**设计模式项目文档**

**——星爸爸咖啡工厂**

1952647 黄开宇

1953975 袁新杰

1950087 陈泽琦

1951336 张培知

1950067 娄天宇

1951966 张济显

1951722 刘创

1950739 申博

1951015 余世璇

1951504 王凌

2021年11月6日

**目录**

[1.项目简介 4](#_Toc86152856)

[2.Design Pattern汇总表 4](#_Toc86152857)

[3.设计模式详述 5](#_Toc86152858)

[3.1 Abstract Factory抽象工厂模式 5](#_Toc86152859)

[3.1.1设计模式简述 5](#_Toc86152860)

[3.1.2 DarkWorkshop和WhiteWorkshop实现API 6](#_Toc86152861)

[3.1.3类图 7](#_Toc86152862)

[3.6 Command命令模式 7](#_Toc86152863)

[3.6.1设计模式简述 7](#_Toc86152864)

[3.6.2 Product实现API 8](#_Toc86152865)

[3.6.3类图 8](#_Toc86152866)

[3.15 Command命令模式 8](#_Toc86152867)

[3.15.1设计模式简述 8](#_Toc86152868)

[3.6.2 Product实现API 9](#_Toc86152869)

[3.6.3类图 9](#_Toc86152870)

[3.24 Business Delegate业务代表模式 10](#_Toc86152871)

[3.24.1设计模式简述 10](#_Toc86152872)

[3.24.2 WarehouseDelegate实现API 10](#_Toc86152873)

[3.24.3类图 11](#_Toc86152874)

[3.24.4参考资料 11](#_Toc86152875)

[3.25 Callback回调模式 11](#_Toc86152876)

[3.25.1设计模式简述 11](#_Toc86152877)

[3.25.2 SceneSelector实现API 11](#_Toc86152878)

[3.25.3类图 12](#_Toc86152879)

[3.25.4参考资料 12](#_Toc86152880)

[3.26 Converter转换器模式 13](#_Toc86152881)

[3.26.1设计模式简述 13](#_Toc86152882)

[3.26.2 VisitorConverter实现API 13](#_Toc86152883)

[3.26.3类图 14](#_Toc86152884)

[3.26.4参考资料 14](#_Toc86152885)

[3.27 COW写时复制模式 14](#_Toc86152886)

[3.27.1设计模式简述 14](#_Toc86152887)

[3.27.2 HandmadeTool实现API 15](#_Toc86152888)

[3.27.3类图 16](#_Toc86152889)

[3.27.4参考资料 17](#_Toc86152890)

[3.28 Data Access Object数据访问对象模式 17](#_Toc86152891)

[3.28.1设计模式简述 17](#_Toc86152892)

[3.28.2 StaffDaoImpl类实现API 17](#_Toc86152893)

[3.28.3类图 18](#_Toc86152894)

[3.28.4参考资料 18](#_Toc86152895)

[3.29 Dirty Flag脏标记模式 19](#_Toc86152896)

[3.29.1设计模式简述 19](#_Toc86152897)

[3.29.2 CharlieFactory/TimeSystem实现API 19](#_Toc86152898)

[3.29.3类图 20](#_Toc86152899)

[3.29.4参考资料 20](#_Toc86152900)

[3.30 Extension Objects扩展对象模式 20](#_Toc86152901)

[3.30.1设计模式简述 20](#_Toc86152902)

[3.30.2 AugProductMachine实现API 20](#_Toc86152903)

[3.30.3类图 21](#_Toc86152904)

[3.30.4参考资料 21](#_Toc86152905)

[3.31 Filter拦截过滤器模式 21](#_Toc86152906)

[3.31.1设计模式简述 21](#_Toc86152907)

[3.31.2 FilterManager实现API 22](#_Toc86152908)

[3.31.3类图 22](#_Toc86152909)

[3.31.4参考资料 23](#_Toc86152910)

[3.32 Immutable不变模式 23](#_Toc86152911)

