如果产品内部发生变化，则建造者也要同步修改，后期维护成本较大。

#### 3.4.1 GamesBuilder 实现 API

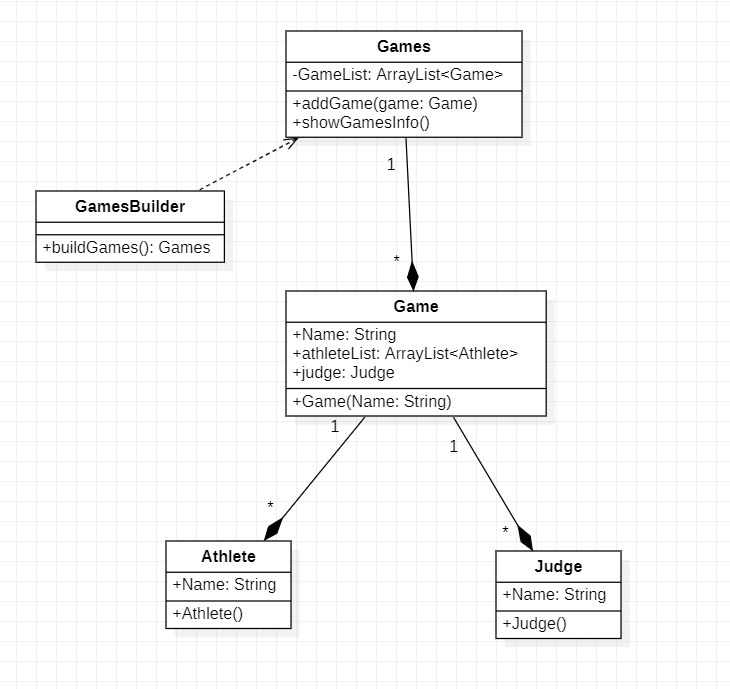
##### 3.4.1.1 API 描述

在我们的类中有一个特定的比赛类，该类主要是为了来模拟现实生活中的运动会比赛拥有多种需要的比赛项目。不同的比赛项目都有相应的参赛运动员、相应的比赛裁判和不同的比赛机制；用于不同的能力值计算方法和成绩转换函数；也用于使用之前生成的不同类型的裁判。

我们可以发现，这些比赛的构建过程都有类似的配置元素，包括参赛动物运动员和该场比赛的裁判，将这些方法抽象出来，既能分解复杂的构造过程，也能允许一个GamesBuilder来对一些常用的比赛构建计划进行封装。

该设计模式的实现满足设计模式六大基本原则中的迪米特原则，类与类之间的耦合度较低，依赖程度较低，提高了模块的相对独立性。

##### 3.4.1.2 类图



##### 3.4.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public Judge orderJudge(String JudgeName) | 如上说明，本函数定义于 Judge Store 类中，分别在 FourHundredMetersSwimming，OneHundredMetersRunning，OneThousandMetersRunning 中进行不同的实现。 |
| private Athlete (Race race, Color color) | 如上说明，本函数定义于 Athlete Factory 类中，分别在 FourHundredMetersSwimming，OneHundredMetersRunning，OneThousandMetersRunning 中进行不同的实现。 |
| public static GamesBuilder getInstance() | 作为建造工厂的接口函数。Game Builder基类中的方法，开放给用户的接口，内部首先生成比赛的运动员和比赛裁判，通过建造者模式建造实现一个比赛。因为各个建造者的实现类中将存有的不同类的对象用于构建，所以他们的 getInstance 函数将具有不同的返回参数。 |

### 3.5 Chain of Responsibility Pattern

设计模式简述

责任链模式（Chain of Responsibility Pattern）使一个请求有机会被不止一个对象处理， 从而避免请求的发送者和请求的接受者的耦合。 把能够处理请求的对象串成一个链条， 然后把请求沿着链条依次传递， 直到一个对象处理它。

在Chain of Responsibilty Pattern中，为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。主要的意图是使多个对象都有机会处理同一个请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。并且对于职责链上的处理者负责处理请求，客户只需要将请求发送到职责链上即可，无须关心请求的处理细节和请求的传递，所以职责链将请求的发送者和请求的处理者解耦。

#### 3.5.1 XXX实现API

##### 3.5.1.1 API描述

责任链模式应用在竞技场场景中，其核心思想为将某一请求交付给某一对象，若此对象无法完成请求，则把相同的请求交给下一对象，这恰好契合了竞技场对战的风格。

在此场景中，设置了父类AbstractOpponent，作为接受挑战请求的对象，继承此父类可创建不同的具体对手接收挑战，AbstractOpponent中有AbstractOpponent类的nextAbstractOpponent属性及相应的set、get函数，并有一个fight函数执行对战。实际代码中实现了一个具体类AthleteFighter作为运动员竞技手，即一场运动员间的擂台赛，这些AthleteFighter被聚集在一个AthleteFighters类中进行对战，通过比较运动员之间的各项属性加以一定算法，运动员进行两两PK，胜者留下接受下一个挑战者，直至最后决出一个胜者。

##### 3.5.1.2 类图（无）

##### 3.5.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| setNextAbstractOpponent() | 为当前选手设置下一个对战选手 |
| getNextAbstractOpponent() | 获取当前选手的对手 |
| fight() | 进行动物擂台赛 |

### 3.6 Command

设计模式简述

命令模式（command Pattern）旨在将方法调用、请求或操作封装到单一对象中，从而根据我们不同的请求对客户进行参数化和传递可供执行的方法调用。此外，这种模式将调用操作的对象与知道如何实现该操作的对象解耦，并在交换出具体类（对象）方面提供更大的整体灵活性；

命令模式为我们提供了一种分离职责的手段，这些职责包括从执行命令的任意地方发布命令以及将该职责转而委托给不同对象；

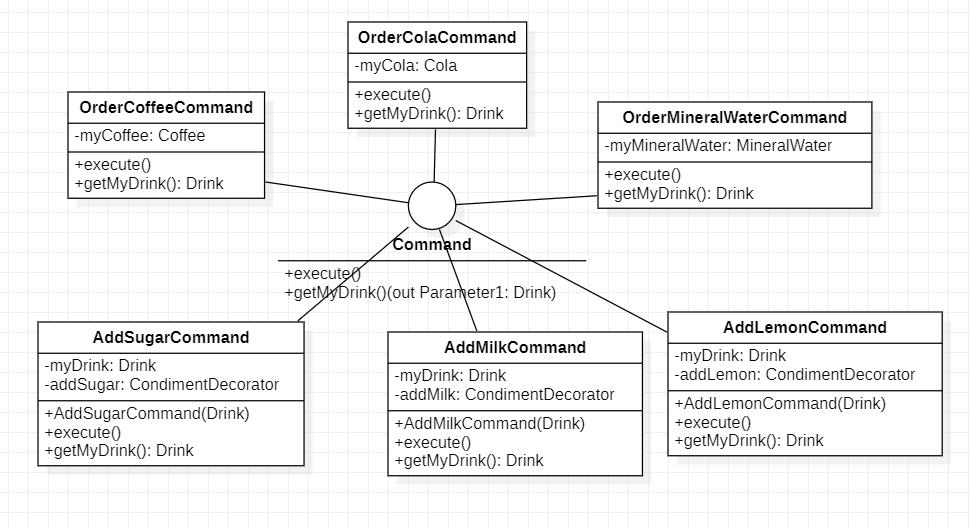
命令模式能将请求包装为一些对象，这些对象包含了有关于改请求的一切必要的信息。通过这样的处理，我们能够延迟对于一些方法的调用。与此同时，通过保存曾经被调用过的 Command 对象，用户能够较为方便的进行撤销 command 的操作。

#### 3.6.1 OrderCondiment实现 API

##### 3.6.1.1 API 描述

在饮品台场景下，运动员可以进行点单，有咖啡、可乐、矿泉水三种选择，选择完毕后，还可以选择添加不同的配料，有白砂糖、牛奶、柠檬三种选择，为此设计了六条command命令，对应六种选择。在此场景下，由于配料的选择没有限制，运动员在理论上可以无限添加配料，因此调用相关添加配料操作的次数较多，选择Command模式有利于选择调用和实际操作的解耦，以及配合Memento进行配料选择的备份，以便于实现undo和redo，增加饮品口味调整的灵活性。与点单和添加配料相关的代码中包含了控制台的菜单输出以及选择输入，实际编程过程中将这些代码封装在OrderCondiment类中。

##### 3.6.1.2 类图



##### 3.6.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| execute() | 接口函数，用于执行具体命令（点单或添加配料） |
| getMyDrink() | 用于返回当前选择的饮品种类 |

### 3.7 Composite

设计模式简述

组合模式（Composite Pattern），又叫部分整体模式，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。这种模式创建了一个包含自己对象组的类。该类提供了修改相同对象组的方式。

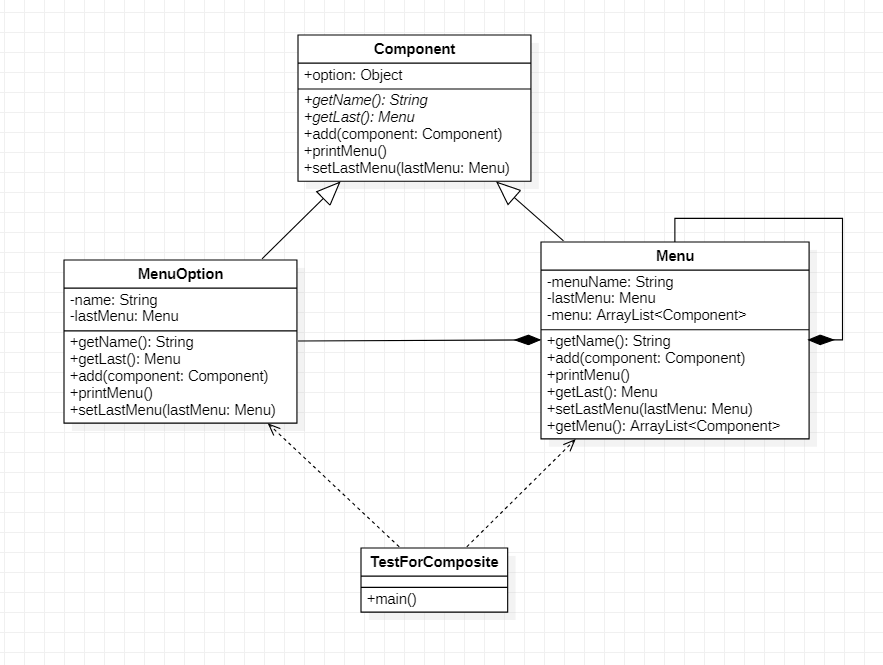
Composite Pattern 将对象组合成树状结构来体现部分与整体的层次关系。 组合模式使得客户端代码可以一致地对待 单体对象和由单体对象组合出的复合对象。

