软件架构与设计模式课程项目

Animal Games

指导老师: **冯巾松老师、侯捷老师**

1[illegible]0477 邓欣凌
1[illegible]653918 闫怡舟
1750226 陆昱珉
1850952 梁荣嘉
1851634 张向龙
1851892 李子涵
1852144 李一凡
1852452 张明哲
1853204 慕林桀
1853562 梁正扬

2020/11/27

Tongji SSE

目录

一.项目简介

本组设计模式项目选题为动物运动会，项目名称 AnimalGames。项目背景是动物界一次盛大的运动会，运动会在特定举办地"动物奥林匹克园"(OlympicsYard)由唯一主办方召开，参与动物有运动员、工作人员、观众等多种身份。

运动会一开始会从总场馆场景初始化，并建立大赛主办方。初始化后进行场馆的搭建，需要建造赛场以及计算不同场馆的面积。

赛前准备方面，本场运动会提供体育赛事报名系统，外加判断选手参赛资格、报名时回退等功能，提供所有工作人员名单，以及可以遍历运动员集合、展示运动员的信息、可查询与修改的信息系统。

后勤方面，运动场搭建了运动员宿舍的安保系统，并且为动物运动员分配训练房间，提供存储体育器材管理。特别地，本赛事专门为动物选手们考虑，习性不同的动物可以自由生活，不必担忧存在捕食关系的选手之间发生不愉快。

作为一场大型运动会，赛场内建立了方便的通讯机制，并且提供不同平台的赛事消息发布以及远程采访。

为提高参赛选手、工作人员以及观众的生活水平，本赛场专门建立了美食广场，提供多种菜品分类，包括各种口味正餐以及冰淇淋等甜品。美食广场支付方便，支持不同动物国货币以及各种支付接口。

比赛方面，运动会从开幕式运动员入场正式开始，此后每个比赛会被分配所需裁判，运动员根据教练点名上场、并需要根据裁判的通知做出响应。运动会有乒乓球、接力等比赛，比赛过程中奖牌榜排名与积分榜实时更新，即时创建成绩单与发放奖牌奖状。比赛允许加时，并且对争议判罚有相应处理措施，运动员可以得到补给并查询成绩。

二.汇总表

1.设计模式实现汇总

编号	设计模式名称	实现个（套）数	何处体现	备注
1	Simple factory	1	main.java.simplefactory	
2	Factory Method	1	main.java.factorymethod	
3	Abstract Factory	1	main,java.abstractfactory	
4	Builder	2	main.java.Builder	实现了经典模式和变种模式两种形式的 Builder
5	Prototype	2	main.java.prototype；main.java.prototypedeepcopy	实现了浅克隆和深克隆两种形式的 prototype

6	Singleton	1	main.java.singleton	
7	Adapter	1	main.java.adapter	
8	Bridge	1	main.java.bridge	
9	Composite	1	main,java.composite	
10	Decorator	2	main.java.decorator	实现了两个装饰器
11	Façade	1	main.java.facade	
12	Flyweight	1	main.java.flyweight	
13	Private Class Data	1	main.java.privateclassdata	
14	Proxy	1	main.java.Proxy	
15	Chain of Responsibility	1	main.java.chainofresponsibility	
16	Command	2	main.java.command	实现了主办方颁发奖牌和裁判向运动员发令
17	Interpreter	1	main.java.interpreter	
18	Iterator	1	main.java,iterator	
19	Mediator	1	main.java.mediator	
20	Memento	1	main.java.memento	
21	Observer	1	main.java.observer	
22	State	1	main.java.state	
23	Strategy	3	main.java.Strategy	实现了动物进食、训练与入场表演三个策略，并与模板模式构成对比
24	Template	1	main.java.template	
25	Visitor	1	main.java.visitor	
26	Active Object	1	main.java.activeobject	

27	Balking	1	main.java.balking	
28	Worker Thread	1	main.java.workerthread	
29	Dirty Flag	1	main.java.DirtyFlag	
30	Lazy loading	1	main.java.lazyloading	
31	Front Controller	1	main.java.frontcontroller	内部可以选择三种对应的控制器和视图
32	Transfer Object	1	main.java.transferobject	
33	Pipeline	1	main.java.pipeline	实现了奖牌榜的排名实时更新
34	Producer Customer	1	main.java.producercustomer	
35	Double Check Locking	1	main.java.doublecheckedlocking	
36	Multiton	1	main.java.Multiton	
37	Extension Objects	1	main.java.extendobjects	
38	MVC	1	main,java,MVC	
39	Filter	1	main.java.filter	
40	Null Object	2	main.java.nullobject；main.java.simplefactory	实现了点名时点到缺席的运动员，买冰淇淋时点了不存在的冰淇淋
41	Specification	1	main.java.specification	
42	Converter	1	main.java.converter	
43	Callback	1	main.java.callback	
44	Business Delegate	1	main.java.businessdelegate	
45	Immutable	1	main.java.immutable	使奖牌实例在多个线程中同时存在保持状态不变，且在访问这些实例时并不需要执行耗时的互斥处理。

共计		52		

2. 非‘ GoF's 23 design patterns’ 的设计模式出处及简介

编号	设计模式	出处	基本介绍
1	Balking	1. https://en.wikipedia.org/wiki/Balk 2. 《图解 Java 多线程设计模式》	多个线程监控某个共享变量，A 线程监控到共享变量发生变化后即将触发某个动作，但是此时发现有另外一个线程 B 已经针对该变量的变化开始了行动，因此 A 便放弃了准备开始的动作。
2	Filter	https://www.tutorialspoint.com/design_pattern/filter_pattern.htm	使开发人员可以使用不同的条件过滤一组对象，并通过逻辑操作以分离的方式链接它们。这种类型的设计模式属于结构模式，因为该模式组合了多个条件以获得单个条件。
3	Private Class Data	https://sourcemaking.com/design_patterns/private_class_data	使用私有类数据设计模式可以防止不必要的操作，防止类将其属性（类变量）暴露给操作。通过最小化其属性（数据）的可见性来保护类状态。
4	Active Object	https://en.wikipedia.org/wiki/Active_object	Active Object 模式是一种异步编程模式。它通过对方法的调用(Method Invocation)与方法的执行(Method Execution)进行解耦(Decoupling)来提高并发性。
5	Extension Objects	https://java-design-patterns.com/patterns/extension-objects/	预期对象的接口将在未来被扩展。通过额外的接口来定义扩展对象。
6	Specification	https://en.wikipedia.org/wiki/Specification_pattern	规约模式（Specification Pattern）将业务规则（通常是隐式业务规则）封装成独立的逻辑单元，从而将隐式业务规则提炼为显式概念，并达到代码复用的目的。同时不同的规约可以通过布尔运算组成新的规约。

7	Worker Thread	https://en.wikipedia.org/wiki/Thread_pool	在 Worker Thread 模式中，工人线程 Worker thread 会逐个取回工作并进行处理，当所有工作全部完成后，工人线程会等待新的工作到来。
8	Double Checked Locking	https://zh.wikipedia.org/wiki/%E5%8F%8C%E9%87%8D%E6%A3%80%E6%9F%A5%E9%94%81%E5%AE%9A%E6%A8%A1%E5%BC%8F	用于在测试之前通过测试锁定条件（“lock hint”）来减少获取锁定的开销。 仅当锁定条件检查结果表明需要锁定时，才会发生锁定。常用于减少在多线程环境中实现“惰性初始化”时的锁定开销。
9	Producer Customer	《图解 Java 多线程设计模式》	生产者和消费者在同一时间段内共用同一存储空间，生产者向空间里生产数据，而消费者取走数据。且数据存储有上下限的限制。
10	Multiton	《Java 与模式》（阎宏 著）第十七章 专题：多例（Multiton）模式与多语言支持	单例模式的扩展。多例类可以有多个实例；多例类必须自己创建，管理自己的实例，并向外界提供自己的实例；实例数目可以为有限个也可以没有上限。
11	Dirty Flag	1. https://java-design-patterns.com/patterns/dirty-flag/ 2. 《J2EE Design Patterns》	用一个标志位（flag）来表示一组数据的状态，这些数据要么是用来计算，或者用来需要同步。在满足条件的时候设置标志位，然后需要的时候检查（check）标志位。如果设置了标志位，那么表示这组数据处于 dirty 状态，这个时候需要重新计算或者同步。如果 flag 没有被设置，那么可以不计算（或者利用缓存的计算结果）。
12	Pipeline	https://java-design-patterns.com/patterns/pipeline/	私有类数据模式（Private Class Data）封装类的初始化数据，控制对类的属性的更改，并保持类数据与使用数据的方法间的隔离。
13	Immutable	《图解 Java 多线程设计模式》	Immutable模式中存在这确保实例状态不发生改变的类(immuatable类），在访问这些实例时并不需要执行耗时的互斥操作，因此可以提高程序性能。
14	Simple Factory	Factory (java-design-patterns.com)	提供封装在称为 factory 的类中的静态方法，以隐藏实现逻辑，使客户端代码专注于使用，而不是初始化新对象。

15	Front Controller	https://java-design-patterns.com/patterns/front-controller/	为一个网站的所有请求引入一个共同的处理程序。这样，我们就可以在一个地方封装诸如安全、国际化、路由和日志等共同功能。
16	Lazy Loading	https://www.geeksforgeeks.org/lazy-loading-design-pattern/	延迟加载是一种设计模式，通常用于将对象的初始化延迟到需要它的点。如果使用得当，它可以提高程序运行的效率。
17	MVC	https://zh.wikipedia.org/wiki/MVC	对用户提供一种查看信息的方法，不需要知道具体的类接口，只要由控制器返回视图即可，常见用在 B/S 架构
18	Null Object	https://java-design-patterns.com/patterns/null-object/	在大多数面向对象的语言（如 Java 或 C#）中，引用可能为空。在调用任何方法之前，需要检查这些引用以确保它们不是 null，因为通常不能在 null 引用上调用方法。该模式不使用 null 引用来表示缺少对象（例如，不存在的客户），而是使用实现预期接口但方法正文为空的对象。此方法的优点是，Null 对象非常可预测，没有副作用：它不执行任何操作。
19	Transfer Object	https://java-design-patterns.com/patterns/data-transfer-object/	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。
20	Converter	https://java-design-patterns.com/patterns/converter/	转换器模式的目的是提供一种通用的、通用的方法来在相应的类型之间进行双向转换，从而允许一个干净的实现，其中类型不需要相互知道。此外，转换器模式引入了双向集合映射，将样板代码减少到最小。
21	Callback	https://java-design-patterns.com/patterns/callback/	Callback 是一段可执行代码，它作为参数传递给其他代码，其他代码希望在方便的时候回调（执行）参数。

22	Business Delegate	https://java-design-patterns.com/patterns/business-delegate/	业务代表模式（Business Delegate Pattern）用于对表示层和业务层解耦。它基本上是用来减少通信或对表示层代码中的业务层代码的远程查询功能。

三.设计模式详述

3.1 Simple Factory

3.1.1 实现 API 描述

3.1.1.1 设计思路

运动会上有给观众卖冰淇淋的。有香草冰淇淋、巧克力冰淇淋、抹茶冰淇淋。

店员使用工厂方法制作冰淇淋。只需接收冰淇淋名字，内部逻辑根据冰淇淋名字，创建对应对象并返回。

3.1.1.2 API 实现

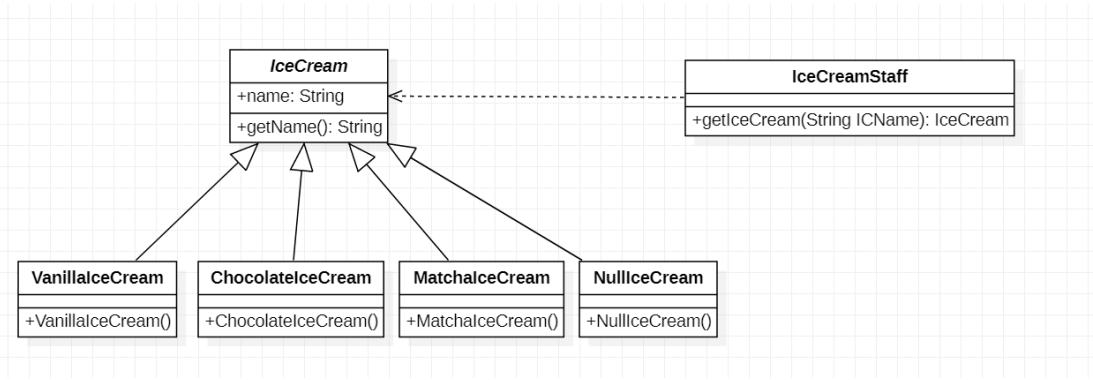
IceCreamStaff 拥有 getIceCream 方法，用于接收冰淇淋名，返回冰淇淋对象。

IceCream 为抽象类，子类有 VanillaIceCream，ChocolateIceCream，MatchaIceCream，NullIceCream，可共用 IceCream 引用。

3.1.1.3 函数功能对照表

函数名	作用
getName():String	获取冰淇淋名
getIceCream(String ICName):IceCream	获取输入冰淇淋名，获取对应的冰淇淋对象

3.1.2 类图



3.1.3 优缺点分析

（1）优点：

A. 工厂类包含必要的逻辑判断，可以决定在什么时候创建哪一个产品的实例。客户端可以免除直接创建产品对象的职责，很方便的创建出相应的产品。

B. 工厂和产品的职责区分明确。

C. 客户端无需知道所创建具体产品的类名，只需知道参数即可。

D. 也可以引入配置文件，在不修改客户端代码的情况下更换和添加新的具体产品类。

（2）缺点：

A. 简单工厂模式的工厂类单一，负责所有产品的创建，职责过重，一旦异常，整个系统将受影响。且工厂类代码会非常臃肿，违背高聚合原则。

B. 使用简单工厂模式会增加系统中类的个数（引入新的工厂类），增加系统的复杂度和理解难度

C. 系统扩展困难，一旦增加新产品不得不修改工厂逻辑，在产品类型较多时，可能造成逻辑过于复杂

D. 简单工厂模式使用了 static 工厂方法，造成工厂角色无法形成基于继承的等级结构。

3.2 Factory Method

3.2.1 实现 API 描述

3.2.1.1 设计思路

用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。它具有很强的灵活性，对于新产品的创建，只需多写一个相应的工厂类。这是典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

运动项目记分员需要为比赛选手创建成绩单，并收录好。创建成绩单采用工厂方法，高层模型不需要知道成绩单具体内容，即可使用 createResult 方法获取成绩单，进行相关操作。具体是什么运动项目的成绩单，成绩单记录的内容与打印方式，由子类（不同运动项目的成绩单）实现。

3.2.1.2 API 实现

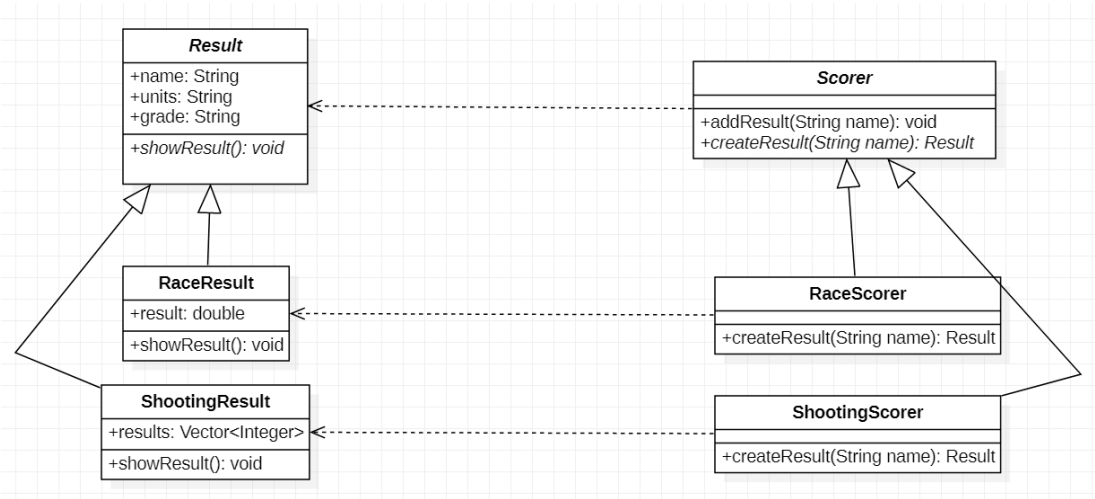
抽象类有 Result，Scorer，具体项目的 Result 和 Scorer 将从它俩继承。

每增加一个项目，只需增加两个抽象类的子类，使用抽象类的指针引用即可，非常之 OCP！

3.2.1.3 函数功能对照表

函数名	作用
showResult():void	打印成绩单
addResult(String name):void	为某运动员创建成绩单并添加至统计表
createResult(String name):Result	为某运动员创建成绩单

3.2.2 类图



3.2.3 优缺点分析

（1）优点：

a. 用户只需要知道具体工厂的名称就可得到所要的产品，无须知道产品的具体创建过程。

b. 灵活性增强，对于新产品的创建，只需多写一个相应的工厂类。

c. 典型的解耦框架。高层模块只需要知道产品的抽象类，无须关心其他实现类，满足迪米特法则、依赖倒置原则和里氏替换原则。

（2）缺点：

a. 类的个数容易过多，增加复杂度

b. 增加了系统的抽象性和理解难度

c. 抽象产品只能生产一种产品，此弊端可使用抽象工厂模式解决。

3.3 Abstract Factory

3.3.1 实现 API 描述

3.3.1.1 设计思路

运动项目记分员需要为男女子比赛选手创建成绩单。但男女子成绩单在成绩等级划分上有所区别，这里设定男子运动员有五级评定，女子运动员有四级评定。将这两种成绩单视为不同产品，属于同一个产品族。

抽象工厂定义每一个子类工厂（运动项目记分员）都可以生产男子成绩单、女子成绩单，具体成绩单由成绩单子类实现，而在高层模型中，仅使用抽象类（父类）“成绩单”来引用。

3.3.1.2 API 实现

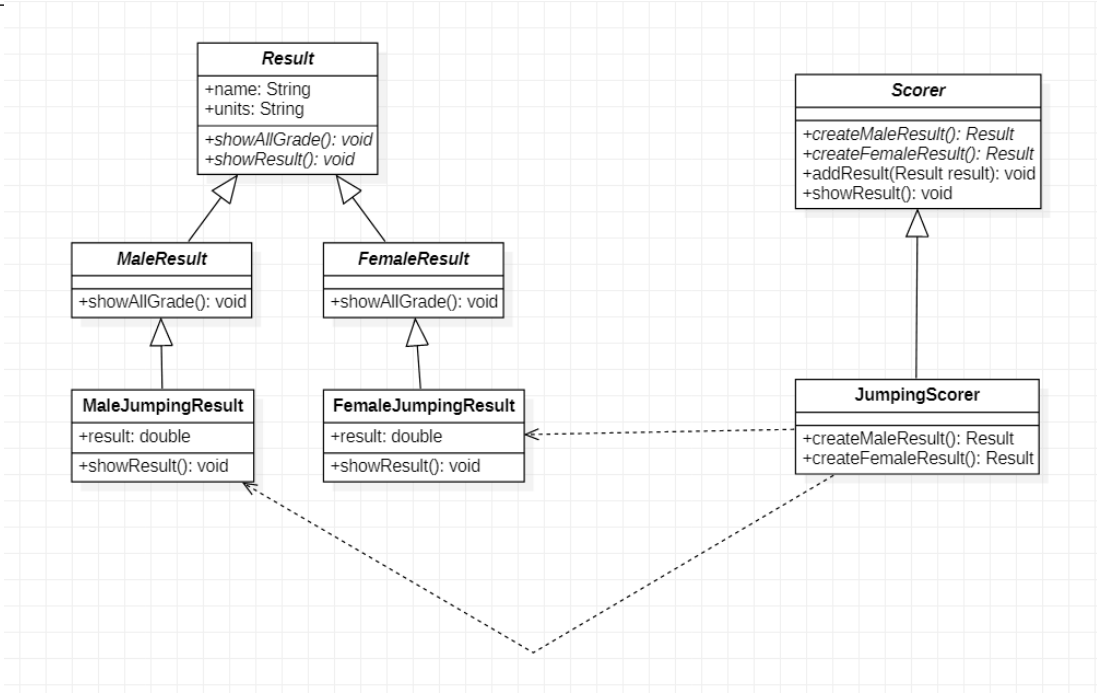
两个抽象类：Result，Scorer。具体项目的 Result 和 Scorer 从它俩继承。