[3.32.1设计模式简述 23](#_Toc86152912)

[3.32.2 AbstractReceipt实现API 23](#_Toc86152913)

[3.32.3类图 24](#_Toc86152914)

[3.32.4参考资料 24](#_Toc86152915)

[3.33 IOC控制反转模式 24](#_Toc86152916)

[3.33.1设计模式简述 24](#_Toc86152917)

[3.33.2 WorkerAddingController实现API 25](#_Toc86152918)

[3.33.3类图 25](#_Toc86152919)

[3.33.4参考资料 25](#_Toc86152920)

[3.34 Monostate单一状态模式 26](#_Toc86152921)

[3.34.1设计模式简述 26](#_Toc86152922)

[3.34.2 Screen 实现API 26](#_Toc86152923)

[3.34.3类图 26](#_Toc86152924)

[3.34.4参考资料 26](#_Toc86152925)

[3.35 Multition多例模式 27](#_Toc86152926)

[3.35.1设计模式简述 27](#_Toc86152927)

[3.35.2 Product实现API 27](#_Toc86152928)

[3.35.3类图 28](#_Toc86152929)

[3.35.4参考资料 28](#_Toc86152930)

[3.36 Null Object空对象模式 28](#_Toc86152931)

[3.36.1设计模式简述 28](#_Toc86152932)

[3.36.2 EmptyStaff实现API 28](#_Toc86152933)

[3.36.3类图 29](#_Toc86152934)

[3.36.4参考资料 29](#_Toc86152935)

[3.37 Value Object值对象模式 29](#_Toc86152936)

[3.37.1设计模式简述 29](#_Toc86152937)

[3.37.2 MilkChocolate类实现API 29](#_Toc86152938)

[3.37.3类图 30](#_Toc86152939)

[3.37.4参考资料 30](#_Toc86152940)

[3.38 Transfer Object传输对象模式 30](#_Toc86152941)

[3.38.1设计模式简述 30](#_Toc86152942)

[3.38.2 Product 实现API 30](#_Toc86152943)

[3.38.3类图 32](#_Toc86152944)

[3.38.4参考资料 32](#_Toc86152945)

# 1.项目简介

本次项目选题为巧克力工厂，项目名称为查理的巧克力工厂。查理的巧克力工厂中有各种各样不同的场所，由一个总经理全权掌管，每个场所中有许多机器同时运作，由相应的经理进行管理，工人们通过对原料进行加工生产出不同的巧克力和巧克力制品。巧克力工厂还会开放展览室和体验室，欢迎世界各地的游客前往参观。

查理的巧克力工厂中涉及到的人有股东、经理、工人、参观者。其中股东是巧克力工厂的所有者，总经理只有一个，是巧克力工厂的总管理者，给各个部门和车间的经理分配任务；经理有食堂经理、办公室经理、车间经理、展览室经理等，负责管理该部门或车间的运作、给该部门或车间的工人分配工作并收集工作过程中工人反馈的问题，向总经理汇报。此外，还有来自世界各地的参观者，他们来到巧克力工厂，体验巧克力制作、品尝新品巧克力，参观巧克力展，乐趣无穷。

查理的巧克力工厂设施十分完善，除了生产巧克力及巧克力制品必须的加工车间和存储原料的仓库外，还有专门堆放生产废料的垃圾站。同时，工厂中的员工待遇也十分优厚，经理可以在自己的办公室工作，员工们不仅有专属的食堂，还提供了员工住宅区，包括普通公寓及小型别墅。

巧克力整体的生产过程由中央控制电脑总控，主要的生产过程包括加工原料生产出产品、对产品进行包装。生成一个巧克力制品，需要先通过加工机器进行生产，再通过包装机器包装产品。同时，工厂中还有质检机器对生成的产品进行质量检查，也有不同运输机器可以运输产品或乘载员工。