#### 3.7.1 Component 实现API

##### 3.7.1.1 API描述

该api通过设计MenuOption和Menu两种继承自Component的类来表现菜单选项中的单选项以及二级菜单，模拟了文件与文件夹的方式。从根本上看，Menu与MenuOption具有极高的相似性，二者都为存储该菜单选项的类，且相当适合打包在一起以减少结构的冗杂感，因此选择组合模式进行设计，并充分利用了组合模式中“将相似的对象打包在一起”的思想，大大的减少了开销，对外显示仅为一个Menu对象。将复杂且相似的对象组合在一起，有效防止在对外表现时的杂乱无章现象，能够对同一组对象进行很好的管理与控制。通过Add等操作向现有Menu对象中添加新的菜单选项，并会在合适时机进行打印和做相应的跳转选项。

##### 3.7.1.2 类图



##### 3.7.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| Add(component : Component) | 向该对象中添加新的菜单选项，可以是Menu或MenuOption |
| PrintMenu() | 打印菜单 |

### 3.8 Decorator / Wrapper

设计模式简述

装饰模式（Decorator Pattern）又名包装模式（Wrapper Pattern），它以对客户端透明的方式扩展对象的功能，允许向一个现有的对象添加新的功能，同时又不改变其结构，是继承关系的一个替代方案。运用该设计模式可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

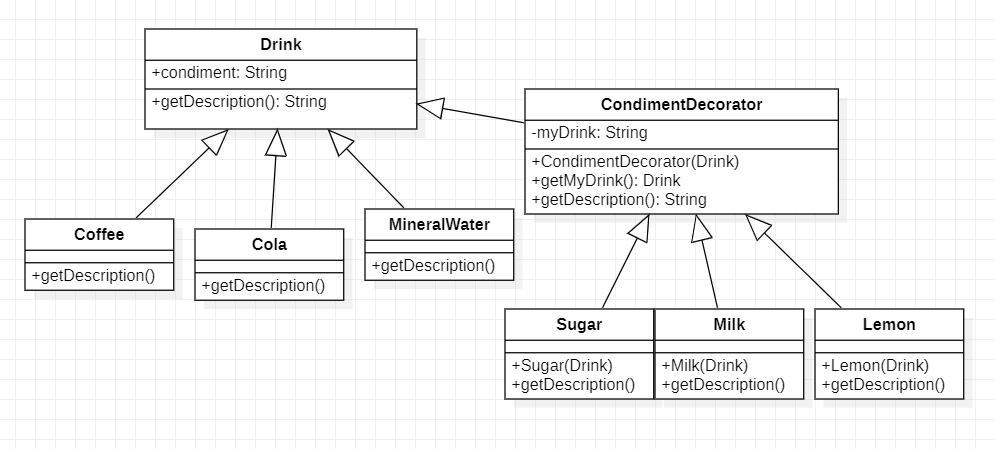
这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

#### 3.8.1 XXX 实现 API

##### 3.8.1.1 API 描述

该模式应用在饮品台场景下，有咖啡、可乐、矿泉水三种饮品，白砂糖、牛奶、柠檬三种配料作为decorator装饰饮品，不同的饮品可以为该动物带来不同的属性加成和状态恢复，帮助动物在运动会中取得更好的成绩。三种饮品继承于一个抽象类Drink，Drink中有一个String类属性“condiment”表示饮品所添加的配料。同时有一个CondimentDecorator继承于Drink，这个CondimentDecorator就是三种配料的父类，用于给饮品类加入配料，即修改饮品类的condiment属性。三种配料可以组合添加并可以无限添加。

##### 3.8.1.2 类图



##### 3.8.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| getDescription() | 返回Drink的配料 |

### 3.9 Facade

设计模式简介

外观（Facade）模式又叫作门面模式，是一种通过为多个复杂的子系统提供一个一致的接口，而使这些子系统更加容易被访问的模式。该模式对外有一个统一接口，通过对外观的包装，使客户端只能看到外观对象，而不会看到具体的细节对象，这样会大大降低应用程序的复杂度，提高了程序的可维护性。换言之，在外观模式中，子系统的外部调用者必须通过一个统一的 Facade 对象才能和子系统内部通信。

日常编码工作中，我们都在有意无意的大量使用外观模式。只要是高层模块需要调度多个子系统（2个以上的类对象），我们都会自觉地创建一个新的类封装这些子系统，提供精简的接口，让高层模块可以更加容易地间接调用这些子系统的功能。  
  
外观（Facade）模式是“迪米特法则”的典型应用，它有以下主要优点。

1. 降低了子系统与客户端之间的耦合度，使得子系统的变化不会影响调用它的客户类。
2. 对客户屏蔽了子系统组件，减少了客户处理的对象数目，并使得子系统使用起来更加容易。
3. 降低了大型软件系统中的编译依赖性，简化了系统在不同平台之间的移植过程，因为编译一个子系统不会影响其他的子系统，也不会影响外观对象。

外观（Facade）模式的主要缺点如下。

1. 不能很好地限制客户使用子系统类，很容易带来未知风险。
2. 增加新的子系统可能需要修改外观类或客户端的源代码，违背了“开闭原则”。

#### 3.9.1 SignUpFacade 实现 API

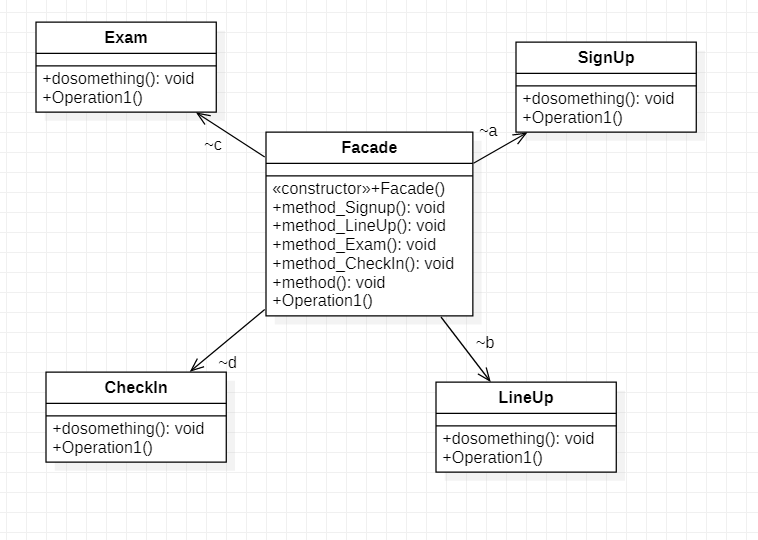
##### 3.9.1.1 API描述

当用户选择了相应的运动员动物后，将会进入报名等一系列流程，在此项目中，用户将会经过，报名，排队，体检，检录等四个行为步骤，最终正式开始参加比赛。这四个小步骤流程都属于报名过程的分支，它们的最终作用是趋于一致的。所以我们可以将这四个小功能进行抽象化，并且设置一个凌驾于这四个小功能之上的大接口，以简化对用户而言复杂的报名流程。在该API中，可以让对于用户来说不重要，或者没必要一定知道的某些步骤，例如排队，体检等过程隐形于用户。

该API为四个子系统中的一组对用户的接口提供一个一致的界面，外观模式将会定义一个更高层的接口，这个接口的存在使得这一子系统更加容易使用。这降低了访问这个对于用户来说较为复杂系统的内部子系统的复杂度，简化了客户端和与之的接口。用于我们的用户并不需要知道系统内部的报名，排队，体检，检录的复杂联系，所以我们向用户提供了一个Facade的接口，这定义了系统的入口。这使得我们的在进入系统即将开始比赛时，就能够快速高效地完成，并不会因为子系统的问题发生耽误，也更加符合实际应用，有利于管理，提升用户的项目体验。

Facade是典型的迪米特法则和依赖倒转原则的应用，但是他并不符合开放性原则，所以用户是没有办法知道子系统内部的行为模式，如果发生问题，会比较麻烦。同时，由于用户不知道且无法对内部的行为做出更改，将会使继写或者重写都变得困难。但它可以极大的优化用户的功能体验，将报名，排队，体检，检录过程进行整合，创建一个更高层的接口，用户将直接调用这个接口，完成复杂的报名流程，更加方便快捷。

##### 3.9.1.2 类图



##### 3.9.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void dosomething() | 作为子系统的行为实现函数，用户没有需要知道每一个子系统的具体行为，子系统的行为将在此提供。 |
| public void method() | 作为外观模式的接口函数，提供给用户调用四个子系统功能的接口，其下包含四个子系统提供给外部的访问方法，用以一次性完成四个子系统的功能。 |

### 3.10 Factory Method

设计模式简述

工厂方法模式（Factory Pattern）又称工厂模式，是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

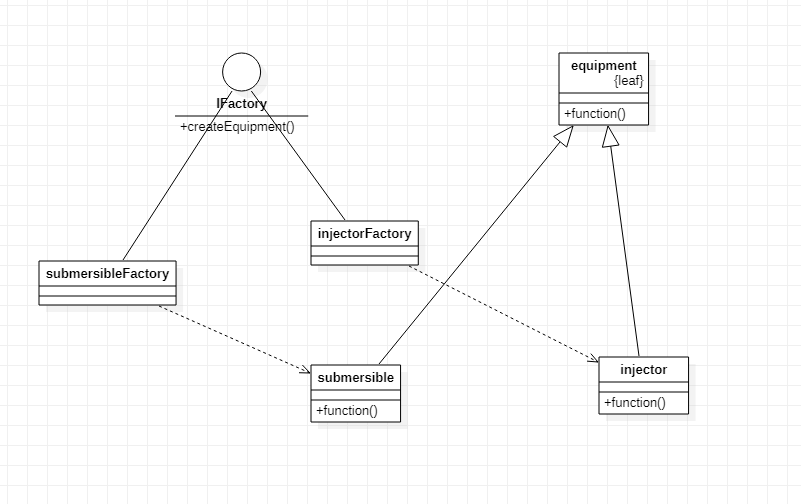
在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。

