设计模式汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 设计模式名称 | 实现个数 | 例程序个数 | 备注说明 |
| 1 | Abstract Factory  抽象工厂模式 | 1 | 1 | 使用抽象工厂创建不同系列主题（北极熊主题，企鹅主题，海豹主题）的游乐设施：碰碰车、过山车、餐厅等 |
| 2 | Adapter  适配器模式 | 1 | 1 | 使得游乐设施的椅子可以适配未成年人，拓展了游乐设施的实用性。 |
| 3 | Bridge  桥接模式 | 1 | 1 | 使得整个娱乐场的公告板的状态保持统一 |
| 4 | Builder  建造者模式 | 1 | 1 | 保证整个娱乐场的时间只有一个实例 |
| 5 | Chain of Responsibility责任链模式 | 1 | 1 | 娱乐园中会面临解决游客投诉的情况，该模式形成了根据可以解决该投诉的责任链来对投诉进行处理。 |
| 6 | Command  命令模式 | 1 | 1 | 娱乐园中的游乐设施会有安全保护设备，该模式记录了安全设备的状态，并且可供操作员下达“升起”“放下”的命令。 |
| 7 | Composite  组合模式 | 1 | 1 | 记录将娱乐园中的员工的信息，并允许他们根据职位形成一条从属链/一棵从属树。 |
| 8 | Decorator  装饰器模式 | 1 | 1 | 通过多种装饰器实现娱乐园门票套餐的自由组合，增加了套餐生成的灵活性。 |
| 9 | Facade  外观模式 | 1 | 1 | 建立检票点作为一个接口，简化检票行为。 |
| 10 | Factory  工厂模式 | 1 | 1 | 可供拥有者下达建造新设施的命令，然后进行建造。 |
| 11 | Fly Weight  享元模式 | 1 | 1 | 共享游戏币实例 |
| 12 | Interpreter  解释器模式 | 1 | 1 | 模拟购买饮料的场景，根据输入的合法表达式计算出总价值 |
| 13 | Iterator  迭代器模式 | 1 | 1 | 提供一种方法可以分类别（老年人，军人，儿童，成年人）对排队的来客进行迭代顺序访问，而又无需暴露该队列的内部表示 |
| 14 | Mediator  中介者模式 | 1 | 1 | 使用旅行团作为中介，降低旅行团中各个成员互相通信时的复杂度使得进行团体行动的行为变得更清晰 |
| 15 | Memento  备忘录模式 | 1 | 1 | 游戏机（GamePlayer）重玩时需要恢复初始状态时使用Memento模式 |
| 16 | Observer  观察者模式 | 1 | 1 | 减少在对娱乐园中各种各样的设备进行测试时，需要的用于创建对象所用的代码 |
| 17 | Prototyoe  原型模式 | 1 | 1 | 使用原型模式大量生成入场券对象，其提供了一个创建对象的最佳方式。 |
| 18 | Proxy  代理模式 | 1 | 1 | 冰雪天地娱乐园内的总经理掌管许多方面的事务，而关于财务方面的工作可以由娱乐园内的财务经理代为管理，即使用代理模式。 |
| 19 | Singleton  单例模式 | 1 | 1 | 保证整个娱乐场的时间只有一个实例 |
| 20 | State  状态模式 | 1 | 1 | 使用旅行团作为中介，降低旅行团中各个成员互相通信时的复杂度，使得进行团体行动的行为变得更清晰 |
| 21 | Strategy  策略模式 | 1 | 1 | 有经验的游客对于景点间的移动方式轻车熟路，游客类的旅行行为可以在运行时根据目的地进行更改 |
| 22 | TemplateMethod  模板方法模式 | 1 | 1 | 游乐场里的小火车轨道上有很多种车厢——虽然长得不一样，能进行的服务也不一样，但是它们都是火车的车厢。维护人员可以从一个车的模板中创造出新的服务车厢来满足游客的新需求。 |
| 23 | Visitor  访问者模式 | 1 | 1 | 游乐场有一个经理。经理下属有几个分管各个模块的副经理。每个副经理下属有多个员工。可以从一个人（树中的节点）身上找到和他对应的别人，比如从一个员工往上找到他的经理。 |
| 24 | Active Object  主动对象模式 | 1 | 1 | 通过使意见处理中心持有一个线程使得游客的反馈意见可以被异步处理，让游客可以在意见被处理后得到通知。 |
| 25 | Balking  犹豫模式 | 1 | 1 | 减少服务员向顾客发起无用的点单请求。 |
| 26 | CallBack  回调模式 | 1 | 1 | 旅行团在游玩过程中有人会需要上厕所，上厕所结束的游客会告知旅行团其他人 |
| 27 | Converter  转换器模式 | 1 | 1 | 使用转换器将会员信息（MemebershipDto）转换成可供售票窗口使用的会员实例（Membership） |
| 28 | DataAccessObject  数据访问对象模式 | 1 | 1 | 持久化游乐园顾客数据到本地文件系统 |
| 29 | Delegation  委托模式 | 1 | 1 | 委托售票处判断游客购买的是儿童票还是成人票，使请求处理更加简洁。 |
| 30 | Dependency Injection  依赖注入模式 | 1 | 1 | 通过依赖注入，游客游玩不同的游乐设施。 |
| 31 | Dirty Flag  脏标记模式 | 1 | 1 | 更新电子游戏机电子菜单界面 |
| 32 | Private Class Data  私有类数据 | 1 | 1 | 游玩娱乐场的各种项目可以在兑奖券上收集奖章，奖章数量收齐后可以尝试兑换奖品 |
| 33 | Data Transfer Object  数据传输对象模式 | 1 | 1 | 创建数据传输对象简化滑雪工具的租借。 |
| 34 | Feature Toggle  特性切换模式 | 1 | 1 | 游乐园内的设有滑雪赛道，共有两条；通过不同的赛道类进行特性的切换 |
| 35 | Front Controller前端控制器模式 | 1 | 1 | 电子游戏机根据对应的用户角色（管理员/用户）分发对应的视图 |
| 36 | Future/Promise回调模式 | 1 | 1 | 顾客下单菜品（Dishes）后，厨房将需求发完之后，继续处理下一个订单，厨房接到订单后开始处理，处理后返回并提醒厨房中等待的进程。 |
| 37 | immutable  不变模式 | 1 | 1 | 游乐园内的娱乐设施位置是固定的，不能进行更改。 |
| 38 | lazy-loading  延迟加载模式 | 1 | 1 | 舞台是一个能够供客人观赏的场所，它有个内置类演员持有人用来对演员进行调度 |
| 39 | Marker Interface  标记接口模式 | 1 | 1 | 区分娱乐园中的保安与小偷 |
| 40 | Monostate  单态模式 | 1 | 1 | 使得整个娱乐场的公告板的状态保持统一 |
| 41 | Multition  多例模式 | 1 | 1 | 游乐设施的椅子使用多例模式创建并维护。有固定数目，单独提供服务。 |
| 42 | MVC模式 | 1 | 1 | 使用MVC展示用户评价 |
| 43 | NullObject  空对象模式 | 1 | 1 | 游乐园的每个项目都要检查票根，但通过票根查询通票的时候可能遇到非法的票据，此时返回空票据(NullTicket）代替（null) |
| 44 | Object Mother | 1 | 1 | 减少在对娱乐园中各种各样的设备进行测试时，需要的用于创建对象所用的代码 |
| 45 | Object Pool  对象池 | 1 | 1 | 电影院中使用的眼镜成本巨大，想重新生产一个成本巨大，所以我们创建了一个眼镜池，使用一个就从里面拿一个。 |
| 46 | Pipeline  管道模式 | 1 | 1 | 维护游乐园内的设备，通过停用检修维护测验复用等流程组成管道实现 |
| 47 | Publish-Subscribe  发布-订阅者模式 | 1 | 1 | 在冰雪天地娱乐园内，游客可以通过游乐园的App订阅相关游乐项目信息，游乐园的相关项目也可以对订阅其的游客进行消息的发布。 |
| 48 | ReadWritelock  读写锁模式 | 1 | 1 | 在冰雪天地娱乐园的影院中，存在影院的观众与维修幕布的工人，当维修工人进行幕布的维修时，观众不能进行幕布上电影的观看，而所有观众可以一起进行电影的观看。 |
| 49 | Specification (Filter, Criteria)  过滤器模式 | 1 | 1 | 游乐场内有多种设施，每种设施有不同的特征，比如设施类型（https://baike.baidu.com/item/%E6%B8%B8%E4%B9%90%E8%AE%BE%E6%96%BD），设施危险度，设施适合年龄段等等。用户根据所需特征来筛选出他所需要的游乐设施的列表。如果不满足则返回空列表。 |
| 50 | Thread Pool  线程池模式 | 1 | 1 | 众所周知，游乐场负责接待游客的人员新找一个要花费大量成本和时间。我们在游乐园开园的时候维护一个接待人员的队列，每次需要提供服务就从队列里面把人拿出来。服务结束后，人员回到队列里待命。当所有人都在接待时，客人需要等待。 |
| 51 | Transfer Object  传递对象模式 | 1 | 1 | 提供一种方法可以分类别（老年人，军人，儿童，成年人）对排队的来客进行迭代顺序访问，而又无需暴露该队列的内部表示 |
| 52 | Value Object  值对象模式 | 1 | 1 | 比较游戏中英雄的属性值 |

目录

[1 抽象工厂(Abstract Factory)模式 14](#_Toc57491435)

[2 适配器(Adapter)模式 16](#_Toc57491436)

[3 桥接(Bridge)模式 19](#_Toc57491437)

[4 建造者(Builder)模式 21](#_Toc57491438)

[5 责任链(Chain of Responsibility)模式 24](#_Toc57491439)

[6 命令(Command)模式 26](#_Toc57491440)

[7 组合(Composite)模式 29](#_Toc57491441)

[8 装饰器(Decorator)模式 31](#_Toc57491442)

[9 外观(Facade)模式 33](#_Toc57491443)

[10 工厂(Factory)模式 35](#_Toc57491444)

[11 享元（FlyWeight）模式 37](#_Toc57491445)

[12 解释器(Interpreter)模式 39](#_Toc57491446)

[13 迭代器(Iterator)模式 41](#_Toc57491447)

[14 中介者(Mediator)模式 44](#_Toc57491448)

[15 备忘录(Memento)模式 47](#_Toc57491449)

[16 观察者(Observer)模式 49](#_Toc57491450)

[17 原型(Prototype)模式 52](#_Toc57491451)

[18 代理(Proxy)模式 55](#_Toc57491452)

[19 单例(Singleton)模式 57](#_Toc57491453)

[20 状态(State)模式 60](#_Toc57491454)

[21 策略(Strategy)模式 63](#_Toc57491455)

[22 模板(Template Method)模式 66](#_Toc57491456)

[23 访问者(Visitor)模式 68](#_Toc57491457)

[24 主动对象(Active Object)模式 70](#_Toc57491458)

[25 犹豫（Balking）模式 72](#_Toc57491459)

[26 回调(Callback)模式 74](#_Toc57491460)

[27 转换器（Converter）模式 76](#_Toc57491461)

[28 数据访问对象(DAO)模式 78](#_Toc57491462)

[29 委托(Delegation)模式 80](#_Toc57491463)

[30 依赖注入(Dependency Injection)模式 82](#_Toc57491464)

[31 脏标记(Dirty Flag)模式 84](#_Toc57491465)

[32 私有类数据（Private Class Data）模式 87](#_Toc57491466)

[33 数据传输对象(dto)模式 90](#_Toc57491467)

[34 特性切换（Feature Toggle）模式 92](#_Toc57491468)

[35 前端控制器(Front Controller)模式 94](#_Toc57491469)

[36 回调(Future/Promise)模式 96](#_Toc57491470)

[37 不变(immutable)模式 98](#_Toc57491471)

[38 延迟加载 (lazy-laoding) 模式 100](#_Toc57491472)

[39 标记接口(Marker Interface)模式 102](#_Toc57491473)

[40 单态(MonoState)模式 105](#_Toc57491474)

[41 多例（Multition）模式 107](#_Toc57491475)

[42 MVC模式 109](#_Toc57491476)

[43 空对象（Null Object）模式 112](#_Toc57491477)

[44 Object Mother模式 114](#_Toc57491478)

[45 对象池(Object Pool)模式 118](#_Toc57491479)

[46 管道(Pipeline)模式 120](#_Toc57491480)

[47 发布-订阅（Publish-Subscribe）模式 122](#_Toc57491481)

[48 读写锁(Read-Write Lock)模式 126](#_Toc57491482)

[49 过滤器(Specification/Filter)模式 130](#_Toc57491483)

[50 线程池(Thread Pool)模式 132](#_Toc57491484)

[51 传输对象(Transfer Object)模式 134](#_Toc57491485)

[52 值对象(Value Object)模式 137](#_Toc57491486)

# 抽象工厂(Abstract Factory)模式

增加游乐设施创建的可拓展性

## 模式简介

抽象工厂模式为一个产品家族提供了统一的创建接口。当需要这个产品家族的某一系列的时候，可以从抽象工厂中选出相对系的系列来创建一个具体的工厂类别。抽象工厂属于创建型设计模式。

设施工厂（FacilityFactory）是一个接口，它提供三个方法：（createBumperCar）创建碰碰车，创建过山车（createRollerCoaster），创建餐馆（createRestaurant）。PenguinFacilityFactory，PolarBearFacilityFactory，SealFacilityFactory均继承自设施工厂（FacilityFactory），分别可以实例化不同风格的游乐设施。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

FacilityFactory. createBumperCar ()

**说明：**创建碰碰车实例

**参数：**无

**返回**：返回一个碰碰车实例

FacilityFactory. createRollerCoaster ()

**说明：**创建过山车实例

**参数：**无

**返回**：返回一个过山车实例

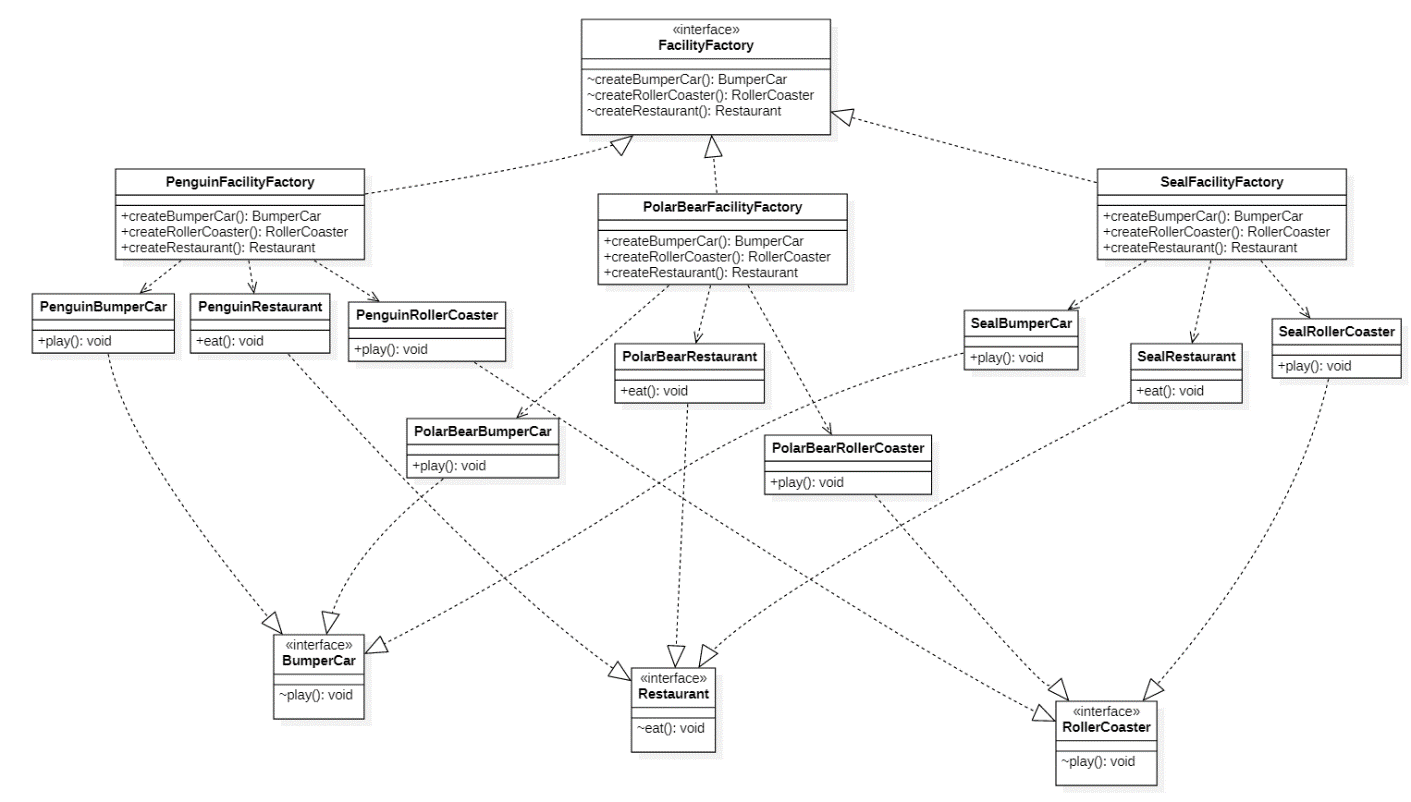
FacilityFactory. createRestaurant ()

**说明：**创建餐馆实例

**参数：**无

**返回**：返回一个餐馆实例

## 类图



# 适配器(Adapter)模式

为未成年人设置游乐设施适配器

## 模式简介

适配器模式（有时候也称包装样式或者包装）可以将将一个类的接口适配成用户所期待的。一个适配允许通常因为接口不兼容而不能在一起工作的类工作在一起，做法是将类自己的接口包裹在一个已存在的类中。

冰雪娱乐场中的各种游乐设施的座椅（Chair）有着一定的身高要求，成年人（Adult）可以直接测量身高并检测是否可以坐下。对于未成年人（Child），可以通过未成年人适配器（ChildAdapter）来加高座椅之后再检测身高能否满足游玩要求。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

Chair.sit(Adult person)

**说明：**人在椅子上坐下

**参数：**person表示试图坐在椅子上的人

**返回**：无

Adult. getAdultHeight ()

**说明：**获得成年人身高

**参数：**无

**返回**：一个Double，表示当前成年人的身高

Child. getChildHeight ()

**说明：**获得未成年人身高

**参数：**无

**返回**：一个Double，表示当前未成年人的身高

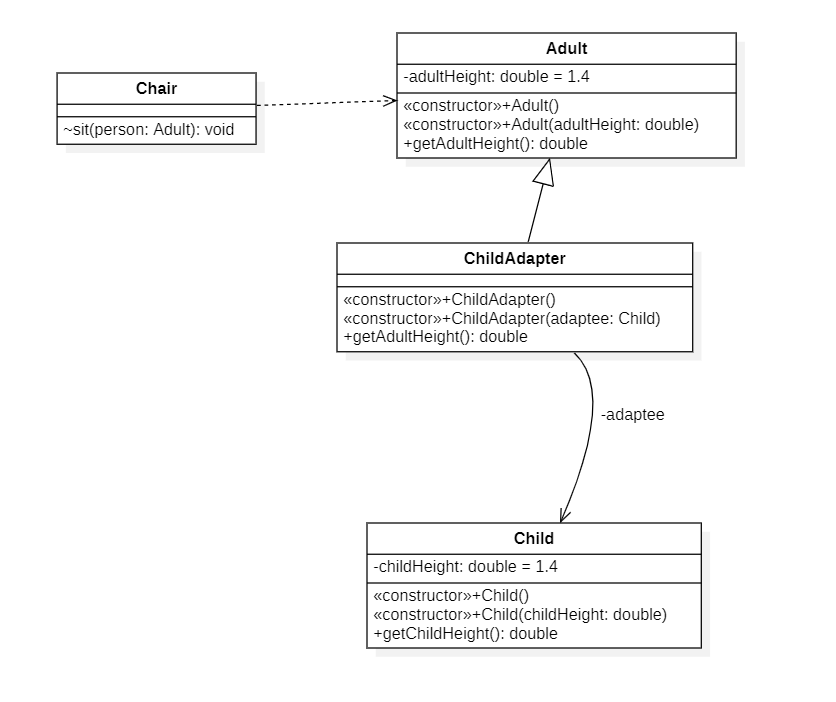
ChildAdapter. getAdultHeight ()