# 2.Design Pattern汇总表

|  |  |  |  |
| --- | --- | --- | --- |
| 编号 | 设计模式名称 | 数量 | 何处体现 |
| 1 | Abstract Factory | 1 | DarkWorkshop/WhiteWorkshop |
| 2 | Adapter | 1 | UAV |
| 3 | Bridge | 1 | Bridge |
| 4 | Builder | 1 | MilkChocController |
| 5 | Chain of Responsibility | 1 | FixWorker |
| 6 | Command | 1 | Product |
| 7 | Composite | 1 | QualityTestSystem |
| 8 | Decorator | 1 | ResidenceDecorator |
| 9 | Facade | 1 | SecurityCheckTask |
| 10 | Factory Method | 1 | ChocolatesMachineAssemblyLine |
| 11 | Flyweight | 1 | ChocolateMix |
| 12 | Interpreter | 1 | Expression |
| 13 | Iterator | 1 | WorkerListIterator |
| 14 | Mediator | 1 | Product |
| 15 | Memento | 1 | ProductMemento |
| 16 | Observer | 1 | Payroll |
| 17 | Prototype | 1 | RawMaterial |
| 18 | Proxy | 1 | PortableBattery |
| 19 | Singleton | 3 | GeneralManager/CharlieFactory/TransferObject |
| 20 | State | 1 | ProductState |
| 21 | Strategy | 1 | WrapStrategy |
| 22 | Template Method | 1 | TransportMachine |
| 23 | Visitor | 1 | UtilityWorker |
| 24 | Business Delegate | 1 | WarehouseDelegate |
| 25 | Callback | 1 | SceneSelector |
| 26 | Converter | 1 | VisitorConverter |
| 27 | COW | 1 | HandmadeTool |
| 28 | Data Access Object | 1 | StaffDaoImpl |
| 29 | Dirty Flag | 1 | CharlieFactory/TimeSystem |
| 30 | Extension object | 1 | AugProductMachine |
| 31 | Filter | 1 | FilterManager |
| 32 | Immutable | 1 | AbstractReceipt |
| 33 | IOC | 2 | WorkerAddingController/CentralControlComputer |
| 34 | Monostate | 1 | Screen |
| 35 | Multition | 1 | Product |
| 36 | Null Object | 1 | EmptyStaff |
| 37 | Value Object | 1 | MilkChocolate |
| 38 | Transfer Object | 1 | Product |

# 3.设计模式详述

## 3.1 Abstract Factory抽象工厂模式

### 3.1.1设计模式简述

抽象工厂模式的定义是一种为访问类提供一个创建一组相关或相互依赖对象的接口，且访问类无须指定所要产品的具体类就能得到同族的不同等级的产品的模式结构。抽象工厂模式是工厂方法模式的升级版本，工厂方法模式只生产一个等级的产品，而抽象工厂模式可生产多个等级的产品。

抽象工厂模式除了具有工厂方法模式的优点外，其他主要优点如下：

* 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。

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。

### 3.1.2 DarkWorkshop和WhiteWorkshop实现API

工厂生产的产品包括巧克力（Chocolate）和巧克力制品（ChocolateProduct）。巧克力可具体为白巧克力、黑巧克力和牛奶巧克力等，巧克力制品可具体为巧克力蛋糕、巧克力曲奇和巧克力华尔夫等，巧克力曲奇又可分为小狗形状曲奇、小猫形状曲奇等。