#### 3.10.1 IFactory实现API

##### 3.10.1.1 API 描述

作为桥接的接口函数。AbstractAnimalEquip基类中的方法，选择装备的安装完成度（糟糕的安装或者完美的安装）

##### 3.10.1.2 类图



##### 3.10.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void function() | 作为工厂方法的接口函数。Equipment基类中的抽象方法，开放给用户的接口，用来获取装备的功能。 |
| public String getEquipmentName(); | 作为工厂方法的接口函数。AbstractAnimalEquip基类中的抽象方法，开放给用户的接口，返回装备的种类名称。 |

### 3.11 Flyweight Pattern

设计模式简述

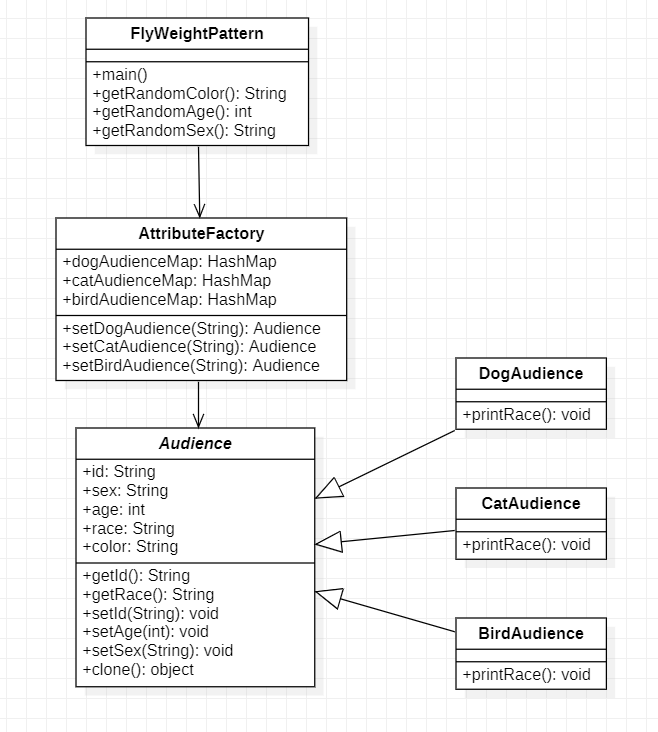
享元模式（Flyweight Pattern）运用共享技术有效地支持大量细粒度的对象，主要用于减少创建对象的数量，以减少内存占用和提高性能。由于 享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式，它是一种 对象结构型模式，它提供了减少对象数量从而改善应用所需的对象结构的方式。享元模式尝试重用现有的同类对象，如果未找到匹配的对象，则创建新对象。

#### 3.11.1 ？？？实现API

##### 3.11.1.1 API 描述

绝大多数情况下，每个观众除了种族信息以外还会具有一些其他属性。因此，我们可以定义一个工厂类AttributeFactory用来为之前创建过的用户附加相应的属性，该工厂类中使用键名为每个实体类的颜色的HashMap。每次接受创建观众请求时都会为该观众附加一个颜色属性。并在之后检查HashMap中创建的相应实体类的对象，如果找到相应的对象便直接返回相应的对象，否则会先创建一个目标对象并将其存储之HashMap中，然后再将该对象返回。最后在Test时使用AttributeFactory来获取一个新的Audiecnce对象，完成一个具有特定颜色观众对象的创建。

##### 3.11.1.2 类图



##### 3.11.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void setDogAudience(color)等… | 作为FlyWeight的接口函数。AttributeFactory基类中的方法，开放给用户的接口，用户可以通过调用该方法为一个观众设置颜色。 |
| public void setAge(int) | Audience基类中的方法，开放给用户的接口，用户可以通过调用该方法为目标观众设置年龄。 |
| public void setSex(String) | Audience基类中的方法，开放给用户的接口，用户可以通过调用该方法为目标观众设置性别。 |
| public void printRace() | Audience基类中的方法，开放给用户的接口，用户可以通过调用该方法输出该观众的身份信息。 |

### 3.12 Interpreter Pattern

设计模式简述

解释器（Interpreter）模式的定义：给分析对象定义一个语言，并定义该语言的文法表示，再设计一个解析器来解释语言中的句子。也就是说，用编译语言的方式来分析应用中的实例。这种模式实现了文法表达式处理的接口，该接口解释一个特定的上下文。

这里提到的文法和句子的概念同编译原理中的描述相同，“文法”指语言的语法规则，而“句子”是语言集中的元素。

在软件开发中，会遇到有些问题多次重复出现，而且有一定的相似性和规律性。如果将它们归纳成一种简单的语言，那么这些问题实例将是该语言的一些句子，这样就可以用“编译原理”中的解释器模式来实现了。但对于满足以上特点，且对运行效率要求不是很高的应用实例，如果用解释器模式来实现，其效果是非常好的。

解释器模式是一种类行为型模式，其主要优点如下。

1. 扩展性好。由于在解释器模式中使用类来表示语言的文法规则，因此可以通过继承等机制来改变或扩展文法。
2. 容易实现。在语法树中的每个表达式节点类都是相似的，所以实现其文法较为容易。

解释器模式的主要缺点如下。

1. 执行效率较低。解释器模式中通常使用大量的循环和递归调用，当要解释的句子较复杂时，其运行速度很慢，且代码的调试过程也比较麻烦。
2. 会引起类膨胀。解释器模式中的每条规则至少需要定义一个类，当包含的文法规则很多时，类的个数将急剧增加，导致系统难以管理与维护。
3. 可应用的场景比较少。

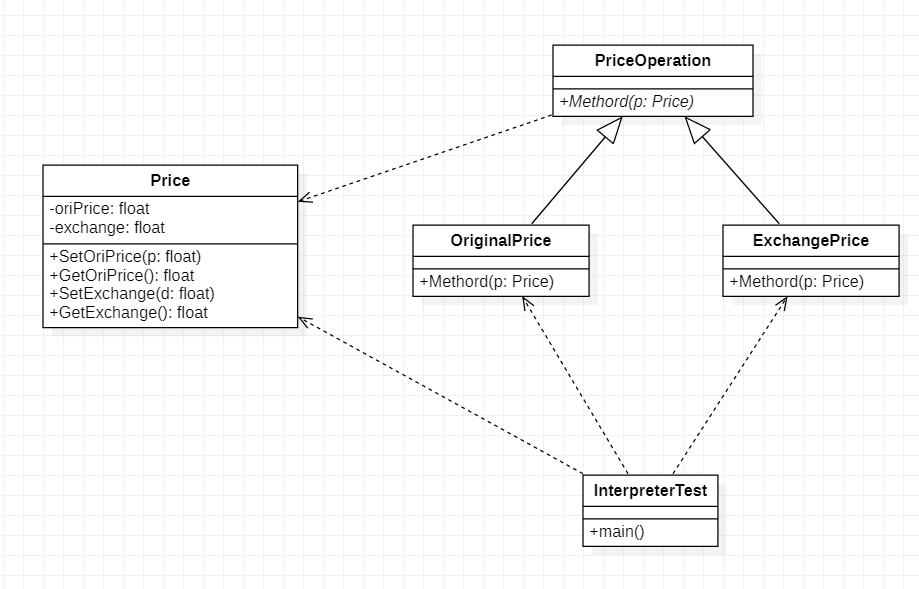
#### 3.12.1 PriceOperation实现API

##### 3.12.1.1 API 描述

对于像价格这样重复出现但又在不同场景有不同变化的对象，我们可以通过设计Price这样一个存储了价格和汇率的类，以及几个对Price进行不同解析方式的解释器来输出对应价格。在合适的场合设置Price的汇率，利用对应场合的价格解释器，将Price对象输入到解释器中即可输出该场合所需的最终价格。由解释器来帮我们进行这样重复但又有不同输入对象的操作。

该模式应用在饮品台场景下，动物们在付款时应用的运动货币与实际流通货币不是1：1等值的，而是存在一定的货币汇率，在此设定为1：10，若一杯咖啡10元，则动物们实际需要支付的货币为100，而这一转换过程需要Interpreter来实现，该模式解读了当前货币的汇率并将之直接转换为动物需要支付的运动货币，从而便于动物付款购买饮品。

##### 3.12.1.2 类图



##### 3.12.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| Method() | 定义解释方法，计算汇率并返回真实支付数额，由不同的子类针对不同场合进行方法具体实现。 |

### 3.13 Iterator Pattern

设计模式简述

迭代器模式（Iterator Pattern）是 Java 和 .Net 编程环境中非常常用的设计模式。这种模式用于顺序访问集合对象的元素，不需要知道集合对象的底层表示。

在现实生活以及程序设计中，经常要访问一个聚合对象中的各个元素，如“数据结构”中的链表遍历，通常的做法是将链表的创建和遍历都放在同一个类中，但这种方式不利于程序的扩展，如果要更换遍历方法就必须修改程序源代码，这违背了 “开闭原则”。