**说明：**获得未成年人适配后的对应成年人身高

**参数：**无

**返回**：一个Double，表示适配后的身高

## 类图



# 桥接(Bridge)模式

多种舞台表演多种节目

## 模式简介

桥接模式是将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interfce)模式。

冰雪娱乐场中的各地都有着大小不同的舞台(Stage)，分为大舞台(BigStage)，小舞台(SmallStage)，而对于每个舞台，都有不同的节目(Performance)可以表演，预设的两个节目为歌唱表演(SingPerformance)，跳舞表演(DancePerformance)，为了让多个舞台可以表演不同的表演，我们使用桥接模式将舞台和节目桥接。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

Stage.modifyActivity(Performance newPerformance)

**说明：**更换表演的节目

**参数：**newPerformance 新的节目

**返回**：无

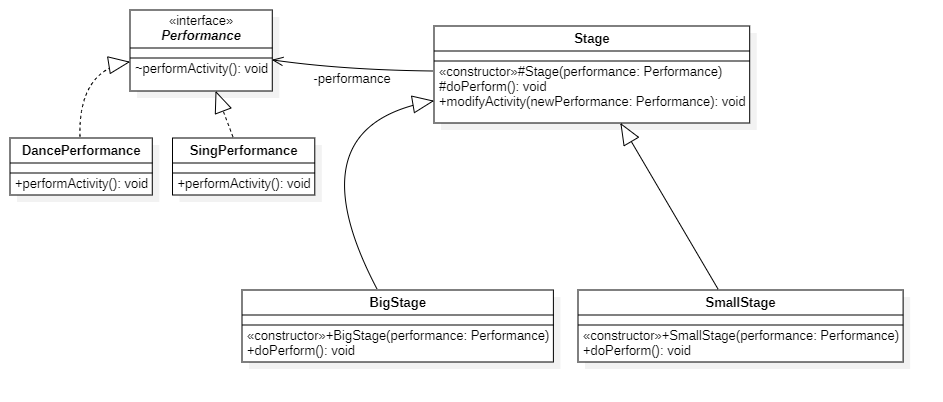
Stage.doPerform()

**说明：**表演预设的节目

**参数：**无

**返回**：无

## 类图



# 建造者(Builder)模式

游乐园内快餐店套餐

## 模式简介

建造者模式就是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

在游乐园中，一份套餐(Meal)是由服务员(Waiter)通过预设的套餐(Meal Builder)提供，预设的套餐是由一个汉堡(Burger)和一杯饮料(drink)组成的。汉堡(Burger)分为经典汉堡(Hamburger)或鸡肉汉堡(Chicken Burger)，饮料分为果汁(Juice)和可乐(Cola)。它们各自由不同的外包装打包，其中果汁是用杯子(Cup)打包，汉堡是用纸盒(Wrapper)打包。

具体来说，我们将创建一个表示食物条目的 Item 接口和实现 Item 接口的实体类，以及一个表示食物包装的 Packing 接口和实现 Packing 接口的实体类。

然后我们创建一个 Meal 类，带有 Item 的 ArrayList 和一个通过结合 Item 来创建不同类型的 Meal 对象的 MealBuilder。Waiter类使用 MealBuilder 来创建一个 Meal。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

Waiter. setMealBuilder(MealBuilder mealBuilder)

**说明：**设置需要准备的套餐

**参数：**特定套餐 mealBuilder

**返回**：无

Waiter.prepareMeal()

**说明：**按照mealBuilder准备相应的套餐

**参数：**无

**返回**：无

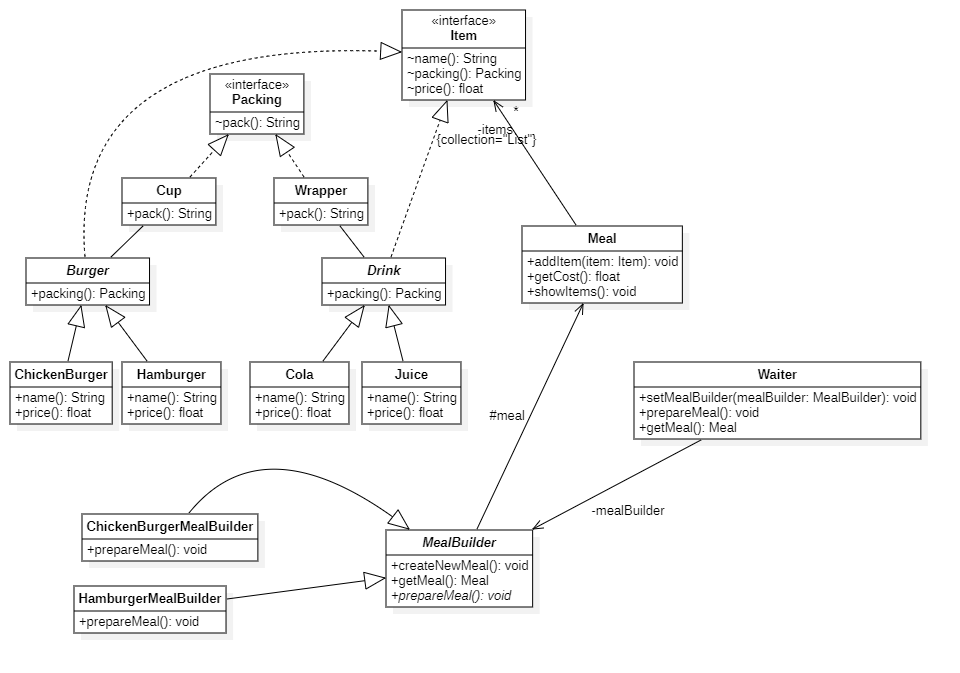
Waiter.getMeal()

**说明：**获取准备完毕的套餐

**参数：**无

**返回**：套餐Meal，若未执行prepareMeal()，则会返回空对象

## 类图



# 责任链(Chain of Responsibility)模式

处理游客对娱乐园的投诉

## 模式简介

责任链模式（Chain of Responsibility Pattern）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以形成一条责任链，责任链上的对象都有权限处理请求，并且请求会沿着该链传递知道该请求被解决。

在冰雪天地娱乐园中，需要面对处理游客投诉的情况，而处理投诉的人不止一个，是有一个自下而上传递的过程的。所以运用责任链模式，对游客的投诉进行传递，直到它被处理。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 223ff. ISBN 0-201-63361-2.

## API描述

ComplaintHandler. handleComplaint(Complaint complaint)

**说明：**解决投诉

**参数：**游客的具体投诉

**返回**：无

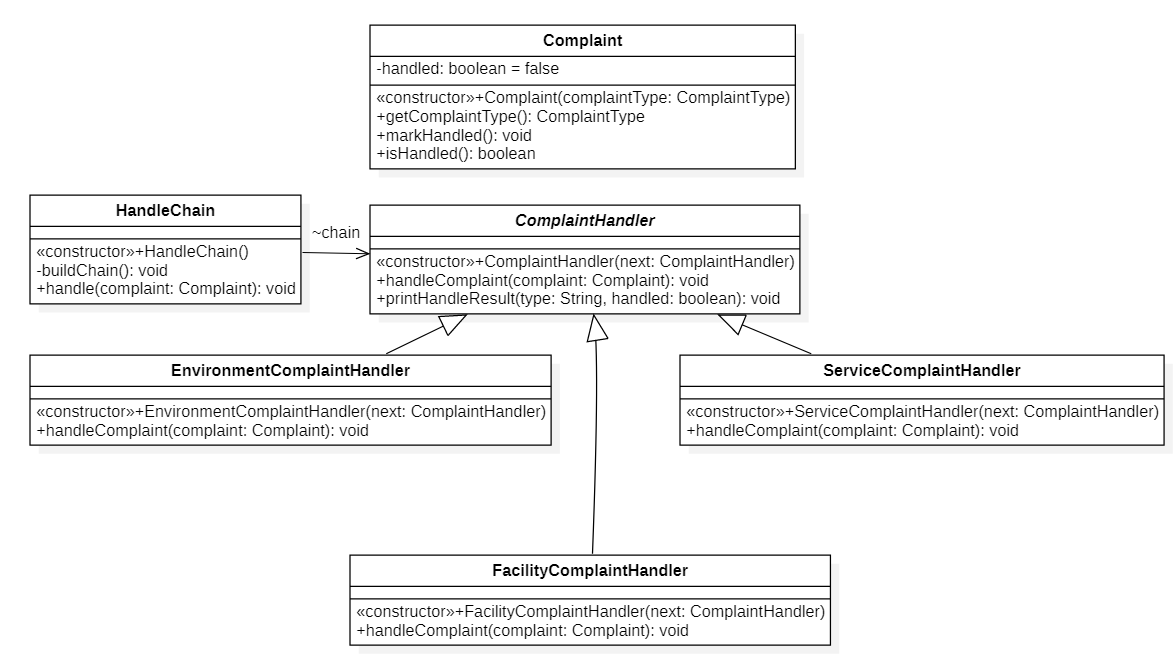
HandleChain. buildChain()

**说明：**形成具体的责任链

**参数：**无

**返回**：无

## 类图



# 命令(Command)模式

对娱乐园内部分设施的安全保护设备实施命令

## 模式简介

命令模式（Command Pattern）将用户的请求封装为一个对象，然后将这个封装起来的对象传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。该模式可以通过不同的命令对某一对象进行参数化。

在冰雪天地娱乐园中，会有不同的游乐设施，其中有的设施会有保护游客安全的安全设备，有升起、放下两种形态。设施的管理员通过操作台对安全设备下达升起和放下两种命令，运用命令模式来模拟该场景。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

Command. putUp()

**说明：**改变安全设备的状态为升起并输出状态

**参数：**无

**返回**：无

Command. putDown()

**说明：**改变安全设备的状态为放下并输出状态

**参数：**无

**返回**：无

Operator. takeCommand(Order order)

**说明：**调用执行命令的接口来解决请求

**参数：**给定的命令

**返回**：无

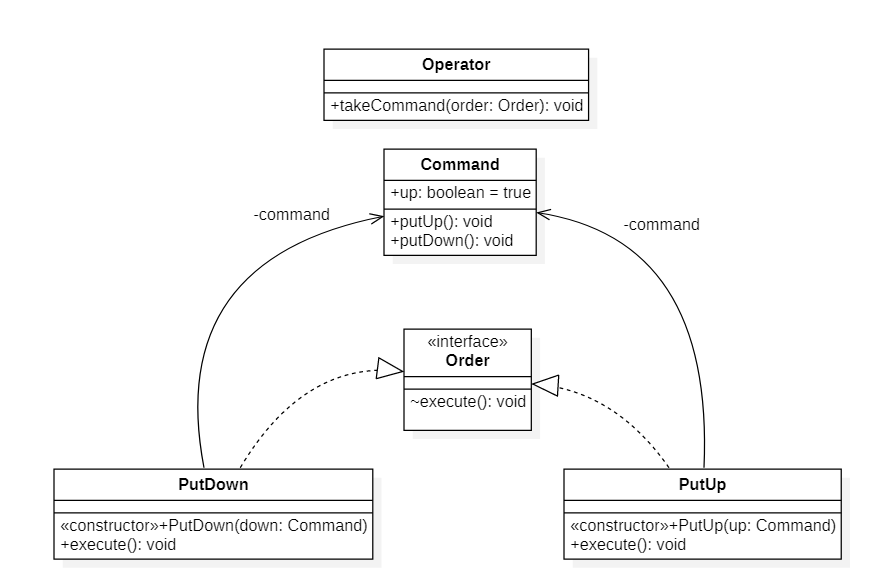
Order. execute()

**说明：**执行命令

**参数：**无

**返回**：无

## 类图



# 组合(Composite)模式

建立娱乐园的所有员工的一个从属链

## 模式简介

组合模式（Composite Pattern），又叫部分整体模式。它把一组相似的对象当作一个单一的对象，然后给它们的属性进行不同的赋值。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，体现出了一个层级的关系。

在冰雪天地娱乐园中，整个娱乐园的员工有各自的姓名职位职责薪水年龄等等，同时他们按照职位会形成一条从属链，或者说形成一棵树。组合模式可以帮助新建新的员工对象，以及添加或者删除他们的职位的从属关系。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

Employee. Employee(String name, String pos, String res, int salary, int age)

**说明：**构造函数，新建一个员工对象时对各个属性进行赋值

**参数：**用于对属性进行赋值

**返回**：无

Employee. add(Employee employee)

**说明：**添加新的从属关系

**参数：**被添加的员工

**返回**：无

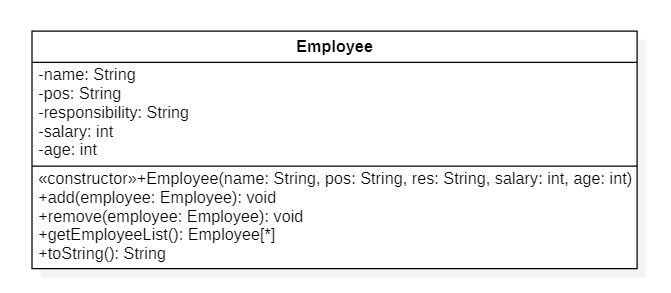
Employee. remove(Employee employee)

**说明：**删除从属链上的某个员工及其下属

**参数：**被删除的员工

**返回**：无

## 类图



# 装饰器(Decorator)模式

灵活组合各种套餐门票

## 模式简介

装饰模式指的是在不必改变原类文件和使用继承的情况下，动态地扩展一个对象的功能。它是通过创建一个包装对象，也就是装饰来包裹真实的对象。装饰模式属于结构型模式。

参观套餐（VisitPackage）是一个接口，它提供了获取价格（getPrice），获取内容（getContent）两个方法，门票（Ticket）继承于VisitPackage，它可以存储门票的基本价格和基本内容。基础装饰器（BaseDecorator）也继承于VisitPackage，作为其余装饰器的父类,其中BumperCarDecorator,IceSculptureExhibitionDecorator,RollerCoasterDecorator都继承于BaseDecorator。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

VisitPackage. getPrice ()

**说明：**获取套餐价格

**参数：**无

**返回**：一个Double表示价格

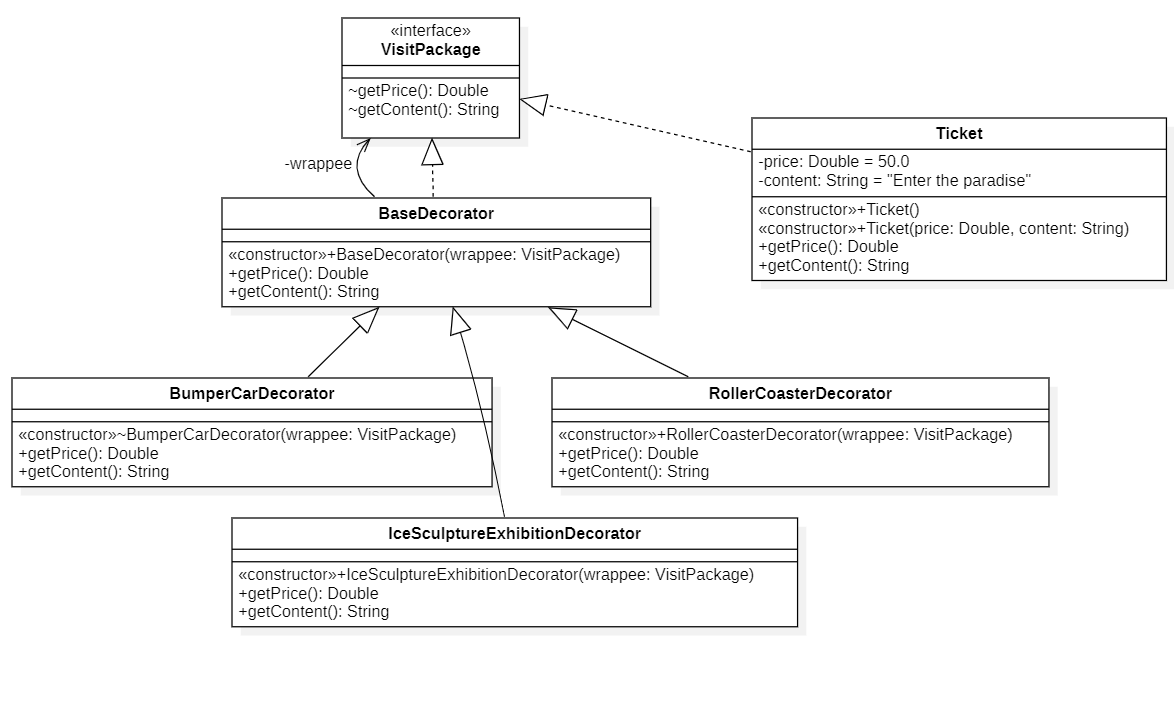
VisitPackage. getContent ()

**说明：**获取套餐内容

**参数：**无

**返回**：一个String表示内容

## 类图



# 外观(Facade)模式

游客入园检票

## 模式简介

外观模式（Facade Pattern）隐藏系统的复杂性，并向客户端提供了一个客户端可以访问系统的接口。这种类型的设计模式属于结构型模式，它向现有的系统添加一个接口，来隐藏系统的复杂性。这种模式涉及到一个单一的类，该类提供了客户端请求的简化方法和对现有系统类方法的委托调用。

进入冰雪天地娱乐园需要检票，我们添加一个检票点作为接口，对游客进行娱乐园入园的检票。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

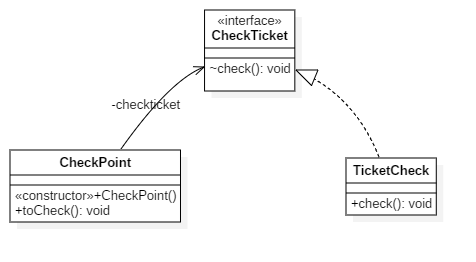
checkCollector.toCheck()

**说明：**对入园的游客进行检票

**参数：**无

**返回**：无

## 类图

****

# 工厂(Factory)模式

建造新的游乐设施

## 模式简介

工厂模式（Factory Pattern）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。注意在工厂模式中，在创建对象时不会对用户暴露创建逻辑，只提供一个接口来让用户创建对象，然后由这个接口自己的子类去判定创建什么具体的实例。

在冰雪天地娱乐园中，需要修建不同的游乐设施。对于娱乐园的拥有者而言，他只需要规定创建什么设施，而不需要知道该设施如何建造，使用工厂模式可以实现这一点。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

BuildFacility. build(String name)

**说明：**根据不同的设施名称返回不同的设施实例

**参数：**设施名称

**返回**：设施实例

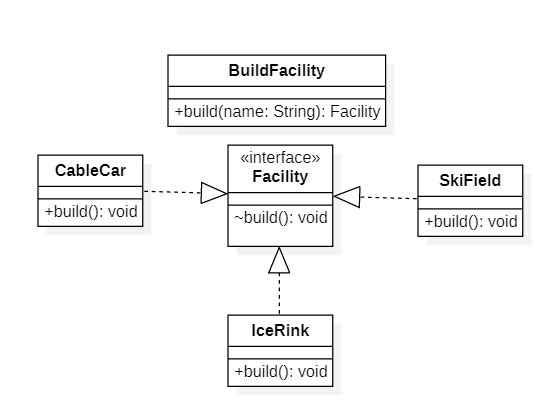
Facility. build()