每个项目的 Scorer 都需要实现 createMaleResult 和 createFemaleResult，这是一个产品结构。

3.3.1.3 函数功能对照表

函数名	作用
createFemaleResult(String name):Result	创建女子成绩单
createMaleResult(String name):Result	创建男子成绩单
addResult(Result result):void	添加成绩单
showResult():void	打印成绩单

3.3.2 类图



3.3.3 优缺点分析

（1）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。

当需要产品族时，抽象工厂可以保证客户端始终只使用同一个产品的产品组。

抽象工厂增强了程序的可扩展性，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。

（2）缺点：

其缺点是当产品族中需要增加一个新的产品时，所有的工厂类都需要进行修改。增加了系统的抽象性和理解难度。

3.4 Builder

3.4.1 实现 API 描述

3.4.1.1 设计思路

建造者模式可以将复杂事物创建的过程抽象出来，该抽象的不同实现方式不同，创建出的对象也不同。

本场景中使用建造者模式建造比赛场地，抽象父类的方法一致为建造场馆不同的部分，针对特定类型场馆实现时建造特定种类的部分。StadiumBuilder 包含创建场馆各个子区域的抽象方法，具体建造类实现了抽象建造者的建造方法，监工类调用建造者中的方法完成复杂场馆的创建。调用建造者进行建造的过程中建造步骤和传参类型一致，不同场馆实现细节封装在具体建造类中。

具体地，代码中实现了经典建造者和变种建造者两种建造模式：经典模式下建造出的同种场馆完全一致，无法传入参数自定义属性；变种建造者模式实现了链式调用传入参数，建造的每种场馆可以自行控制细节实现。

3.4.1.2 API 实现

StadiumBuilder 是抽象建造者类，FlyingVenueBuilder、RacingTrackBuilder 和 SwimmingPoolBuilder 是具体建造类，StadiumDirector 是建造监工类。

使用者先实例化 StadiumDirector 和所需类型的 StadiumBuilder，然后调用 setStadiumBuilder 方法将要建造的场馆指定给 StadiumDirector，再调用 buildNewStadium 建造场馆。使用变种模式时可以链式调用 CompetitionAreaCount 和 SpectatorAreaCount 设置比赛区和观众席数量。

3.4.1.3 函数功能对照表

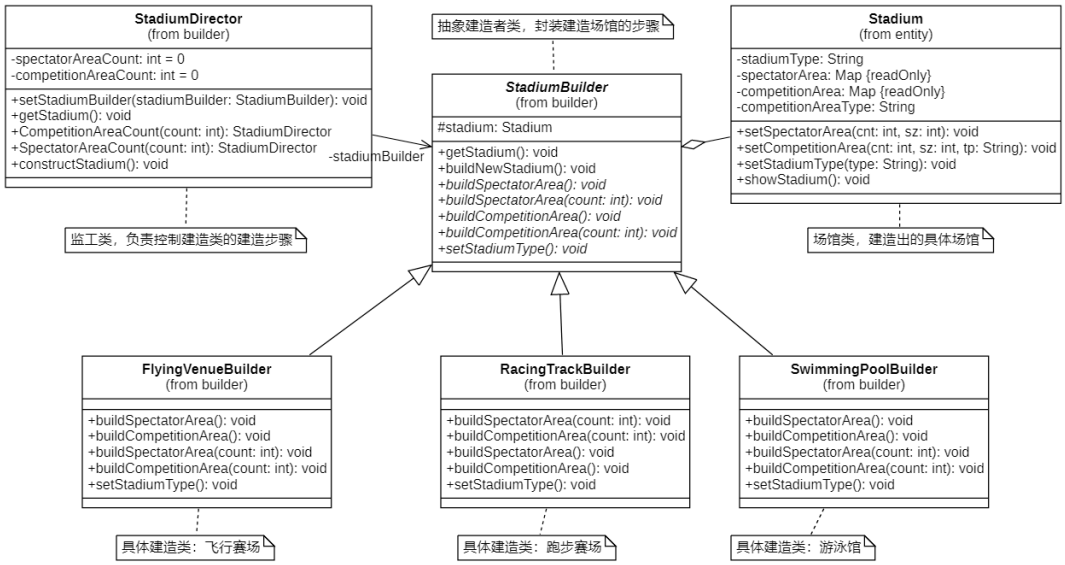
1. 经典模式

函数名	作用
setStadiumBuilder(StadiumBuilder):void	给监工类设置建造的具体场馆
buildNewStadium():void	创建新场馆
setStadiumType():void	设置场馆类型
buildCompetitionArea():void	建造竞赛区
buildSpectatorArea():void	建造观众席
getStadium():void	展示建造好的场馆

2. 变种模式

函数名	作用
setStadiumBuilder(StadiumBuilder):void	给监工类设置建造的具体场馆
buildNewStadium():void	创建新场馆
setStadiumType():void	设置场馆类型
CompetitionAreaCount(int count):StadiumDirector	设置竞赛区数量
SpectatorAreaCount(int count):StadiumDirector	设置观众席数量
buildCompetitionArea():void	建造竞赛区
buildSpectatorArea():void	建造观众席
getStadium():void	展示建造好的场馆

3.4.2 类图



3.4.3 优缺点分析

（1）优点：

使用 builder 时设置每个属性后返回本身，采用链接的编程风格，使用起来更加方便快捷。每次设置属性时都是具名的方法操作，对开发人员来说更容易使用；不需要给宿主类增加相应的 set 方法，即宿主类在创建之后是不可修改的，在一些安全模式比较高的程序中十分重要。

（2）缺点：

在使用构建器的时候，必须利用另外一个类来实例化另外一个类，会使创建的时候创建更多的对象，损耗一定的性能。

3.5 Prototype

3.5.1 实现 API 描述

3.5.1.1 设计思路

原型模式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。为了更快的印制奖状上的文字，采用原型模式直接 clone()批量生产奖状，可以逃避构造函数的约束并提高性能。同时，在过程中对姓名、运动项目、奖项等级等按需进行替换，以高效创建对象。

具体地，代码里实现了深克隆和浅克隆两种克隆模式。

浅克隆：创建一个新对象，新对象的属性和原来对象完全相同，对于非基本类型属性，仍指向原有属性所指向的对象的内存地址。

深克隆：创建一个新对象，属性中引用的其他对象也会被克隆，不再指向原有对象地址。

3.5.1.2 API 实现

Certificate 是具体原型类，Java 中的 Cloneable 接口时是抽象原型类。Certificate 实现了 Cloneable 接口，重写了 clone()方法用于批量复制奖状。访问类可以通过调用 clone()方法复制多张奖状，并通过 Certificate 提供的三个函数修改奖状的具体内容，从而达到批量生产奖状（奖状本身有很多相同的内容和属性可以通过 clone()直接完成）并且适当根据需要修改部分属性的目的。

3.5.1.3 函数功能对照表

1. 浅克隆

函数名	作用
setName(String name):void	修改奖状获得者姓名
setPrizeLevel(AwardLevel prizeLevel):void	修改奖状等级
setSportsType(String sportsType):void	修改运动类型
printCertificate():void	打印奖状
fitPrize(AwardLevel awardLevel):Prize	适配奖状等级和奖品类型
clone():Object	克隆函数，实现浅克隆

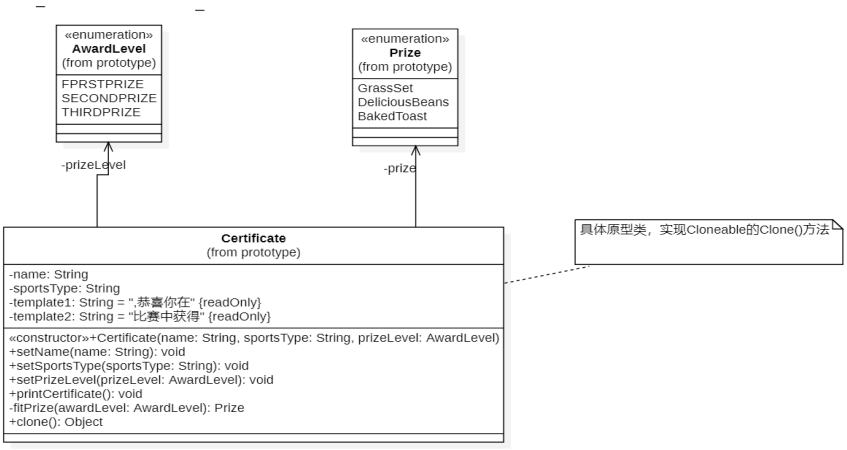
2. 深克隆

函数名	作用
setName(String name):void	修改奖状获得者姓名
setPrizeLevel(AwardLevel prizeLevel):void	修改奖状等级
setSportsType(String sportsType):void	修改运动类型

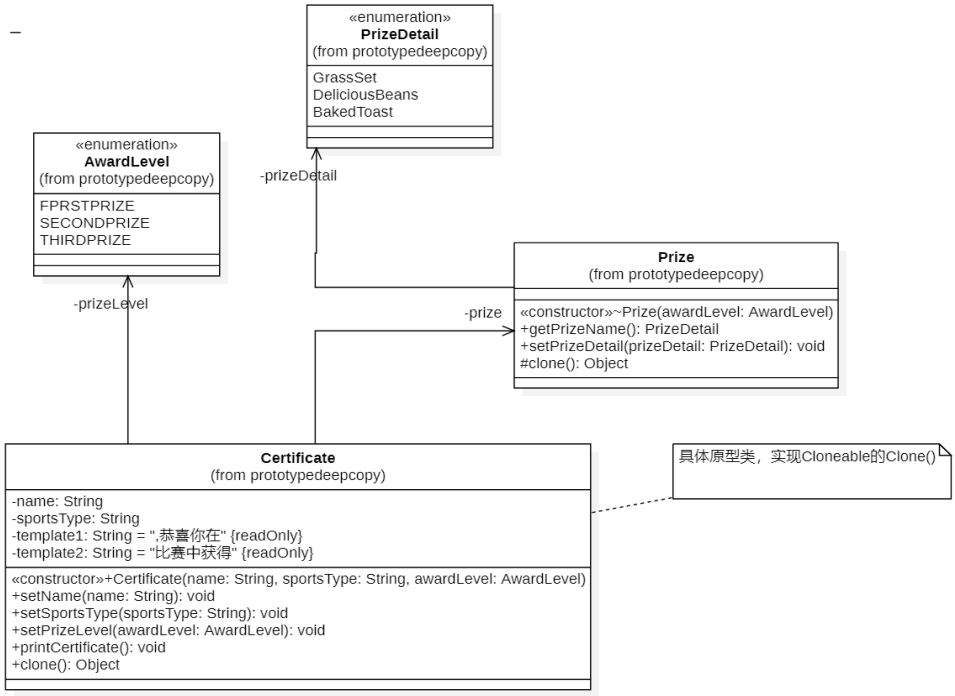
printCertificate():void	打印奖状
clone():Object	克隆函数，实现深克隆

3.5.2 类图

a. 浅克隆



b. 深克隆



3.5.3 优缺点分析

（1）优点：

Java 自带的原型模式基于内存二进制流的复制，在性能上比直接 new 一个对象更加优良。

（2）缺点：

当实现深克隆时，需要编写较为复杂的代码，而且当对象之间存在多重嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来会比较麻烦。因此，深克隆、浅克隆需要运用得当。

3.6 Singleton

3.6.1 实现 API 描述

3.6.1.1 设计思路

单例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。

使用单例模式构建运动会的主办方。由于规定一次运动会只能有唯一的主办方，采取单例模式来构建唯一的主办方实例。

3.6.1.2 API 实现

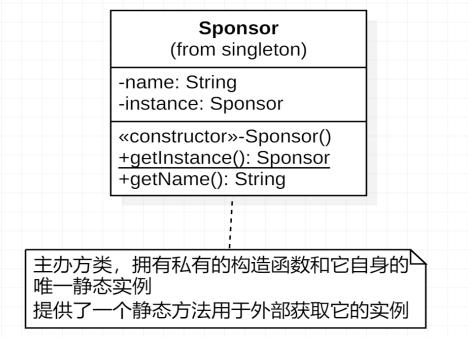
Sponsor 主办方类，其构造函数为私有函数，保证其不会在外部被实例化。外部通过 getInstance() 静态方法来获取。

其有一个 getName() 的公共方法，用于获取主办方的名称。

3.6.1.3 函数功能对照表

函数名	作用
getInstance(): Sponsor	获取 Sponsor 实例
getName(): String	获取 Sponsor 的名称属性

3.6.2 类图



3.6.3 优缺点分析

（1）优点：

唯一的实例，减少内存开销，避免了反复创建和销毁实例。

（2）缺点：

无法被继承，没有接口。实例化由外部完成而不是由其内部。

3.7 Adapter

3.7.1 实现 API 描述

3.7.1.1 设计思路

适配器模式用于整合不同来源的现存接口，现存接口可能因为提供方不同而有着不同的函数名，调用方式，利用一个中间的适配器，对于外部用户而言，只需要调用一个统一的接口，即可完成不同的接口的调用。

在我们的实现场景中，在美食广场中，动物们需要支付购买食物，adapter 接口主要用于进行移动支付，对外提供了 login 和 pay 两个统一的接口,其内部实现整合了两个来自"alibaba"和“wechat”的支付接口（均是模拟接口，自己实现的），两个支付接口具有不同的命名风格，但是对用户暴露的是一个统一的接口。

3.7.1.2 API 实现

Payment 为一个接口 Interface，其内部主要有两个函数 login 以及 pay，这两个函数即为对用户提供的两个功能。

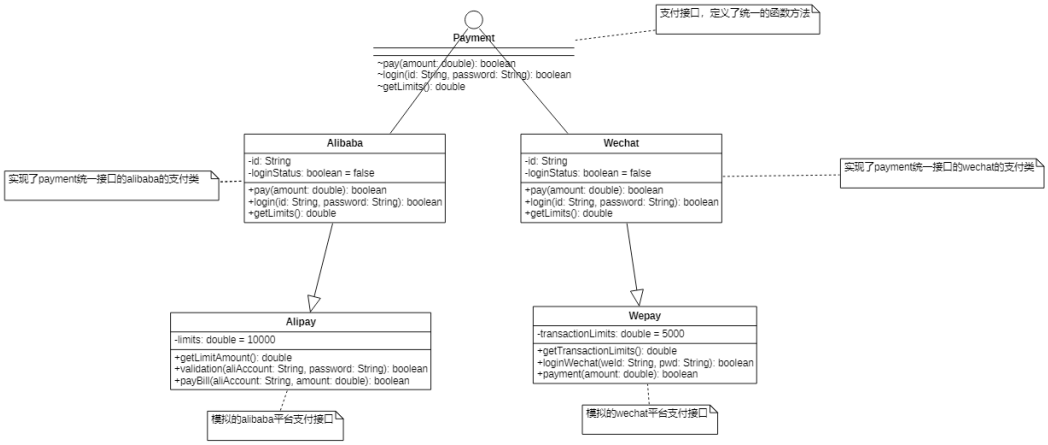
AliPay 和 Wepay 是模拟的两个由支付宝和微信两个支付平台提供的不同接口，两个接口具有上述所写的两个功能，也就是支付以及登录，不过两个接口命名风格差距较大，用户调用难度大。

Alibaba 和 Wechat 两个是实现了 Payment 接口的两个类，两个类中均实现 Payment 接口中的两个函数，在两个函数中，分别调用了 Alipay 和 Wepay 两个平台提供系统的接口，这样就实现了接口的统一。

3.7.1.3 函数功能对照表

函数名	作用
pay(double amount):boolean	pay 是 Payment 接口中对外提供的支付函数
login(String id,String password):boolean	login 是 Payment 接口中对外提供的登录函数
getTransactionLimits():double	获得微信支付单笔支付的上限
loginWechat(String weId,String pwd):boolean	Wepay 提供的用户登录接口
payment(double amount):boolean	Wepay 提供的用户支付接口
getLimitAmount():double	获得支付宝单笔支付的上限
validation(String aliAccount,String password):boolean	Alipay 提供的用户登录接口
payBill(String aliAccount,double amount)	Alipay 提供的用户支付接口

3.7.2 类图



3.7.3 优缺点分析

（1）优点：

可以统一系统的接口，使得调用者和外部系统提供的现有接口解耦。对于程序员而言，省去了很多重新实现接口的时间。在长期过程而看，适配器也满足了开闭原则。

（2）缺点：

增加了代码的可读性，使得代码变得复杂，过多的适配器使得系统代码变得凌乱。同时也增加了系统本身的复杂度。

3.8 Bridge

3.8.1 实现 API 描述

3.8.1.1 设计思路

桥接模式（Bridge Pattern）适用于将抽象部分与它的实现部分分离，使它们都以独立地变化。当某种抽象部分未来新增类别较多，应该选用桥接模式将其独立出来。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。

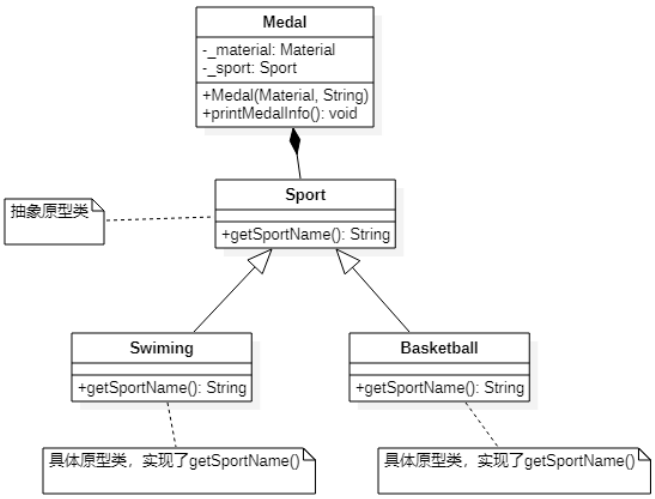
3.8.1.2 API 实现

在动物运动会这一场景下，奖牌（Model）应该表示为各种不同的运动项目，且运动项目可能在未来会新增许多种类，因此采用桥接模式。将运动项目抽象为 Sport 抽象类，运动项目的实例继承该抽象类并实现相应函数，奖牌类中包含 Sport 类成员现在包含 Swimming 类和 Basketball 类，实现了抽象和实现可变维度的耦合度。

3.8.1.3 函数功能对照表

函数名	作用
getSportName()：String	获取运动的名称
printMedalInfo()：void	输出奖牌信息

3.8.2 类图



3.8.3 优缺点分析

（1）优点：

- 抽象与实现分离，扩展能力强
- 符合开闭原则
- 符合合成复用原则
- 其实现细节对客户透明

（2）缺点：

由于聚合关系建立在抽象层，要求开发者针对抽象化进行设计与编程，能正确地识别出系统中两个独立变化的维度，这增加了系统的理解与设计难度。

3.9 Composite

3.9.1 实现 API 描述

3.9.1.1 设计思路

将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

抽象类 EquipmentRoom 代表体育器材的存放处。由 Locker 储物柜和 Equipment 体育器材继承，储物柜中可以存放储物柜或者体育器材。可以调用函数打印存放路径。体育器材可以设置价值，调用函数每个储物柜会计算自己柜子下所有的体育器材的价值

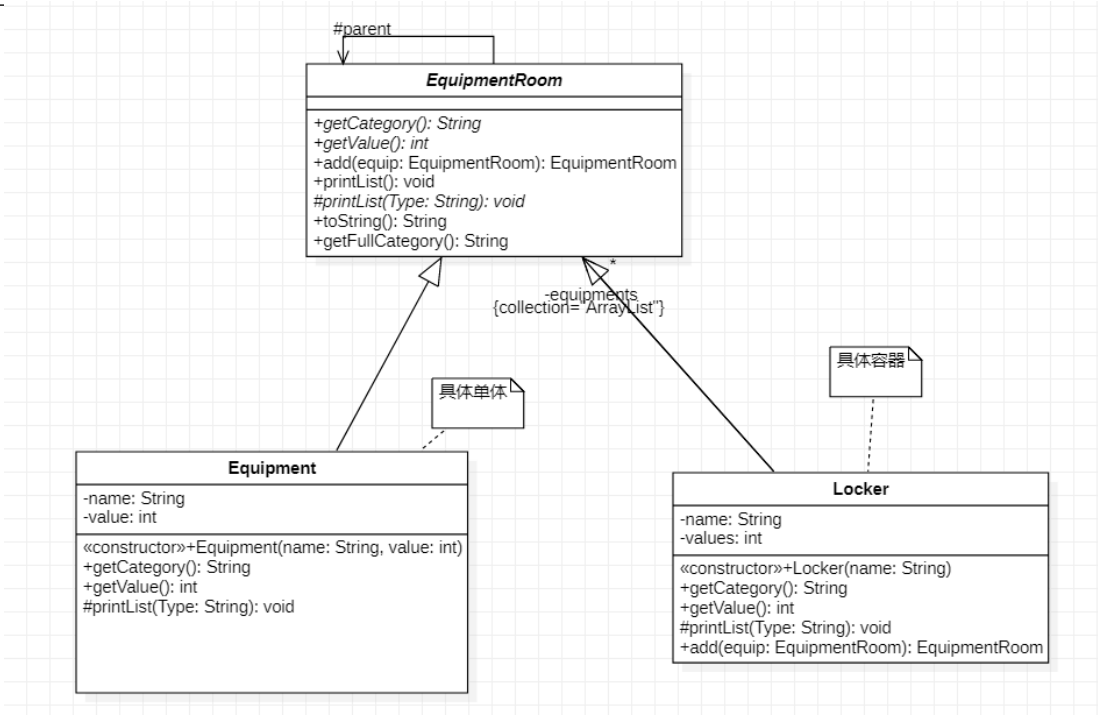
3.9.1.2 API 实现

EquipmentRoom 作为抽象类被 Equip 和 Locker 继承，Locker 调用 add 函数来添加 Equip 或者 Locker.两个子类都可以使用 printList 来打印对应的信息。