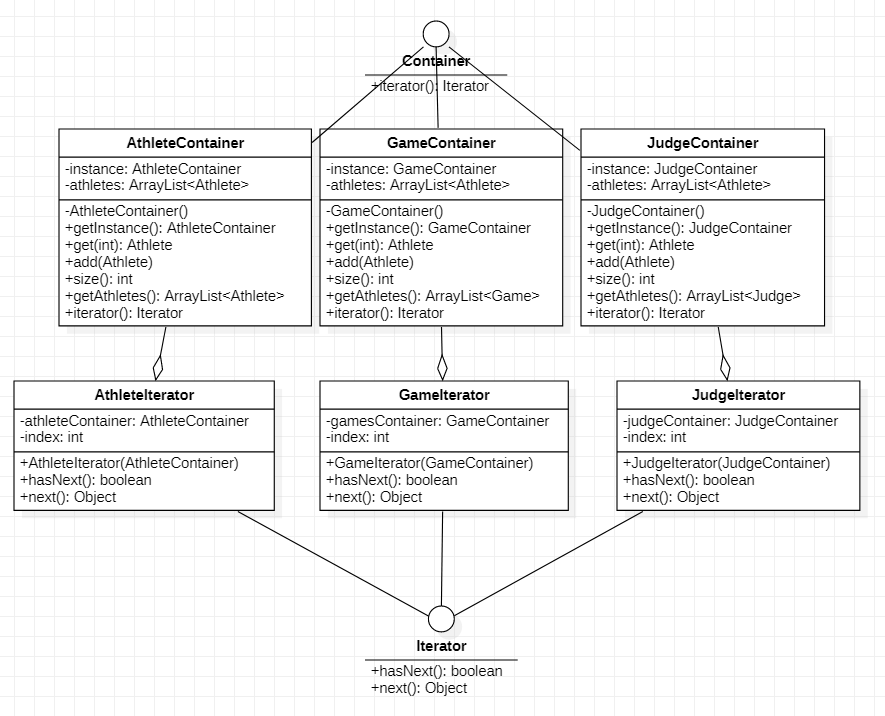
“迭代器模式”能较好地克服以上缺点，它在客户访问类与聚合类之间插入一个迭代器，这分离了聚合对象与其遍历行为，对客户也隐藏了其内部细节，且满足“单一职责原则”和“开闭原则”。

#### 3.13.1 XXX实现API

##### 3.13.1.1 API 描述

迭代器模式（Iterator Pattern）使运动员、裁判和比赛等的遍历操作更加简便，使代码编写者无需知道各种容器的大小，就能够直接遍历整个容器并获取容器中的元素。在Iterator接口中有两个主要函数hasNext()和next()，hasNext()用于判断当前指针下是否还有下一个元素存在，next()用于提取元素并使指针后移。在该项目中，有三个具体的运动员、比赛和裁判Iterator实现了Iterator接口，并创建了对应的三个具体Container用于存储相应元素及实现Iterator的初始化，每个Container作为对应具体Iterator的私有属性，并实现Container接口，其中有get、add、size等操作用于添加获取元素。

##### 3.13.1.2 类图



##### 3.13.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| iterator() | 初始化相应Container的iterator |
| hasNext(); | 判断容器中是否还有下一个元素 |
| next() | 获取下一个元素 |

### 3.14 Mediator Pattern

设计模式简述

中介者（Mediator Pattern）模式：定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，使代码易于维护且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。其主要有点如下：  
- 降低了对象之间的耦合性，使得对象易于独立地被复用。  
- 将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，使得系统易于维护和扩展。  
其主要缺点是：当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

#### 3.14.1 AthleteAndOthersMediator实现API

##### 3.14.1.1 API 描述

抽象中介者是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。具体中介者实现中介者接口，定义一个列表来管理同事对象，协调各个同事角色（Athlete和Others）之间的交互关系，因此它依赖于同事角色。

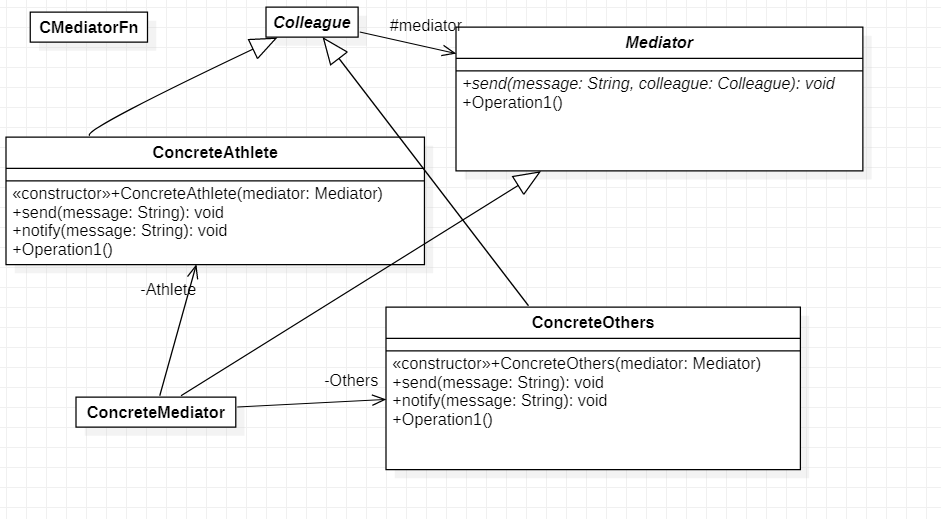
抽象同事类定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。具体同事类是抽象同事类的实现者，当需要与其它同事对象交互时，由中介者对象负责后续的交互。

在逻辑中，用户参与比赛之后可以在专属的大厅进行休息，喝饮料等恢复能量的行为，此时，如果发生需要参加的比赛，运动员将无法获得消息，此时，需要Mediator设计模式成立中介者。

将运动员动物和其他来通知消息的动物分别进行抽象化，设置抽象同事类，同时创建抽象中介者类和具体中介者类，用于类和类之间的消息传送。用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

由于对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。简单来说，中介类（不管是具体中介者还是抽象中介者），作为一个桥梁搭建于两个user:Athlete和Others之间，运动员用户先通过抽象出抽象中介者类，通过抽象中介者模式，将消息传递给抽象其他动物，再通过抽象转化为具体同事类，以处理不同类之间的通信，并支持松耦合，使代码易于维护。

##### 3.14.1.2 类图



##### 3.14.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void send(String message) | 具体同事类、中介者类等当中，普通动物观众或者是其他运动员动物，通过中介者使用该函数，向用户动物发送消息。 |
| public void notify(String message) | 具体同事类运动员动物类当中用于发送消息message的函数，以便于修改message的内容。 |
| public void setOthers(ConcreteOthers Others) | 在中介者类中，设置具体的同事类1和2，该函数设置了用户以外的其他动物，将同事类当中的Others 与中介者模式中的相对应，setAthlete()与该函数同理，不再赘述。 |
| public void send(String message, Colleague colleague) | 中介者类中进行发送消息功能的函数，用来在两个具体同事类Others和Athlete之间发送消息。 |

### 3.15 Memento Pattern

设计模式简述

备忘录（Memento）模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式 。  
备忘录模式是一种对象行为型模式，其主要优点如下。  
- 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。  
- 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。  
- 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

其主要缺点是：资源消耗大。如果要保存的内部状态信息过多或者特别频繁，将会占用比较大的内存资源。

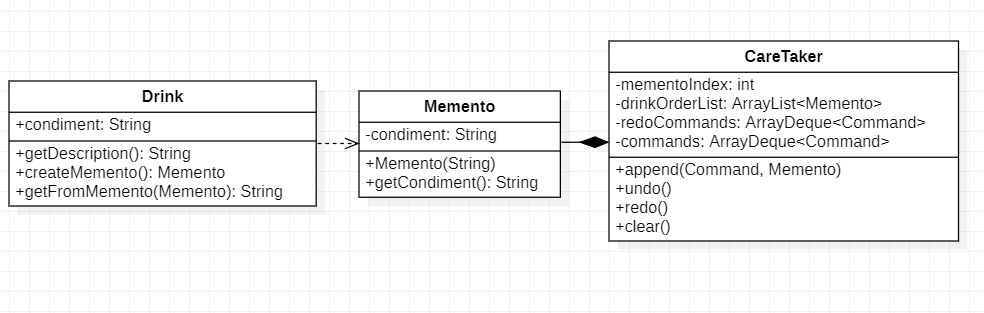
#### 3.15.1 OrderCondiment实现API

##### 3.15.1.1 API 描述

在饮品台场景下，Memento为Drink的备份类，CareTaker用于维护一个以Memento为元素的List，以便随时调取和增加备份，支持指令的undo和redo。在饮品种类点单完毕后，饮品的初始态（无添加配料）被保存在CareTaker中的MementoList中，进入配料选择后，每一次选择都将改变饮品的状态，每个状态都会被创建为一个新的Memento对象并存入List中。

当运动员选择undo时，CareTaker会从List中选取最近一次的先前备份，通过getCondiment()函数返回此饮品的先前状态下的配料选择，并赋值给当前选择的饮品，完成undo。同时CareTaker中包含了一个commandsList和一个redoCommandsList，分别存储了点单命令和被撤回的点单命令，当执行undo时，pop出commandsList的一条命令并push进redoCommandsList，当执行redo时，pop出redoCommandsList中的一条命令并push进commandsList，当有新的点单命令时，清空redoCommandsList中的所有命令，此时只能执行undo，不能执行redo。与点单和添加配料相关的代码中包含了控制台的菜单输出以及选择输入，实际编程过程中将这些代码封装在OrderCondiment类中。

##### 3.15.1.2 类图



##### 3.15.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| createMemento() | 创建Drink的备份，存储Drink的配料 |
| getFromMemento() | 从备份中获得Drink的配料 |
| append() | 将新创建的Memento加入存储List |
| undo() | 撤销命令，恢复Drink的先前状态 |
| redo() | 重新执行撤销的命令 |

### 3.16 Observer Pattern

设计模式简述

在现实世界中，许多对象并不是独立存在的，其中一个对象的行为发生改变可能会导致一个或者多个其他对象的行为也发生改变。例如，某种商品的物价上涨时会导致部分商家高兴，而消费者伤心；还有，当我们开车到交叉路口时，遇到红灯会停，遇到绿灯会行。这样的例子还有很多，例如，股票价格与股民、微信公众号与微信用户、气象局的天气预报与听众、小偷与警察等。

在软件世界也是这样，例如，Excel 中的数据与折线图、饼状图、柱状图之间的关系；MVC 模式中的模型与视图的关系；事件模型中的事件源与事件处理者。所有这些，如果用观察者模式来实现就非常方便。