**说明：**提供建造新的设施的接口

**参数：**无

**返回**：无

## 类图



# 享元（FlyWeight）模式

共享游戏币实例

## 模式简介

从实际业务场景的角度分析，冰雪天地娱乐园中有很多顾客，每个顾客都可以拥有属于自己的游戏币(Coin). 但从内存管理的角度分析，同类的游戏币不会因为在哪个顾客手中而有所不同。因此，可以利用享元（FlyWeight）模式共享游戏币实例来节省内存占用。

在享元模式中实现了简单的Abstract Factory设计模式的金币工厂，享元模式的实现嵌套在金币工厂中。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

GoldCoinFactory.createBigCoin()

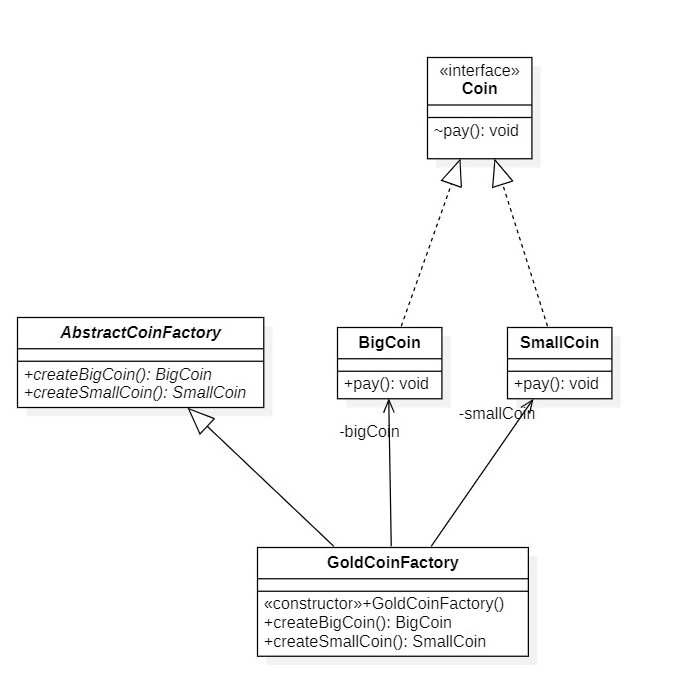
GoldCoinFactory.createSmallCoin()

**说明：**创建游戏币

**参数：**无

**返回**：游戏币Coin

## 类图



# 解释器(Interpreter)模式

解析表达式计算总价值

## 模式简介

项目模拟了一个出售饮料的场景，解释器模式提供了计算饮料总价值的功能。我们定义了自己的一套语法规则，并在Parser中实现了语法分析功能，根据输入的合法表达式生成了对应的语法树，在代码实现中表示为一个ExpressionNode，调用ExpressionNode中的interpret方法和execute方法解析并计算饮料的总价值。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

ExpressionNode.interpret()

**说明：**解析表达式

**参数：**无

**返回**：无

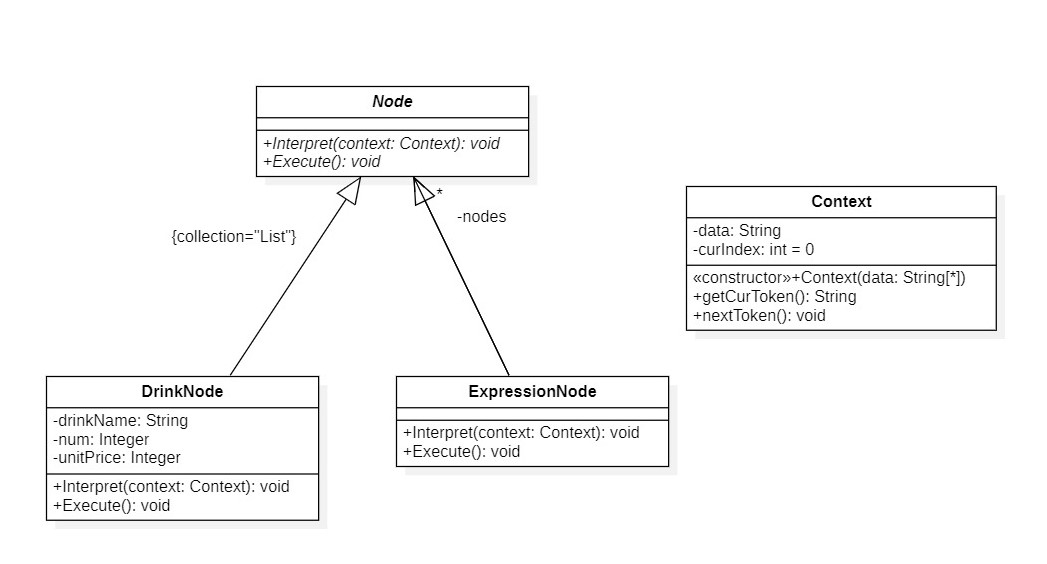
ExpressionNode.execute()

**说明：**计算已解析表达式的值

**参数：**无

**返回**：计算结果

## 类图



# 迭代器(Iterator)模式

分类别对排队的游客进行顺序访问

## 模式简介

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。

在冰雪天地娱乐场中（Ice Paradise），有着各种各样的游客在检票处排队（TouristsInLine），里面有老人（ELDERLY），儿童(CHILDREN)，成年人(ADULT)和光荣的退伍军人（MILITARY）。他们会有各自的队伍，但是对于正在排队的游客这一个聚合而言，我们可以简单地使用游客迭代器（TouristInLineIterator）对其进行不同方式的迭代，且不暴露聚合的内部结构。

此外，Iterator 是迭代器的接口，描述了迭代器模式基本的两个方法，判断是否有下一个对象（hasNext()）以及获得下一个对象（Next()）。游客类型（TouristType）则是一个枚举类，代表各种游客的类别。

参考出处：  
1. <https://java-design-patterns.com/patterns/iterator/>

2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

TouristInLineIterator.hasNext()

**说明：**判断队伍中是否还有下一个对象

**参数：**无

**返回**： 一个boolean表示是否有下一个对象

TouristInLineIterator.next()

**说明：**获取队伍中的下一个对象

**参数：**无

**返回**：一个游客（Tourist）实例

TouristsInLine.append(TouristType touristType, String name)

**说明：**在队伍的末尾新来了一个特定类型（touristType）的名为name的游客

**参数：**游客类型 touristType 游客姓名 name

**返回**：无

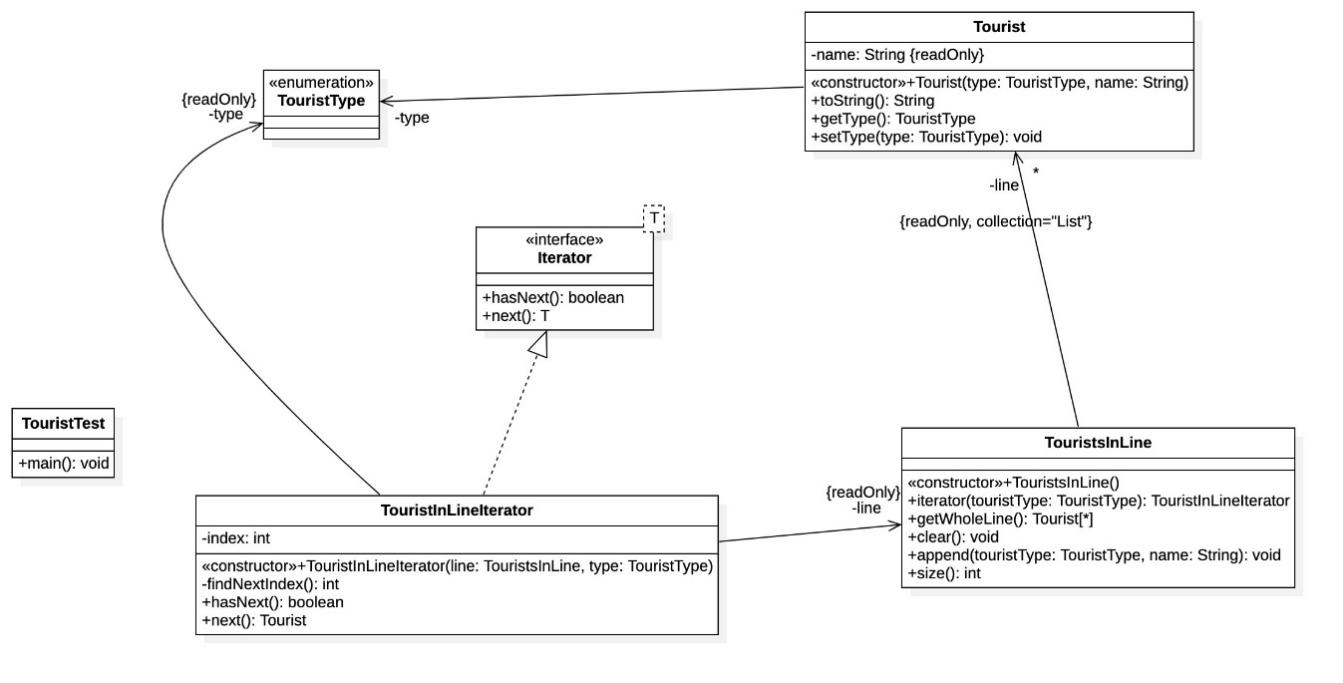
TouristsInLine.clear()

**说明：**清空队伍

**参数：**无

**返回**：无

## 类图



# 中介者(Mediator)模式

简化旅行团的团队行动

## 模式简介

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。

Action是一个枚举类，是一组常量，表示旅行团可以进行的行为。旅行团（TourGroup）和旅行团成员(GroupMember)是两个接口，描述了旅行团和成员各自的方法。相对应的，TourGroupImpl 和 GroupMemberBase 是旅行团和旅行团成员各自的实现。

每一个旅行团（TourGroupImpl）可以有多个旅行团成员（GroupMemberBase），通过addMember() 方法将成员加入至某一个旅行团中。Son，Father，Mother 和Tour Guide均继承自旅行团成员（GroupMemberBase），分别代表着各类的旅行团成员。

当旅行团（TourGroupImpl）中的某一个旅行团成员（GroupMemberBase）希望进行某一个行为时，他会通过旅行团（TourGroupImpl）通知旅行团中的其他旅行团成员，与其一起进行该行为。中介者模式在此处将旅行团作为旅行团中各个成员的中介，有效将原先的复杂的通讯网络简化为星形结构。

参考出处：

1. <https://java-design-patterns.com/patterns/mediator/>

2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

TourGroup.addMember(GroupMember member)

**说明：**旅行团添加新成员

**参数：**新添加的团队成员，一个GroupMember 实例

**返回**：无

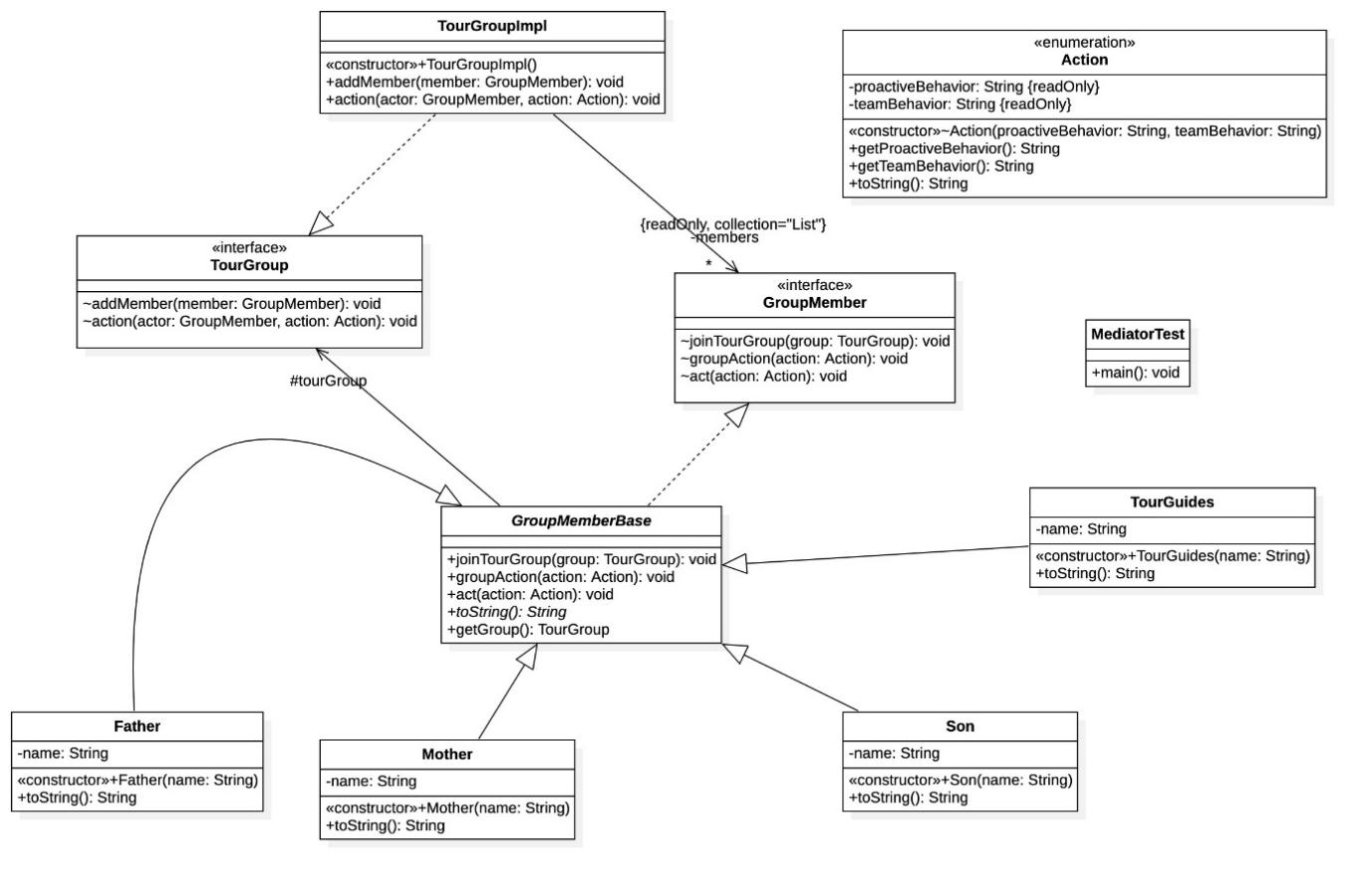
GroupMember.act(Action action)

**说明：**某个旅行团成员自发的进行某一个行为

**参数：**Action，取值为 Enum 类所限定

**返回**：无

## 类图



# 备忘录(Memento)模式

重启游戏

## 模式简介

游戏机（GamePlayer）是一个能够供客人游玩电子游戏的机器，它有一个内置类游戏备份类（GameMemento）用来创建存档或是保存初始游戏状态。

游戏机类在创建时创建一个游戏备份，当需要重新游戏时，只需要载入游戏备份即可。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

GameMemento.getBoard()

**说明：**对游戏机的棋盘状态进行存档

**参数：**空

**返回值：**空

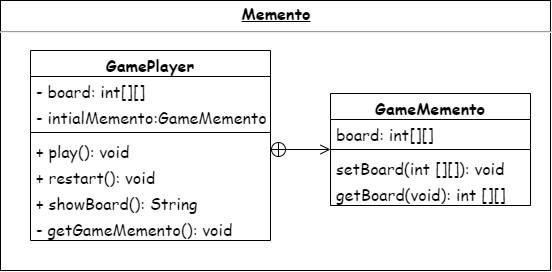
GameMemento.setBoard()

**说明：**将游戏机的棋盘恢复到存档状态

**参数：**无

**返回值：**无

## 类图



# 观察者(Observer)模式

游乐园内人数实时更新

## 模式简介

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

在本场景中，人数(PeopleNum)作为主题对象，而园内人数实时显示屏(NumScreen)和入口负责员工(EntranceEmployee)需要实时监测人数，显示屏需要实时更新人数，而员工则需要在人数过多时进行限流。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

PeopleNum.setVal(int val)

**说明：**更新peopleNum的值

**参数：**新的人数值val

**返回**：无

PeopleNum.getVal()

**说明：**获取peopleNum的当前值

**参数：**无

**返回**：peopleNum当前值 val

PeopleNum.attach(NumObserver newObserver)

**说明：**添加新的观察者(Observer)

**参数：**新的观察者newObserver

**返回**：无

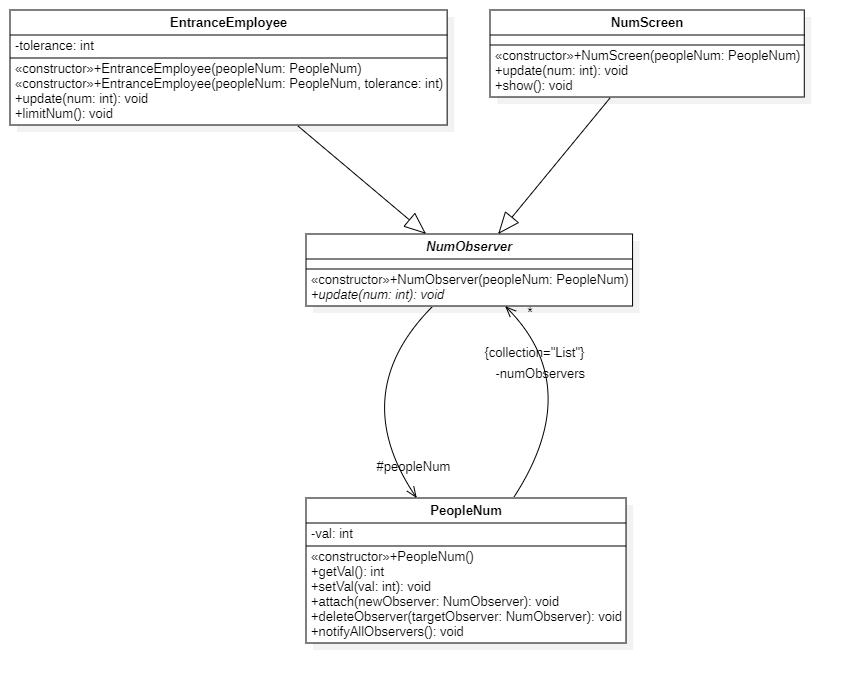
PeopleNum.deleteObserver(NumObserver targetObserver)

**说明：**删除指定观察者(Observer)

**参数：**需要删除的观察者targetObserver

**返回**：无

## 类图



# 原型**(Prototype)模式**

入场券的生成

## 模式简介

原型模式（Prototype Pattern）可以在保证性能的同时用于创建重复的对象，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。

在冰天雪地娱乐园中，入场券（Ticket）需要大量重复生成，每张同种类的入场券属性相同，在Ticket类内实现clone()方法，用于快速生成相同对象，

同时，Ticket类还提供深拷贝deepclone()方法，用于对Ticket类内的的数组和引用对象进行拷贝。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API 描述

Tikcet.clone()

**说明：**对入场券对象进行复制。

**参数：**空

**返回值：**新生成的入场券对象。

Ticket.deepclone()

**说明：**对入场券对象及其内部的数组与引用对象进行拷贝。

**参数：**无

**返回值：**新生成的深拷贝入场券对象。

Ticket.getInfo()

**说明：**获取入场券的基本信息。

**参数：**无

**返回值：**入场券对象基本信息的字符串。

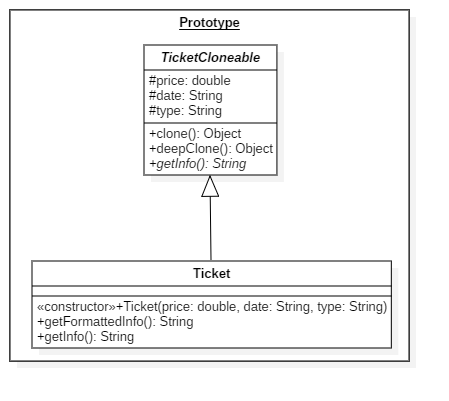
Ticket.getFormattedInfo()