3.9.1.3 函数功能对照表

函数名	作用
add(EquipmentRoom equip):EquipmentRoom	向储物柜中添加储物柜或者体育器材返回抽象类可以进行链式操作。
getFullCategory():String	输出储物柜或者体育器材的存放路径
toString():String	重写输出展示物品名称和对应价值
printList():void	储物柜重写实现打印储物柜下所有的储物柜和体育器材，体育器材重写实现打印自己的名称和价值。
getValue():int	储物柜重写实现打印储物柜下所有体育器材的价值总和，体育器材重写实现打印自己的价值。

3.9.2 类图



3.9.3 优缺点分析

（1）优点：

- 对外表现完全相同的接口，不需要去具体了解整体和部分的区别。
- 在使用上更加方便

（2）缺点：

对于体育器材无法实现的添加物品操作返回空值，使用的时候需要进行判断

3.10 Decorator

3.10.1 实现 API 描述

3.10.1.1 设计思路

装饰器模式（Decorator Pattern）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。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。

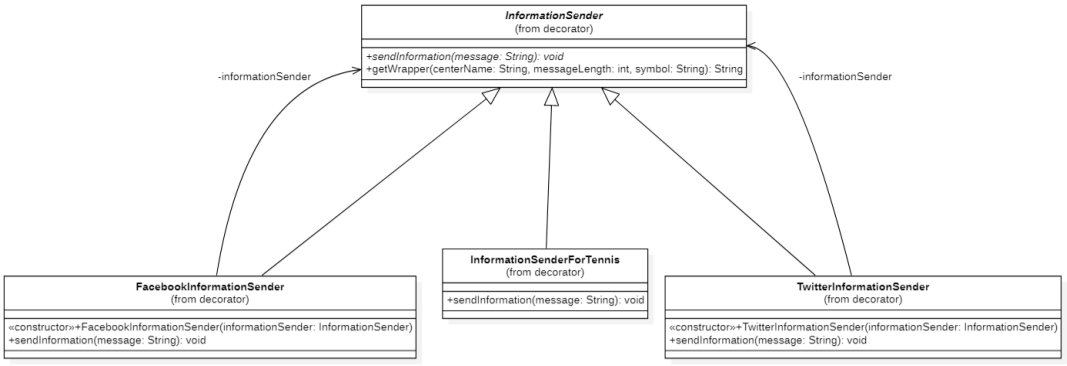
3.10.1.2 API 实现

本设计模式使用赛事消息经由不同平台发布这个场景实现。通过创建原始的消息发布对象，我们可以发送基本的赛事消息。如果想经由其他平台发布赛事消息，如 Facebook，可以新建 Facebook 消息发布对象来装饰原始消息发布对象来实现通过不同平台发布赛事消息。

3.10.1.2 函数功能对照表

函数名	作用
InformationSender.sendInformation(String message):void	发送未经修饰的信息
FacebookInformationSender.sendInformation(String message):void TwitterInformationSender.sendInformation(String message):void	在原信息基础上进行修饰并发送

3.10.2 类图



3.10.3 优缺点分析

（1）优点：

- 装饰器是继承的有力补充，比继承灵活，在不改变原有对象的情况下，动态的给一个对象扩展功能，即插即用
- 通过使用不用装饰类及这些装饰类的排列组合，可以实现不同效果
- 装饰器模式完全遵守开闭原则

（2）缺点：

装饰模式会增加许多子类，过度使用会增加程序得复杂性。

3.11 Façade

3.11.1 实现 API 描述

3.11.1.1 设计思路

外观设计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，外部应用程序不用关心内部子系统的具体细节，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。

外观（Facade）模式包含以下主要角色。

- 外观（Facade）角色：为多个子系统对外提供一个共同的接口。
- 子系统（Sub System）角色：实现系统的部分功能，客户可以通过外观角色访问它。
- 客户（Client）角色：通过一个外观角色访问各个子系统的功能

本项目中将该设计模式应用于体育赛事的报名系统，各个运动项目的报名分别为子系统，为他们设置报名表这一外观角色，为各项体育赛事报名提供统一的接口，动物是使用报名表进行报名的客户角色。

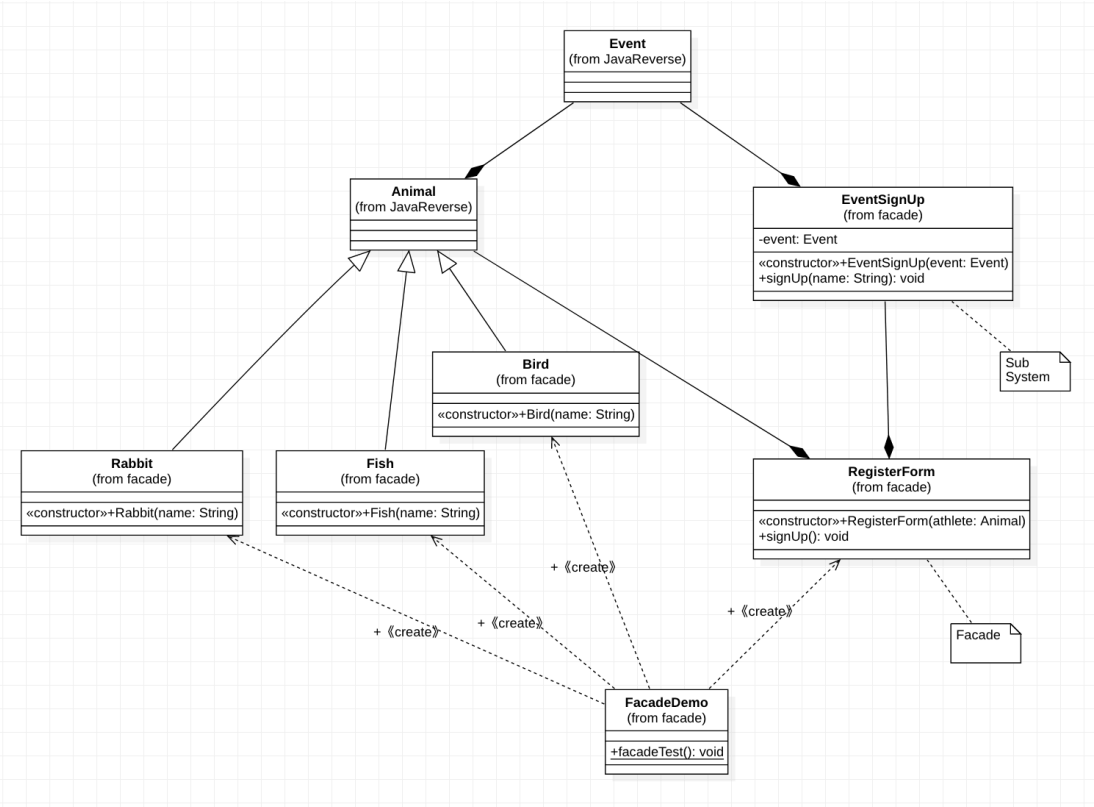
3.11.1.2 API 实现

本用例中，各项体育赛事的报名系统（EventSignUp）设定为模式中的各个子系统，通过设置报名表(RegisterForm）这一外观角色，为各项体育赛事报名(EventSignUp）提供统一的接口，动物（Animal）们是使用外观角色（Facade）进行报名的客户角色。动物类包含一个 eventToSignUP 属性，报名表（RegisterForm）根据动物(Animal）的该属性，选择对应运动项目的子系统，协助进行报名。

3.11.1.3 函数功能对照表

函数名	作用
EventSignUp.signUp(String name) -> void	处理 Animal 的运动项目报名请求。
RegisterForm.signUp() -> void	根据 Animal 的 eventToSignUp 属性选择运动项目报名子系统。

3.11.2 类图



3.11.3 优缺点分析

外观（Facade）模式是“迪米特法则”的典型应用。

（1）优点：

- 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。
- 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。
- 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。

（2）缺点：

- 不能很好地限制客户使用子系统类，很容易带来未知风险。
- 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”。

3.12 Flyweight

3.12.1 实现 API 描述

3.12.1.1 设计思路

享元模式（Flyweight Pattern）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。这种类型的设计模式属于结构型模式，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。

该模式的实现场景是为比赛分配裁判， 当新开一个比赛并需要分配裁判时，会优先从已有的该类型比赛裁判中寻找空闲的裁判， 如果没有则需要招聘（新建）裁判并放入队列

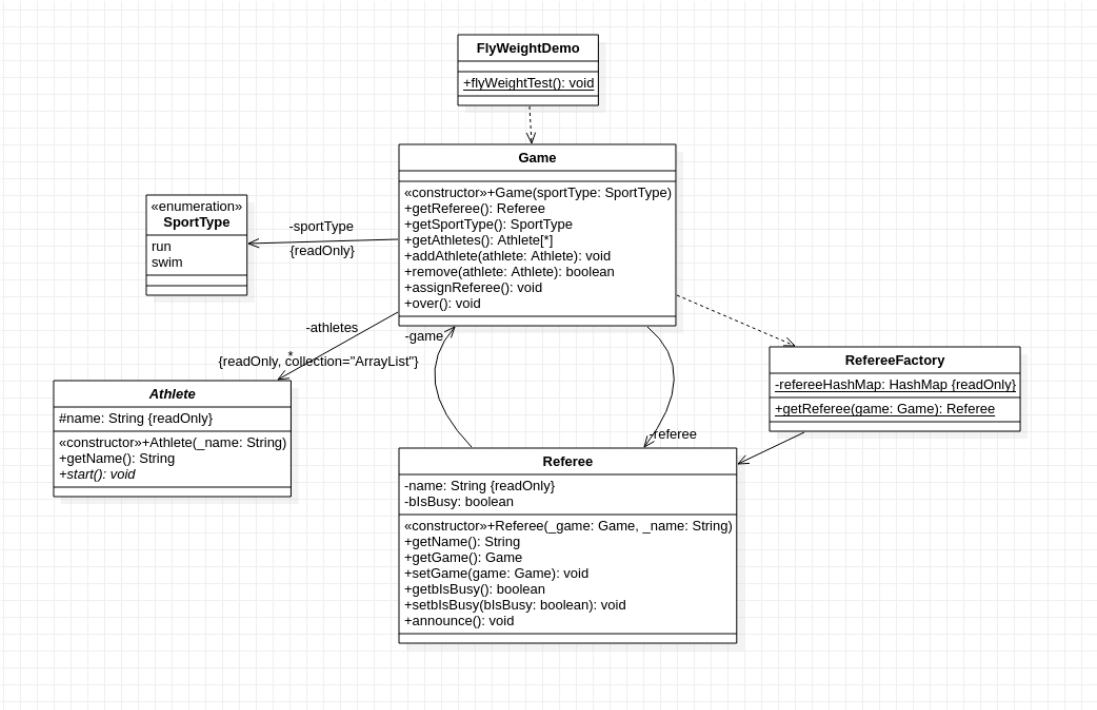
3.12.1.2 API 实现

RefereeFactory 中有一个 hashmap 存储各种比赛类型目前已有的裁判， 当我们创建比赛并尝试分配裁判时， 会调用 **RefereeFactory** 的 getReferee 方法，先去遍历上述 map 中对应的比赛类型的所有裁判， 寻找是否有目前空闲的裁判， 如果有就直接把裁判分配给比赛， 并把裁判状态设置为忙碌， 如果没有就新建一个裁判放入队列并分配给比赛

3.12.1.3 函数功能对照表

函数名	作用
Game.Game(SportType type):	创建指定类型的比赛
Game.assignReferee(): void	为某一场比赛分配裁判，将裁判状态置为忙碌
Game.over(): void	结束比赛，将分配的裁判状态置为空闲

3.12.2 类图



3.12.3 优缺点分析

（1）优点：

- 相同对象只要保存一份，这降低了系统中对象的数量，从而降低了系统中细粒度对象给内存带来的压力。

（2）缺点：

- 为了使对象可以共享，需要将一些不能共享的状态外部化，这将增加程序的复杂性。
- 读取享元模式的外部状态会使得运行时间稍微变长。

3.13 Private Class Data

3.13.1 实现 API 描述

3.13.1.1 设计思路

私有类数据模式（Private Class Data）封装类的初始化数据，控制对类的属性的更改，并保持类数据与使用数据的方法间的隔离。当类的初始化数据是一次性不可修改的数据、需要控制对初始化数据的更改以及需要预防对初始化数据的不必要的更改时可以选用私有类数据。可以减少类对外界暴露的属性，实现对初始化数据的封装。

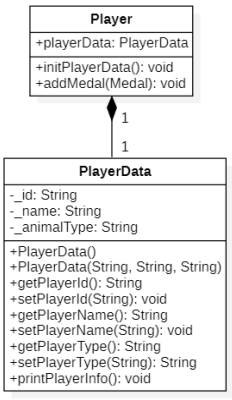
3.13.1.1 设计思路

动物运动会中，运动员的注册信息，如名称_name，动物种类_animalType，运动员编号_id 这类信息是在注册为运动员时就需要生成且之后不应轻易改动的。因此选用私有类数据模式来封装运动员注册时的数据，即 PlayerData 类。Player 类可以调用 PlayerData 成员变量进行访问和更改 PlayerData 类信息。

3.13.1.3 函数功能对照表

函数名	作用
getPlayerId():String	获取运动员编号
setPlayerId():void	更改运动员编号
getPlayerName():String	获取运动员姓名
setPlayerName():void	更改运动员姓名
getPlayerType():String	获取运动员动物种类
setPlayerType():void	更改运动员动物种类
printPlayerInfo():void	打印运动员注册信息

3.13.2 类图



3.13.3 优缺点分析

（1）优点：

- 减少类对外暴露的属性。
- 从类中移除了对数据的写权限。

（2）缺点：

- 使类更加复杂，使用不当调用代码可能冗余。

3.14 Proxy

3.14.1 实现 API 描述

3.14.1.1 设计思路

在代理模式中，一个类代表另一个类的功能，属于结构型模式。代理类为其他对象提供一种代理以控制对这个对象的访问。

本设计模式展示的场景为远程采访场景，记者要采访运动员，但整个采访环节很麻烦，运动员不能浪费时间一个人完成，以及考虑到疫情期间减少接触外来记者，由代理把记者的问题录制下来，由代理准备不必运动员亲自参与的采访环节，运动员只需要亲自回答问题即可。

使用代理类控制运动员的访问，以隔离并保护采访方法，只将关键逻辑在真实采访中实现，而其他逻辑由代理类完成。

3.14.1.2 API 实现

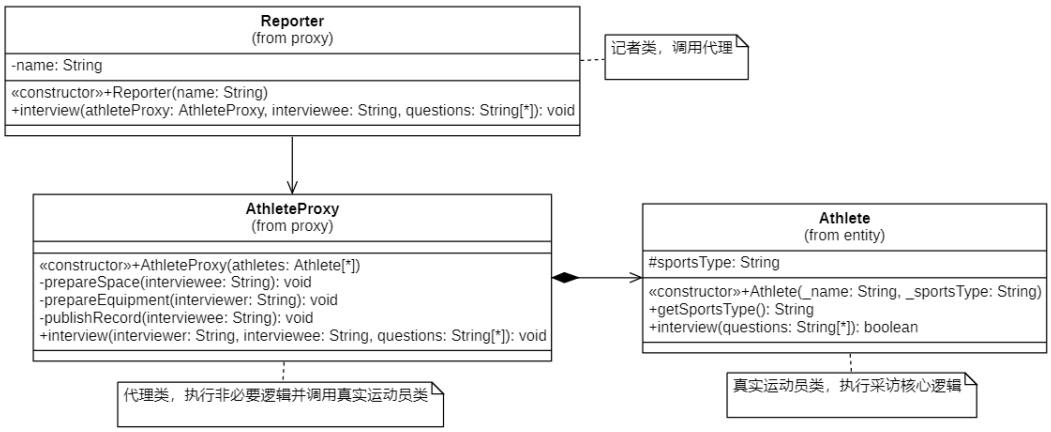
Athlete 是真实运动员类，在本场景中只负责完成采访的核心逻辑——回答问题。AthleteProxy是采访代理类，负责对外提供采访的接口，完成不必运动员亲自完成的逻辑。Reporter是记者类，可以调用采访代理类进行采访。

使用者先实例化 Reporter、Athlete 列表和问题列表，再使用 Athlete 列表构造 AthleteProxy，然后记者类调用 interview 即可完成 AthleteProxy 中封装的采访。

3.14.1.3 函数功能对照表

函数名	作用
interview(AthleteProxy athleteProxy, String interviewee, String[] questions):void	记者调用代理采访类采访
interview(String interviewer, String interviewee, String[] questions) :void	代理采访类调用真实采访类进行采访
prepareSpace(String interviewee):void	准备采访地点
prepareEquipment(String interviewer):void	准备采访设备
interview(questions):void	真实记者类录制采访
publishRecord(String interviewee):void	发布采访

3.14.2 类图



3.14.3 优缺点分析

（1）优点：

代理模式在客户端与目标对象之间起到一个中介作用和保护目标对象的作用，通过代理类访问真实类，一些非法、不合理的调用将被代理类拦截，如采访不正确名称的运动员。将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性。本例中代理模式还完成了核心逻辑与非核心逻辑的分离，让代码逻辑更加清晰。

（2）缺点：

代理模式会造成系统设计中类的数量增加，在客户端和目标对象之间增加一个代理对象，会造成请求处理速度变慢。代理模式也不可避免地增加了系统的复杂度。

3.15 Chain of Responsibility

3.15.1 实现 API 描述

3.15.1.1 设计思路

责任链模式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。在这种模式中，通常每个接收者都包含对另一个接收者的引用。如果一个对象不能处理该请求，那么它会把相同的请求传给下一个接收者，依此类推。

用责任链模式实现对于争议的判罚：如果争议出现时，主裁判不能给出判罚，则参考助理裁判的意见；若助理裁判仍不能判断，则参考机器裁判（如门线、鹰眼技术）的判罚。

3.15.1.2 API 实现

AbstractReferees 是抽象裁判类，拥有两个保护级别的属性：myJudgement 表示该裁判的判罚，nextReferee 表示下一位裁判，即责任链的下级。

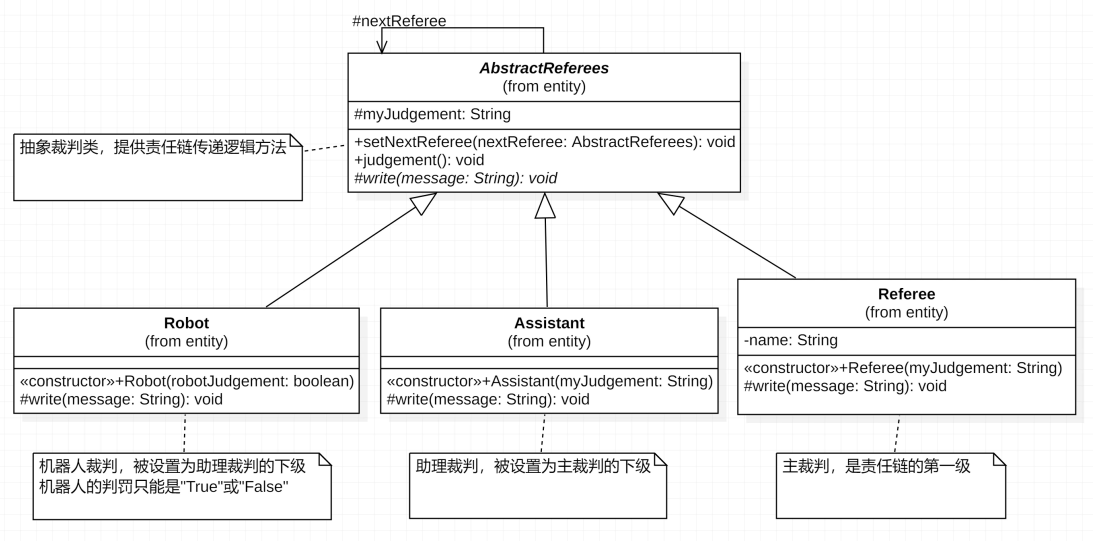
setNextReferee() 方法用于初始化 nextReferee 属性，指定下一位裁判。

judgement() 方法用于输出判罚，并且当该裁判的判罚为 "Unknown" 时，向责任链下级的裁判请求判罚。

3.15.1.3 函数功能对照表

函数名	作用
setNextReferee(): AbstractReferee	设置下一位裁判作为责任链下级
judgement(): void	进行判罚
write(): void	打印该裁判的判罚