观察者模式（Observer Pattern），又叫发布-订阅模式（Publish / Subscribe Pattern），定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。其优点如下：

1、观察者和被观察者是抽象耦合的。

2、建立一套触发机制。

其主要缺点是：

1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3、观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

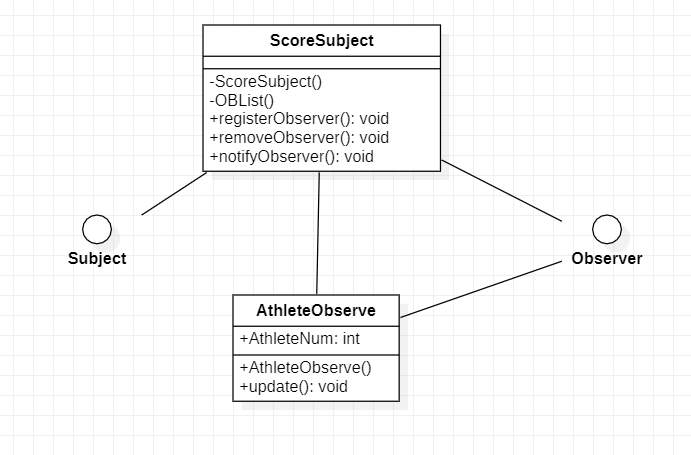
#### 3.16.1 Subject实现API

##### 3.16.1.1 API 描述

将查看运动员成绩的动物抽象为观察者的概念，并将查看成绩的操作也抽象化。如此一来，每个动物都可以查看运动会某一项目的参赛者以及他们所取得的成绩。

在该API中，可以进行注册观察者与删除观察者的操作。在项目排名发生变化时，观察者会接收到这一信息。从用户角度来看，我们为运动会制作公示板，符合真实场景中公示板上显示本场比赛排名的场景，有利于用户体验。

##### 3.16.1.2 类图



##### 3.16.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void registerObserver() | 如上说明，本函数定义于Subject类中，在ScoreSubject类中进行实现。 |
| public void removeObserver(); | 如上说明，本函数定义于Subject类中，在ScoreSubject类中进行实现。 |
| public void notifyObserver(); | 如上说明，本函数定义于Subject类中，在ScoreSubject类中进行实现 |

### 3.17 Prototype Pattern

设计模式简述

原型模式即用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节。

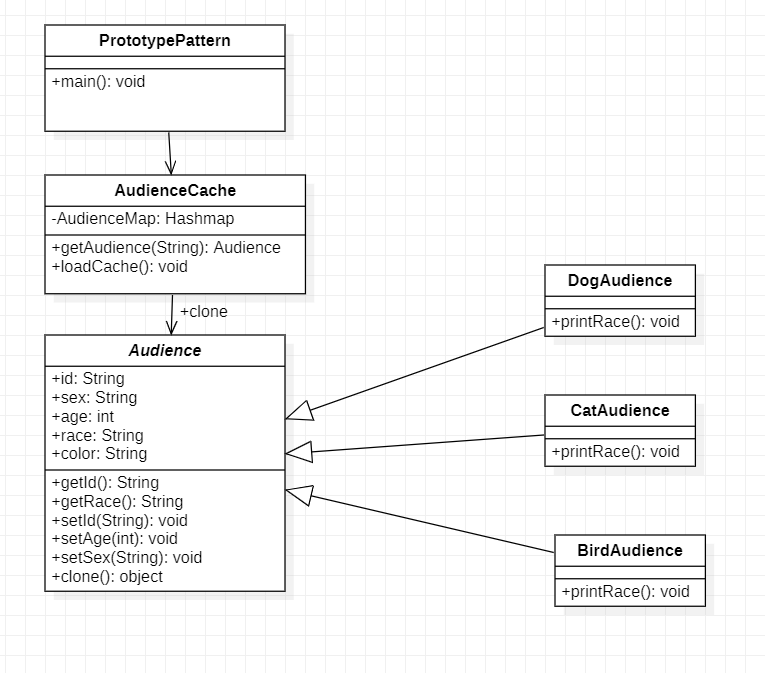
#### 3.17.1 ？？？实现API

##### 3.17.1.1 API 描述

从逻辑上来讲，每个观众除了种族信息以外其他信息（如：年龄、性别等）也是全部具备的，如果每次加入一个新观众是都从头开始进行一次创建的话，其过程是十分麻烦和浪费的。因此我们可以先使用HashMap创建一个容器，其中分别存储不同种族的观众，这样一来，每当想要创建一个新的观众时我们便可以直接通过不同种族的不同id来clone出一个特定种族的新观众。

在这里，用户可以通过id直接生成不同种族的观众，而每个种族的观众相互独立，不存在相互影响的情况，如果用户向添加一个种族为“狗”的观众，用户只需要调用getAudience方法，并且仅通过指定观众种族的id即可完成创建工作。

##### 3.17.1.2 类图



##### 3.17.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| Public void getAudience(String) | 作为Prototype的接口函数。AudienceCache基类中的方法，开放给用户的接口，用户可以通过调用该方法clone一个根据id指定的特定种族的观众。 |

### 3.18 Proxy Pattern

设计模式简述

在代理模式（Proxy Pattern）中，一个类代表另一个类的功能。这种类型的设计模式属于结构型模式。

我们由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。 代理模式的结构比较简单，主要是通过定义一个继承抽象主题的代理来包含真实主题，从而实现对真实主题的访问。

主要缺点是：

 1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。

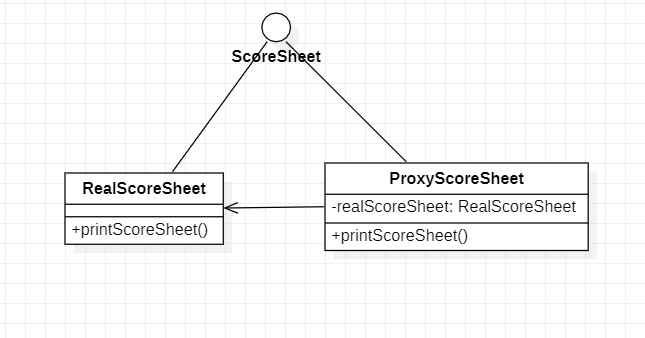
2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

#### 3.18.1 ScoreSheet实现API

##### 3.18.1.1 API 描述

我们为成绩表单的查询做一个代理，玩家用户通过与代理的交互来完成成绩表单的获取。这也符合了现实生活中，在运动会上运动员无法直接查看真实成绩表单，只能在电子屏幕上，查看“代理”的成绩表单。  
1. 抽象主题（Score Sheet）类：通过接口实现成绩表单的获取功能  
2. 真实主题（Real Score Sheet）类：Score Sheet即tool类本身  
3. 代理（Proxy Score Sheet）类：提供了和Score Sheet本身相同的方法，有所扩展

##### 3.18.1.2 类图



### 3.19 Singleton Pattern

设计模式简述

单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。  
单例模式有 3 个特点：

单例类只有一个实例对象；

该单例对象必须由单例类自行创建；

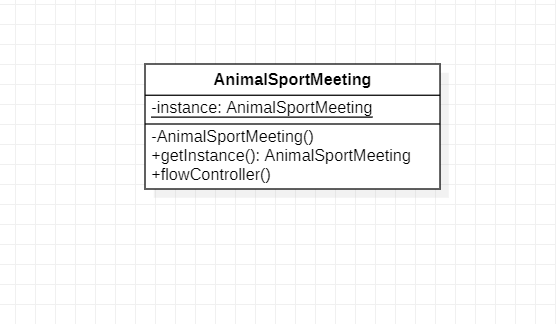
单例类对外提供一个访问该单例的全局访问点；

#### 3.19.1 Animal Sport Meeting实现API

##### 3.19.1.1 API 描述

在我们的设计中，此动物运动会项目只有一个动物运动会对象，因此AnimalSportMeeting采用Singleton来实现。AnimalSportMeeting的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.1.2 类图



##### 3.19.1.3 相关函数及其作用

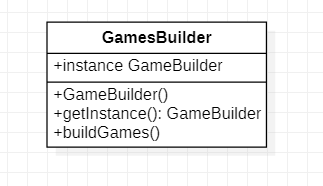
|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| public void flowController() | 执行具体的运动会流程指令 |
| private AnimalSportMeeting() | 让构造函数为 private，这样该类就不会被实例化 |

#### 3.19.2 Games Builder实现API

##### 3.19.2.1 API 描述

在我们的设计中，此动物运动会项目只有一个比赛建造者对象，因此Games Builder采用Singleton来实现。Games Builder的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.2.2 类图



##### 3.19.2.3 相关函数及其作用

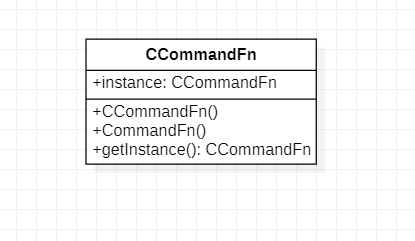
|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| public void buildGames() | 执行具体的比赛构造流程指令 |
| private GamesBuilder() | 让构造函数为 private，这样该类就不会被实例化 |

#### 3.19.3 CCommandFn实现API

##### 3.19.3.1 API 描述

根据我们的设计，这个动物运动会只存在一个饮品台对象，因此CCommandFn采用Singleton来实现。CCommandFn的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.3.2 类图



##### 3.19.3.3 相关函数及其作用

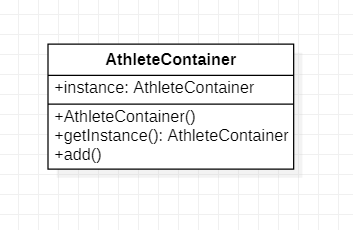
|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| public void CommandFn() | 执行具体的饮品台使用流程指令 |
| private CCommandFn() | 让构造函数为 private，这样该类就不会被实例化 |

#### 3.19.4 Athlete Container实现API

##### 3.19.4.1 API 描述

根据我们的设计，这个动物运动会只存在一个运动员管理器对象，因此Athlete Container采用Singleton来实现。Athlete Container的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.4.2 类图