**说明：**获取格式化的入场券基本信息。

**参数：**无

**返回值：**格式化后的入场券对象基本信息字符串。

## 类图



# 代理**(Proxy)模式**

财务经理代理总经理管理财务

## 模式简介

在代理（Proxy）模式中，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。冰天雪地娱乐园内的总经理掌管许多方面的事务，而关于财务方面的工作可以由娱乐园内的财务经理代为管理，即使用代理模式。

因此，GeneralManager类和FinanceManagerProxy均实现了IFinanceManager接口中的manageFinace()方法，其中财务经理对象是调用总经理对象的管理财务方法以实现自己的管理财务方法。

参考出处：Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API 描述

GeneralManager.manageFinace();

**说明：**总经理执行管理财务方法。

**参数：**无

**返回值：**无

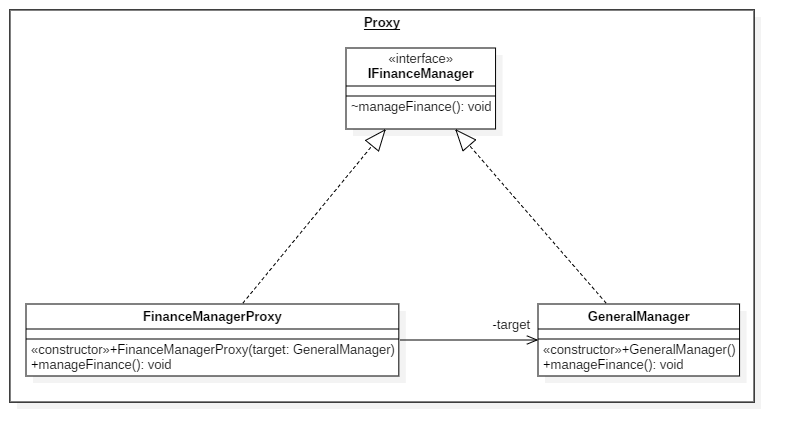
FinanceManagerProxy.manageFinace();

**说明：**财务经理执行代理管理财务方法。

**参数：**无

**返回值：**无

## 类图



# 单例(Singleton)模式

设置和获得当前的时间信息

## 模式简介

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。单例类只能有一个实例。

冰雪娱乐场中各地的时间是唯一的，管理人员可以通过时间类（Clock）设置时间，游客也可以通过时间类（Clock）获得时间，但无论如何，他们操作的时间类的实例都是同一个。

参考出处：

1. <https://www.runoob.com/design-pattern/singleton-pattern.html>

2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

Clock. getClock()

**说明：**获得同一个时间单例类的实例，只有一个实例

**参数：**无

**返回**：时间单例类的实例

Clock. setTime(String currentTime)

**说明：**设置当前的时间

**参数：**当前的时间currentTime

**返回**：无

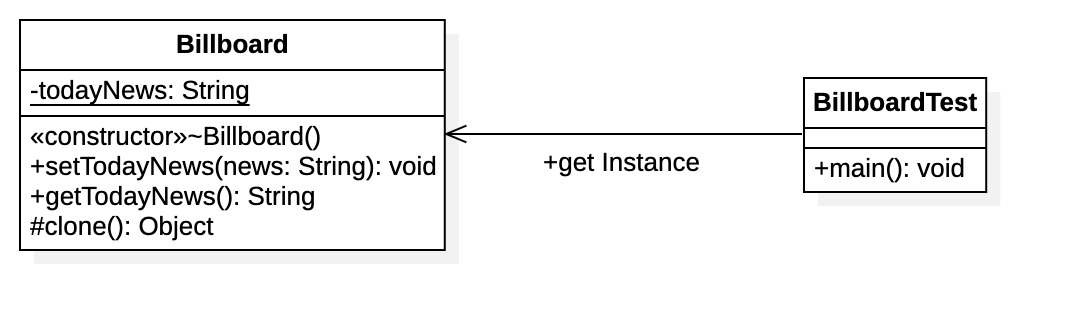
Clock. getClock()

**说明：**获得当前的时间

**参数：**无

**返回**：当前的时间currentTime

## 类图



# 状态(State)模式

保存设备不同状态

## 模式简介

状态模式是用最简洁的代码通过接口、抽象类、普通类、继承、委托、代理模式等方式，将状态抽象为类，然后通过控制状态的逻辑委托不同的状态去做不同的事情，对于每一个状态来说又再次委托控制状态的逻辑做出相应的动作和修改。这使得程序非常的简洁，各个状态分工明确，密切配合。

Device是设备的实体类，包含一个State对象和若干交互方法。

State本身是一个接口，它在各种详细的State类中得到实现，OnState代表开启状态，OffState代表关闭状态，ErrState代表错误状态，这些状态都实现了各自的交互方法。

参考出处：GOF《Design Patterns: Elements of Reusable Object-Oriented Software》

## API描述

Device.getState()

**说明：**获取设备当前状态

**参数：**无

**返回**：状态字符串

Device.doUse()

**说明：**尝试使用当前设备

**参数：**无

**返回**：无

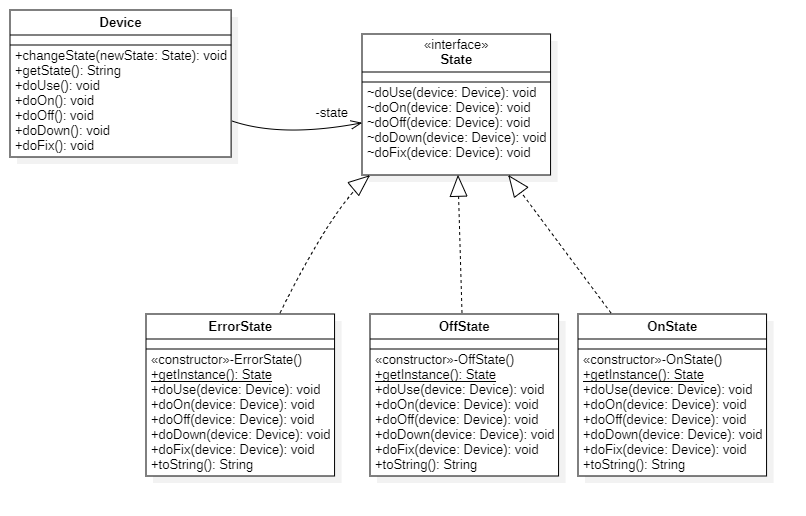
Device.doOn/Off/Down/Fix()

**说明：**尝试开启/关闭/损坏/修复当前设备

**参数：**无

**返回**：无

## 类图



# 策略(Strategy)模式

游客的游玩策略可以根据目的地任意替换

## 模式简介

在策略模式（Strategy Pattern）中，一个类的行为或其算法可以在运行时更改。这种类型的设计模式属于行为型模式。

冰雪娱乐场中有一部分经验丰富的游客（ExperiencedTourist），他们经常来冰雪娱乐场进行游玩，以至于他们对于如何在各个地点之间周转（MovingMethod）轻车熟路，他们知道如何快速到达另一个目的地（MovingMethod.execute()）。

比如到达冷冰冰餐厅（goToChillingRestaurant）的最佳时间是通过步行，到达冰雪峡谷（goToIceCanyon） 的最佳实践是乘坐缆车，到达的半山腰的小木屋（goToMidHillsideChalet）的最佳实践是坐狗拉雪橇，到达冰雪娱乐场出口（goToParadiseExit）的最佳实践是乘坐电动汽车。

参考出处：

1. <https://java-design-patterns.com/patterns/strategy/>

2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 233ff. ISBN 0-201-63361-2.

## API描述

ExperiencedTourist(MovingMethod method)

**说明：**构造函数，实例化一个希望进行 method 行动的有经验的游客（ExperiencedTourist）

**参数：**method即将进行的行动方式

**返回**：一个ExperiencedTourist的实例

ExperiencedTourist.changeMethod(MovingMethod method)

**说明：**变更该游客的目的地

**参数：**method即将进行的行动方式

**返回**：无

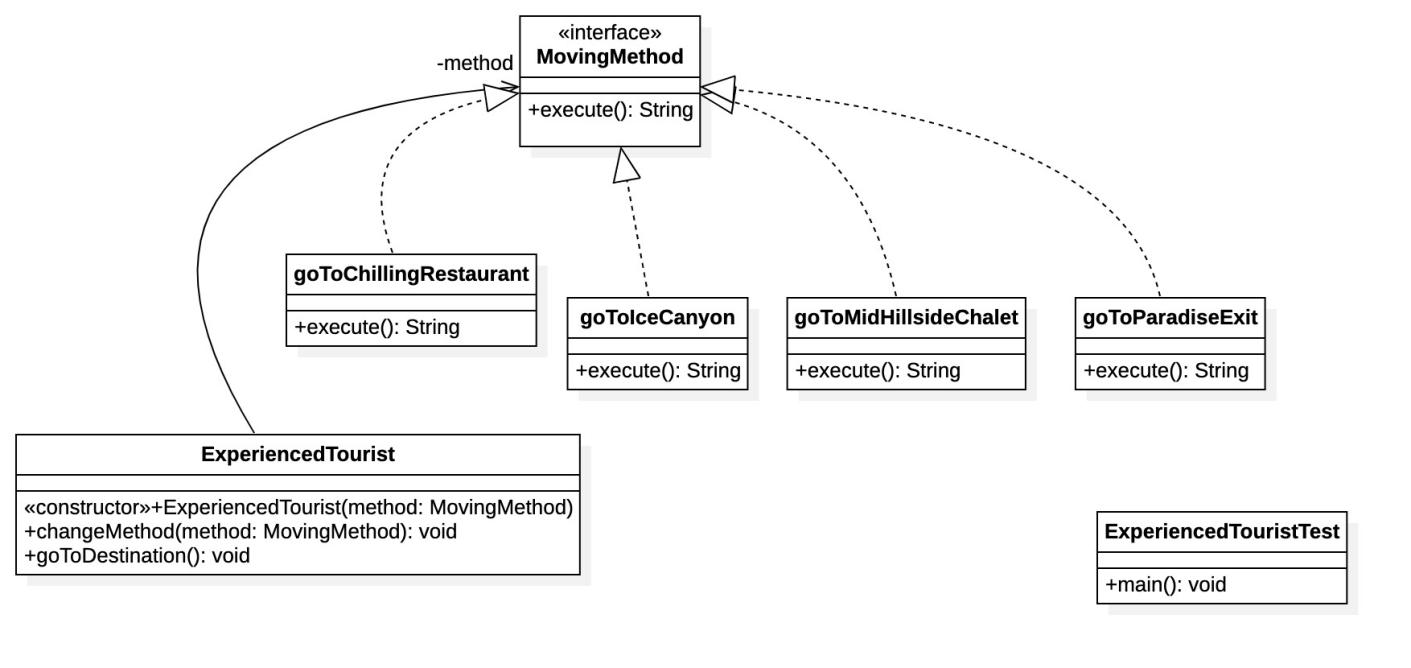
ExperiencedTourist. goToDestination()

**说明：**前往某一个目的地

**参数：**无

**返回**：无

## 类图



# 模板(Template Method)模式

定义一个模板结构，将具体内容延迟到子类去实现

## 模式简介

游乐场里的小火车轨道上有很多种车厢——虽然长得不一样，能进行的服务也不一样，但是它们都是火车的车厢。维护人员可以从一个车的模板中创造出新的服务车厢来满足游客的新需求。

在模板模式中，HalflingCabin作为一个抽象类公开定义了一个方法visitedBy ()，即为模板。当人进入车厢被服务时，需要调用此函数，并改变车厢种类。由于不同的车厢服务不同，子类重写visitedBy()方法，同时保证调用以HalflingCabin抽象类中定义的方式进行, 使部分步骤在抽象类实现，部分步骤在子类实现。

参考出处：Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). "Template Method". Design Patterns. Addison-Wesley. pp. 325–330. ISBN 0-201-63361-

## API描述

HalflingCabin.visitedBy()

**说明：**车厢被访问

**参数：**客户名字

**返回**：无

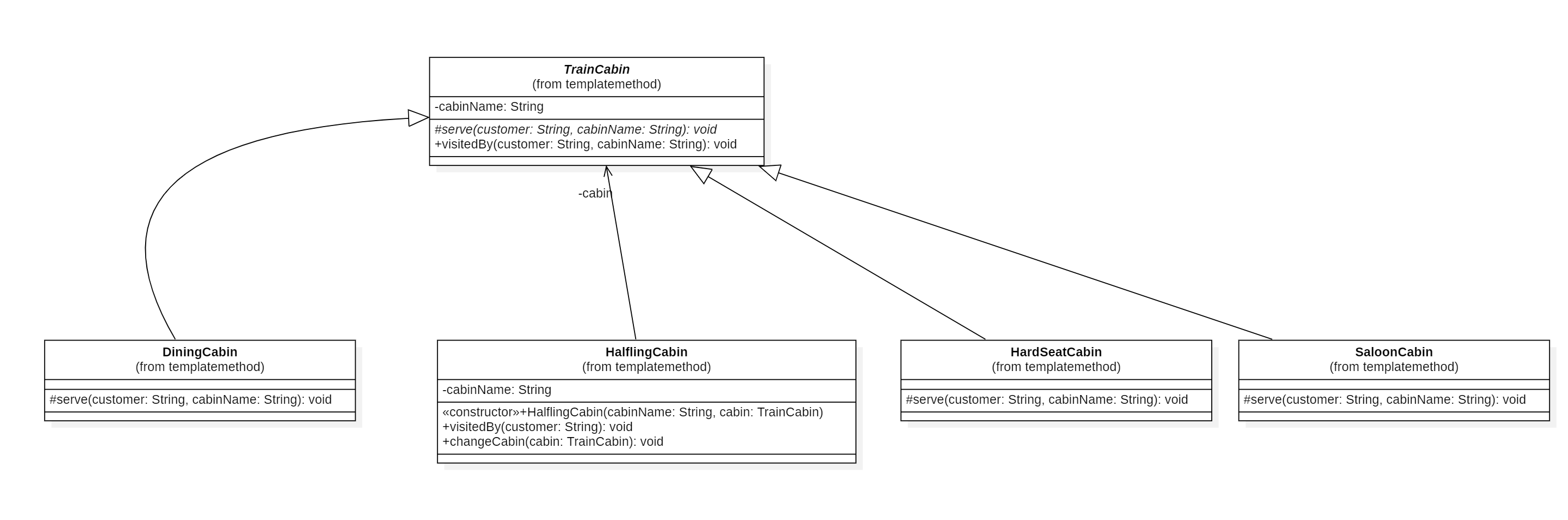
HalflingCabin.changeCabin()

**说明：**车厢被改变

**参数：**车厢类型

**返回**：无

## 类图



# 访问者(Visitor)模式

解耦合

## 模式简介

访问者模式通过增加Visitor接口，实现了操作与具体对象的解耦。在本系统中，雇员访问者接口（EmployeeVisitor） 用于访问雇员，即对雇员对象执行一系列操作。实现其接口的子类需要分别实现对成员的visit函数。对于新增功能，只需新增EmployeeVisitor子类，而无需改动原有代码。

参考出处：The Visitor Family of Design Patterns at the Wayback Machine (archived October 22, 2015). Additional archives: April 12, 2004, March 5, 2002. A rough chapter from The Principles, Patterns, and Practices of Agile Software Development, Robert C. Martin, Prentice Hall

## API描述

EmployeeVisitor ()

**说明：**获得一个访问者实例，用于访问整个雇员树并跟他们打招呼。

**参数：**无

**返回**：EmployeeVisitor

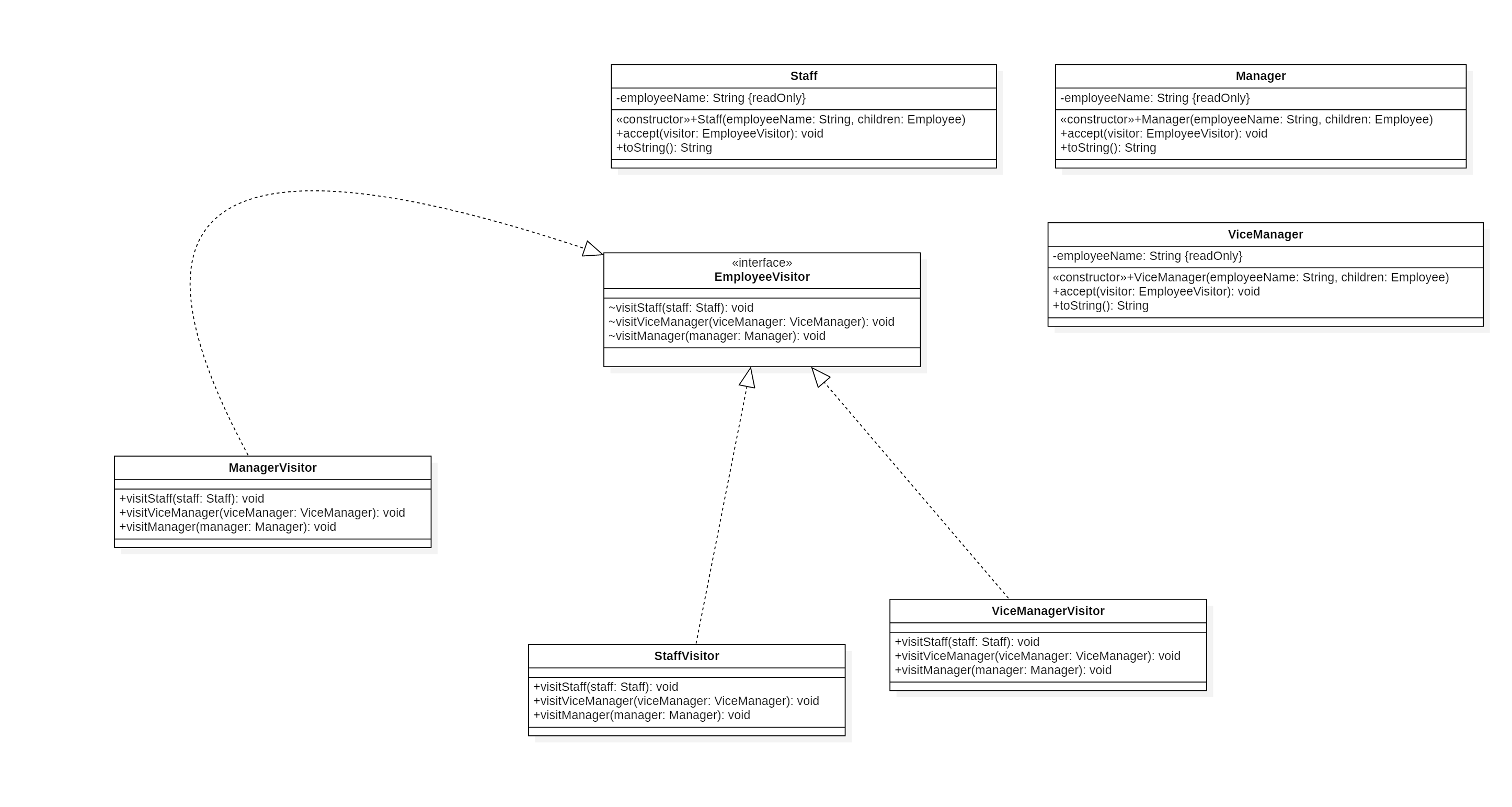
Employee.accept()

**参数：** EmployeeVisitor用以访问雇员

**说明：**使用访问者访问雇员，并继续往下找人。

**返回**：无

## 类图



# 主动对象(Active Object)模式

实现消息中心对游客意见的异步处理

## 模式简介

主动对象(Active Object)模式一种异步编程思想，允许任务的提交（相当于对异步方法的调用）和任务的执行（相当于异步方法的真正执行）分离，实现了异步编程，有利于提高并发性，从而提高系统的吞吐率。主动对象模式是一种多线程设计模式。