3.15.2 类图



3.15.3 优缺点分析

（1）优点：

责任链的传递对于调用者可以是透明的。

可以动态调整或增删责任。

将请求的发送和接收者解耦。

（2）缺点：

对系统性能有所影响。

3.16 Command

3.16.1 实现 API 描述

3.16.1.1 设计思路

命令（Command）模式的定义如下：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。这样两者之间通过命令对象进行沟通，这样方便将命令对象进行储存、传递、调用、增加与管理。

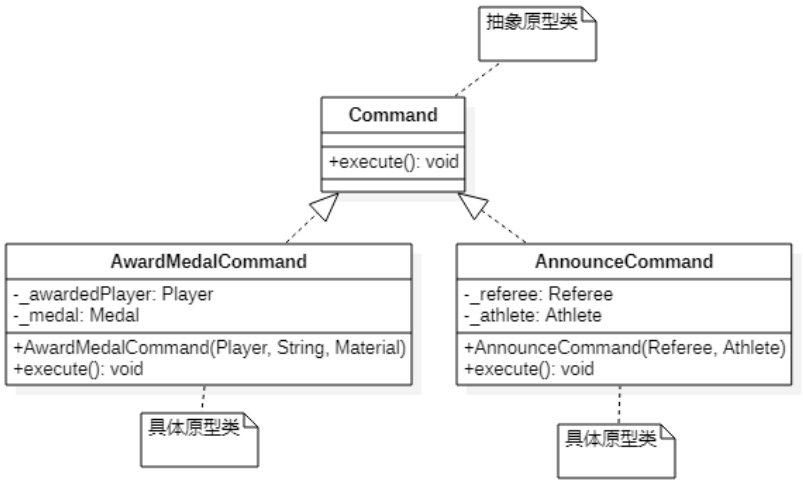
3.16.1.2 API 实现

动物运动会场景下，有许多种命令方式，现在选择实现主办方向运动员颁发奖牌和裁判向运动员发出起跑指令两种命令。AnnounceCommand 在按照构造函数创建实例后可以通过 execute（）函数进行调用

3.16.1.3 函数数功能对照表

函数名	作用
execute():void	执行该命令

3.16.2 类图



3.16.3 优缺点分析

（1）优点：

- 通过引入中间件（抽象接口）降低系统的耦合度。
- 扩展性良好，增加或删除命令非常方便。采用命令模式增加与删除命令不会影响其他类，且满足“开闭原则”。
- 可以实现宏命令。命令模式可以与组合模式结合，将多个命令装配成一个组合命令，即宏命令。

（2）缺点：

- 可能产生大量具体的命令类。因为每一个具体操作都需要设计一个具体命令类，这会增加系统的复杂性。

3.17 Interpreter

3.17.1 实现 API 描述

3.17.1.1 设计思路

解释器（Interpreter）模式，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。

项目中为实现该模式，模拟总场馆的餐厅中任务购买餐饭的场景，解释器模式提供了计算所点餐食的总价值的功能。定义了语法规则，并且在 MatchExpression 中实现了语法分析的功能，可以根据输入的合法表达式生产对应语法树，并最终给出总价。

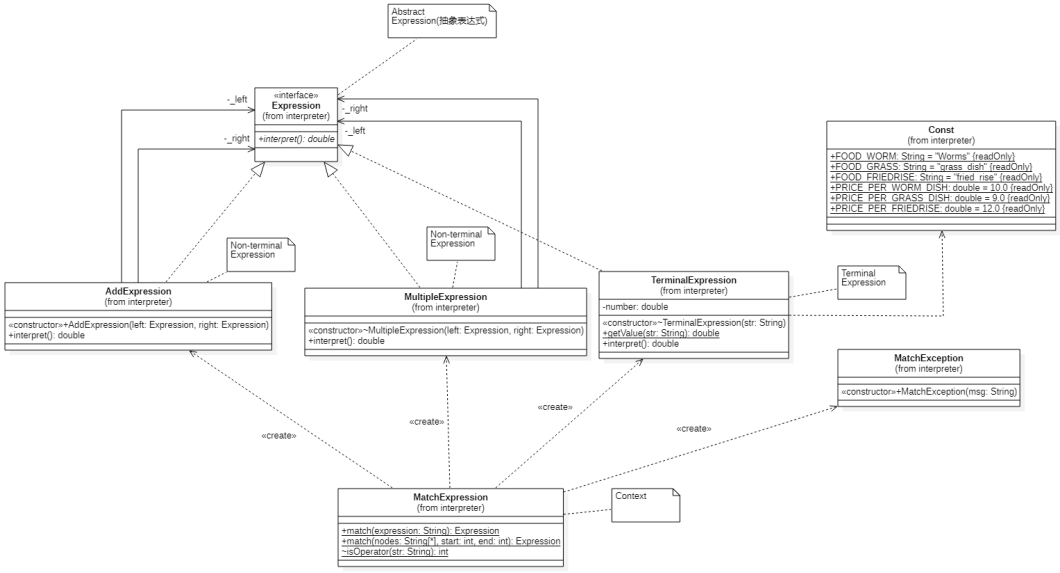
3.17.1.2 API 实现

定义了一个抽象表达式(Expression)接口，包含了解释方法 interpret()。定义了一个终结符(Terminal Expression)类，实现抽象表达式接口中的解释方法 interpret()，用于返回最终计算结果。定义了两个非终结符(Non-Terminal Expression)类，分别为 AddExpression 和 MultipleExpression，均是抽象表达式类的子类，实现了 interpret()方法，分别返回加法操作和乘法操作。MatchExpression 为环境类，完成对终结符表达式的初始化处理，并定义了 match(String expression)的方法调用表达式对象的解释方法来对该表达式进行分析解释。

3.17.1.3 函数功能对照表

函数名	作用
MatchExpression.match(String expression):Expression	接受 expression 表达式整体，将表达式每个部分拆分并记录在 nodes[]里
MatchExpression.match(String[] nodes,int start,int end):Expression	接受拆分后的表达式的元素 nodes[]，然后进行语法树的分析
AddExpression.interpret():double	完成该语法表达式的左边部分和右边部分的计算操作，这里为加法操作
MultipleExpression.interpret():double	完成该语法表达式的左边部分和右边部分的计算操作，这里为乘法操作
TerminalExpression.interpret():double	返回最终计算结果

3.17.2 类图



3.17.3 优缺点分析

（1）优点：

i. 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。

ii. 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。

（2）缺点：

i. 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。

ii. 可能会引起类膨胀。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。

iii. 可应用的场景比较少。在软件开发中，需要定义语言文法的应用实例非常少，所以这种模式很少被使用到。

3.18 Iterator

3.1.1 实现 API 描述

3.1.1.1 设计思路

提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。设计一个运动员类 SportMan，然后两个对应的集合类 ManCollection，对应实现的迭代器来遍历运动员集合.

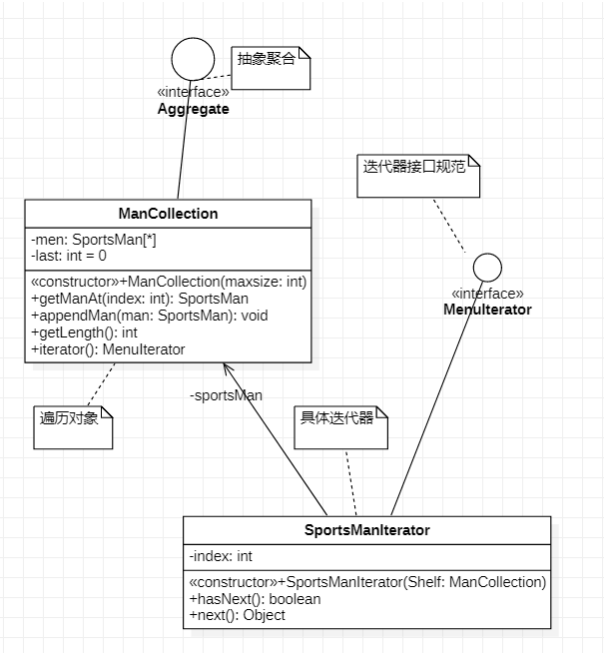
3.9.1.2 API 实现

ManCollection 调用 appendMan 来构成一个运动员序列，用 iterator 函数创建一个指针遍历集合，在 hasNext 为真的情况下来用 next 函数一个一个遍历集合中的元素

3.9.1.3 函数功能对照表

函数名	作用
hasNext():boolean	判断是否集合有下一个元素
next():Object	返回迭代器指向的下一个元素。
appendMan(SportsMan man)： void	增添一名运动员
iterator():MenuIterator	返回一个指向集合的迭代器

3.1.2 类图



3.1.3 优缺点分析

（1）优点：

- 它支持以不同的方式遍历一个聚合对象。
- 迭代器简化了聚合类。
- 在同一个聚合上可以有多个遍历。

- 在迭代器模式中，增加新的聚合类和迭代器类都很方便，无须修改原有代码。

（2）缺点：

由于迭代器模式将存储数据和遍历数据的职责分离，增加新的聚合类需要对应增加新的迭代器类，类的个数成对增加，这在一定程度上增加了系统的复杂性。

3.19 Mediator

3.19.1 实现 API 描述

3.19.1.1 设计思路

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

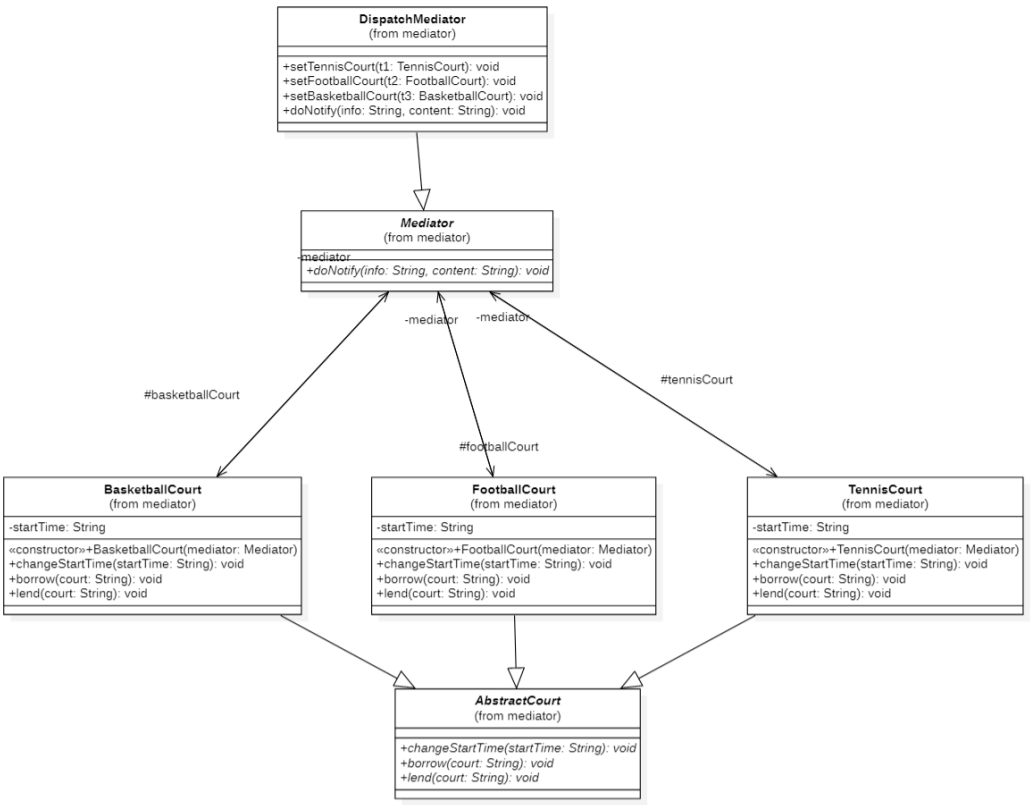
3.19.1.2 API 实现

本设计模式使用不同网球场之间的互相通讯这个场景实现。每一个网球场为一个对象，不同网球场之间的通讯（如借用设备），通过向消息中介(DispatchMediator)发送消息，由消息中介(DispatchMediator)再来处理并向其他球场发出相应请求。

3.19.1.3 函数功能对照表

函数名	作用
DispatchMediator.doNotify(String info, String content):void	接受传入中介者的信息并执行相应业务逻辑
FootballCourt.changeStartTime(String startTime):void BasketballCourt.changeStartTime(String startTime):void TennisCourt.changeStartTime(String startTime):void	调用 DispatchMediator.doNotify 以设置开始使用场地的时间，通过 DispatchMediator 通知其他场地调用者的状态
FootballCourt.changeStartTime(String court):void BasketballCourt.changeStartTime(String court):void TennisCourt.changeStartTime(String court):void	调用 DispatchMediator.doNotify 以改变开始使用场地的时间，通过 DispatchMediator 通知其他场地调用者的状态
FootballCourt.lend(String court):void BasketballCourt.lend(String court):void TennisCourt.lend(String court):void	调用 DispatchMediator.doNotify 以通知调用者要借用设备的场地

3.19.2 类图



3.19.3 优缺点分析

中介者模式是一种对象行为型模式。

（1）优点：

- 类之间各司其职，符合迪米特法则。
- 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。

（2）缺点：

中介者模式将原本多个对象直接的相互依赖变成了中介者和多个同事类的依赖关系。当同事类越多时，中介者就会越臃肿，变得复杂且难以维护。

3.20 Memento

3.20.1 实现 API 描述

3.20.1.1 设计思路

备忘录模式指的是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。

备忘录模式的主要角色如下：

a. 发起人（Originator）角色：记录当前时刻的内部状态信息，提供创建备忘录和恢复备忘录数据的功能，实现其他业务功能，它可以访问备忘录里的所有信息。

b. 备忘录（Memento）角色：负责存储发起人的内部状态，在需要的时候提供这些内部状态给发起人。

c. 管理者（Caretaker）角色：对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

在本用例当中，针对某一个运动项目的报名表(EventEntryForm)，考虑到报名参赛的运动员所属物种(Species)之间可能存在相互捕食的关系，为安全因素考虑，在运动员填写报名表(EventEntryForm)时，设计备忘录模式，用以避免在同一场比赛当中有相互捕食关系的动物同时参赛的情况。

3.20.1.2 API 实现

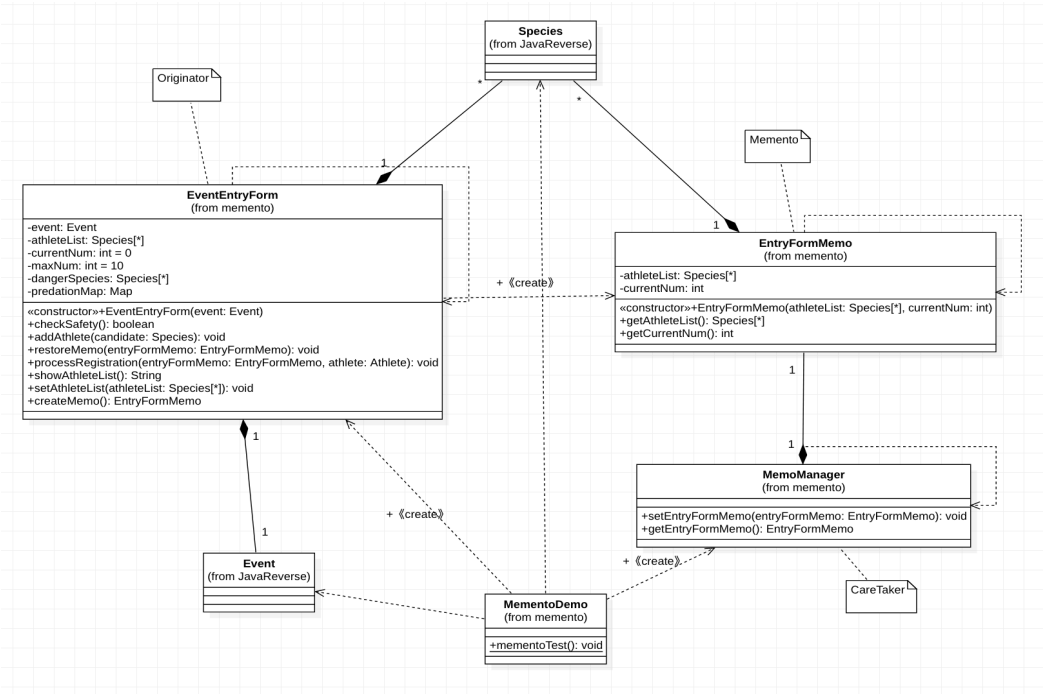
报名表(EventEntryForm)作为发起人（Originator）角色，每当运动员(Athlete)报名之前，报名表先将已报名运动员信息（这里指一个记录参赛运动员物种的数组athleteList）存到备忘录(EntryFormMemo)当中，存完之后将欲报名运动员填到数组当中去，并判断数组当中是否有不安全因素存在（捕食关系物种同时存在），如果没有则不做处理，报名成功；如果有则利用管理者类(MemoManager)将从备忘录(EntryFormMemo)中恢复原数组内容，并通知“不安全，报名失败”。

3.20.1.3 函数功能对照表

函数名	作用
EventEntryForm.processRegistration (EntryFormMemo entryFormMemo, Athlete athlete) -> void	用于处理运动员的报名操作，主要工作为检查调用其他接口检查是否安全，以判断是否通过报名
EventEntryForm.checkSafety() -> boolean	用于检查运动员报名表中有无相互捕食的选手同时存在
EventEntryForm.addAthlete(Species candidate) -> void	用于向运动员报名表中添加运动员（只添加运动员物种类别）
EventEntryForm.createMemo() ->EntryFormMemo	用于生成记载当前运动员报名表信息的备忘录
EventEntryForm.restoreMemo(EntryFormMemo entryFormMemo) -> void	用于从备忘录中恢复运动员报名表信息

EventEntryForm.showAthleteList() -> String	用于获取展示运动员报名表内容
MemoManager.getEntryFormMemo() -> EntryFormMemo	用于管理备忘录，获得到之前存储的备忘录
MemoManager.setEntryFormMemo(EntryFormMemo entryFormMemo) -> void	用于管理备忘录，存储备忘录
EntryFormMemo.getAthleteList() ->Species[]	用于获取备忘录中记载的运动员报名表信息