##### 3.19.4.3 相关函数及其作用

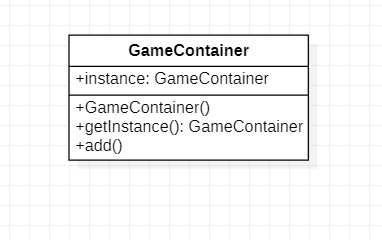
|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| get(int index) | 根据指针返回容器对应的对象 |
| private AthleteContainer() | 让构造函数为 private，这样该类就不会被实例化 |
| void add() | 在容器末尾增加对象 |
| int size() | 返回运动员管理容器的大小 |
| getAthletes() | 返回运动员容器对象本身 |
| iterator() | 生成运动员容器迭代器 |

#### 3.19.5 Game Container实现API

##### 3.19.5.1 API 描述

根据我们的设计，这个动物运动会只存在一个比赛管理器对象，因此GameContainer采用Singleton来实现。GameContainer的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.5.2 类图



##### 3.19.5.3 相关函数及其作用

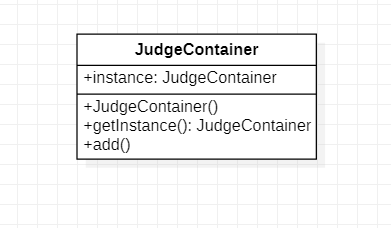
|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| private GameContainer() | 让构造函数为 private，这样该类就不会被实例化 |
| void add() | 在容器末尾增加对象 |

#### 3.19.6 JudgeContainer实现API

##### 3.19.6.1 API 描述

根据我们的设计，这个动物运动会只存在一个裁判管理器对象，因此JudgeContainer采用Singleton来实现。JudgeContainer的实例生成只在第一次调用其构造函数的时候使用。使用getInstance函数创建唯一实例。

##### 3.19.6.2 类图



##### 3.19.6.3 相关函数及其作用

|  |  |
| --- | --- |
| 函数名 | 作⽤ |
| getInstance() | 创建唯一实例 |
| private JudgeContainer () | 让构造函数为 private，这样该类就不会被实例化 |
| void add() | 在容器末尾增加对象 |

### 3.20 State Pattern

设计模式简述

状态模式（State Pattern），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。

对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。状态模式的解决思想是：当控制一个对象状态转换的条件表达式过于复杂时，把相关“判断逻辑”提取出来，放到一系列的状态类当中，这样可以把原来复杂的逻辑判断简单化。

#### 3.20.1 Athlete状态实现API

##### 3.20.1.1 API 描述

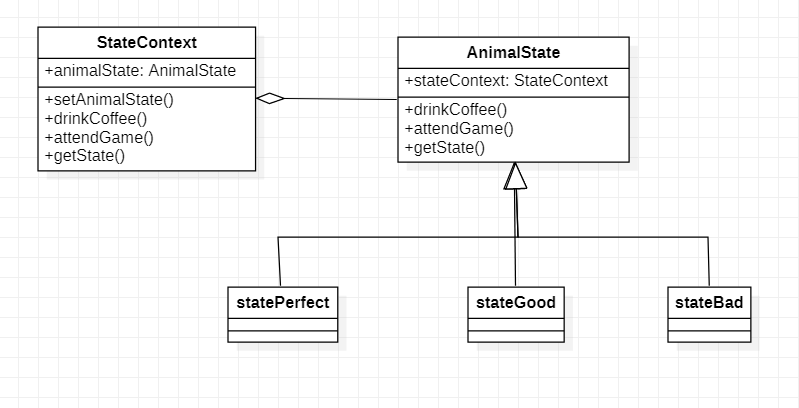
动物运动员根据不同的运动状态拥有不同的状态（完美状态，状态较好，稍有疲惫），状态的下滑会影响动物运动员的比赛成绩，用户可以通过服用饮品来恢复你的动物的运动状态，以帮助您在接下来的比赛中得到好成绩。

环境（StateContext）角色：定义一个环境类来描述运动员Athlete的状态

抽象状态（Animal State）角色：定义一个接口，用以封装环境对象中的特定状态所对应的行为。

具体状态（Concrete State）角色：实现抽象状态所对应的行为。

##### 3.20.1.2 类图



##### 3.20.1.3 相关函数及其作用

|  |  |
| --- | --- |
| 函数名 | 作用 |
| void setAnimalState (  AnimalState animalState) | 为该运动员环境设置状态 |
| void drinkCoffee() | 通过服用coffee，运动员可以恢复一次状态，若当前状态已经达到最佳状态，则不产生效果。 |
| void attendGame() | 运动员参加比赛后，状态值下降，若当前状态已经达到了疲惫状态，则不会发生状态变化。 |
| double getState() | 获得当前状态所对应的状态参数 |

### 3.21 Strategy Pattern

设计模式简述

Strategy模式定义并封装一系列算法，由具体对象根据场景选择不同的策略，从而调用到对应的不同算法。

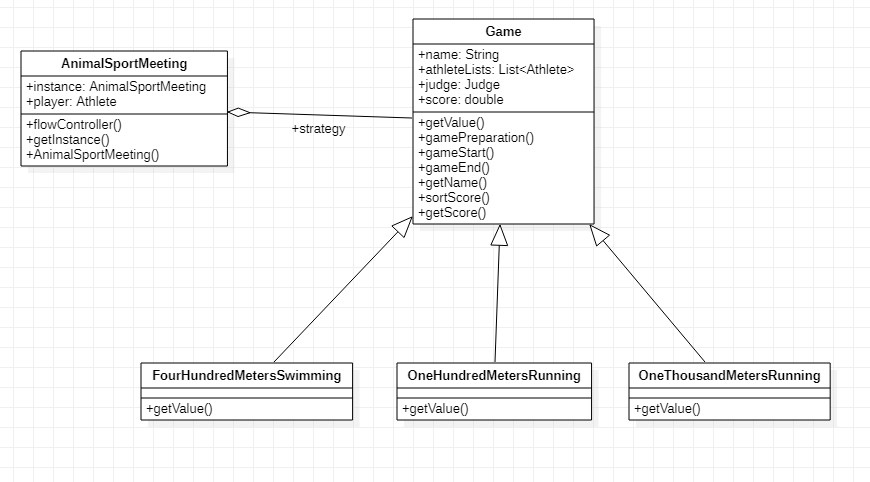
此设计模式分离具体的算法和客户端，使得客户端可以自由切换算法，算法也可以独立于客户端自由进行更改；避免在同一算法中出现大量的条件判断，而是将原本逻辑复杂的算法拆分成多个结构相对简单的独立算法；算法可扩展性良好。但是在结构框架中需要实例化每一个新的策略类，且需要对外暴露所有策略，复杂化了结构。

#### 3.21.1 Games实现API

##### 3.21.1.1 API 描述

我们为运动员的比赛提供了多种比赛选择方案，比如100米跑，1000米跑以及400米游泳比赛分别对应策略类的OneHundredMetersRunning，OneThousandMetersRunning和FourHundredMetersSwimming。动物运动员在选择比赛时，类Games将会根据 game name的不同，为该运动员选择不同的比赛策略，进行相应的比赛操作。

##### 3.21.1.2 类图



##### 3.21.1.3 相关函数及其作用

|  |  |
| --- | --- |
| 函数名 | 作用 |
| void getValue(int no) | 根据动物运动员的物种得到动物相应属性值，再由相应的能力值计算函数得到动物的能力值，结合动物的状态所决定的状态参数，通过成绩计算函数得到该动物的比赛成绩。 |
| void flowController() | 控制整个运动会的流程 |

### 3.22 Template Method

设计模式简述

在模板方法模式（Template Method Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

Template Method Pattern 提供了一种在父类中定义处理流程，在子类中具体实现的处理方式，同时在具体实现时Template Method又允许子类重新定义流程的具体步骤。优点是实现了反向控制和OCP原则，既提高了代码的复用性，又可以便捷的扩展子类群（扩展性），实现无限的可能性。这个设计模式虽然提高了代码的复用性，但是每一个不同的实现都需要一个新的子类实现，从而提升了系统的复杂度。

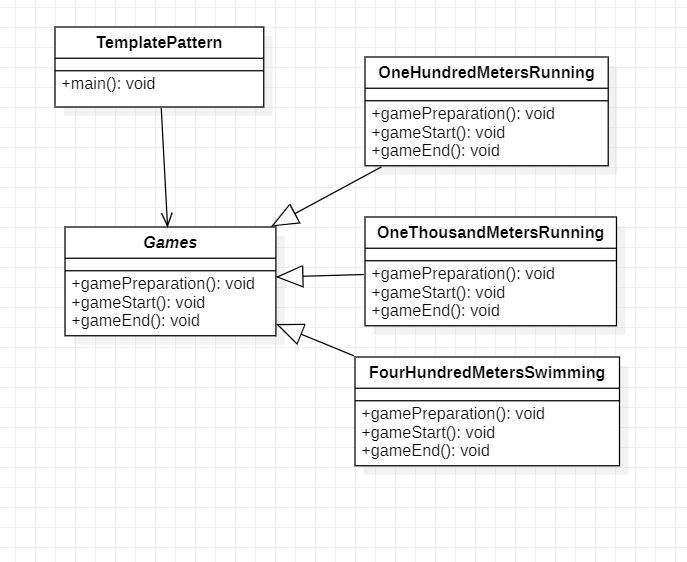
#### 3.22.1 ？？？实现API

##### 3.22.1.1 API描述

从现实角度来讲，尽管每种比赛之间有着许许多多的不同点，但是它们的进行都会有一系列的固定的流程，而这套流程往往具有高度的重复性。因此，我们可以选择写一个模版来展现比赛的流程。在这里可以定义一个Games类，然后在每个具体的game中重写Games中定义的方法。

该设计模式可以未不同的比赛项目制定一套固有的模板用户想要使用该模板时仅需创建或调用一个Games类，然后直接只需要执行play方法便可以直接生成一套该比赛项目的流程模板。

##### 3.22.1.2 类图



##### 3.22.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void play() | 作为Template模式的接口函数。Games基类中的方法，开放给用户的接口，内部首先定义了每个比赛项目中必须具备的流程，并将它们封装在一个play方法中，具体的赛事由用户进行选择。 |