消息中心（Proxy）可以接受游客发起处理意见（solveEvent方法），然后将对该方法的调用转化为事件处理请求（SolveEventRequest）并把处理中心（Servant）传递给该请求，并将该请求交由调度中心调度（SchedulerThread）调度，而调度中心会不断从请求队列（ActivationQueue）中取出请求并尝试处理（SolveEventRequest. execute（））。当游客发起处理意见请求后，游客会立刻取得处理结果的容器（FutureResult），当事件处理完毕后，通知中心（NotifyThread）会将容器内的实际结果（RealResult）取出，通知游客事件处理完毕并告知处理结果，从而实现了游客意见的异步处理。

参考出处：《Java多线程设计模式 》（结城浩著）中国铁道出版社 p295

## API描述

Proxy. SolveEventRequest (String message)

**说明：**游客发起意见处理

**参数：**message表示游客意见

**返回**： 一个Result表示处理结果

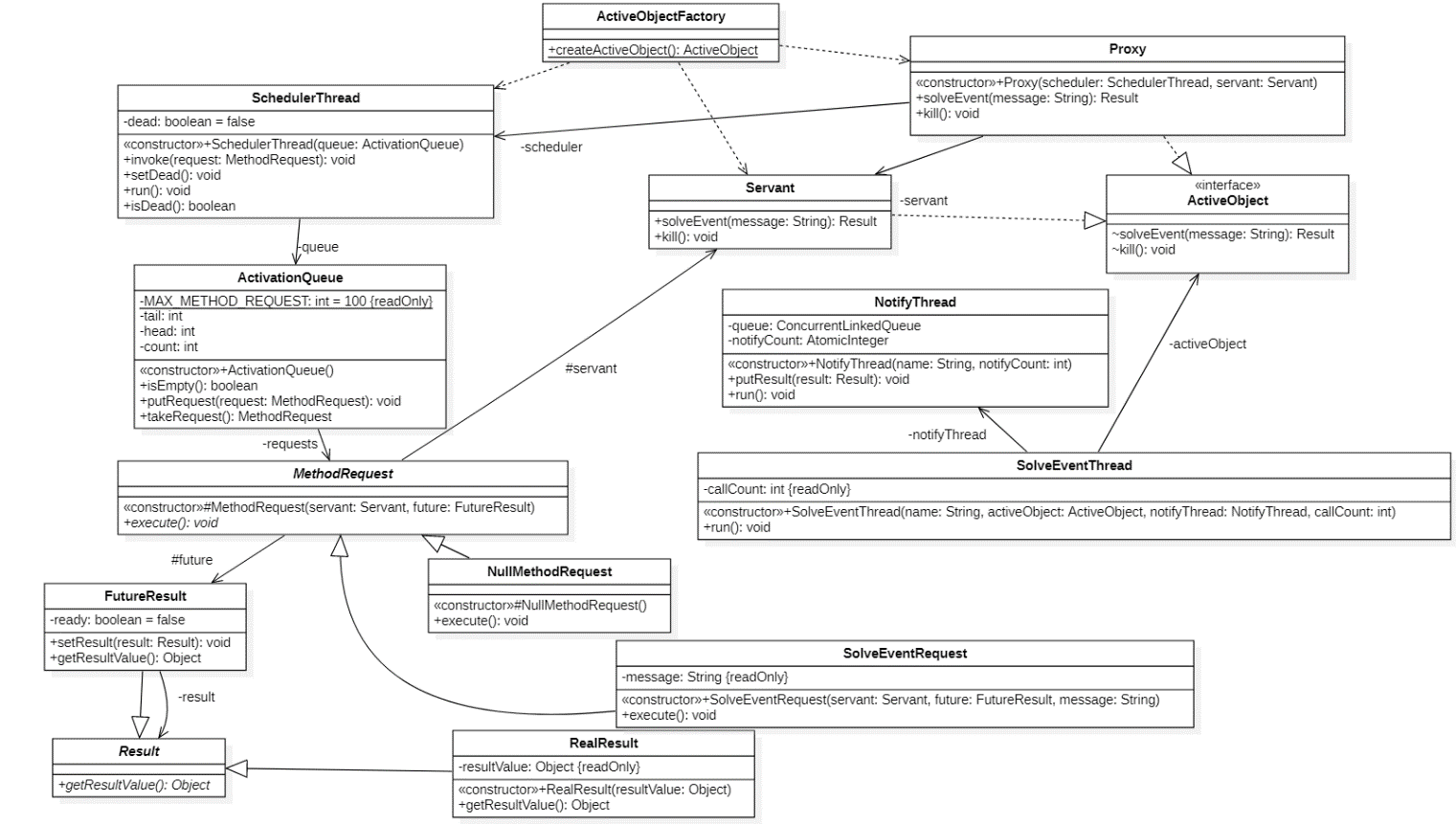
Result. getResultValue ()

**说明：**获取结果的返回值

**参数：**无

**返回**： 一个Object表示返回结果

## 类图



# 犹豫（Balking）模式

减少服务员无效的询问点单，增加效率

## 模式简介

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，我们把这样的线程间交互称为Balking（犹豫）设计模式。

账单（Bill）记录了顾客在娱乐园餐馆中消费菜品的价格，顾客（CustomerThread）在思考一定时间后修改账单上的菜品，期间服务员（WaiterThread）如果来询问点单，则正式下单，顾客之后不主动下单。反之如果一定时间后服务员没来，则顾客会主动下单，之后如果账单未作新更改，则拒绝服务员的询问下单。

参考出处：《Java多线程设计模式 》（结城浩著）中国铁道出版社 p115

## API描述

Bill. addDish (double curPrice)

**说明：**顾客修改账单内容

**参数：**curPrice表示新增菜品价格

**返回**：无

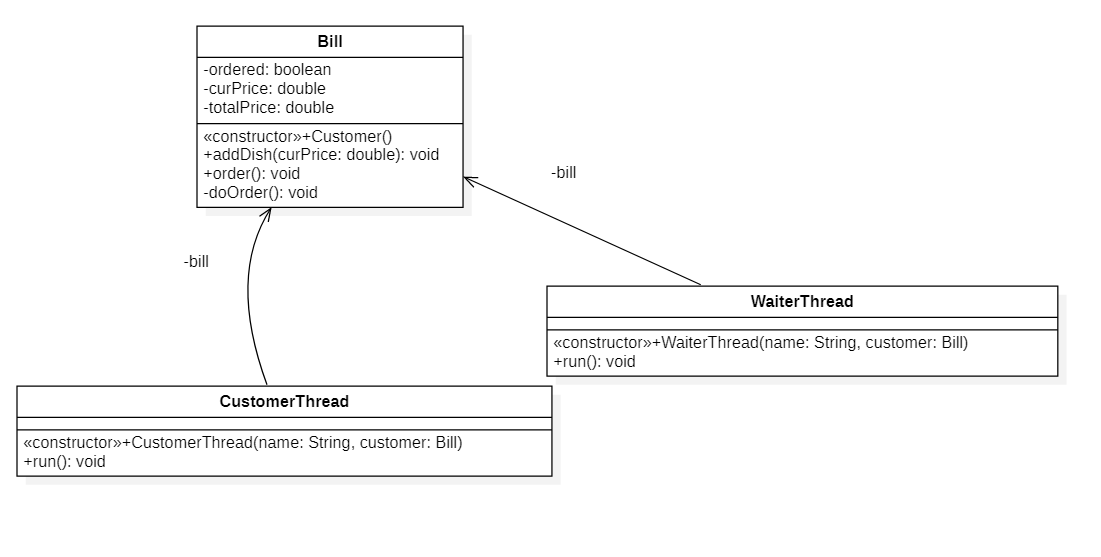
Bill. order ()

**说明：**试图下单，如果当前账单已经下过单则忽略

**参数：**无

**返回**：无

## 类图



# 回调(Callback)模式

## 模式简介

回调是一段可执行的代码，它作为一个参数传递给其他代码，其他代码会在某个方便的时候回调（执行）这个参数。

冰天雪地娱乐园的游客会需要上厕所，每一次上厕所他们会新建一个上厕所的任务（goToTheBathroom），并选择上完厕所之后执行的动作（callback），比如报告领队等。

参考出处：https://java-design-patterns.com/patterns/callback/

## API描述

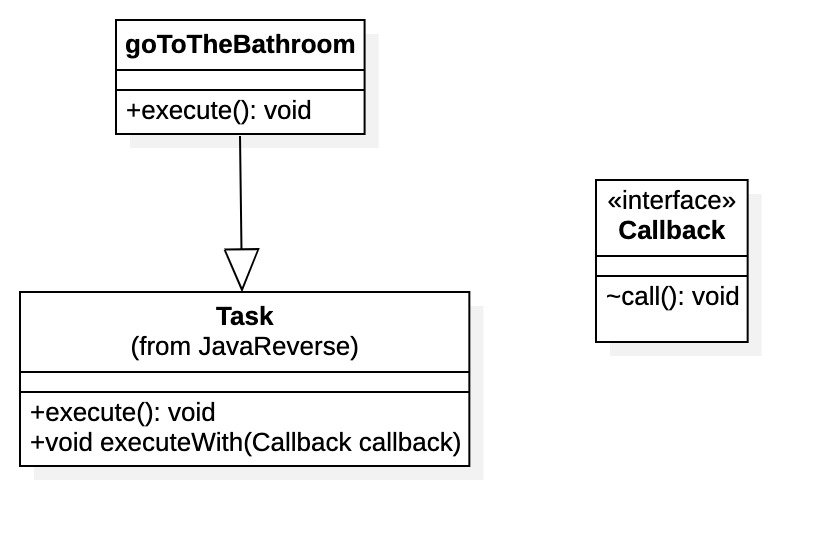
goToTheBathroom.executeWith(callback);

**说明：**一名游客去上厕所，并会在上完厕之后执行callback

**参数：**callback任务完成之后的回调函数

**返回**：无

## 类图



# 转换器（Converter）模式

在会员信息和会员实体类间转换

## 模式简介

对于娱乐园的会员用户(Membership)，娱乐园会提供折扣门票，购票处需要通过查询得到的通用信息类(MembershipDto)(具体关于Dto的内容参见Dto模式)获取到其实体(Membership)。这个过程是由转换器(Converter)来完成的。

参考出处：

1. <https://java-design-patterns.com/patterns/converter/>

2. https://www.javaguides.net/2018/08/converter-design-pattern-in-java.html

## API描述

convertToDto（T）：U

**说明：**从实体类转换到通用信息类使用的函数

**参数：**一个实体类的对象

**返回值：**一个通用信息类的对象

ConvertToEntity（U）：T

**说明：**从通用信息类转换到实体类使用的函数

**参数：**一个通用信息类的对象

**返回值：**一个实体类的对象

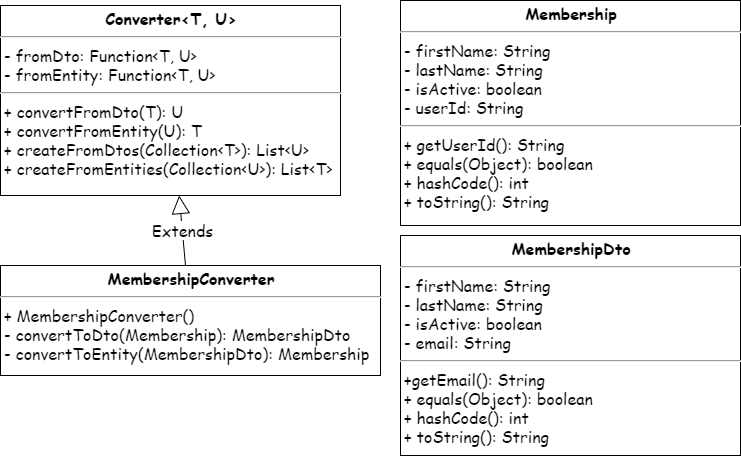
Converter（Function<U,T>, Function<T, U>）：Converter

**说明：**创建一个可以将实体类与通用信息类互相转换的Converter

**参数：**Function<U,T>和Function<T,U>是用来互相转换的函数

**返回值：**一个在特定类中间使用特定转换函数的转换器

## 类图



# 数据访问对象(DAO)模式

数据持久化存储

## 模式简介

DAO 模式用于把低级的数据访问操作从高级的业务服务分离出来，通过新增DAO层将底层数据访问和上层业务逻辑解耦。在本例中，CustomerDao接口定义对应的Customer数据对象的操作函数，CustomerDaoImpl实现CustomerDao接口，从本地持久化文件中读取Farm对象。若未来将Customer数据放入数据库中存储，只需新增类实现CustomerDao接口从数据库中读取信息，无需改动原有代码。

参考出处：

1. <https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm>

2. https://www.baeldung.com/java-dao-pattern

## API描述

CustomDaoImpl.getCustom()

**说明：**读取本地文件，获取顾客数据

**参数：**无

**返回**：Customer

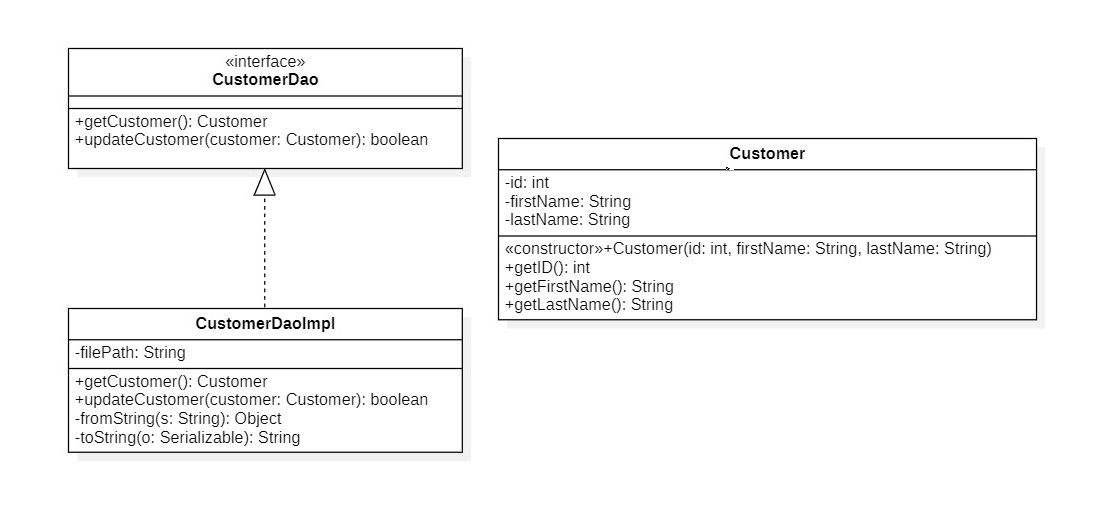
CustomDaoImpl.updateCustomer()

**说明：**更新本地顾客数据，将顾客数据以文件形式持久化到本地系统

**参数：**Customer

**返回**：无

## 类图



# 委托(Delegation)模式

判断购票者是成人/儿童

## 模式简介

委托模式是软件设计模式中的一项基本技巧。在委托模式中，有两个对象参与处理同一个请求，接受请求的对象将请求委托给另一个对象来处理。委托模式是一项基本技巧，许多其他的模式，如状态模式、策略模式、访问者模式本质上是在更特殊的场合采用了委托模式。委托模式使得我们可以用聚合来替代继承，它还使我们可以模拟mixin。

进入冰雪天地娱乐园需要购买门票，门票分为成人票和儿童票，可以将其委托给ticket office判断游客购买的是儿童票还是成人票。

参考出处：Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design patterns : elements of reusable object-oriented software (14. print. ed.). Reading, Mass.: Addison-Wesley. p. 20. ISBN 0-201-63361-2.

## API描述

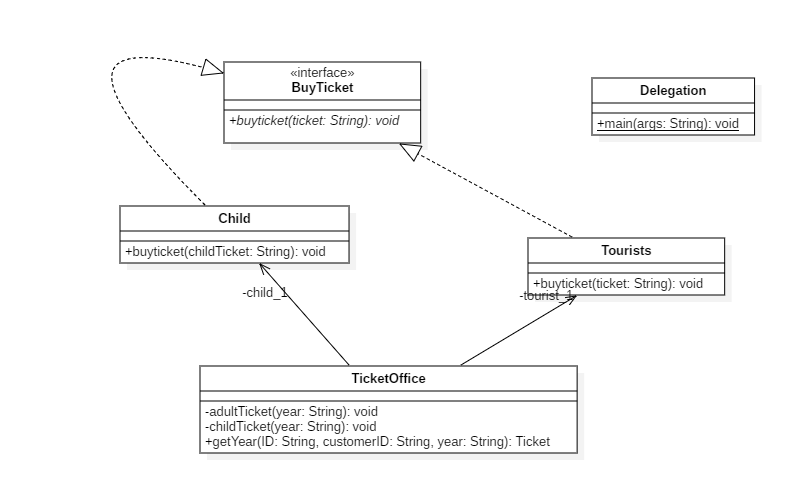
ticketOffice.getYear()

**说明：**获取游客的年龄

**参数：**顾客编号ID,customerID,顾客年龄year

**返回**：购买的门票信息

## 类图



# 依赖注入(Dependency Injection)模式

游玩不同的娱乐设施

## 模式简介

依赖注入（Dependency Injection）是一种软件设计模式，在这种模式中，一个或多个依赖关系（或服务）被注入或通过引用传递到一个依赖对象（或客户端）中，并成为客户端状态的一部分。该模式将客户端依赖关系的创建与自身的行为分离开来，这使得程序设计可以松散耦合，并遵循控制权反转和单一责任原则。

冰雪天地娱乐园中有许多娱乐设施，游客通过一个接口游玩不同的娱乐设施。

参考出处：Mark Seemann, Steven van Deursen . *Dependency Injection Principles, Practices, and Patterns*

## API描述

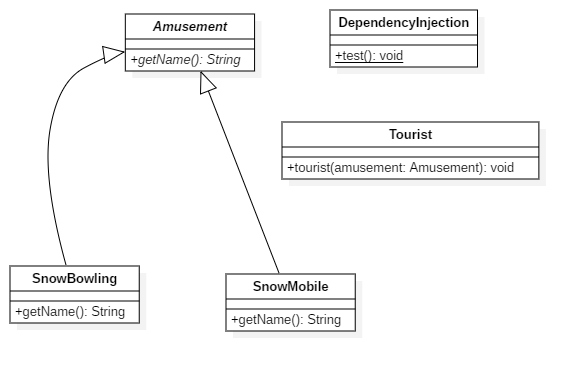
tourist.tourist()

**说明：**确认游客要游玩的娱乐设施

**参数：**游客要游玩的娱乐设施amusement

**返回**：无

## 类图

****

# 脏标记(Dirty Flag)模式

更新电子游戏机电子菜单界面

## 模式简介

DirtyFlag模式用一个标志位表示数据的状态，这些数据在满足条件的时候需要设置标志位，在被需要的时候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同步；如果没有设置标志位，那么就可以不计算或使用缓存的计算结果。在两次check之间，即使有多次标志位的设置，也只需要计算一次。DirtyFlag的目的在于减少无谓的计算或同步。

使用的限制条件：

1. 单次计算的开销足够大

2. 需要计算的概率足够低

3. 延迟计算没有副作用

4. 考虑内存换速度的代价

DirtyFlagPattern属于Optimization Pattern，只有确定瓶颈、需要优化的时候才应该考虑使用这种模式。

这里以电子游戏机的菜单更新模拟DirtyFlag模式，当需要菜单更新的时候，将标志位设置为true；展示菜单时先check标志位，只有当标志位为真时，才会更新菜单内容。

参考出处：

1. https://gameprogrammingpatterns.com/dirty-flag.html

2. <https://java-design-patterns.com/patterns/dirty-flag/>

## API描述

ElectronicMenu.menuChange()