3.20.2 类图



3.20.3 优缺点分析

备忘录模式是一种对象行为型模式。

（1）优点：

- 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。

（2）缺点：

- 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。

3.21 Observer

3.21.1 实现 API 描述

3.21.1.1 设计思路

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

在一场比赛中， 裁判与运动员是一对多的关系， 裁判作为观察者， 运动员作为被观察者。

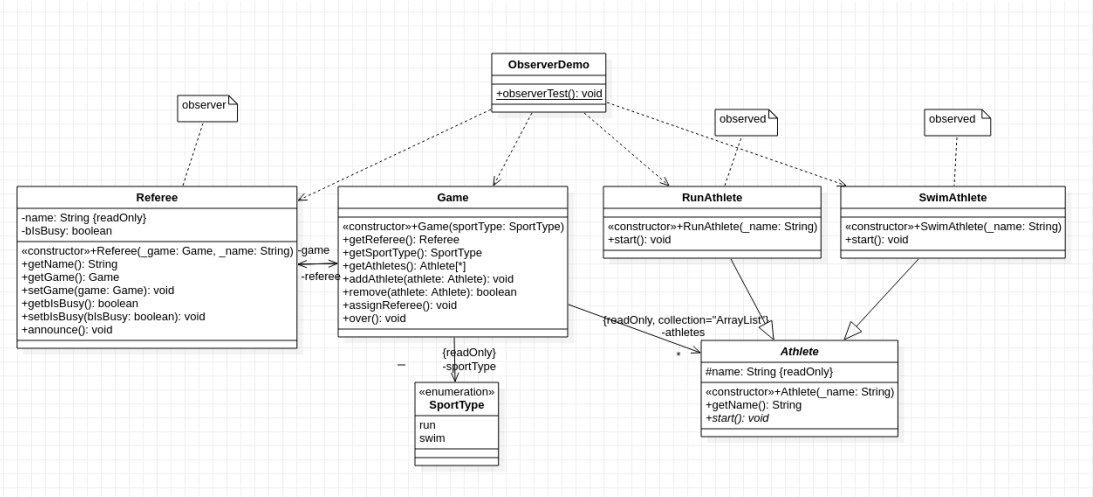
3.21.1.2 API 实现

预先为比赛分配裁判和添加运动员 裁判作为观察者有一个 announce 方法，调用该方法会通知比赛中的所有运动员，遍历他们执行对应的 start 方法

3.21.1.3 函数功能对照表

函数名	作用
Game.Game(SportType type):	创建指定类型的比赛
RunAthlete.RunAthlete(String name):	创建跑步运动员
SwimAthlete.SwimAthlete(String name):	创建游泳运动员
Game.assignReferee(): void	为某一场比赛分配裁判，将裁判状态置为忙碌
Game.addAthlete(Athlete athlete): void	运动员参加比赛
Game.getReferee(): Referee	获取某一比赛分配到的裁判
Referee.annouce():void	裁判宣布比赛开始， 运动员更新状态，开始比赛

3.21.2 类图



3.21.3 优缺点分析

（1）优点：

- 降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。符合依赖倒置原则。
- 目标与观察者之间建立了一套触发机制。

（2）缺点：

- 当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

3.22 State

3.22.1 实现 API 描述

3.22.1.1 设计思路

本例使用乒乓球比赛过程中运动员的状态来说明状态模式。乒乓球运动员有四个状态：比赛中、赛点、胜利、失败。

乒乓球运动员总是持有一个状态类，使用的是抽象类的引用，但可以指向所有从其派生的子类（不同的实际状态）。

每个子类状态，自带一个检查状态的逻辑 checkState，它能检查当前状态是否需要切换，如果需要切换，能够将运动员持有的状态更新，即将该子类状态实例整个替换掉（把自己替换掉），得益于 Java 的自动垃圾回收机制，可以免去 delete 自己（原状态）的代码。

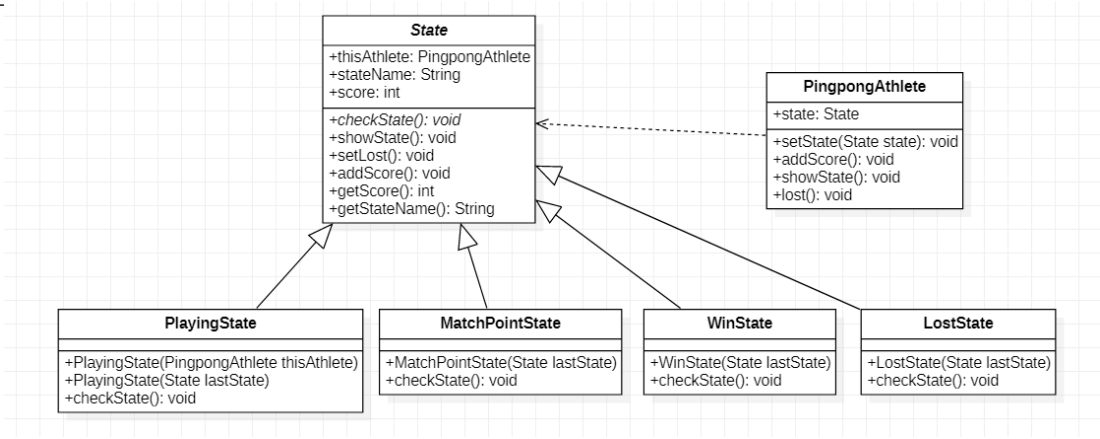
3.22.1.2 API 实现

PingpongAthlete 持有一个状态 State，该状态指针，实际指向的是它的子类，每个子类实现一种状态。根据 State 内部 Score 的增长，State 会自动切换状态，将 PingpongAthlete 持有的 State 指针指向一个新创建出来的状态，这个状态在构造时会复制原状态的必要的信息。

3.22.1.3 函数功能对照表

函数名	作用
setState(State state):void	为运动员设置（或替换）新的状态
addScore():void	为运动员增加一分
PingpongAthlete::showState():void	运动员显示当前状态
lost():void	运动员人数，状态替换成失败状态
checkState():void	检查状态，该切换状态时切换
State::showState:void	打印当前状态
setLost():void	任何一个状态可切换为失败状态
addScore():void	任何一个状态可为状态加一分
getScore():int	获得当前分数
getStateName():String	获得当前状态名

3.22.2 类图



3.22.3 优缺点分析

（1）优点：

1. 结构清晰，状态模式将与特定状态相关的行为局部化到一个状态中，并且将不同状态的行为分割开来，满足“单一职责原则”。
2. 将状态转换显示化，减少对象间的相互依赖。将不同的状态引入独立的对象中会使得状态转换变得更加明确，且减少对象间的相互依赖。
3. 状态类职责明确，有利于程序的扩展。通过定义新的子类很容易地增加新的状态和转换。

（2）缺点：

1. 状态模式的使用必然会增加系统的类与对象的个数。
2. 状态模式的结构与实现都较为复杂，如果使用不当会导致程序结构和代码的混乱。
3. 状态模式对开闭原则的支持并不太好，对于可以切换状态的状态模式，增加新的状态类需要修改那些负责状态转换的源码，否则无法切换到新增状态，而且修改某个状态类的行为也需要修改对应类的源码。

3.23 Strategy

3.23.1 实现 API 描述

3.23.1.1 设计思路

设计模式定义一系列的算法，并把每一个算法封装起来，使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

本场景中不同动物的同种行为由不同的算法实现，将几种动物独特的进食、训练、开幕式表演方法封装起来，使得单个实例调用相应抽象方法时可以按照动物种类自动取用。

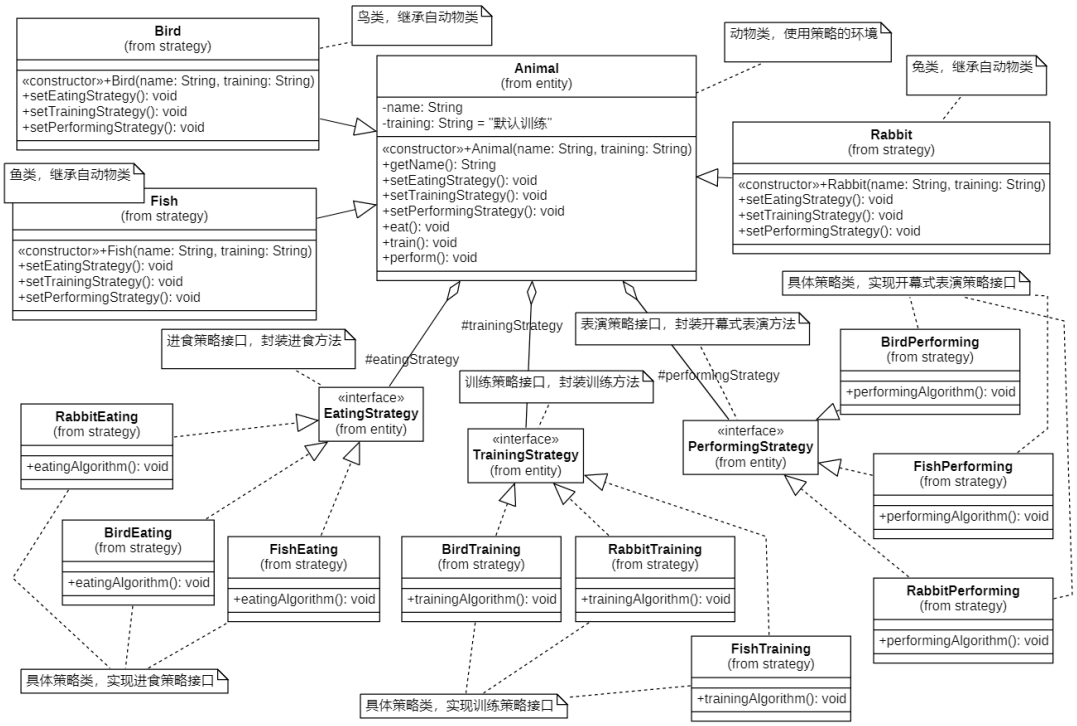
3.23.1.2 API 实现

此场景中 Animal 为环境类，EatingStrategy、TrainingStrategy、PerformingStrategy 为策略接口，三种动物各自的 Eating、Training 和 Performing 类为具体实现算法。Animal 中有三种策略的接口，在三种动物各自的构造函数中将抽象接口设置为具体策略，从而实现了父类代码不变的情况下子类灵活改变方法。具体动物调用父类的 eat、train 和 perform 即可用同种操作实现不同策略。

3.23.1.3 函数功能对照表

函数名	作用
eat():void	动物运动员进食
train():void	动物运动员训练
perform():void	动物运动员入场表演

3.23.2 类图



3.23.3 优缺点分析

（1）优点：

横向扩展性好，灵活性高。

（2）缺点：

客户端需要知道全部策略，若策略过多会导致复杂度升高。

3.23.4 其他

和模板模式的对比：

本项目中使用策略模式和模板模式都实现了动物运动员入场的场景。

策略模式和模板模式有一个很重要的区别，即模板模式一般只针对一套算法，注重对同一个算法的不同细节进行抽象提供不同的实现，而策略模式注重多套算法多套实现，在算法中间不应该有交集，算法和算法只间也不会有冗余代码。因为不同算法只间的实现一般不同很相近。因此我们可以看到，策略模式的关注点更广，模板模式的关注点更深。

3.24 Template

3.24.1 实现 API 描述

3.24.1.1 设计思路

template 模式旨在为外部活动提供一个活动模板，在 template 中，有一系列模板行为，也就是在不同场景中保持不变的行为，同时也会有一部分不定行为，这部分的实现交由使用者自行实现。

该例实现的是开幕式时运动员入场的过程，在入场过程中，进场，等待等等部分都是相同的，唯一的不同在于经过主席台是表演的节目，节目较有特色，所以设置为 template 当中定制的部分。

3.24.1.2 API 实现

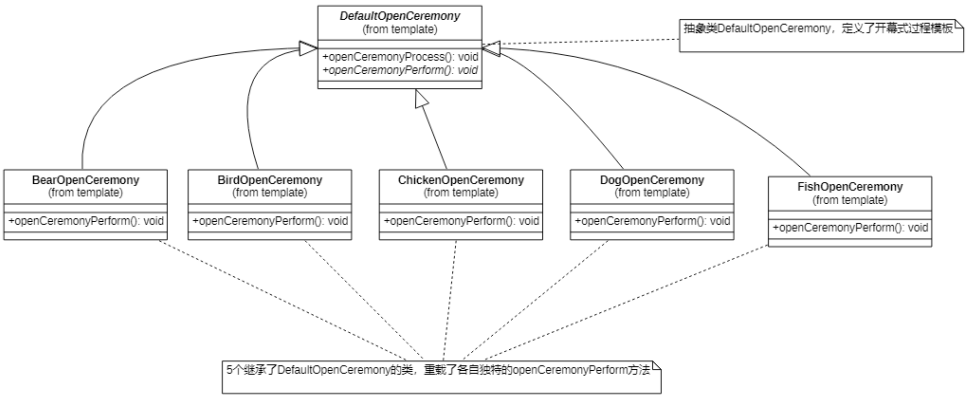
DefaultOpenCeremony 是一个抽象类，其中的 openCeremonyProcess()函数即为上述的模板，在模板中实现了全部固定的部分，即为进场，等待，退场等等的场景。其中 openCeremonyPerform()为一个抽象函数，即为在继承子类中需要具体实现的函数。

其他的具体的动物的继承子类当中，重载了 openCeremonyPerform()函数，具体实现了各种动物各自的表演过程。

3.24.1.3 函数功能对照表

函数名	作用
openCeremonyProcess():void	提供了开幕式的基本模板
openCeremonyPerform():void	实现了各种动物独特的表演

3.24.2 类图



3.24.3 优缺点分析

（1）优点：

i. 它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。

ii. 它在父类中提取了公共的部分代码，便于代码复用。

iii. 部分方法是由子类实现的，因此子类可以通过扩展方式增加相

应的功能，符合开闭原则。

（2）缺点：

i. 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象，间接地增加了系统实现的复杂度。

ii. 由于继承关系自身的缺点，如果父类添加新的抽象方法，则所有子类都要改一遍

3.25 Visitor

3.25.1 实现 API 描述

3.25.1.1 设计思路

Visitor 模式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。

为了实现该场景，在项目初始化之后，初始化建造体育场馆之前需要对场面地建筑面积和周长进行计算。故项目模拟在计算总体育赛事场馆群内不同建筑物的周长和面积时，使用 visitor 模式，将不同的场馆的形状定为 Element（抽象元素角色），不同的操作类设为 Visitor（访问者角色）；

3.25.1.2 API 实现

Visitor 类是抽象访问者，定义了访问具体元素的接口，为每个具体元素提供一个对应的访问操作 visit()，该操作中的参数类型标识了被访问的具体元素。Area 类和 Perimeter 类是具体访问者(Concrete Visitor)角色，实现抽象访问者 Visitor 类中声明的各个访问操作，指出访问者访问一个元素时该做什么。

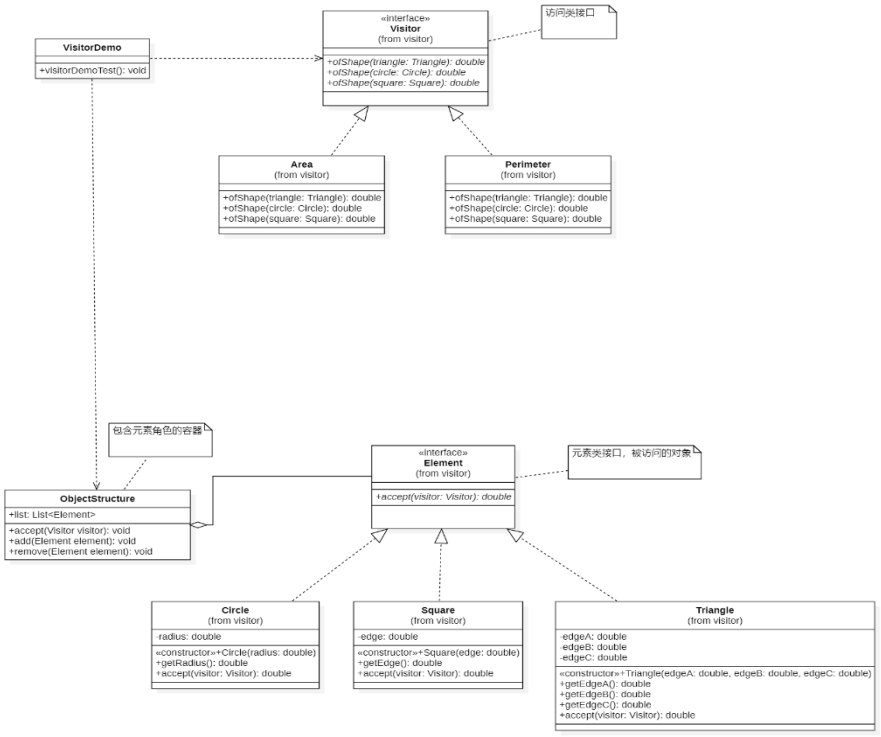
Element 类是抽象元素，声明一个包含接受操作 accept() 的接口，被接受的访问者对象作为 accept() 方法的参数。Circle、Square、Triangle 类均为具体元素(Concrete Element)角色，实现抽象元素角色提供的 accept() 操作，其方法体都是 visitor.ofShape(this) 。

3.25.1.3 函数功能对照表

函数名	作用
Area.ofShape(Triangle triangle):double	计算三角形的面积
Area.ofShape(Circle circle):double	计算圆形的面积
Area.ofShape(Square square):double	计算正方形的面积
Perimeter.ofShape(Triangle triangle):double	计算三角形的周长
Perimeter.ofShape(Circle circle):double	计算圆形的周长
Perimeter.ofShape(Square square):double	计算正方形的周长

Circle.accept(Visitor visitor):double	圆形接受实现 visitor 接口的类，完成该 visitor 对应的操作
Square.accept(Visitor visitor):double	正方形接受实现 visitor 接口的类，完成该 visitor 对应的操作
Triangle.accept(Visitor visitor):double	三角形接受实现 visitor 接口的类，完成该 visitor 对应的操作

3.25.2 类图



3.25.3 优缺点分析

（1）优点：

i. 扩展性好。能够在不修改对象结构中的元素的情况下，为对象结构中的元素添加新的功能。

ii. 复用性好。可以通过访问者来定义整个对象结构通用的功能，从而提高系统的复用程度。

iii. 灵活性好。访问者模式将数据结构与作用于结构上的操作解耦，使得操作集合可相对自由地演化而不影响系统的数据结构。

iv. 符合单一职责原则。访问者模式把相关的行为封装在一起，构成一个访问者，使每一个访问者的功能都比较单一。

（2）缺点：

i. 增加新的元素类很困难。在访问者模式中，每增加一个新的元素类，都要在每一个具体访问者类中增加相应的具体操作，这违背了“开闭原则”。

ii. 破坏封装。访问者模式中具体元素对访问者公布细节，这破坏了对象的封装性。

3.26 Active Object

3.26.1 实现 API 描述

3.26.1.1 设计思路

Active Object 模式是一种异步编程模式。它通过对方法的调用(Method Invocation)与方法的执行(Method Execution)进行解耦(Decoupling)来提高并发性。ActiveObject 模式和 Command 模式的配合使用是实现多线程控制的一项古老的技术，该模式有多种使用方式，为许多工业系统提供了一个简单的多任务核心.