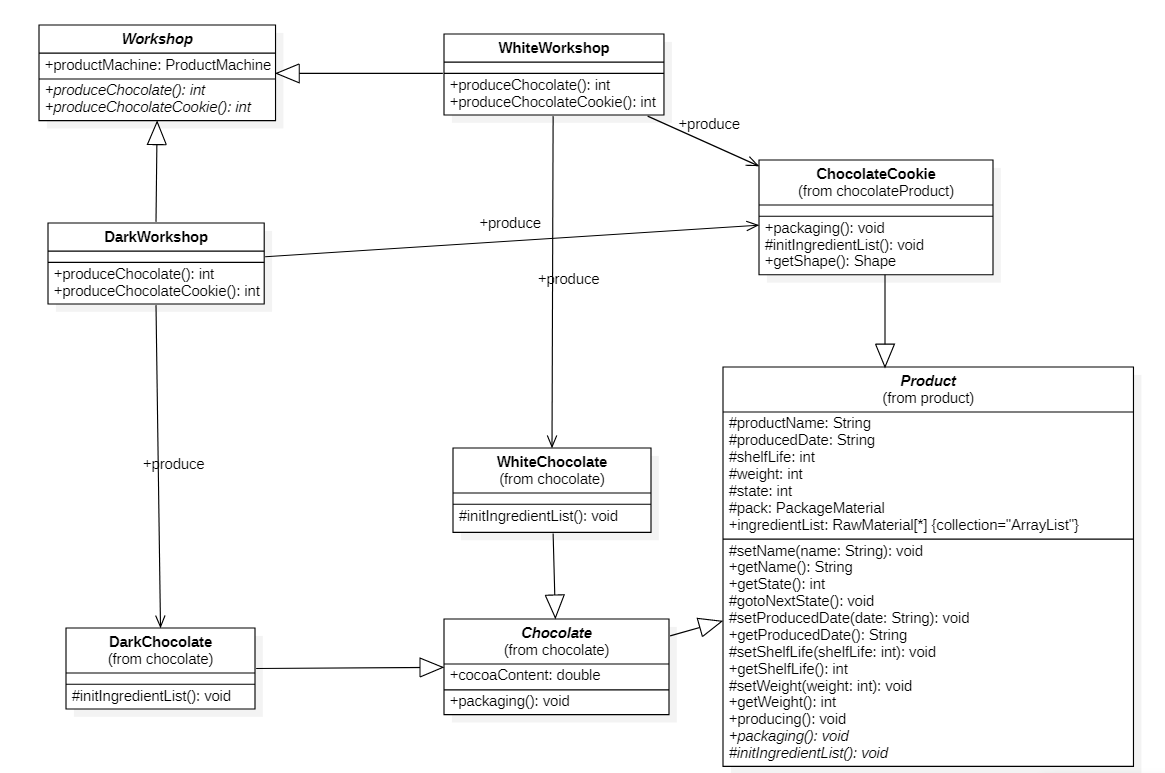
在抽象工厂模式中，通过继承抽象车间类和抽象产品类，可以实现在一个车间（Workshop）里生产一系列同族产品。黑巧克力车间（DarkWorkshop）生产黑巧克力和有黑巧克力夹心的小猫形状曲奇，白巧克力车间（WhiteWorkshop）生产白巧克力和有白巧克力夹心的小狗形状曲奇。

每个车间类都有一个公开成员变量productMachine，代表该车间的生产机器，由机器完成产品的实际生产过程，productMachine的实现对用户隐藏。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| DarkWorkshop | produceChocolate(): int | 说明：生产黑巧克力 参数：无 返回值：完成一次生产的产品产出量 |
| DarkWorkshop | produceChocolateCookie(): int | 说明：生产黑巧克力夹心小猫曲奇 参数：无 返回值：完成一次生产的产品产出量 |
| WhiteWorkshop | produceChocolate(): int | 说明：生产白巧克力 参数：无 返回值：完成一次生产的产品产出量 |
| WhiteWorkshop | produceChocolateCookie(): int | 说明：生产白巧克力夹心小狗曲奇 参数：无 返回值：完成一次生产的产品产出量 |

### 3.1.3类图



## 3.6 Command命令模式

### 3.6.1设计模式简述

定义：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。

主要优点：

* 通过引入中间件（抽象类）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OCP。
* 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。
* 方便实现 Undo 和 Redo 操作。命令模式可以与备忘录模式结合，实现命令的撤销与恢复。
* 可以在现有命令的基础上，增加额外功能。比如日志记录，结合装饰器模式会更加灵活。