**说明：**需要更新菜单时调用此方法，将标志位设置为true

**参数：**无

**返回**：无

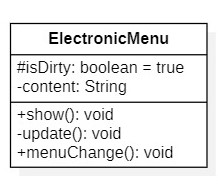
ElectronicMenu.show()

**说明：**展示菜单，若标志位为true，则更新菜单后展示

**参数：**无

**返回**：无

## 类图



# 私有类数据（Private Class Data）模式

## 模式简介

私有类数据可以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。它可以减少类对外暴露的属性，并优雅地从类中移除了对数据的写权限。当以下情况成立时可以使用 Private Class Data 模式：1. 类的初始化数据是一次性的不可修改的数据。2. 需要控制对类的初始化数据的更改。3. 预防对初始化数据的不必要的更改。

冰天雪地娱乐园内有四种娱乐项目可以在兑奖券（Raffle）上累计次数，分别是狗拉雪橇（dogSledding）、滑雪（skiing）、蹦极(bungeeJumping)和大转盘(bigWheel)。对于有效的兑奖券（Raffle），游客可持有他们进行查看（checkout）和兑奖（redeem）；而对于因为各种原因而失效（如过期等）的兑奖券（ImmutableRaffle），游客则只能进行查看（checkout）。

参考出处：

1. <https://java-design-patterns.com/patterns/private-class-data/>

2. <https://en.wikipedia.org/wiki/Private_class_data_pattern>

3. https://sourcemaking.com/design\_patterns/private\_class\_data

## API描述

Raffle.checkout()

**说明：**游客查看手上的兑奖券的状态，类会输出四类游玩项目目前累计的游玩次数

**参数：**无

**返回**：无

Raffle.redeem()

**说明：**游客将兑奖券进行兑奖，类会各扣除一项游玩项目的游玩次数（如果可以扣除，即次数大于1），并检查是否可以兑奖

**参数：**无

**返回**：无

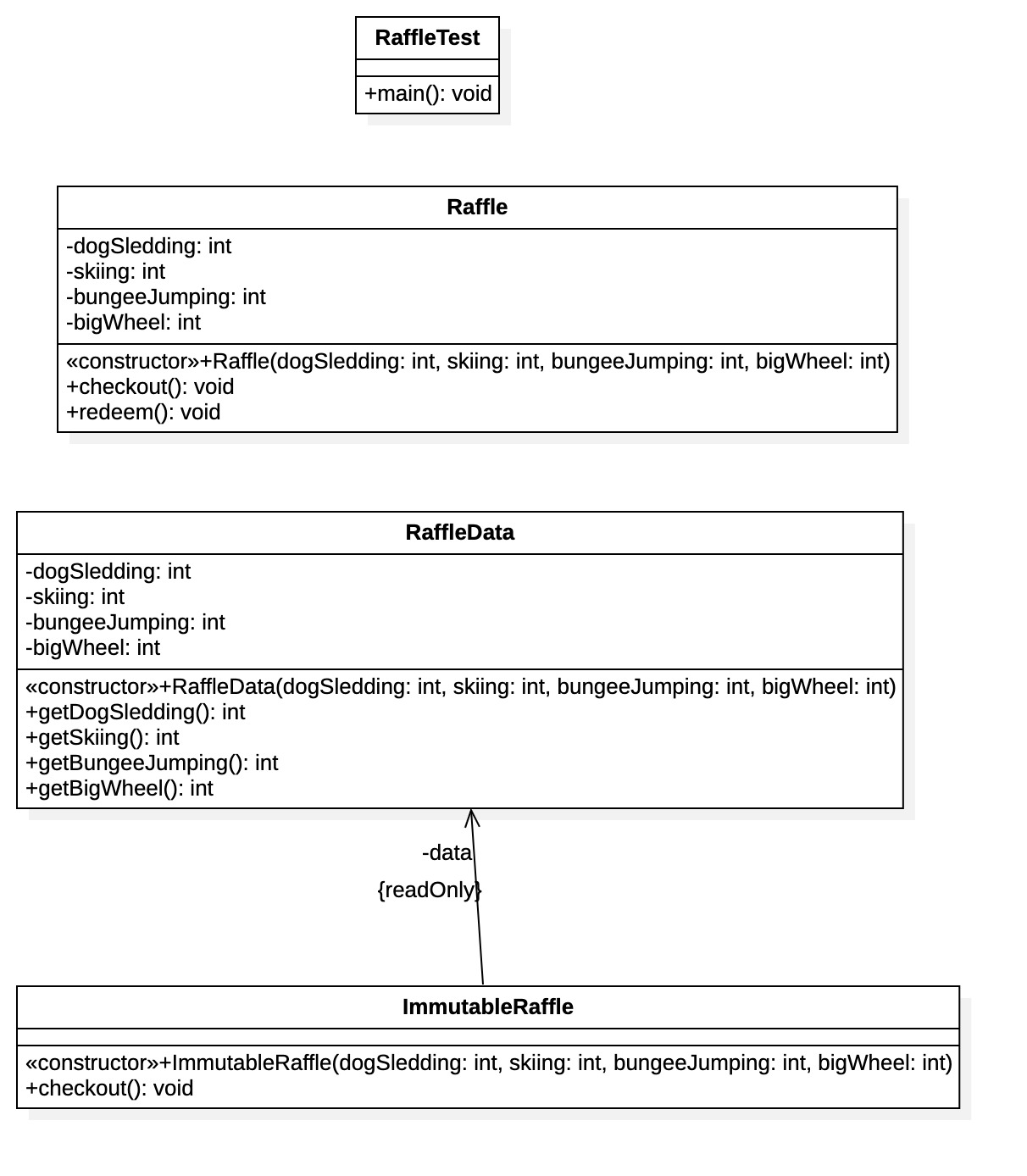
immutableRaffle.checkout()

**说明：**游客查看手上的兑奖券的状态，类会输出四类游玩项目目前累计的游玩次数

**参数：**无

**返回**：无

## 类图



# 数据传输对象(dto)模式

处理游客的滑雪工具租借订单

## 模式简介

数据传输对象模式（DTO)(Data Transfer Object)，是一种设计模式。在调用远程接口时，可以创建一个数据传输对象，容纳调用的所有数据。数据传输目标往往是数据访问对象从数据库中检索数据。数据传输对象与数据交互对象或数据访问对象之间的差异是一个以不具有任何行为除了存储和检索的数据（访问和存取器）。

在冰雪天地娱乐园中滑雪需要租借滑雪工具，滑雪者将订单传给工作人员，工作人员将订单中的滑雪工具打包交给滑雪者，滑雪者再打开包裹拿到滑雪工具。

参考出处： Fowler, Martin (2010). Data Transfer Object. Patterns of Enterprise Application Architecture. Retrieved from <http://martinfowler.com/eaaCatalog/dataTransferObject.html>

## API描述

preprocessingLine.establishPipe()

**说明：**分发滑雪工具

**参数：**滑雪者skier

**返回**：无

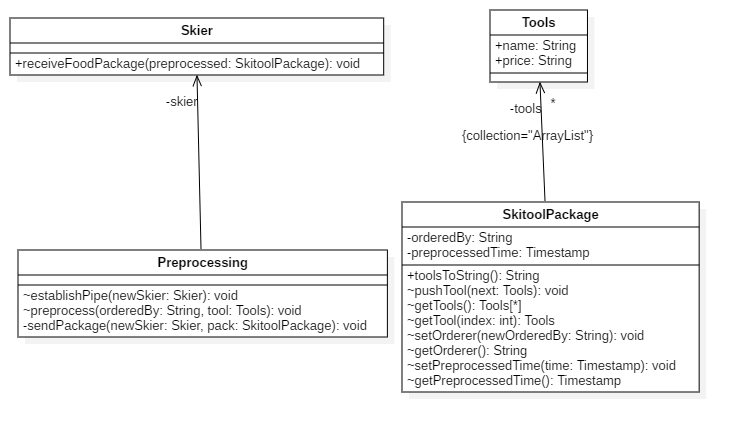
preprocessingLine.preprocess()

**说明：**将工具传入预处理处

**参数：**订单orderBy，滑雪工具tool

**返回**：无

## 类图



# 特性切换（Feature Toggle）模式

滑雪赛道选择

## 模式简介

游乐园内的设有滑雪赛道，共有两条，分别为竞速赛道和障碍赛道；通过不同的赛道类进行特性的切换（这里采用不同发的输出作为样例）

参考出处：

1. <https://martinfowler.com/articles/feature-toggles.html、>

2. https://java-design-patterns.com/patterns/feature-toggle/

## API 描述

PlayerGroup.addPlayerToRaceGroup(Player)

**说明：**向竞速赛道添加参与玩家。

**参数:** 玩家实例（Player)

**返回值：**无

PlayerGroup.addPlayerToSlalomGroup(Player)

**说明：**向障碍赛道添加参与玩家。

**参数:** 玩家实例（Player)

**返回值：**无

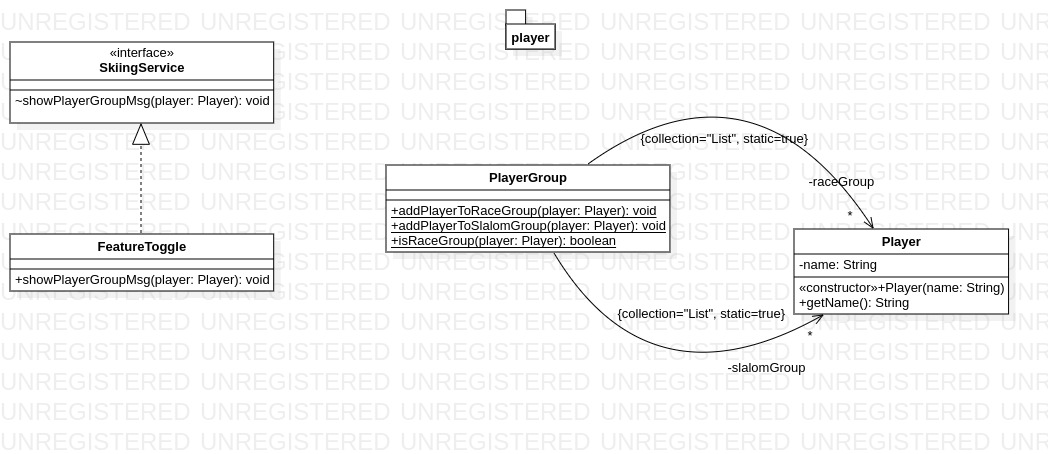
SkiingService.showPlayerGroupMsg(Player player)

**说明：**输出对玩家的欢迎信息

**参数:** 玩家实例（Player)

**返回值：**无

## 类图



# 前端控制器(Front Controller)模式

根据用户角色分发视图

## 模式简介

前端控制器模式将所有请求集中到一个类来处理，该类依据不同的请求来执行相应的动作。FrontController类接收请求并进行权限验证，然后判断请求类型，分发器PlayerDispatcher根据请求类型调用相应的接口。

参考出处：

1. Alur, Deepak; John Crup; Dan Malks (2003). Core J2EE Patterns, Best Practices and Design Strategies, 2nd Ed. Sun Microsystems Press. pp. 650pp. ISBN 0-13-142246-4.

2. <https://www.geeksforgeeks.org/front-controller-design-pattern/>

3. https://www.tutorialspoint.com/design\_pattern/front\_controller\_pattern.htm

## API描述

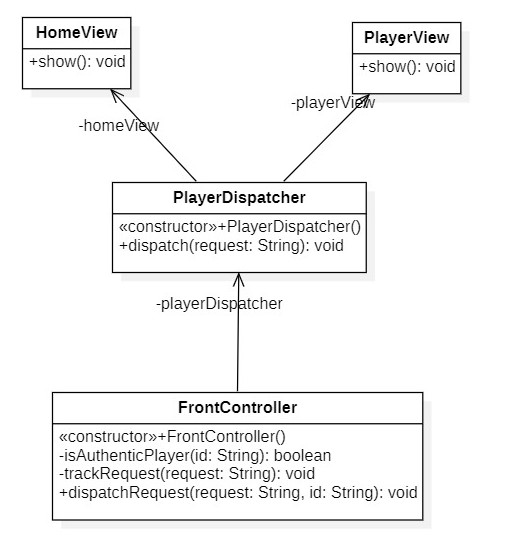
FrontController.dispatchRequest(String request, String id)

**说明：**根据用户id和用户请求分发页面

**参数：**请求访问界面request, 用户id

**返回**：无

## 类图



# 回调(Future/Promise)模式

餐厅下单

## 模式简介

乐园的餐厅内有统一厨房和厨房内得多位厨师；顾客下单菜品（Dishes）后，将菜品订单（oreder）发到厨房（Kitchen），厨房将需求发完之后，继续处理下一个订单；厨房接到订单后开始处理，处理后返回并提醒厨房中等待的进程，提示完成订单，进行下一步操作。

参考出处：

1. <https://www.yld.io/blog/the-callback-pattern/>

2. https://java-design-patterns.com/patterns/callback/

## API 描述

Kitchen.request( Dishes )

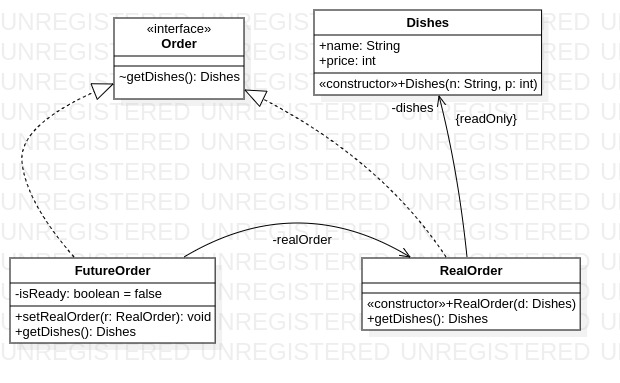
**说明：**厨房发出菜品制作请求

**参数:** 菜品（Dishes）实例

**返回值：**厨师订单实例（Order）

黄文海著《Java 多线程编程实战指南（设计模式篇）》---第 6 章

## 类图

**

# 不变(immutable)模式

管理乐园建筑

## 模式简介

游乐园内的娱乐设施位置是固定的，不能进行更改。在设施创建时，应使用不变模式对设施的位置进行固定。

使用Bulding类来记录设施信息，通过BuildingInfo来进行信息的固定，BuildingInfo内置一个Building对象和其他成员都为final属性，也没有Set方法用于更改成员，Get方法也为final属性。

黄文海著《Java 多线程编程实战指南（设计模式篇）》---第 3 章

## API 描述

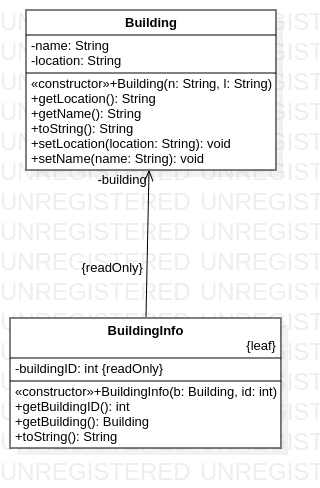
getBuilding();

**说明：**获取建筑信息。

**参数:** 无

**返回值：**Building 建筑实例

## 类图



# 延迟加载 (lazy-laoding) 模式

重启游戏

## 模式简介

舞台（Stage）是一个能够供客人观赏的场所，它有一个内置类演员持有人（actorHolder）用来对演员进行调度。

演员持有人在舞台请求演员后才进行演员的构造。

参考出处：J2EE Design Patterns B-3 Authors: [William Crawford](javascript:;), [Jonathan Kaplan](javascript:;)

## API 描述

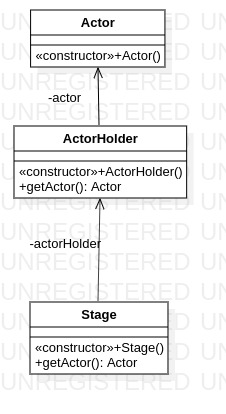
Stage.getActor()

**说明：**创建演员对象

**参数：**空

**返回值：**空

## 类图



# 标记接口(Marker Interface)模式

## 模式简介

标记接口是计算机科学中的一种设计思路，用于给面向对象的编程语言描述对象。因为编程语言本身不支持为类维护元数据，而标机接口可以作为描述类的元数据，弥补了这个功能的缺失。对于实现了标记接口的类，我们就可以在运行时通过反射机制去获取元数据，进而区分需要特别对待的对象。

Permission接口作为标记接口，Guard类实现该接口，而Thief不实现。是否实现Permission接口就成为了区分Guard和Thief的标记。

参考出处：

1. https://www.baeldung.com/java-marker-interfaces

2. Bloch, Joshua (2008). "Item 37: Use marker interfaces to define types". Effective Java (Second edition). Addison-Wesley. p. 179. ISBN 978-0-321-35668-0.

## API描述

Guard.enter()

**说明：**保安进入娱乐园

**参数：**无

**返回：**无

Thief.enter()

**说明：**小偷进入娱乐园

**参数：**无

**返回：**无

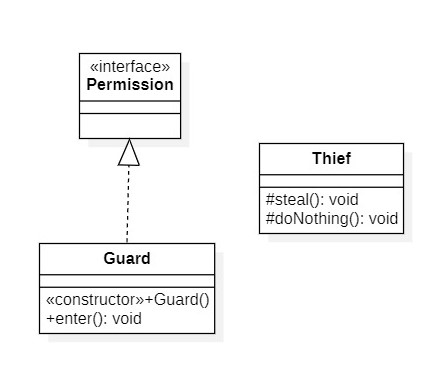
Thief.doNothing()

**说明：**小偷被禁止进入娱乐园

**参数：**无

**返回：**无

## 类图



# 单态(MonoState)模式

设置和获得当前的电子公告板信息

## 模式简介

MonoState模式部分解决了Singleton模式的缺陷,它关注的是Behaviour。该模式具有以下的优势。  
1.透明性.对于创建者而已,创建MonoState类的实例和创建普通类的实例是一致的(通过new()).这减少了沟通成本.  
2.可派生性. MonoState的派生类也具有MonoState性的. 这主要是因为子类和父类共享相同的static数据域. MonoState的继承, 可以看做是基于相同数据下的行为多态性的派生.