### 3.23 Visitor

设计模式简述

访问者模式是一个相对比较简单，但结构又稍显复杂的模式，它讲的是表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

将作用于某种数据结构中的各元素的操作分离出来封装成独立的类，使其在不改变数据结构的前提下可以添加作用于这些元素的新的操作，为数据结构中的每个元素提供多种访问方式。它将对数据的操作与数据结构进行分离，是行为类模式中最复杂的一种模式。

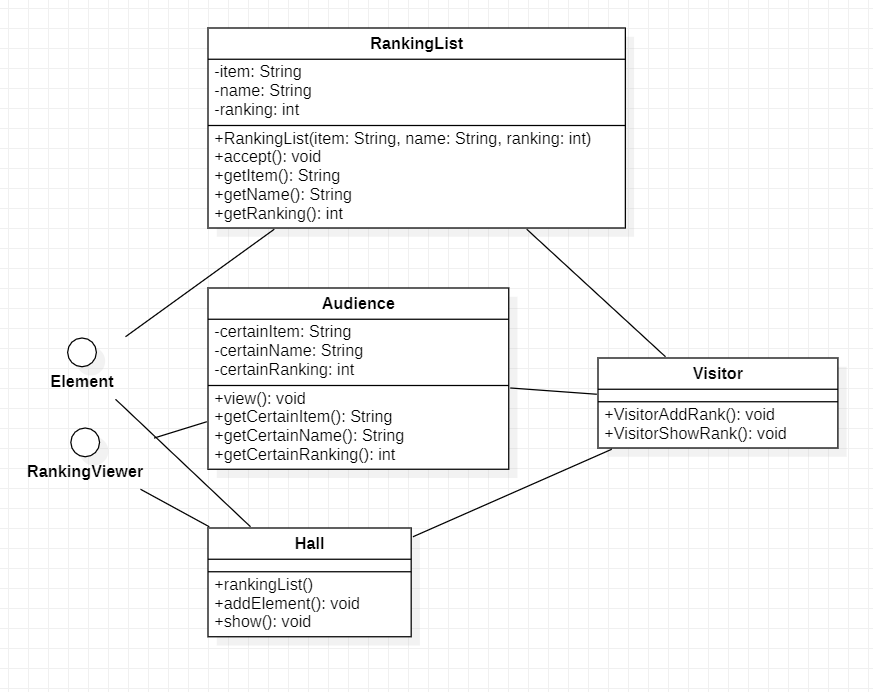
#### 3.23.1 RankingViewer访问实现API

##### 3.23.1.1 API描述

将查询运动员排名的观众抽象为排名查看者，即访问者的概念。将所查看的比赛项目、运动员姓名以及运动员成绩装入排行榜中。观众可以通过访问大厅来查询具体排名信息。

在该API中，可以为不同比赛分别生成排行榜，并储存运动员的排名信息。在需要调用排行榜时，则将相应的排行榜取出并查看。从用户角度来看，我们为每一个项目生成排行榜，符合真实场景中观众可以直接查看本场比赛排名的场景，有利于用户体验。

##### 3.23.1.2 类图



##### 3.23.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| void view(RankingList rangkingList); | 作为访问者的接口函数。Rankingviewer基类中的方法，开放给用户的接口，内部首先生成排行榜，之后通过大厅向观众开放访问渠道。具体的排行榜在具体的产品工厂中提供。 |

### 3.24 MVC

参考资料

<https://www.cnblogs.com/klb561/p/10170383.html>

<https://zh.wikipedia.org/wiki/MVC>

设计模式简述

MVC模式（Model–view–controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC模式最早由Trygve Reenskaug在1978年提出，是施乐帕罗奥多研究中心（Xerox-PARC）在20世纪80年代为程序语言Smalltalk发明的一种软件架构。MVC模式的目的是实现一种动态的程序设计，使后续对程序的修改和扩展简化，并且使程序某一部分的重复利用成为可能。除此之外，此模式透过对复杂度的简化，使程序结构更加直观。软件系统透过对自身基本部分分离的同时也赋予了各个基本部分应有的功能。

在经典MVC模式中，M即模型(Model)是指模型表示业务规则。在MVC的三个部件中，模型拥有最多的处理任务。被模型返回的数据是中立的，模型与数据格式无关，这样一个模型能为多个视图提供数据，由于应用于模型的代码只需写一次就可以被多个视图重用，所以减少了代码的重复性。V即视图(view)是指用户看到并与之交互的界面。比如由html元素组成的网页界面，或者软件的客户端界面。MVC的好处之一在于它能为应用程序处理很多不同的视图。在视图中其实没有真正的处理发生，它只是作为一种输出数据并允许用户操纵的方式。C即控制器(controller)是指控制器接受用户的输入并调用模型和视图去完成用户的需求，控制器本身不输出任何东西和做任何处理。它只是接收请求并决定调用哪个模型构件去处理请求，然后再确定用哪个视图来显示返回的数据。

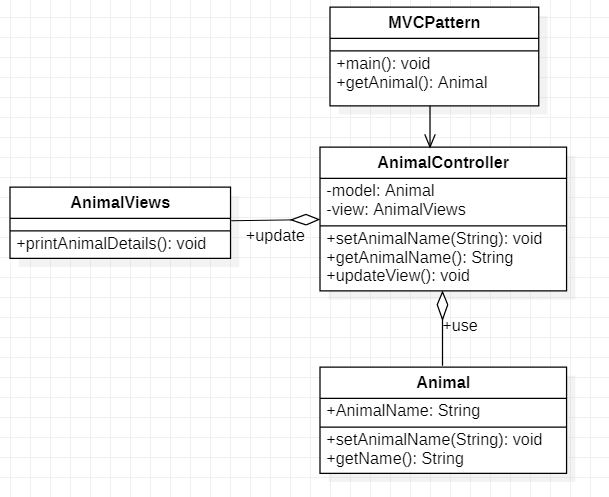
#### 3.24.1 ？？？实现API

##### 3.24.1.1 API描述

从逻辑上来讲，参赛运动员应该具备自己的姓名属性，然而有时会存在忘记设置姓名或是初始化了错误的姓名。这样一来，就需要系统提供一个窗口来帮助有这方面需求的运动员进行修改姓名的操作。

在这里系统为用户提供了修改姓名的功能，用户在需要进行修改姓名操作时可以直接调用setAnimalName修改后，修改操作完成后需要在更新视图来查看修改结果。

##### 3.24.1.2 类图



##### 3.24.1.3 相关函数及其作用

|  |  |
| --- | --- |
| **函数名** | **作用** |
| public void View() | 作为MVC模式的接口函数。AnimalController基类中的方法，开放给用户的接口，用户可以通过调用该方法来更新经过AnimalController处理过的视图 |
| public String getAniamlName() | 作为MVC模式的接口函数。AnimalController基类中的方法，开放给用户的接口，用户可以通过调用该方法来更新试图，获取使用者的当前姓名。 |
| public void setAnimalName(String) | 作为MVC模式的接口函数。AnimalController基类中的方法，开放给用户的接口，用户可以通过调用该方法来更新试图，重新设置使用者的姓名。 |

### 3.25 Servant Pattern

参考资料

<https://www.cnblogs.com/hujingnb/p/10171629.html>

<https://blog.csdn.net/afei__/article/details/80835057>

设计模式简述

雇工模式（Servant Pattern）也叫做仆人模式，是行为模式的一种，其意图是：为一组类提供通用的功能，而不需要类实现这些功能，它是命令模式的一种扩展。

其中有如下三种角色:

IServiced : 用于定义服务内容的接口

Serviced : 具体的服务提供者

Servant : 执行者,即雇工类, 用于执行服务

IServiced用于定义一组类所具有的功能，而Serviced针对不同的服务对象有不同的服务内容，也就是具体实现IServiced接口所定义的功能。功能定义完毕后，我们需要由一个雇工来执行这些功能。简单而言，就是需要有一个执行者，可以把一组功能聚集起来，这个执行者就是servant。

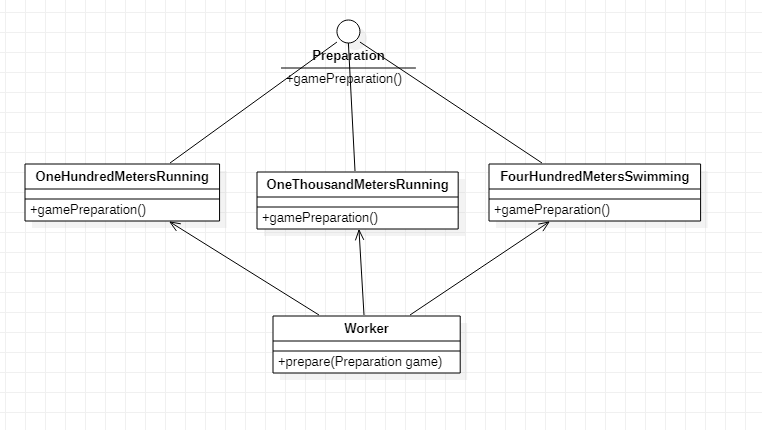
在整个雇工模式中，所有具有IServiced功能的类可以实现该接口，然后由雇工类Servant进行集合，完成一组类不用实现通用功能而具有相应职能的目的。

在日常的开发中, 我们可能已经接触过雇工模式,只是没有把它抽离出来,也没有汇编成册. 雇工模式是命令模式的一种简化, 但它更符合我们实际的需求,更容易引入开发场景中.