在该场景中，不同区域动物宿舍都配备有对应的安保系统（Security System），安保系统内现有警报设施（Alarm），根据动物区域不同，警报的报告时间各不相同，肉食动物区 2s 报告一次，草食动物区 5s 报告一次。实现多线程。

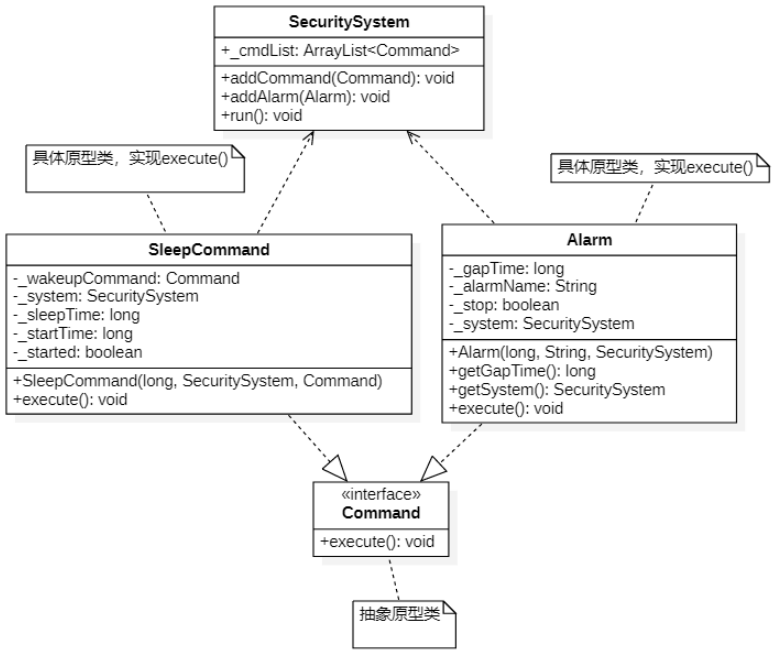
3.26.1.2 API 实现

延迟命令（SleepCommand）添加到 System 后，System 依次读取系统内命令，如果该延迟命令没有到达延迟的时间，则再次将延迟命令复制到 System 尾端。如果 SleepCommand 执行时已经到达延迟时间，则将唤醒命令（wakeupCommand） 即此处的 Alarm 报告命令添加到 System 尾部，当 System 读取时则进行报告。

3.26.1.3 函数功能对照表

函数名	作用
getGapTime():long	获取警报的报告间隔
getSystem():SecuritySystem	获取警报运行的安保系统实例
addCommand(Command):void	向安保系统内添加命令
addAlarm(Alarm):void	向安保系统内添加新的警报

3.26.3 类图



3

3.26.4 优缺点分析

（1）优点：

- 能够实现简单的不同工作分配时间控制

（2）缺点：

- Cpu 时钟与实际时间有偏差
- 依赖于 System 便利的速度，如果同时进行任务过多，则会产生较大偏差

3.27 Balking

3.27.1 实现 API 描述

3.27.1.1 设计思路

Balking 模式：一个线程要去执行某个操作。但是，在发现这个操作已经被别的线程做了，于是，就没有必要再去做这样的一个操作了，于是立即停止。

项目中利用 Balking 模式模拟跑步比赛结束后回收号码背心的具体场景。具体的，跑步比赛结束后，工作人员需要收回运动员手上的号码背心，如果发现运动员主动放回器材回收点则不需要继续向该运动员索要号码背心。

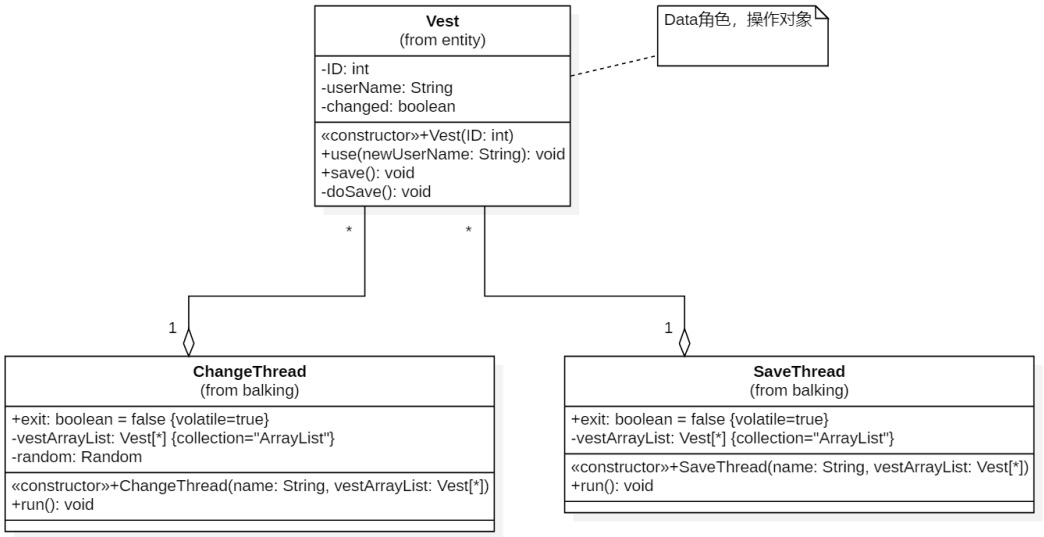
3.1.1.2 API 实现

Vest 类在这里是 Data 角色，是需要被保护的内容。ChangeThread 类模拟运动员比赛后的行为线程，用随机数模拟运动员在使用完号码背心之后到主动回收背心到器材室之间所需要的时间，到时间之后进行 vest.save()操作进行回收。于此同时，SaveThread 类模拟场上工作人员回收号码背心的情景，且设定每隔 1s 回收一件号码背心（执行 vest.save()）。Demo 中设定了一个 ArrayList<Vest>用于表示待回收的背心列，对于同一件背心，如果已经执行了 save()操作修改了其 changed 属性，那么第二次对其执行 save()时实际操作内容为空。

3.1.1.3 函数功能对照表

函数名	作用
ChangeThread.run():void	运行该线程，模拟运动员比赛后的行为
SaveThread.run():void	运行该线程，模拟专门回收背心的体育场工作人员的工作情况
Vest.use(String newUserName):void	某个运动员使用该背心，修改使用状态和详细信息
Vest.save():void	若该背心被使用过，则应当将该背心回收
Vest.doSave()：void	模拟将号码背心回收到体育器材室

3.27.2 类图



3.27.3 优缺点分析

（1）优点：

i. 该设计模式中会存在一个守护条件，当该条件不成立时，会立即停止当前的动作。除本项目实现的场景之外，该设计模式也可以用于文档的自动保存和手动保存的协调工作，提高运行效率同时保证守护条件的准确守护。

3.28 Worker Thread

3.28.1 实现 API 描述

3.28.1.1 设计思路

在线程池模式中，工人线程会逐个取回工作并进行处理，当所有工作全部完成后，工人线程会等待新的工作到来。

3.28.1.2 API 实现

首先构造一个补给点 CheckPoint，并传入参数初始化其中的志愿者数量，然后调用其 startVolunteer() 方法让志愿者就绪。

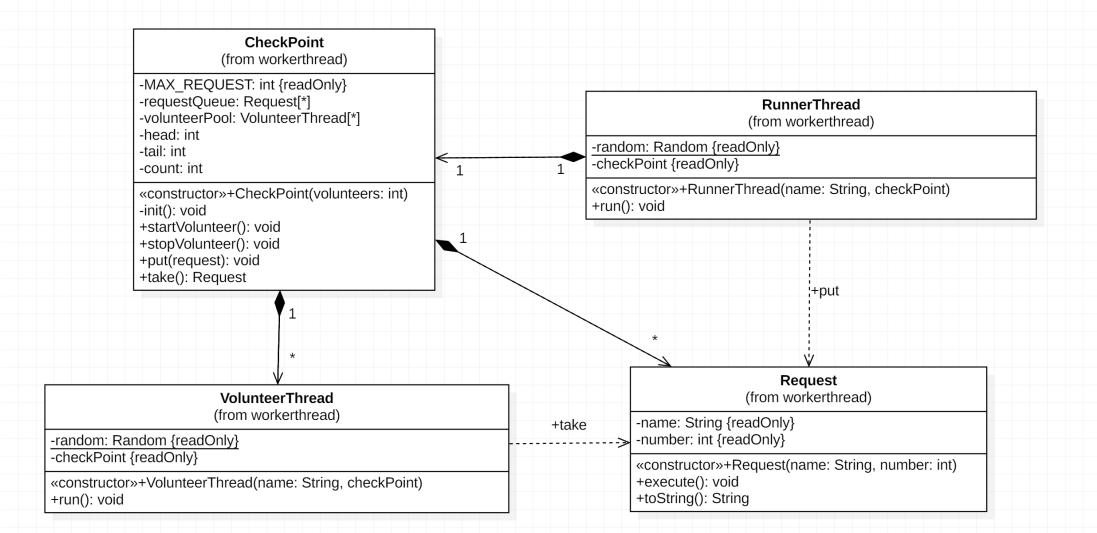
然后构造运动员 RunnerThread，并调用 start() 方法使其开始发送补给请求。

等待所有运动员的所有请求完成后，调用 stopVolunteer() 方法停止志愿者工作。

3.28.1.3 函数功能对照表

函数名	作用
CheckPoint.startVolunteer(): void	开始所有志愿者线程
CheckPoint.stopVolunteer(): void	停止所有志愿者线程
CheckPoint.put(Request request): void	运动员新建补给请求
CheckPoint.take(): Request	志愿者接受请求
Request.execute(): void	执行请求
Request.toString(): String	打印输出执行内容

3.28.2 类图



3.28.3 优缺点分析

（1）优点：

在工作到来之前就预先准备好了工人，减小了线程启动时的影响。

工人线程可复用，减少资源开销。

（2）缺点：

当线程池需要处理的任务之间并非互相独立时，可能会出现死锁。

3.29 Dirty Flag

3.29.1 实现 API 描述

3.29.1.1 设计思路

脏标志模式用一个标志位来表示一组数据的状态，这些数据要用于计算或者同步。在满足条 件的时候设置标志位，需要的时候检查标志位。如果设置了标志位，那么表示这组数据处于 dirty 状态，需要重新计算或者同步。如果 flag 没有被设置，那么可以不计算（或者利用缓存的计算结果）。

该模式的实现为接力跑比赛，模拟跑步的过程中需要定时间片（0.3s）刷新运动员的位置，但在接力棒未传到手时运动员在原地等待、位置不会有变化，运动员完成传递接力棒之后也不会继续跑步、位置同样不再变。为了节省资源设置 dirty flag，当位置信息未变动时不需要重新计算、直接使用计算好的已存储位置。

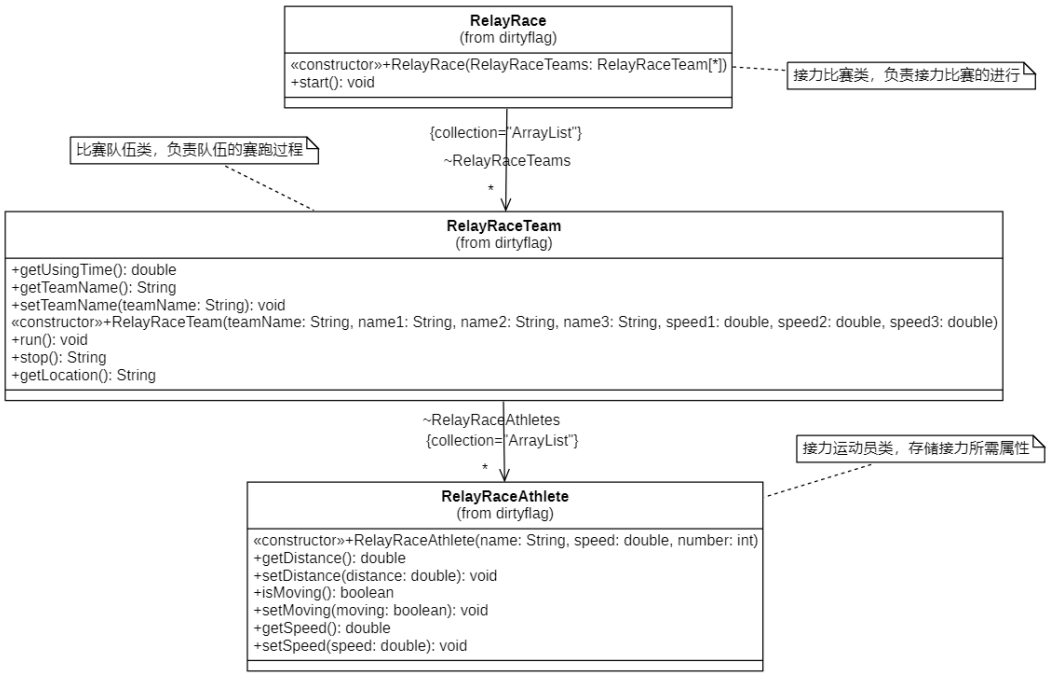
3.29.1.2 API 实现

RelayRaceTeam 为接力比赛队伍，初始化时需要传入运动员信息，若信息不符合规范会被更改为默认参数。RelayRace 为接力比赛类，初始化时需要传入参赛队伍列表。然后使用 start 方法开始接力比赛，命令行输出每一时刻各选手的位置，比赛结束后显示各队伍用时以及获胜队伍。

3.29.1.3 函数功能对照表

函数名	作用
RelayRace.start():void	开始接力比赛
RelayRaceTeam.run():void	每一队伍起跑
RelayRaceTeam.getLocation():String result	每 0.3 秒刷新该队运动员位置，若位置有变化则计算新位置并返回，若无变化则直接使用之前存储的位置
RelayRaceTeam.stop():void	每队到达终点之后停止比赛、重置位置信息
RelayRaceTeam.getUsingTime():double	获取每一队跑完用时

3.29.2 类图



3.29.3 优缺点分析

（1）优点：

Dirty Flag 模式的本质作用在于：延缓计算或数据同步，减少无谓的计算或者同步。本例中每一次位置更新只有没结束比赛的队伍中正处于运动状态的选手位置需要计算，其他选手都可以直接沿用以前存储的位置，节省了运算开支。

（2）缺点：

设置并更新标志位以及进行标志位判断本身也会占用一些存储空间和运算量。

3.30 Lazy Loading

3.30.1 实现 API 描述

3.30.1.1 设计思路

一些数据的加载可能需要大量的时间和内存消耗，而这些数据也许并不是立即需要使用，并且可能并不会使用，这样就造成了资源的浪费以及用户体验的损失，基于此，Lazy Loading 提出的解决办法即是只有当数据真正需要的时候才进行 loading，将载入的操作进行延迟。

在本例当中，模拟的是对于动物狂欢会的工作人员信息的载入，只有真正需要 getStuffList 的时候，才进行数据的导入工作，在这之前，工作人员信息并不会得到导入。

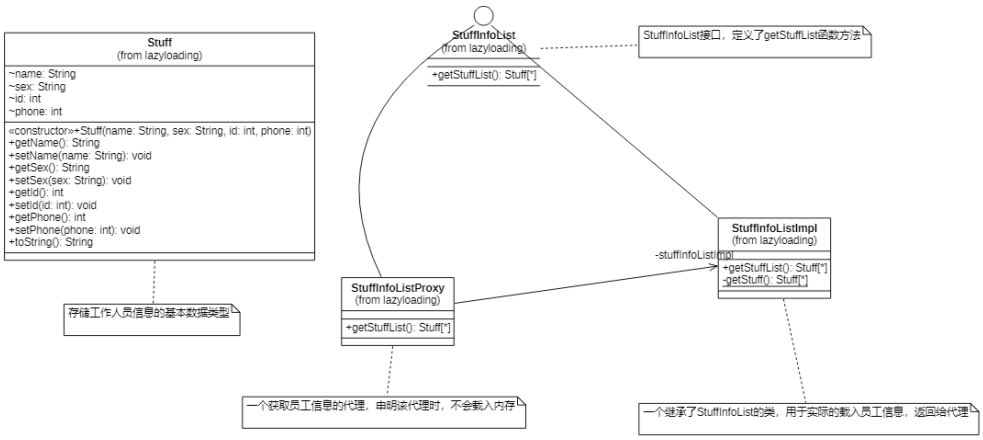
3.30.1.2 API 实现

一个接口类 StuffInfoList，其中提供了一个对外提供员工信息列表的函数 getStuffList()。对于该接口，有两个实现，一个实现是用户可以进行申明的类 StuffInfoListProxy，该类中持有一个 StuffInfoListImpl 类的实例，简单的申明 StuffInfoListProxy 不会进行数据的导入，只有调用了 get 函数之后，才会创建 StuffInfoListImpl 的实例，在进行初始化时导入数据，并进行数据的返回。

3.30.1.3 函数功能对照表

函数名	作用
getStuffInfoList():ArrayList<Stuff>	得到员工信息列表
getStuff():ArrayList<Stuff>	StuffInfoListImpl 导入员工信息

3.30.2 类图



3.30.3 优缺点分析

（1）优点：

i. 只有真的调用 getStuffInfoList 的时候，系统才会进行数据的载入，从而提高了效率以及用户体验感。

ii. 对于数据的载入，并不需要用户调用其他的接口，而是将载入的功能嵌入到 getStuffInfoList 的功能当中。从而这一过程对于用户而言是透明的。

（2）缺点：

i. 需要实现一个额外的代理或者对载入数据的类进行封装，使得代码变得复杂。

3.31 Front Controller

3.31.1 实现 API 描述

3.31.1.1 设计思路

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。直接和 MVC 合并,扩充 MVC 的视图（记分板视图/运动场视图）。前端控制器做一个授权，根据使用者的身份，来给他指定到一个控制器进行操作。运动员只有对应个人可以看个人的记录，管理员可以查看运动场地，其他人可以查看最新的计分板情况。

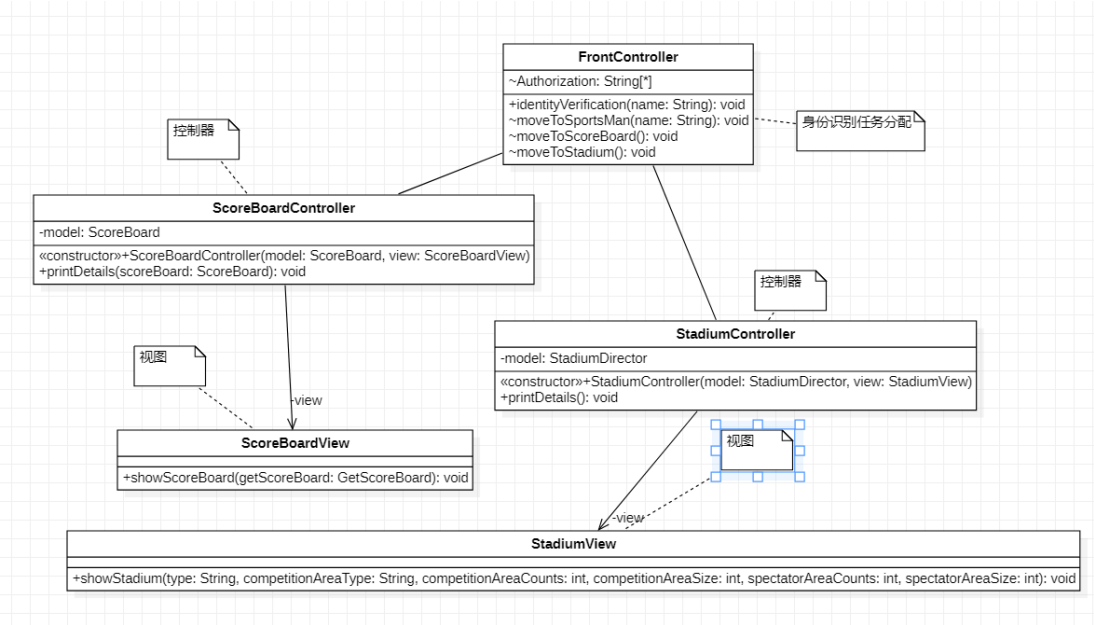
3.31.1.2 API 实现

由 FrontController 来接受输入请求，匹配对应名字的控制器分别启动：moveToSportsMan,moveToScoreBoard,moveToStadium。在其中调用由对应控制器 printDetails 调用 View 中的接口打印相关信息。

3.31.1.2 函数功能对照表

函数名	作用
Stadium ScoreBoard SportMan	这两个实体类及相关的函数请看 callback 和 Builder 设计模式的实现。 SportMan 的相关函数请看 MVC 设计模式的实现，这里只是调用展示。
moveToSportsMan(String name): void	转移到运动员的控制器
moveToScoreBoard(): void	转移到计分板的控制器
moveToStadium(): void	转移到运动场的控制器
printDetails(***): void	类似 MVC 利用 View 来打印具体类的相关信息，模拟视图。
identityVerification(String name):void	接受一个姓名输入，来寻找相应的控制器

3.31.2 类图



3.31.3 优缺点分析

（1）优点：

- 减少了控制器公有的如身份验证的部分
- 使得整个数据请求和展示的逻辑变得更加清晰。

（2）缺点：

对于视图和控制器不多的项目或者没有共同需求的控制器而言是完全没有必要的。

3.31.4 其他

一般前端控制器减少各个控制器公有的部分比如身份验证，日志记录这一类要求。需要多个 MVC 才能体现出好处。这里使用非常牵强，而且我调用另外两位同学的实体类，一些接口也不是很切合 MVC 写法，效果不是很好。

3.32 Transfer Object

3.32.1 实现 API 描述

3.32.1.1 设计思路

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。

3.32.1.2 API 实现

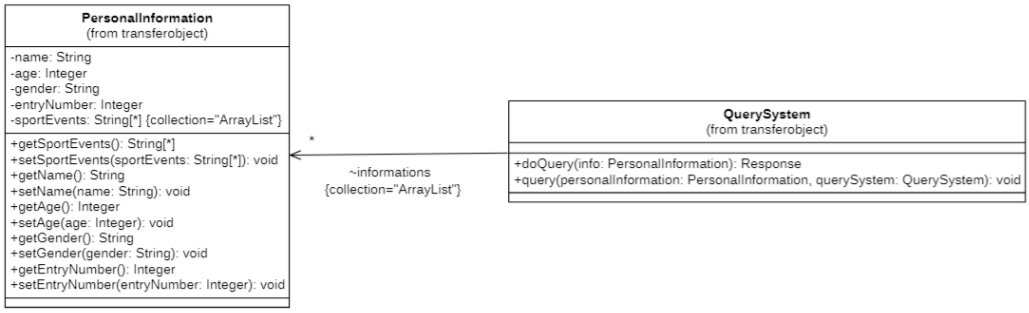
运动员查询和修改个人信息时，将所有信息封装成 POJO 的形式传递给系统，系统根据得到的 POJO 对运动员信息做出相应修改。

3.32.1.2 函数功能对照表

函数名	作用

QuerySystem.doQuery(PersonalInformation info):Response	向查询服务提供含有查询信息的 POJO 执行查询
Response.ok(Integer status, String msg, Object obj):PersonalInformation	向查询者返回