冰雪娱乐场中的各地都有着电子公告板（billboard），控制中心的管理者会在上面统一发布（setTodayNews）一些当前的娱乐场公告，人们则可以获得当前公告板上的信息（getTodayNews）。

参考出处：

1. <https://riptutorial.com/design-patterns/example/21458/the-monostate-pattern>

2. https://wiki.c2.com/?MonostatePattern

## API描述

Billboard.setTodayNews(String news)

**说明：**将公告板上的信息进行更新

**参数：**news 用于更新的新闻

**返回**：无

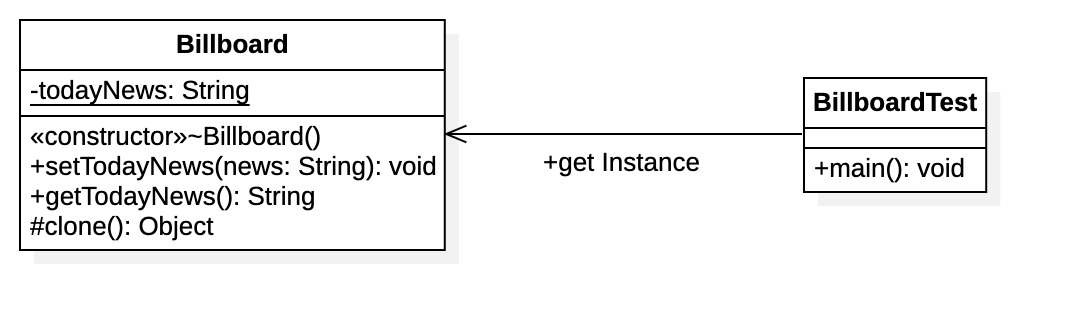
Billboard.getTodayNews()

**说明：**获得当前的新闻

**参数：**无

**返回**：当前的新闻news

## 类图



# 多例（Multition）模式

管理座椅

## 模式简介

游乐园内的娱乐设施的座椅是有固定数目的，对于每个娱乐设施，这个数目决定了该娱乐设施的使用人数上限。在设施创建时，应使用多例模式创建并管理座椅。

因此，在ChairMultition类中维护一个Map<Integer , ChairMultition> 的静态哈希表存储着该类的给定数目的对象。通过ChairMultition.getInstance(int i)接口获得一个特定序数的座椅实例。

出处：P 480，《Patterns of Enterprise Application Architecture》 by Martin Fowler, with Dave Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford，2002

作者博客对该模式的介绍：https://martinfowler.com/eaaCatalog/registry.html

亚马逊书店有售：https://www.amazon.com/gp/product/0321127420?ie=UTF8&tag=martinfowlerc-20&linkCode=as2&camp=1789&creative=9325&creativeASIN=0321127420

## API 描述

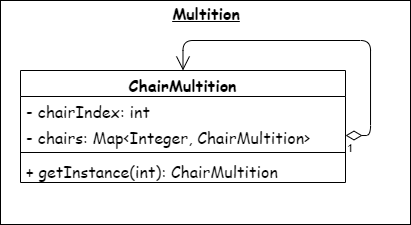
getInstance(int index);

**说明：**获取特定序号的座椅。

**参数：**index，座椅的序号

**返回值：**ChairMultition座椅实例

## 类图



# MVC模式

展示用户评价

## 模式简介

MVC 模式代表 Model-View-Controller（模型-视图-控制器） 模式。这种模式用于应用程序的分层开发。

Model（模型） - 模型代表一个存取数据的对象或 JAVA POJO。它也可以带有逻辑，在数据变化时更新控制器。

View（视图） - 视图代表模型包含的数据的可视化。

Controller（控制器） - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

参考出处：《Gamma, Erich et al. (1994) Design Patterns》

Wikipedia: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

## API描述

UserReviewController.getUserId/Stars/Review()

**说明**：获取用户评价的某个字段

**参数**：无

**返回**：需要获取的字段

UserReviewController.setUserId/Stars/Review(String/int)

**说明**：修改用户评价的某个字段

**参数**：对于UserId与Review为String，对于Stars为int (1~5)

**返回**：无

UserReviewController.updateView()

**说明**：更新视图

**参数**：无

**返回**：无

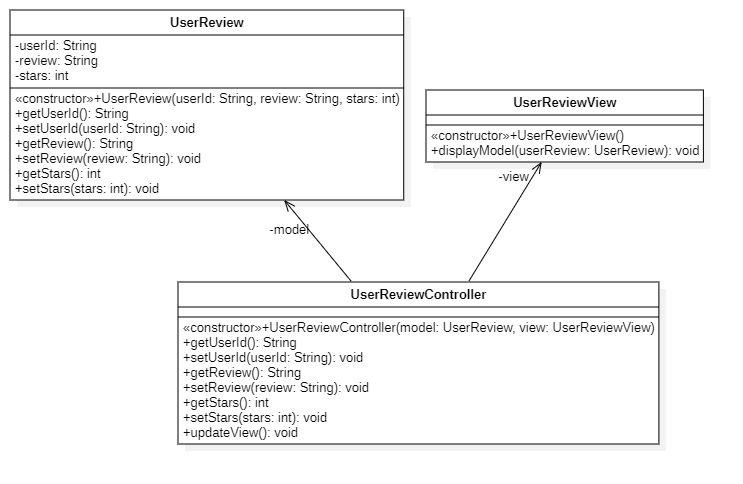
UserReviewView.displayModel(UserReview userReview)

**说明**：展示评价信息

**参数**：需要展示的Review

**返回**：无

## 类图



# 空对象（Null Object）模式

无效门票使用空对象

## 模式简介

游乐园的每个项目都要检查通票来判断客人是否重复游玩了该项目，但客人提供的票根（Stub）未必是合法的票据，需要通过票根使用验票器（TicketFinder）在门票数据库中进行查询，若该票无效返回无效票（NullTicket）。

出处：

Berczuk, Steven (December 1994). "Finding Solutions Through Pattern Languages". [IEEE Computer](https://en.wikipedia.org/wiki/Computer_(magazine)). [IEEE](https://en.wikipedia.org/wiki/Institute_of_Electrical_and_Electronics_Engineers). **27** (12): 75–76. [doi](https://en.wikipedia.org/wiki/Doi_(identifier)):[10.1109/2.335755](https://doi.org/10.1109%2F2.335755)（<https://ieeexplore.ieee.org/document/335755>）

关于该设计模式的简介：https://en.wikipedia.org/wiki/Null\_object\_pattern

## API 描述

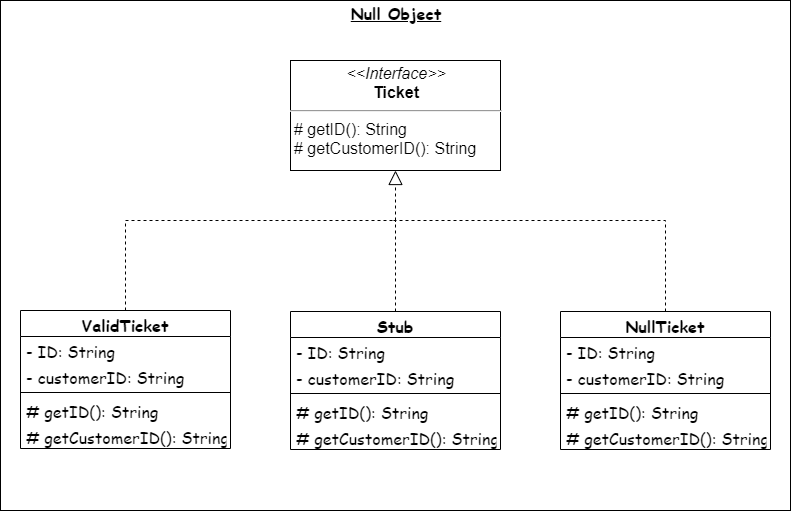
NullTicket();

**说明：**建立通票的空对象，在调用getID(), getCustomerID()时打印无效票的信息。

**参数：**无

**返回值：**无

## 类图



# Object Mother模式

可以快速且简洁地对大量娱乐场内的设施进行测试

## 模式简介

Object Mother是一种用于测试的类，它可以帮助创建用于测试的示例对象。

设施（Facilities）是一个接口，用于描述设施可以存在和调整的状态。电子设施（ElectronicFacilities）和传统设施（TraditionalFacilities）分别是设施接口的两个实现。在建立测试的时候，可以控制其是否损坏（makeRepaired()/makeDameged()），是否开放（makeOpen()/makeClosed()），是否有电（makePowerOn() makePowerOff()，仅限于ElectronicFacilities），并可以尝试进入设施（enter()）。

最后，我们可以通过FacilitiesObjectMotherTest类，调用FacilitiesObjectMother类的方法快速创建各种我们需要的实例。

参考出处：

1. <https://java-design-patterns.com/patterns/object-mother/>

2. <https://martinfowler.com/bliki/ObjectMother.html>

3. https://wiki.c2.com/?ObjectMother

## API描述

FacilitiesObjectMother.createUndamagedTraditionalFacilities()

**说明：**创建一个未损坏的传统设施进行后续测试

**参数：**无

**返回**：一个未损坏的传统设施

FacilitiesObjectMother. createDamagedTraditionalFacilities()

**说明：**创建一个损坏的传统设施进行后续测试

**参数：**无

**返回**：一个损坏的传统设施

FacilitiesObjectMother. createPowerOnDamagedElectronicFacilities()

**说明：**创建一个通电的损坏的电子设施进行后续测试

**参数：**无

**返回**：一个通电的损坏的电子设施

FacilitiesObjectMother. createPowerOnUndamagedElectronicFacilities()

**说明：**创建一个通电的未损坏的电子设施进行后续测试

**参数：**无

**返回**：一个通电的未损坏的电子设施

FacilitiesObjectMother. createPowerOffDamagedElectronicFacilities()

**说明：**创建一个未通电的损坏的电子设施进行后续测试

**参数：**无

**返回**：一个未通电的损坏的电子设施

FacilitiesObjectMother. createPowerOffUndamagedElectronicFacilities()

**说明：**创建一个未通电的未损坏的电子设施进行后续测试

**参数：**无

**返回**：一个未通电的未损坏的电子设施

ElectronicFacilities.enter()

**说明：**尝试进入电子设备

**参数：**无

**返回**：无

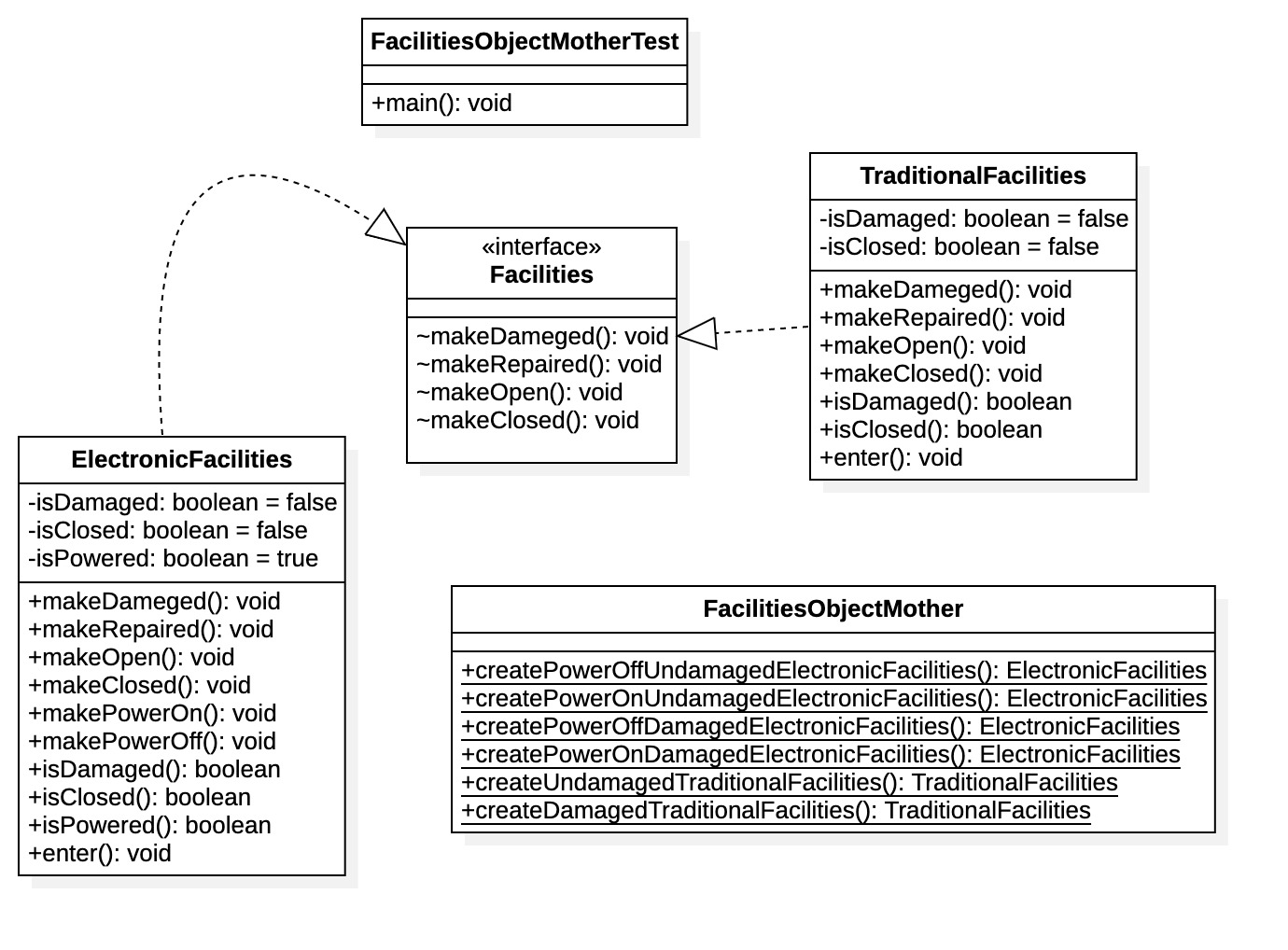
TraditionalFacilities.enter()

**说明：**尝试进入传统设施

**参数：**无

**返回**：无

## 类图



# 对象池(Object Pool)模式

要用的时候先从里面拿

## 模式简介

一个对象池包含一组已经初始化过且可以使用的对象，而可以在有需求时创建和销毁对象。池的用户可以从池子中取得对象，对其进行操作处理，并在不需要时归还给池子而非直接销毁它。这是一种特殊的工厂对象。

冰雪游乐场里总不能没有电影院吧！电影院中使用的眼镜成本巨大，想重新生产一个成本巨大，所以我们创建了一个眼镜池，使用一个就从里面拿一个。

参考出处

• Kircher, Michael; Prashant Jain (2002-07-04). "Pooling Pattern" (PDF). EuroPLoP 2002. Germany. Retrieved 2007-06-09.

## API 描述

PolarizedGlassPool.checkOut()

**说明：**如果对象池中还有剩余对象，则从其中取出一个；否则创建一个新对象。

**参数：**无

**返回**：对象（眼镜）

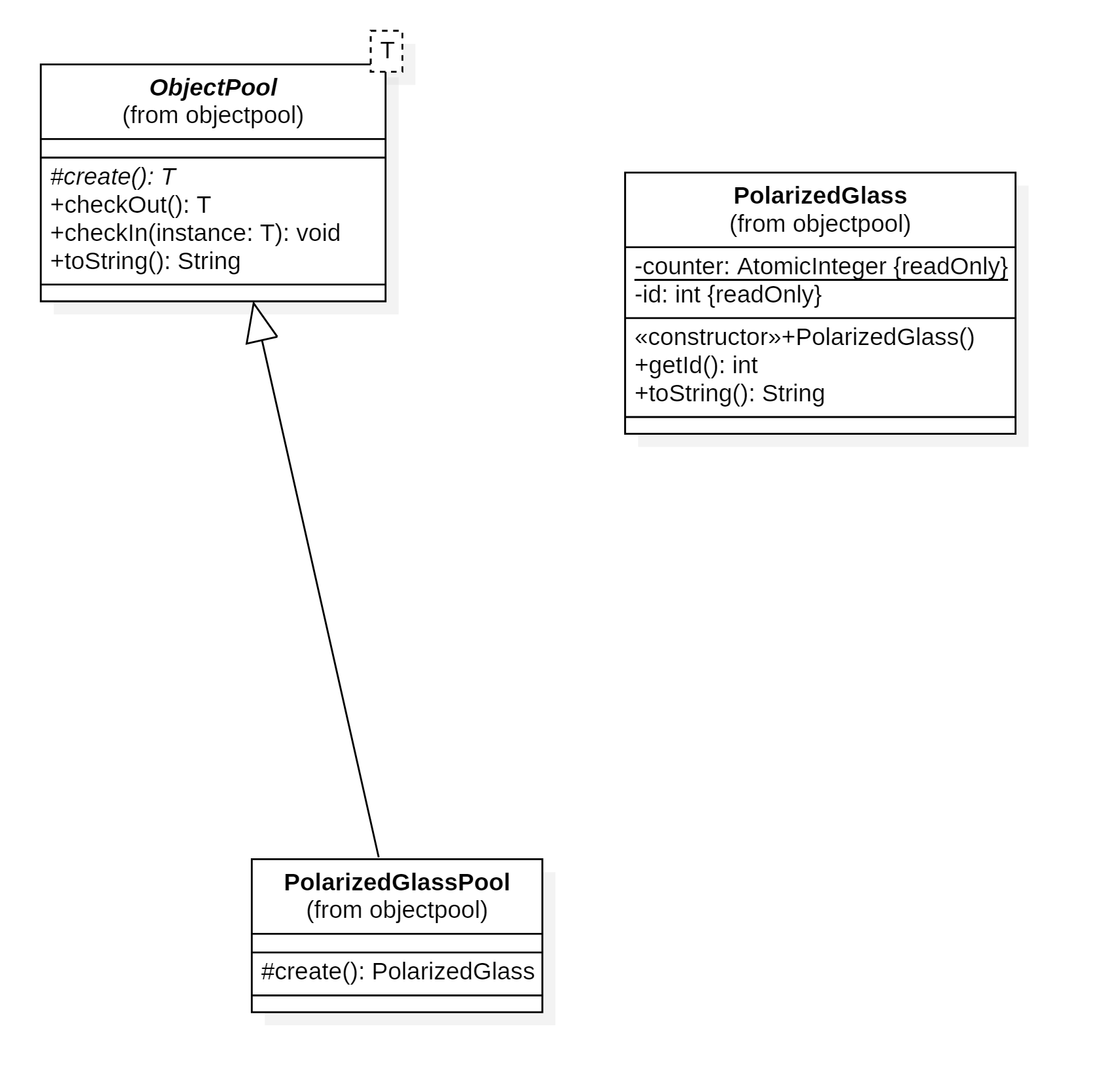
PolarizedGlassPool.checkIn()

**说明：**归还对象

**参数：**对象（眼镜）

**返回**：无

## 类图



# 管道(Pipeline)模式

设备维护

## 模式简介

管道模式将一个复杂的流程分割为若干子段（Handler），每个段接受一个输入，产生一个输出，下一个段使用上一个段的输出作为输入，从而对数据进行一系列的处理。

本例将设备维护流程（EquipmentMaintenancePipeline）分割为数个子过程，分别为停用（ProhibitEquipment）、检查（ExamineEquipment）、维修（MaintainEquipment）、测试（TestEquipment），验收（CommitEquipment）。一个设备（Equipment）依次经过这5个步骤完成完整的设备维护流程。

出处：

《Pipeline Rules of Thumb Handbook》 A Manual of Quick, Accurate Solutions to Everyday Pipeline Engineering Problems -8th Edition

by E.W. McAllister

## API 描述

addHandler(Handler<O,K> nextHandler):

EquipmentMaintenancePipeline <I,K>

**说明：**向管道添加子段

**参数：**将添加的下一子段

**返回值：**添加过该段后的管道

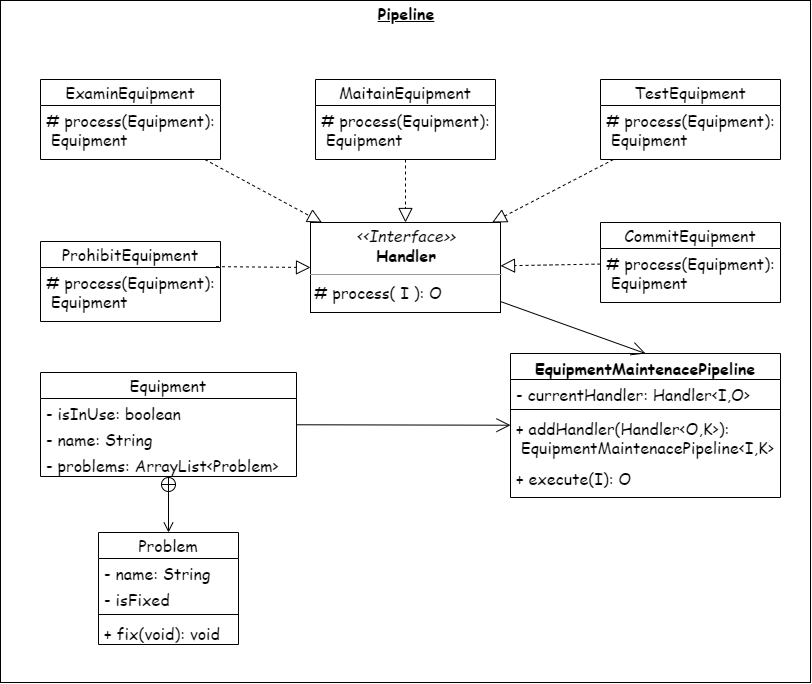
execute(I input): O

**说明：**执行以input:I为输入，产生O类型输出的管道

**参数：**input: I任意类型的输入

**返回值：**管道规定类型的输出。

## 类图



# 发布-订阅（Publish-Subscribe）模式

游客通过娱乐园App订阅游乐项目信息

## 模式简介

发布-订阅（Publish-Subscribe）模式定义了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将得到通知，在冰天雪地娱乐园内，游客可以通过游乐园的App订阅相关游乐项目信息，游乐园的相关项目也可以对订阅其的游客进行消息的发布。

参考出处：Birman, K. and Joseph, T. "Exploiting virtual synchrony in distributed systems" in Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87), 1987. pp. 123–138.