#### 3.25.1 ???实现API

##### 3.25.1.1 API描述

##### 3.25.1.2 类图



##### 3.25.1.3 相关函数及其功能

### 3.26 Null Object Pattern

参考资料

<http://www.cs.oberlin.edu/~jwalker/nullObjPattern/>

<https://www.cnblogs.com/sco1234/p/7954814.html>

<https://blog.csdn.net/qiumengchen12/article/details/44923139>

设计模式简述

空对象模式（Null Object Pattern）为缺少给定类型的对象提供一个对象作为替代。Null对象提供了智能的“不执行任何操作”行为，向其协作者隐藏了细节。

在空对象模式中，一个空对象取代 NULL 对象实例的检查。Null 对象不是检查空值，而是反应一个不做任何动作的关系。这样的 Null 对象也可以在数据不可用的时候提供默认的行为。

我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类，还创建一个未对该类做任何实现的空对象类，该空对象类将无缝地使用在需要检查空值的地方。

空对象模式的优点如下：

（1）它可以加强系统的稳固性，能有有效地防止空指针报错对整个系统的影响，使系统更加稳定。

（2）它能够实现对空对象情况的定制化的控制，能够掌握处理空对象的主动权。

（3）它并不依靠Client来保证整个系统的稳定运行。

（4）它通过isNull对==null的替换，显得更加优雅，更加易懂。

#### 3.26.1 ???实现API

##### 3.26.1.1 API描述

##### 3.26.1.2 类图

##### 3.26.1.3 相关函数及其功能

### 3.27 Simple Factory Pattern

参考资料

<https://blog.csdn.net/u012156116/article/details/80857255>

<https://www.cnblogs.com/ygsworld/p/10649491.html>

设计模式简述

简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。因为在简单工厂模式中用于创建实例的方法是静态（static）方法，因此简单工厂模式又被称为静态工厂方法（Static Factory Method）模式，它属于类创建型模式。

**优点**

1. 工厂类是整个模式的关键.包含了必要的逻辑判断,根据外界给定的信息,决定究竟应该创建哪个具体类的对象.
2. 通过使用工厂类,外界可以从直接创建具体产品对象的尴尬局面摆脱出来,仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的．
3. 明确了各自的职责和权利，有利于整个软件体系结构的优化。

**缺点**

1. 由于工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则，将全部创建逻辑集中到了一个工厂类中；
2. 它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。
3. 当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求．
4. 这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利；

这些缺点在工厂方法模式中得到了一定的克服。

#### 3.27.1 ？？？实现API

##### 3.27.1.1 API描述

##### 3.27.1.2 类图

##### 相关函数及其作用

### 3.28 Transfer Object Pattern

参考资料

设计模式简述

传输对象模式（Transfer Object Pattern）用于从客户端向服务器一次性传递带有多个属性的数据。传输对象也被称为数值对象。传输对象是一个具有 getter/setter 方法的简单的 POJO 类，它是可序列化的，所以它可以通过网络传输。它没有任何的行为。服务器端的业务类通常从数据库读取数据，然后填充 POJO，并把它发送到客户端或按值传递它。对于客户端，传输对象是只读的。客户端可以创建自己的传输对象，并把它传递给服务器，以便一次性更新数据库中的数值。以下是这种设计模式的实体:

业务对象（Business Object） - 为传输对象填充数据的业务服务。

传输对象（Transfer Object） - 简单的 POJO，只有设置/获取属性的方法。

客户端（Client） - 客户端可以发送请求或者发送传输对象到业务对象。

#### 3.28.1 XXX实现API

##### 3.28.1.1 API描述

##### 3.28.1.2 类图

##### 3.28.1.3 相关函数及其作用

### 3.29 Filter Pattern

参考资料

<https://github.com/ZQCard/design_pattern>

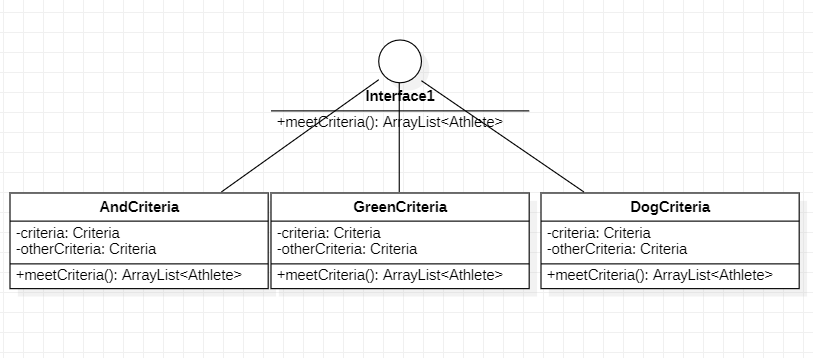
设计模式简述

过滤器模式（Filter Pattern）也叫做标准模式（Criteria Pattern），它允许开发人员使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来。这种类型的设计模式属于结构型模式，它结合多个标准来获得单一标准。

#### 3.29.1 XXX实现API

##### 3.29.1.1 API描述

##### 3.29.1.2 类图



##### 3.29.1.3 相关函数及其作用

### 3.30 Intercepting Filter Pattern

参考资料

设计模式简述

拦截过滤器模式（Intercepting Filter Pattern）用于对应用程序的请求或响应做一些预处理/后处理。定义过滤器，并在把请求传给实际目标应用程序之前应用在请求上。过滤器可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

过滤器（Filter）：过滤器在请求处理程序执行请求之前或之后，执行某些任务。

过滤器链（Filter Chain）：过滤器链带有多个过滤器，并在 Target 上按照定义的顺序执行这些过滤器。

Target：Target 对象是请求处理程序。

过滤管理器（Filter Manager）：过滤管理器管理过滤器和过滤器链。

客户端（Client）：Client是向 Target 对象发送请求的对象。

#### 3.30.1 XXX实现API

##### 3.30.1.1 API描述

##### 3.30.1.2 类图

##### 3.30.1.3 相关函数及其作用

### 3.31 DAO

参考资料

设计模式简述

DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。

在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。

用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。

DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。有如下优点：

1、隔离了数据访问代码和业务逻辑代码。业务逻辑代码直接调用DAO方法即可，完全感觉不到数据库表的存在。分工明确，数据访问层代码变化不影响业务逻辑代码,这符合单一职能原则，降低了藕合性，提高了可复用性。

2、隔离了不同数据库实现。采用面向接口编程，如果底层数据库变化，如由 MySQL 变成 Oracle 只要增加 DAO 接口的新实现类即可，原有 MySQ 实现不用修改。这符合 "开-闭" 原则。该原则降低了代码的藕合性，提高了代码扩展性和系统的可移植性

#### 3.31.1 XXX描述API

##### 3.31.1.1 API描述

##### 3.31.1.2 类图

##### 3.31.3 相关函数及其作用

### 3.32 Composite Entity Pattern

参考资料

设计模式简述

组合实体模式（Composite Entity Pattern）用在 EJB 持久化机制中。一个组合实体是一个 EJB 实体 bean，代表了对象的图解。当更新一个组合实体时，内部依赖对象 beans 会自动更新，因为它们是由 EJB 实体 bean 管理的。以下是组合实体 bean 的参与者。

组合实体（Composite Entity）：它是主要的实体 bean。它可以是粗粒的，或者可以包含一个粗粒度对象，用于持续生命周期。

粗粒度对象（Coarse-Grained Object）：该对象包含依赖对象。它有自己的生命周期，也能管理依赖对象的生命周期。

依赖对象（Dependent Object）：依赖对象是一个持续生命周期依赖于粗粒度对象的对象。

策略（Strategies）：策略表示如何实现组合实体。

#### 3.32.1 XXX实现API

##### 3.32.1.1 API描述

##### 3.32.1.2 类图

##### 3.32.1.3 相关函数及其作用

### 3.33 Front Controller Pattern

参考资料

设计模式简述

前端控制器模式（Front Controller Pattern）是用来提供一个集中的请求处理机制，所有的请求都将由一个单一的处理程序处理。该处理程序可以做认证/授权/记录日志，或者跟踪请求，然后把请求传给相应的处理程序。以下是这种设计模式的实体。

前端控制器（Front Controller）：处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 web 的应用程序，也可以是基于桌面的应用程序。

调度器（Dispatcher）：前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序。

视图（View）：视图是为请求而创建的对象。

#### 3.33.1 XXX实现API

##### 3.33.1.1 API描述

##### 3.33.1.2 类图

##### 3.33.1.3 相关函数及其作用

### 3.34 Delegate Pattern

参考资料

设计模式简述

委托模式(Delegate Pattern): 多个对象接收并处理同一请求，他们将请求委托给另-一个对象统一处理请求。其典型为事件委托机制。

它是通过委托者将请求委托给被委托者去处理实现的。因此委托模式解决了请求与委托者之间的耦合。通过被委托者对接收到的请求的处理后，分发给相应的委托者去处理。

委托模式和代理模式有很大程度上的相似性，都是把业务的需要实现的逻辑交给一个目标实现类来完成。但也有一定细微的区别，通俗的理解是，代理模式的代理类和实现类是上下级关系，而委托模式的委托类和被委托类是平级关系。这就很好的解释了委托模式和代理模式在类与类之间的关系上存在差异。

#### 3.34.1 XXX实现API

##### 3.34.1.1 API描述

##### 3.34.1.2 类图

##### 3.34.1.3 相关函数及其作用

### 3.35 Lazy initialization

参考资料

设计模式简述

延迟初始化（Lazy initialization）也称延迟实例化，是指在使用组件或对象时才进行初始化。一个对象的延迟初始化意味着该对象的创建将会延迟至第一次使用该对象时。延迟初始化主要用于提高性能，避免浪费计算，并减少程序内存要求。

延迟初始化是延迟到需要域的值时才将它初始化的这种行为。如果永远不需要这个值，这个域就永远不会被初始化。虽然延迟初始化主要是一种优化，但它也可以用来打破类和实例初始化中的有害循环。

就像大多数的优化一样，对于延迟初始化，最好建议“除非绝对必要，否则就不要这么做”。延迟初始化就像一把双刃剑。它降低了初始化类或者创建实例的开销，却增加了访问被延迟初始化的域的开销。根据延迟初始化的域最终需要初始化的比例、初始化这些域要多少开销，以及每个域多久被访问一次，延迟初始化实际上降低了性能。

也就是说，延迟初始化有它的好处。如果域只在类的实例部分被访问，并且初始化这个域的开销很高，可能就值得进行延迟初始化。要确定这一点，唯一的办法就是测量类在用和不同延迟初始化时的性能差异。

#### 3.35.1 XXX实现API

##### 3.35.1.1 API描述

##### 3.35.1.2 类图

##### 3.35.1.3 相关函数及其作用