3.32.2 类图



3.32.3 优缺点分析

（1）优点：

- 当 model 发生变化时不需要改动业务代码和接口

（2）缺点：

- 各种变化的需求导致各种各样的 Transfer Object 混用，导致项目维护困难

3.33 Pipeline

3.33.1 实现 API 描述

3.33.1.1 设计思路

管道模式使用有序阶段来处理一系列输入值。每个已实现的任务由流水线的一个阶段表示。您可以认为管线类似于工厂中的装配线，装配线中的每个项目都是分阶段构造的。部分组装的物品从一个组装阶段传递到另一组装阶段。装配线的输出与输入的顺序相同。

奖牌榜排名实时更新，定义结构体 medalBoard 用来存储当前各动物代表队的金银铜牌数目以及他们的排名顺序，分别定义阶段类 goldSort，silverSort，brozenSort，它们传入传出数据均是 medalBoard，分别对奖牌榜上动物代表队按照所获得的金牌，银牌，铜牌数目进行排序。

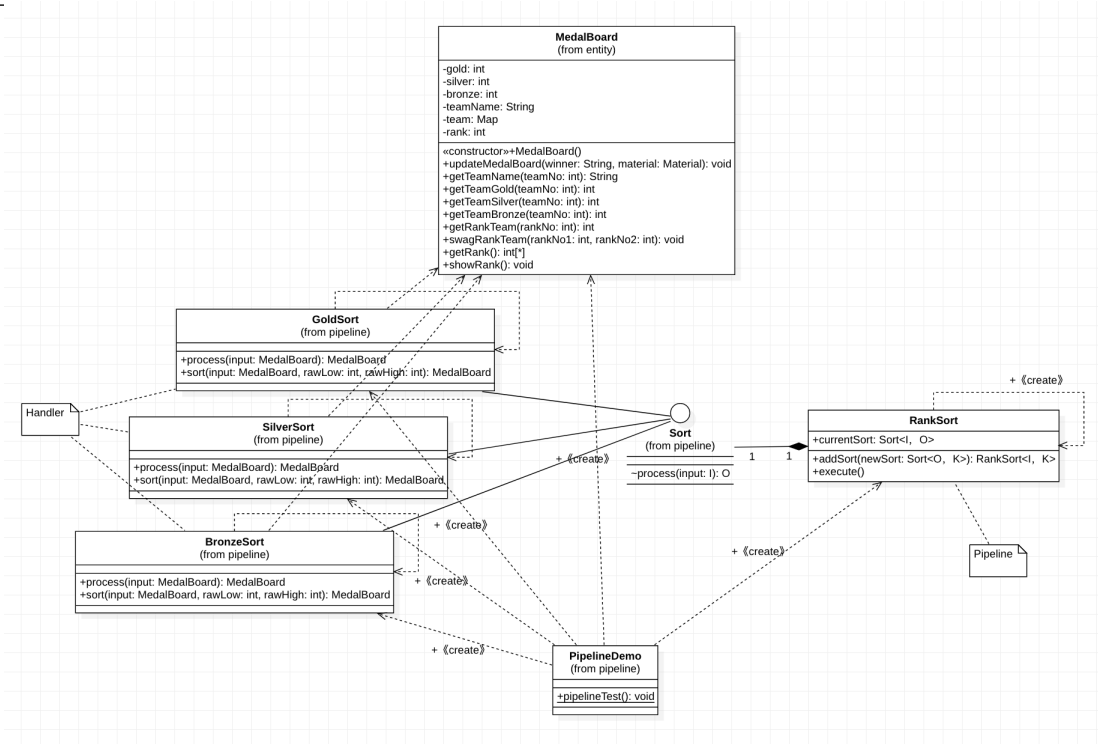
3.33.1.2 API 实现

定义管道类 rankSort， 用于将各阶段类按顺序组装到一起，并调用其 execute 方法来排序动物代表队。其执行顺序按照组装阶段顺序，先调用 goldSort 的 process 按照金牌数目排序动物代表队，再调用 silverSort 的 process 按照银牌数目排序动物代表队，最后调用 bronzeSort 的 process 按照铜牌数目排序动物代表队，全部执行完成之后即为正确拍好顺序后的奖牌榜。

3.33.1.3 函数功能对照表

函数名	作用
MedalBoard.updateMedalBoard(String winner, Material material) -> void	用于更新代表队奖牌数目
MedalBoard.swagRankTeam(int rankNo1, int rankNo2) -> void	用于交换奖牌榜上两只代表队的排名顺序
MedalBoard.showRank() -> void	用于公示奖牌榜排名及奖牌数目
GoldSort.process(MedalBoard input) -> MedalBoard	用于将奖牌榜上各代表队按照金牌数目进行排序（银牌、铜牌阶段类均有此方法，且作用相同）
GoldSort.sort(MedalBoard input, int rawLow, int rawHigh) -> MedalBoard	奖牌榜上各代表队按照金牌数目排序所使用的排序算法（银牌、铜牌阶段类均有此方法，且作用相同）
RankSort.addSort(Sort<O, K> newSort) -> RankSort<I, K>	用于为管道类添加阶段类
RankSort.execute(I input) -> O	用于按照添加阶段类的顺序执行 process 方法，以排序奖牌榜

3.33.2 类图



3.34 Producer Customer

3.34.1 实现 API 描述

3.34.1.1 设计思路

Producer-Customer 模式，即生产者消费者模式中，Producer（生产者）指生成数据的线程；Consumer（消费者）指使用数据的线程。通常平台上可用数据量会有上限和下限的限制。该模式可以保证生产者安全地将数据交给消费者，并且消除生产者和消费者处理速度差异可能引起的差异。

项目中，该模式用于实现运动员从桌子上拿矿泉水，桌子只能放置 n 瓶水，若桌上没水，则需要等待放置后才拿，若桌上水满，则需要等待某些水被拿走后再放置。

3.34.1.2 API 实现

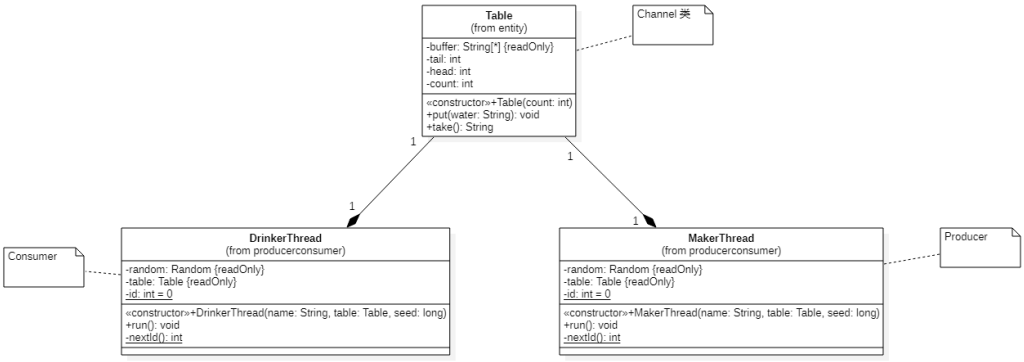
Table 类是数据存放容器，限定同时最多只能存放 n 瓶矿泉水。DrinkerThread，消费者线程，模拟运动员拿取矿泉水；MakerThread，生产者线程，模拟往桌上放置矿泉水的工作人员。消费者线程和生产者线程类均重写 run()方法，分别调用 table.take()和 table.put()模拟放置矿泉水和拿取矿泉水的过程。

3.34.1.3 函数功能对照表

函数名	作用
DrinkerThread.run():void	模拟运动员，可以拿取桌子上的水
MakerThread.run():void	模拟工作人员，可以往桌子上放置矿泉水
Table.put(String water):void	放置矿泉水

Table.take():String	拿取矿泉水

3.34.2 类图



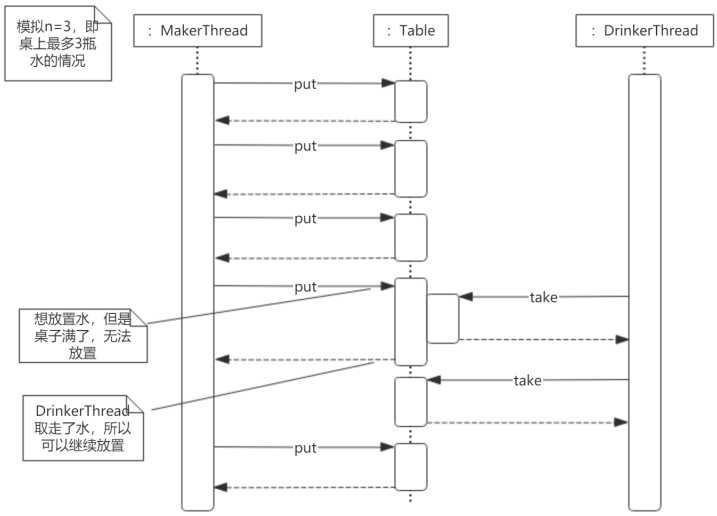
3.34.3 优缺点分析

（1）优点：

i. 添加了一个“桥梁角色”，用于消除线程间处理速度的差异

ii. MakerThread（生产者线程） 和 DrinkerThread（消费者线程）并不依赖于 Table 类的具体实现。

3.34.4 时序图



3.35 Double Locked Checking

3.35.1 实现 API 描述

3.35.1.1 设计思路

在实现单例模式时，如果未考虑多线程的情况，就容易在多线程情况下出现多个实例。对此的改进方法有一个演进过程：加锁 ----> 双重锁。

加锁虽然解决了问题，但是因为用到了 synchronized，会导致很大的性能开销，并且加锁其实只需要在第一次初始化的时候用到，之后的调用都没必要再进行加锁，所以用 Double Checked Locking 设计模式。

项目中在模拟初始化 OlympicsYard 的时候使用该设计模式确保总体育园场景只初始化一次。

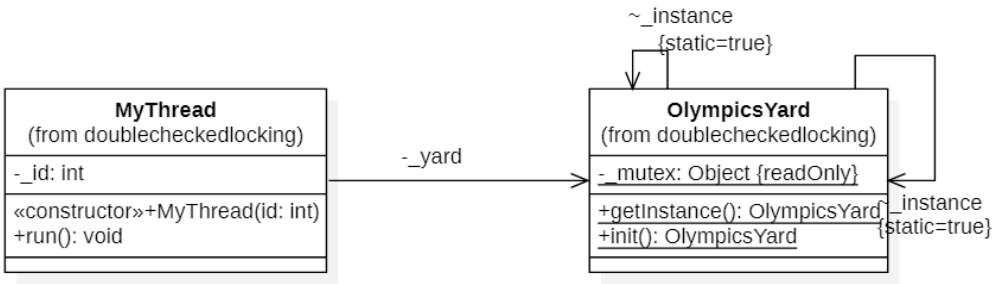
3.35.1.2 API 实现

OlympicsYard 类是只能初始化 1 次且只能存在 1 个实例的实体类。可调用 static 的 getInstance()方法获取 OlympicsYard 实例，若没有实例被初始化则调用 init()初始化实例，若实例已经存在则返回实例。MyThread 类模拟需要 OlympicsYard 实例的线程，重写 run()方法调用 getInstance()方法获取实例。

3.35.1.3 函数功能对照表

函数名	作用
MyThread.run():void	启动线程，调用 OlympicsYard.*getInstance*() 获得 OlympicsYard 实例
static OlympicsYard.getInstance()	获得 OlympicsYard 类的唯一实例，若该实例还未初始化则调用 init()
static OlympicsYard.init()	初始化

3.35.2 类图



3.35.3 优缺点分析

（1）优点：

i. 减少了不必要的性能开销，提高了执行效率，同时保证了多线程情况下不会多次初始化实例。

3.36 Multiton

3.36.1 实现 API 描述

3.36.1.1 设计思路

多例模式是单例模式的扩展。多例类可以有多个实例；多例类必须自己创建，管理自己的实例，并向外界提供自己的实例；实例数目可以为有限个也可以没有上限。

本场景中使用多例模式分配训练房间，固定数量的房间是由私有构造函数编译阶段静态创建的，只能由多例类分配而不能再新建。

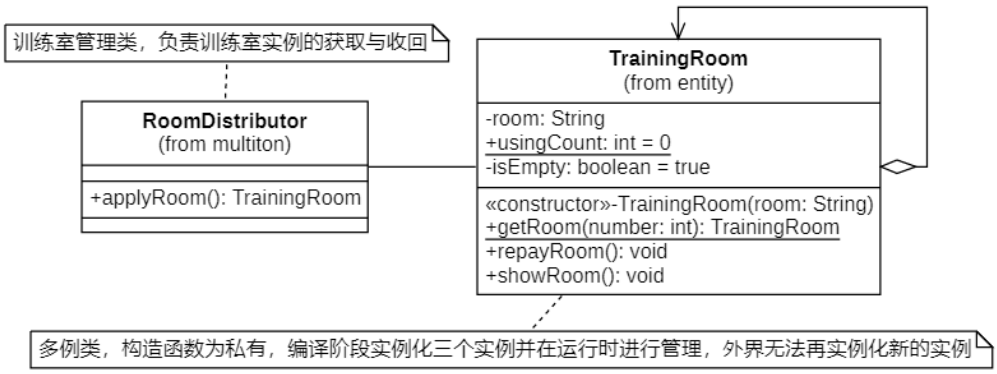
3.36.1.2 API 实现

TrainingRoom 类是在编译阶段实例化了特定数量（三个）个实例的实体类，它的构造函数为私有，无法从外界再实例化新的实例。通过 getRoom(int)方法可以获取已经存在的实例，repayRoom()可以返还获取的实例，即使得实例回到未使用状态供下次取用。

3.36.1.3 函数功能对照表

函数名	作用
roomDistributor.applyRoom():TrainingRoom	获取一个 TrainingRoom 实例
roomDistributor.showRoom():void	展示获取的房间信息
roomDistributor.repayRoom():void	归还获得的房间

3.36.2 类图



3.36.3 优缺点分析

（1）优点：

单例模式和多例模式类的构造方法都是是私有的，从而避免了外部利用构造方法直接创建实例。多例模式在单例模式的基础上进行扩展，将初始化时生成的实例个数从一个扩展为多个，使得在同样确保多例类安全性的基础上使用更加灵活。

（2）缺点：

逻辑更加复杂，开发难度增加。

3.37 Extension Objects

3.37.1 实现 API 描述

3.37.1.1 设计思路

扩展对象模式中的预期对象的接口将在未来被扩展。通过额外的接口来定义扩展对象。

主体对象为比赛，拥有查询扩展对象（加时赛）的接口。加时赛是基于比赛上的扩展，可以对比赛添加加时赛。

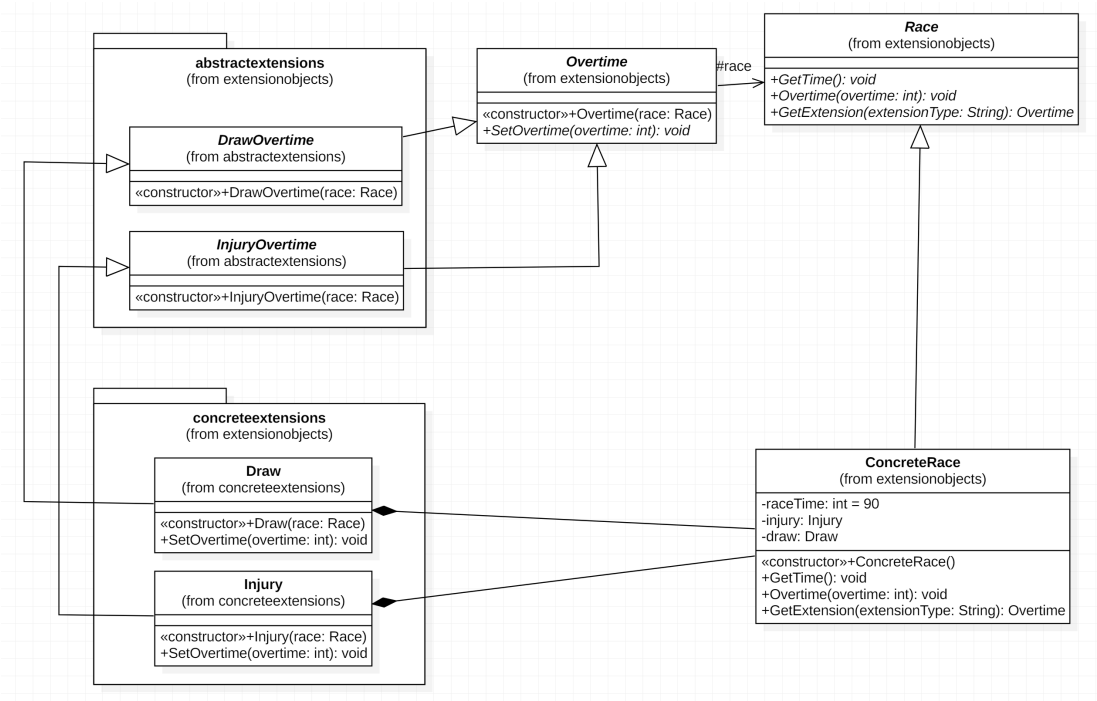
3.37.1.2 API 实现

首先构造一个比赛对象，通过其 GetExtension() 方法获取其加时赛扩展，最后通过扩展对象的 SetOvertime() 方法来完成加时赛的设置。

3.37.1.3 函数功能对照表

函数名	作用
GetTime(): void	打印本场比赛的时长
Overtime(int overtime): void	构造指定时长的加时赛
GetExtension(String extensionType): Overtime	获取加时赛扩展对象
SetOvertime(int overtime): void	指定加时赛的时长

3.37.2 类图



3.37.3 优缺点分析

（1）优点：

使得主体类的抽象接口不会过于膨胀。

可以增加全新的扩展功能。

（2）缺点：

调用时的类变得更加复杂。

需要控制对扩展对象的滥用。

3.38 MVC

3.38.1 实现 API 描述

3.38.1.1 设计思路

提供控制器和视图来访问运动员的具体信息，不需要了解类的具体信息。控制器可以限制外界对运动员的操作权限，视图决定向外界展示那部分信息。 一个 controller 类，一个 view 类。controller 中有一个 model 代表具体的运动员,View 来展示 model 中的信息。

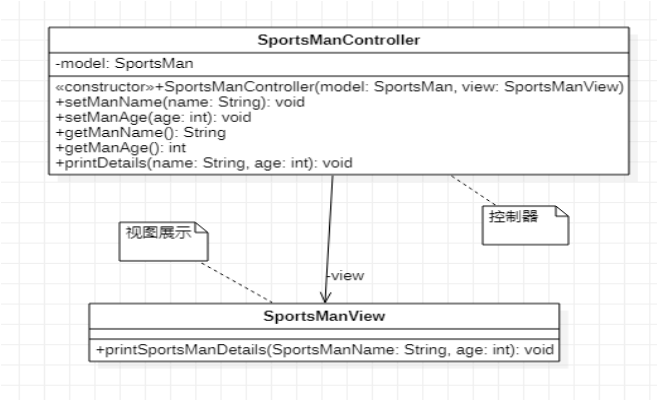
3.38.1.2 API 实现

构造控制器 SportsManController 以后，首先调用 printDetails 获取信息，然后两个 set 函数修改信息，再调用 printDetails 查看是否成功修改。

3.38.1.2 函数功能对照表

函数名	作用
printDetails(String name,int age): void	利用 View 来打印对应信息
setManName setManAge getManName getManAge	调用 model(Sportmans)的对应方法来完成对运动员信息的访问和修改。

3.38.2 类图



3.38.3 优缺点分析

（1）优点：

- 视图层和业务层分离，这样就允许更改视图层代码而不用重新编译模型控制器代码。同样，一个应用的业务流程或者业务规则的改变只需要改动 MVC 的模型层即可，因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。