关于该设计模式的简介：https://en.wikipedia.org/wiki/ Publish–subscribe\_pattern

## API 描述

Entertainment.getName();

**说明：**用于获取游乐项目的名称。

**参数：**无

**返回值：**游乐项目的名称。

Visitor.getName();

**说明：**用于获取游客的名字。

**参数：**无

**返回值：**游客的名字。

Information.getInformation();

**说明：**用于获取发布信息的内容。

**参数：**无

**返回值：**发布信息的内容。

VisitorController.subscribe();

**说明：**游客进行订阅后调用，向Application类发送订阅请求。

**参数：**订阅的游客名字visitorName,该游客订阅的游乐项目名字entertainmentName.

**返回值：**无

VisitorController.unsubscribe();

**说明：**游客进行取消订阅后调用，向Application类发送取消订阅请求。

**参数：**取消订阅的游客名字visitorName,该游客取消订阅的游乐项目名字entertainmentName.

**返回值：**无

EntertainmentController.publish();

**说明：**App提交发布消息请求后调用，向Application类发送发布信息请求。

**参数：**发布信息的游乐项目名entertainmentName，发布的消息内容info.

**返回值：**无

Application.subscribe();

**说明：**执行订阅操作，将该条订阅关系存储。

**参数：**订阅的游客对象visitor,被订阅的游乐项目对象entertainment.

**返回值：**无

Application.unsubscribe();

**说明：**执行取消订阅操作，将相应的订阅关系删除。

**参数：**取消订阅的游客对象visitor,被取消订阅的游乐项目对象entertainment.

**返回值：**无

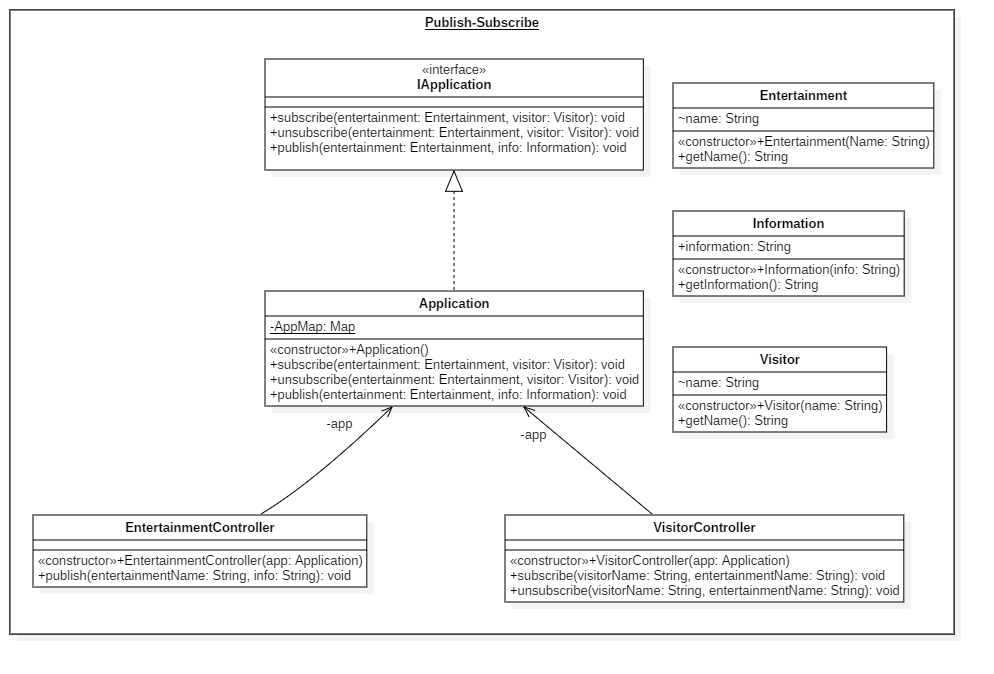
Application.publish();

**说明：**订阅的游客对象visitor,被订阅的游乐项目对象entertainment.

**参数：**发布信息的游乐项目对象entertainment，发布的消息内容info.

**返回值：**无

## 类图



# 读写锁(Read-Write Lock)模式

影院观众观影与维修工人维修幕布

## 模式简介

在多个线程共享一个实例的时候，会存在仅仅读取实例状态的读者，并且也会有改变实例的线程的写者，此时便需要用到读写锁模式，在冰天雪地娱乐园的影院中，存在影院的观众与维修幕布的工人，当维修工人进行幕布的维修时，观众不能进行幕布上电影的观看，而所有观众（Reader）可以一起进行电影的观看。

参考出处：“Taming Java Threads” by Allen Holub/ISBN1-89311510-0;

https://java-design-patterns.com/patterns/reader-writer-lock/

## API 描述

Film.read();

**说明：**对电影进行读，场景下即对电影进行观看。

**参数：**无

**返回值：**无

Film.write();

**说明：**对电影进行写，场景下即对幕布就行维修。

**参数：**无

**返回值：**无

Film.getFilmname();

**说明：**获取电影的名字。

**参数：**无

**返回值：**电影的名字。

AudienceThread.run();

**说明：**观众进行电影的观看。

**参数：**无

**返回值：**无

RepairManThread.run();

**说明：**维修工人进行幕布的维修。

**参数：**无

**返回值：**无

ReadWriteLock.readLock();

**说明：**锁住读线程，在读操作开始时执行。

**参数：**无

**返回值：**无

ReadWriteLock.readUnlock();

**说明：**解锁读线程，在读操作结束后执行。

**参数：**无

**返回值：**无

ReadWriteLock.writeLock();

**说明：**锁住写线程，在写操作开始时执行。

**参数：**无

**返回值：**无

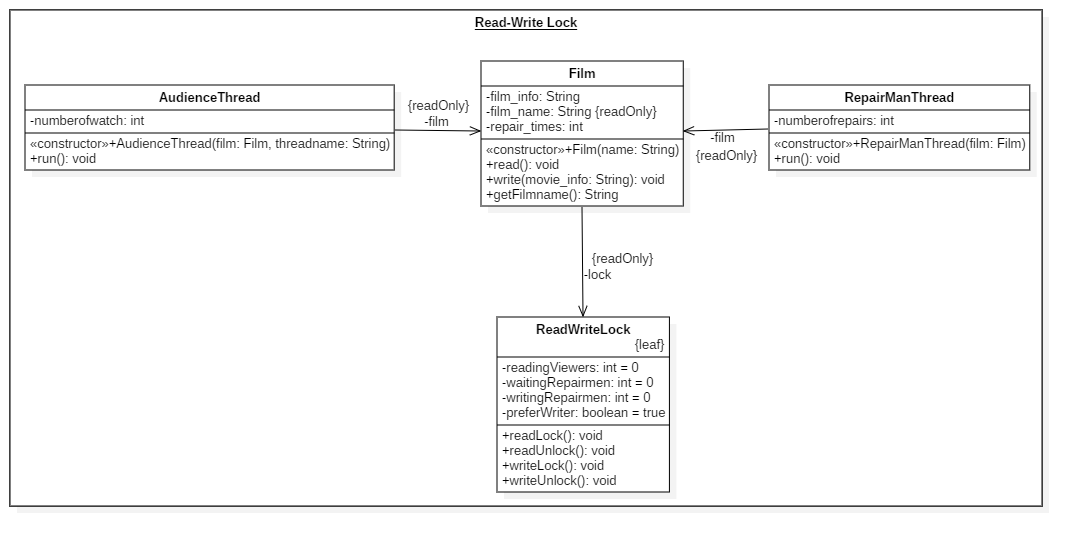
ReadWriteLock.writeUnlock();

**说明：**解锁写线程，在写操作结束后执行。

**参数：**无

**返回值：**无

## 类图



# 过滤器(Specification/Filter)模式

如何匹配一个候选者

## 模式简介

过滤器模式（Filter Pattern）或标准模式（Criteria Pattern）是一种设计模式，这种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。

游乐场内有多种设施，每种设施有不同的特征，比如设施类型，设施危险度，设施适合年龄段等等。用户根据所需特征来筛选出他所需要的游乐设施的列表。如果不满足则返回空列表。

参考出处：Evans, Eric (2004). Domain Driven Design. Addison-Wesley. p. 224.

## API描述

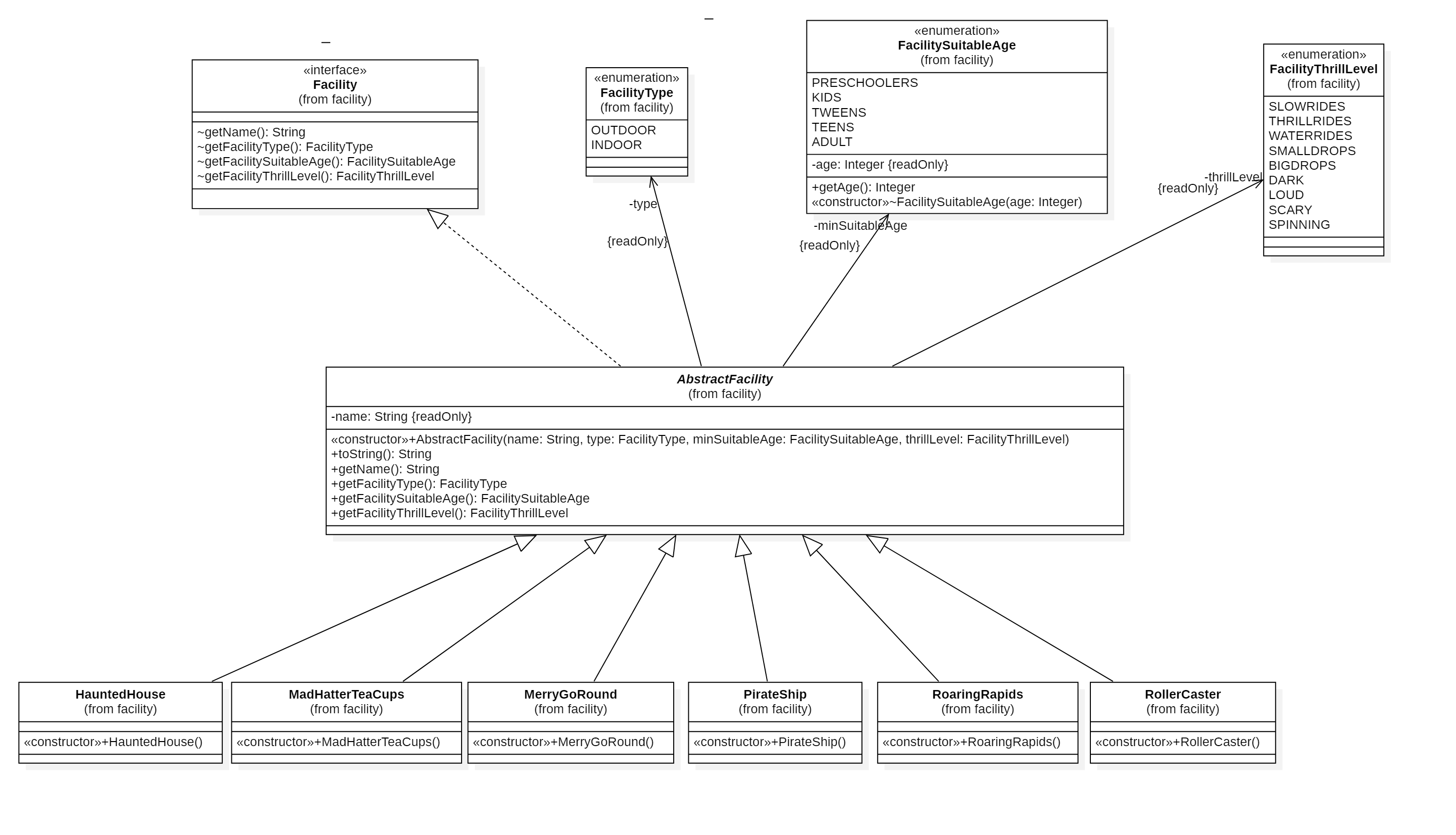
List<Facility>.stream().filter(new \*\*\*Selector(\*\*\*.\*\*\*)).collect(Collectors.toList());

**说明：**从列表中筛选出符合所给筛选条件的元素

**参数：**选择器和对应参数

**返回**：符合筛选条件的元素列表

## 类图



# 线程池(Thread Pool)模式

预定义一定数量的线程来处理工作负载

## 模式简介

对于每个命令或请求，都分配一个线程，由这个线程执行工作。它将委托消息的一端和执行消息的一端用两个不同的线程来实现。

众所周知，游乐场负责接待游客的人员新找一个要花费大量成本和时间。我们在游乐园开园的时候维护一个接待人员的队列，每次需要提供服务就从队列里面把人拿出来。服务结束后，人员回到队列里待命。当所有人都在接待时，客人需要等待。

参考出处："Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java" by Binildas C. A.

## API描述

Task()

**说明：**初始化一个任务

**参数：**timeMs 任务执行时间

**返回**：无

\*\*\*Task ()

**说明：**初始化一个具体的任务，继承 Task 并使用 super() 初始化任务所需时间

**参数：**numPeople 任务所需人数

**返回**：无

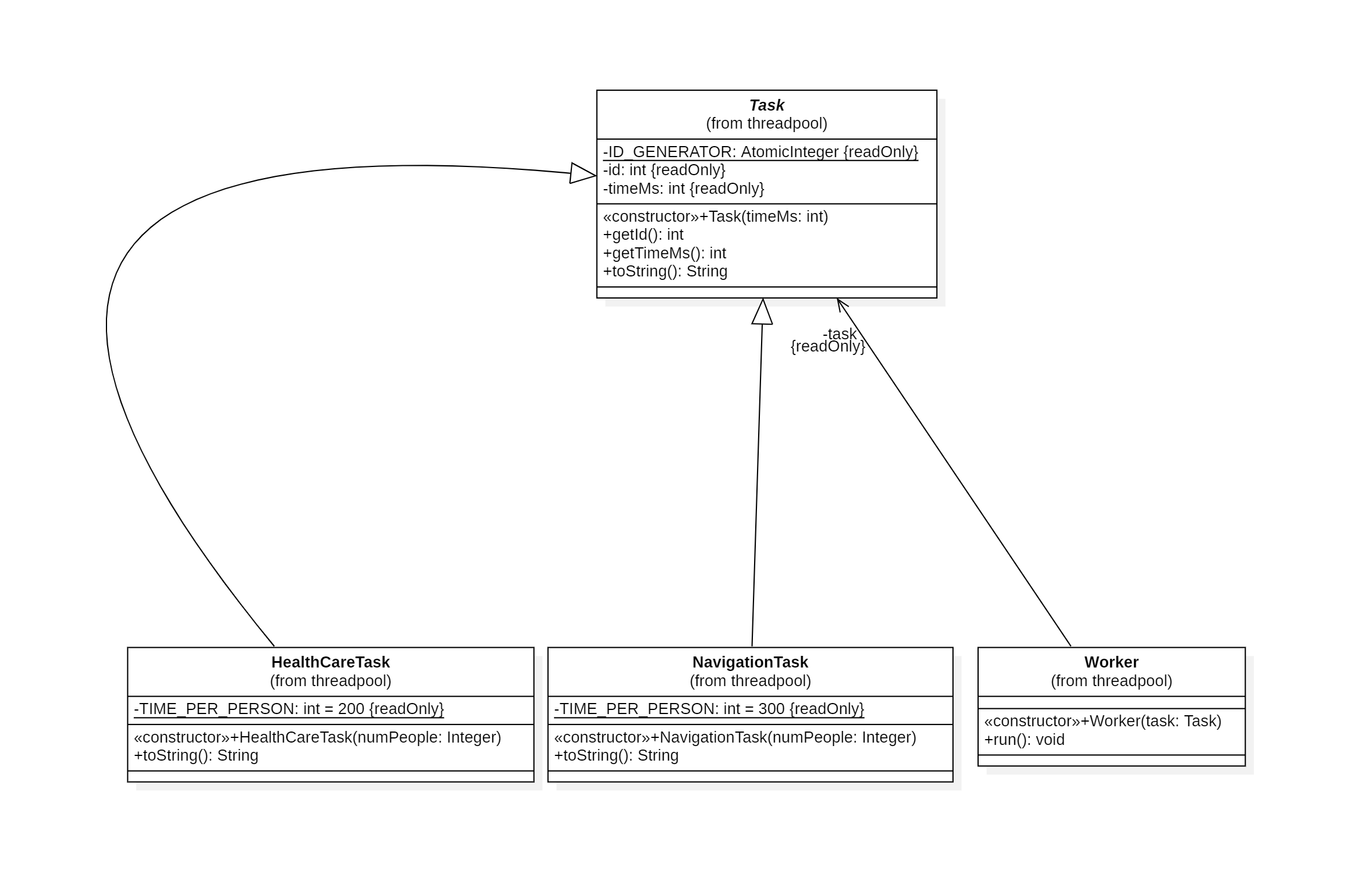
Worker()

**说明：**使用任务初始化一个工作线程

**参数：**task 任务

**返回**：无

## 类图



# 传输对象(Transfer Object)模式

传输用户信息

## 模式简介

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。

其中 UserVO代表User的数值对象，UserBO代表User的业务对象。

参考出处： MSDN (2010). Data Transfer Object. Microsoft MSDN Library. Retrieved from <https://msdn.microsoft.com/en-us/library/ms978717.aspx> .

## API描述

UserBO.deleteUser(UserVO user)

**说明：**删除业务对象中的某个数值对象

**参数：**需要删除的对象user

**返回**：无

UserBO.addUser(UserVO user)

**说明：**添加某个数值对象到业务对象

**参数：**需要添加的对象user

**返回**：无

UserBO.getUser(int index)

UserBO.getUser(String userId)

**说明：**获取某个user的数值对象

**参数：**索引index 或 用户ID userId

**返回**：查询到的UserVO

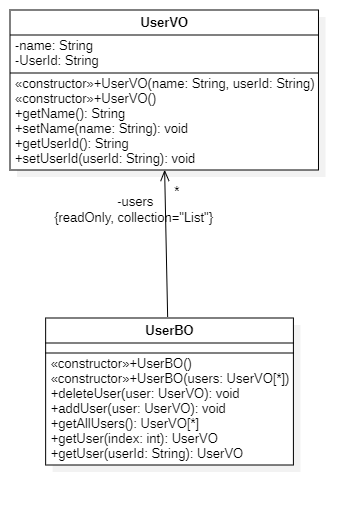
UserBO.getAllUser()

**说明：**获取所有user数值对象组成的列表

**参数：**无

**返回**：userVO列表

## 类图



# 值对象(Value Object)模式

## 模式简介

针对一个对象，我们更强调这个对象的值而不是对象本身，即判定两个对象是否相等时，必须是两个对象的值相等，而不是两者是同一个对象。

对于游戏机中的英雄(Hero)，具有力量(strength)、智力(intelligence)、幸运(luck)三个属性。英雄属性(HeroStat)的比较强调对象的值是否相同，故采用值对象模式。

## API描述

HeroStat.valueOf（strength, intelligence, luck）

**说明：**根据给定属性生成英雄状态

**参数：**英雄的力量值、智力值、幸运值

**返回：**无

## 类图