（2）缺点：

1、增加了系统结构和实现的复杂性

2、视图与控制器间的过于紧密的连接

3.38.4 其他

其实 MVC 主要运用到 B/S 服务架构，分离前端 html 和后端数据处理数据库，用在这里非常的不合适写起来也很别扭。

3.39 Filter

3.39.1 实现 API 描述

3.39.1.1 设计思路

过滤器模式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。

在本项目中使用过滤器模式来对美食广场的菜品进行味系分类， 通过过滤器来得到不同味系的菜单

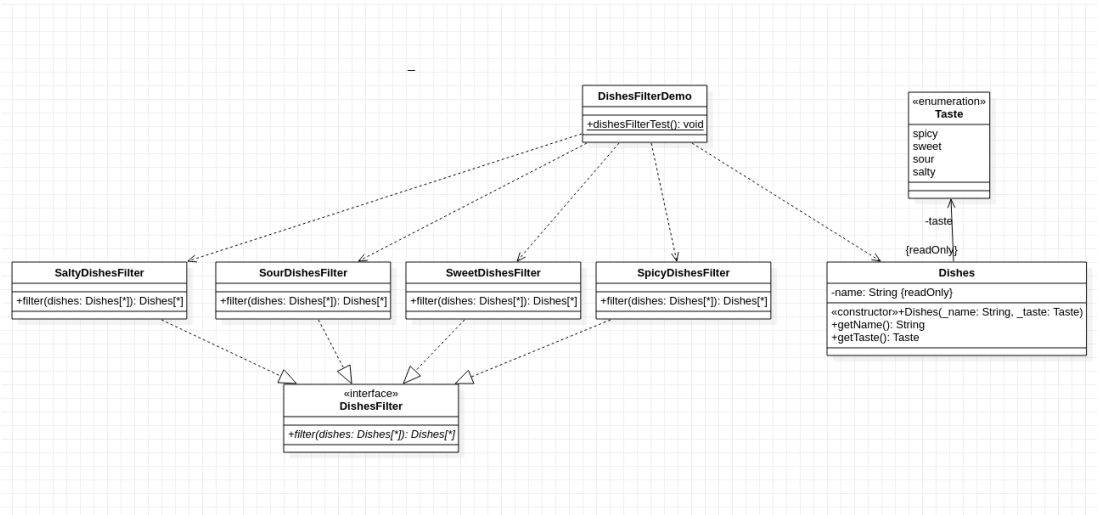
3.39.1.2 API 实现

先多次调用 dishes 的构造函数生成不同菜品，放入同一个 arraylist 作为总菜单，然后以总菜单为参数运行每个过滤器的 filter 方法，该方法会遍历 arraylist 并找到符合该过滤器条件的菜品，最后输出过滤好的 arraylist

3.39.1.3 函数功能对照表

函数名	作用
Dishes.DIshes(String name, Taste taste):	构造指定名字和味道的菜品
SpicyDishesFilter.SpicyDishesFilter():	构造辣味菜品过滤器
SourDishesFilter.SourDishesFilter():	构造酸味菜品过滤器
SaltyDishesFilter.SaltyDishesFilter():	构造咸味菜品过滤器
SweetDishesFilter.SweetDishesFilter():	构造甜味菜品过滤器
DishesFIlter.filter(ArrayList<DIshes> dishes): ArrayList<DIshes>	用过滤器对给定的菜单进行过滤

3.39.2 类图



3.39.3 优缺点分析

（1）优点：

- 体现了各功能模块的“黑盒”特性及高内聚、低耦合的特点。
- 支持软件功能模块的重用。
- 支持并行操作，每个过滤器可以作为一个单独的任务完成。

（2）缺点：

- 通常导致系统处理过程的成批操作。
- 需要设计者协调两个相对独立但又存在关系的数据流。
- 可能需要每个过滤器自己完成数据解析，从而导致系统性能下降，并增加了过滤器具体实现的复杂性。

3.40 Null Object

3.40.1 实现 API 描述

3.40.1.1 设计思路

本例使用教练点名运动员上场来说明空对象模式。有一个实到队伍，教练如果点到其中的名字，对应的运动员对象则做出答到、上场的动作；如果点到一个名字不在实到队伍里面，将生成一个“空运动员”，它也具有答到、上场的方法，但它的实际行为是声明“该运动员缺席”。

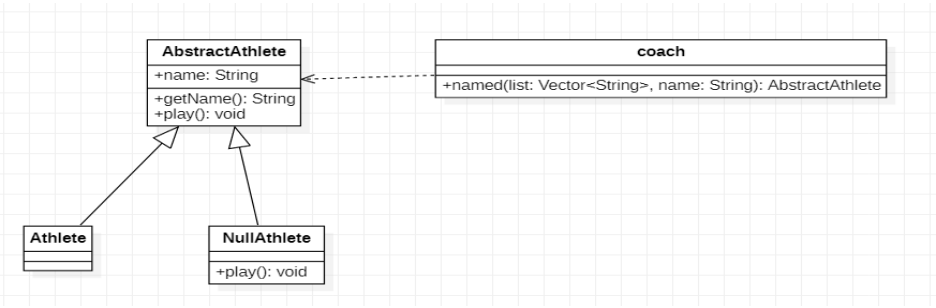
3.49.1.2 API 实现

Coach 类将调用 named 方法进行点名，如果点到存在的运动员，将返回运动员对象，如果点到不存在的运动员，将返回空运动员对象。

3.40.1.3 函数功能对照表

函数名	作用
play():void	打印上场消息
getName():String	获取运动员名
named(Vector<String> list, String name) :AbstractAthlete	教练点名

3.40.2 类图



3.40.3 优缺点分析

（1）优点：

在空对象模式中，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，从而无需检查空值。

（2）缺点：

增加了一个空类，成本是否提升算是见仁见智。

3.41 Specification

3.41.1 实现 API 描述

3.41.1.1 设计思路

规约模式将业务规则（通常是隐式业务规则）封装成独立的逻辑单元，从而将隐式业务规则提炼为显式概念，并达到代码复用的目的。同时不同的规约可以通过布尔运算组成新的规约。

用规约模式实现对于选手参赛资格的判断：对于不同的比赛要求创建不同的规约，通过调用规约判断选手能否参赛。

3.41.1.2 API 实现

AbstractSelector 是各种规约的抽象基类，可以通过 and(), or(), not() 来实现各种规约的复合。

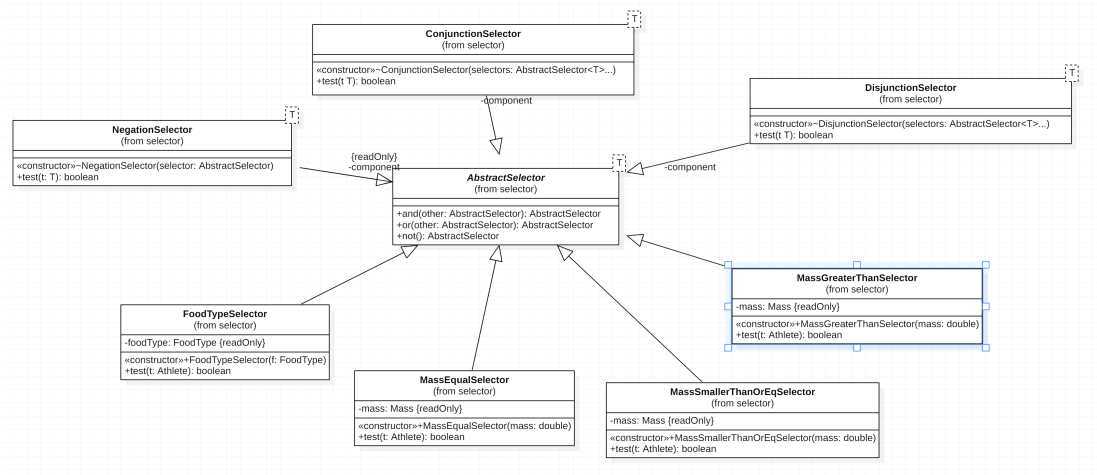
除了这 3 个以外的 Selector 则是具体的业务逻辑规约。

各个规约接收 Athelete 实体，判断其是否符合规约。

3.41.1.3 函数功能对照表

函数名	作用
getInstance(): Sponsor	获取 Sponsor 实例
getName(): String	获取 Sponsor 的名称属性

3.41.2 类图



3.41.3 优缺点分析

（1）优点：

将隐式的逻辑转化为显式的概念，便于理解和维护。

通过抽象实现了复合规约。

（2）缺点：

滥用规约会使得代码复杂度增加。

3.42 Converter

3.42.1 实现 API 描述

3.42.1.1 设计思路

在编写程序的过程中，通常会存在多种类型的数据的转换，这些数据类型通常有着固定的数据格式以及字段，则这种数据类型之间的转换可以通过转换器这个模式，提供两种数据类型的双向转换。

3.42.1.2 API 实现

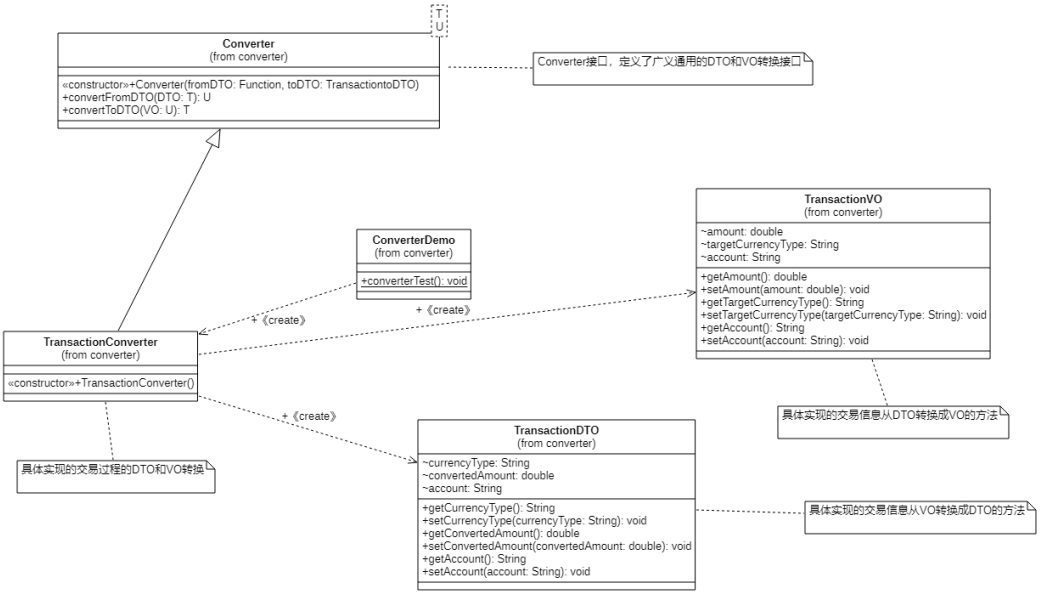
Converter 类是一个抽象的父类，其提供了一个对用户提供的进行转换的标准接口，同时泛型的形式提供了一个双向转换的模板，可以在子类中进行具体的实践。其中 convertFromDTO 和 convertToDTO 是提供给用户的两个转换口，两个转换口调用内部的两个自定义实现的转换函数，进行相对应的转换。

用户在外部调用 converter 的 convertFromDTO 和 convertToDTO 即可完成相应的转换。

3.42.1.3 函数功能对照表

函数名	作用
fromDTO.apply(final T DTO):final U	具体实现 DTO 到 VO 之间的转换
toDTO.apply(final U VO):final T	具体实现 VO 到 DTO 之间的转换
convertFromDTO(final T DTO):final U	提供给用户调用的 DTO 转换成 VO 的接口
convertToDTO(final U VO):final T	提供给用户调用的 VO 转换成 DTO 的接口

3.42.2 类图



3.42.3 优缺点分析

（1）优点：

i. 对外部提供统一的接口，使得调用者和内部的具体实现解耦。
ii. 内部采用泛型的思想，便于多种数据类型的添加和存在，使得内部具有可扩充性，并且在后续的添加中也符合开闭原则。

（2）缺点：

i. 由于没有强制要求指明类别，后期的类型识别可能变得低效。
ii. 所有数据类型间的转换都需要单独实现，使得程序后期变得复杂。

3.43 Callback

3.43.1 实现 API 描述

3.43.1.1 设计思路

回调机制是一种常见的设计模型，他把工作流内的某个功能，按照约定的接口暴露给外部使用者，为外部使用者提供数据，或要求外部使用者提供数据。

callback 设计模式的实现借助一个任务类和一个回调类，任务完成后自动调用回调类的函数以及功能。本次设计中，任务即为监听外部比赛，获取了比赛结束的消息后，调用更新接口，进行积分榜的更新。

3.43.1.2 API 实现

回调的整个工作流可以分为两步，第一步是完成一个指定任务，然后调用传递进去的回调函数。整个流程封装在 executer()函数中，executer()函数首先调用一个抽象函数 exec()，接着调用传递给 executer 函数的 callback 类型的回调函数。

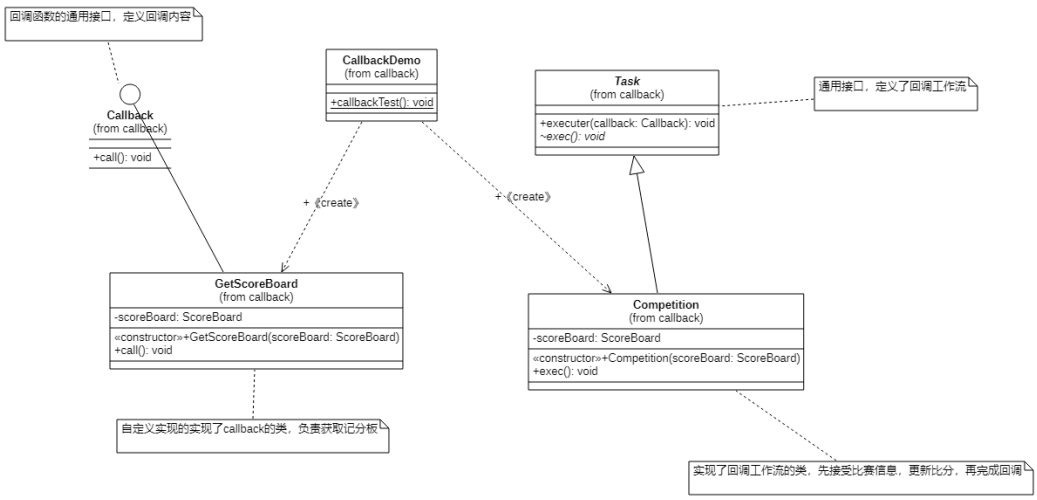
exec()函数是实现具体的回调工作流当中的第一个任务。

callback 是一个接口类，其中的 call()即为用户指定的完成了回调工作流的任务后需要执行的回调步骤。

3.43.1.3 函数功能对照表

函数名	作用
call():void	callback 中的回调函数
exec():void	在回调函数之前执行的任务
executer(Callback):void	该函数完成了回调的整个工作流，先执行 exec 的任务，然后实现回调。

3.43.2 类图



3.43.3 优缺点分析

（1）优点：

i. executer 封装了多种回调函数，使得更加灵活

ii. 整个工作流被封装，用户可以自定义其中的具体动作。

（2）缺点：

i. 每一个任务都需要重写 executer，增加了程序的复杂度。

3.44 Business Delegate

3.44.1 实现 API 描述

3.44.1.1 设计思路

业务代表模式（Business Delegate Pattern）用于对表示层和业务层解耦。它基本上是用来减少通信或对表示层代码中的业务层代码的远程查询功能。

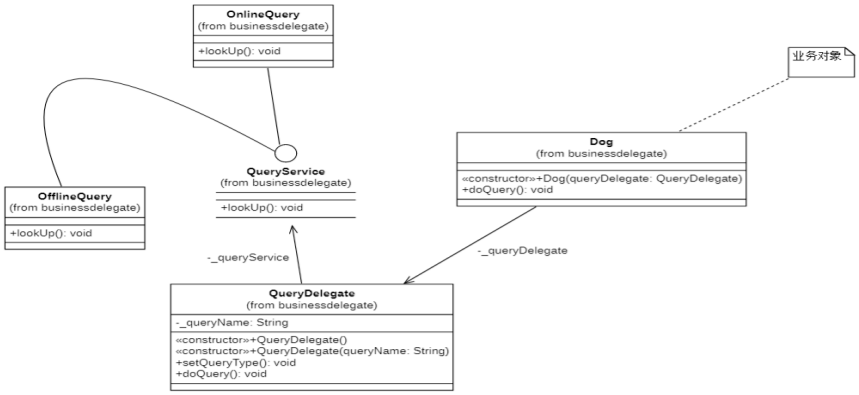
3.44.1.2 API 实现

以运动员查询成绩作为场景来实现 Business Delegate 设计模式。定义类 Dog、QueryDelegate 类来演示业务代表模式的用法。新建 QueryDelegate 类，在构造函数中传入"offline"或"online"字符串表明对象负责线上或是线下查询，随后 QueryDelegate 类传入委托方 Dog 类中，在 Dog 类中调用事先定义的查询方法即可完成查询。

3.44.1.3 函数功能对照表

函数名	作用
QueryDelegate.setQueryType():void	获取查询类实体对象
lookUp():void	执行查询
Dog.doQuery():void	实体类执行查询

3.44.2 类图



3.44.3 优缺点分析

（1）优点：

- 做到了业务代表表示层与业务层的解耦

3.45 Immutable

3.45.1 实现 API 描述

3.45.1.1 设计思路

Immutable 就是不变的、不发生改变的意思。Immutable 模式中存在着确保实例状态不发生改变的类(immutable 类)。在访问这些实例时并不需要执行耗时的互斥处理，因此若能巧妙地利用该模式，提高程序性能。

在项目中，考虑到奖牌 Medal 的属性一经创建就不再改变。我们将 Medal 类定义为 immutable 类。具体实现方法见下。

3.45.1.2 API实现

将Medal类定义为immutable类，首先将 Medal 类声明为 final 类型，这表明不能为其创建子类，虽然这并不是 Immutable 模式的必要条件，但却是防止子类修改其字段值的一种措施；其次将 Medal 类的_material 和_sport 字段的可见性都设为 private，这样这两个字段都只有从该类的内部才能访问，这也不是 Immutable 模式的必要条件，而是防止子类修改其字段值的一种措施；另外，Medal 类的字段_material 和_sport 字段都声明为 final，意即一旦字段被赋值一次，就不会再被赋值。这时，即便多个线程同时访问同一个实例，Medal 类也是安全的。

PrintMedalThread类用于持续显示构造函数中传入的Medal类实例。显示的字符串格式如下：

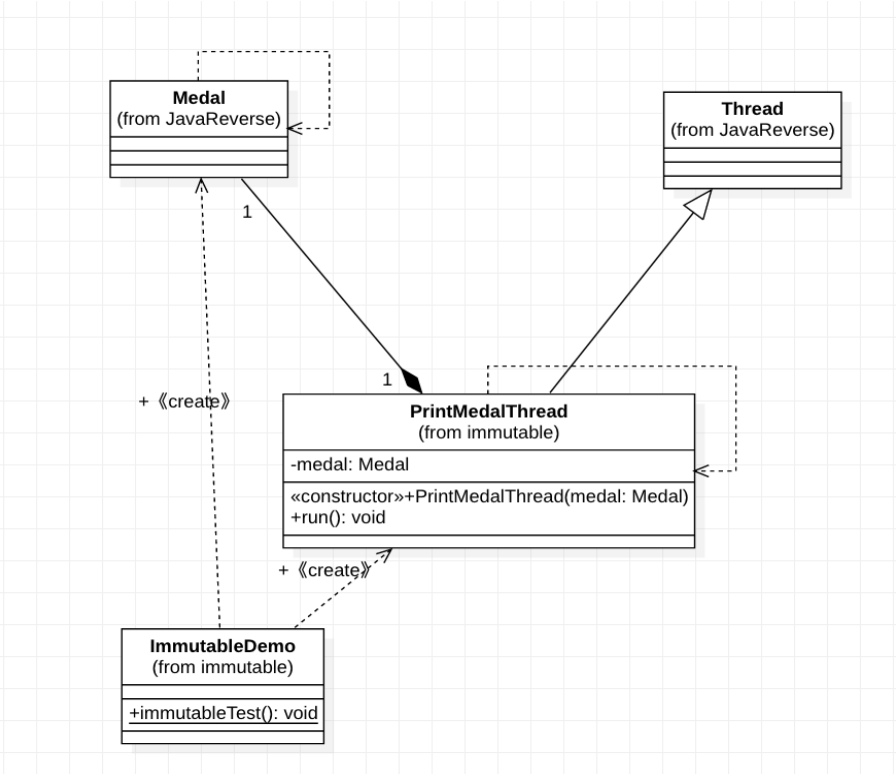
Thread.currentThread().getName() + " prints " + medal

同一实例传入多个构造函数当中，打印出的上述字符串应该只有线程名是不同的。

3.45.1.3 函数功能对照表

函数名	作用
PrintMedalThread.run()	用于打印线程名称及线程中所使用的 Medal 实例

3.45.2 类图



3.45.3 优缺点分析

（1）优点：

- 对于适用于 Immutable 模式的类（immutable 类），我们无需再使用 synchronized 方法执行线程的互斥处理，因此即便不使用 synchronized，也能确保安全性。

（2）缺点：

确保类的不可变性有时是一项出乎意料的难题。