主要缺点：

* 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。
* 命令模式的结果其实就是接收方的执行结果，但是为了以命令的形式进行架构、解耦请求与实现，引入了额外类型结构（引入了请求方与抽象命令接口），增加了理解上的困难。不过这也是设计模式的通病，抽象必然会额外增加类的数量，代码抽离肯定比代码聚合更加难理解。

### 3.6.2 Product实现API

本项目中的实现：

* 工人（Worker）是命令的调用者
* 产品（Product）是命令基类
* 巧克力的各种子类（DarkChocolate、MilkChocolate等）是具体的命令类
* 各种巧克力制品（ChocolateCake、ChocolateWafer等）是具体的命令类
* 机器（BasicProductMachine、WapperMachine）是命令的接收者

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Worker | +produce*Something*(): void (something可以换成任意产品类名，如DarkChocolate) | 下达生产某产品的命令，通过调用具体的命令类的producing()方法向接收方机器发送生产该产品的指令。 |
| Product | +producing(): void | 生产该产品的具体命令，调用相应接收方机器的processing()方法，传递命令，使之开始执行指令生产该产品。具体是哪个机器作为接收者需要在子类中确定。如下面的类图所示。 |
| BasicProductMachine | +processing(product: Product, productNum: int): void | 具体的生产过程，当命令类的对象调用到此类对象的方法时，开始具体的生产过程。 |
| WapperMachine | +processing(product: Product, productNum: int): void | 具体的生产过程，当命令类的对象调用到此类对象的方法时，开始具体的生产过程。 |

### 3.6.3类图

## 3.15 Command命令模式

### 3.15.1设计模式简述

定义：不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。该模式又叫快照模式。

主要优点：

* 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。

主要缺点：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。

### 3.15.2 ProductMemento实现API

在工厂的实际生产过程中，对一些历史记录进行存档是很普遍的现象。于是我们选择使用备忘录模式，来记录本项目中的产品数据。

本项目中，我们设置Product为发起人角色，其属性weight即记录了当前产品的重量状态，Product可以通过创建和恢复备忘录中的weight数据；ProductMemento为备忘录角色，负责储存weight数据，以及在Product需要恢复数据的时候提供数据给它；设置ProductCaretaker为管理者角色，它能够对ProductMemento中的weight数据进行管理，提供保存与获取备忘录数据的功能。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Worker | +produce*Something*(): void (something可以换成任意产品类名，如DarkChocolate) | 下达生产某产品的命令，通过调用具体的命令类的producing()方法向接收方机器发送生产该产品的指令。 |
| Product | +producing(): void | 生产该产品的具体命令，调用相应接收方机器的processing()方法，传递命令，使之开始执行指令生产该产品。具体是哪个机器作为接收者需要在子类中确定。如下面的类图所示。 |
| BasicProductMachine | +processing(product: Product, productNum: int): void | 具体的生产过程，当命令类的对象调用到此类对象的方法时，开始具体的生产过程。 |
| WapperMachine | +processing(product: Product, productNum: int): void | 具体的生产过程，当命令类的对象调用到此类对象的方法时，开始具体的生产过程。 |

### 3.15.3类图

## 3.24 Business Delegate业务代表模式

### 3.24.1设计模式简述

业务代表模式（Business Delegate）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。业务代表模式中有以下实体：

1. **客户端（Client）**：表示层代码可以是 JSP、servlet 或 UI java 代码。
2. **业务代表（Business Delegate）**：一个为客户端实体提供的入口类，它提供了对业务服务方法的访问。
3. **查询服务（LookUp Service）**：查找服务对象负责获取相关的业务实现，并提供业务对象对业务代表对象的访问。
4. **业务服务（Business Service）**：业务服务接口。实现了该业务服务的实体类，提供了实际的业务实现逻辑。

### 3.24.2 WarehouseDelegate实现API

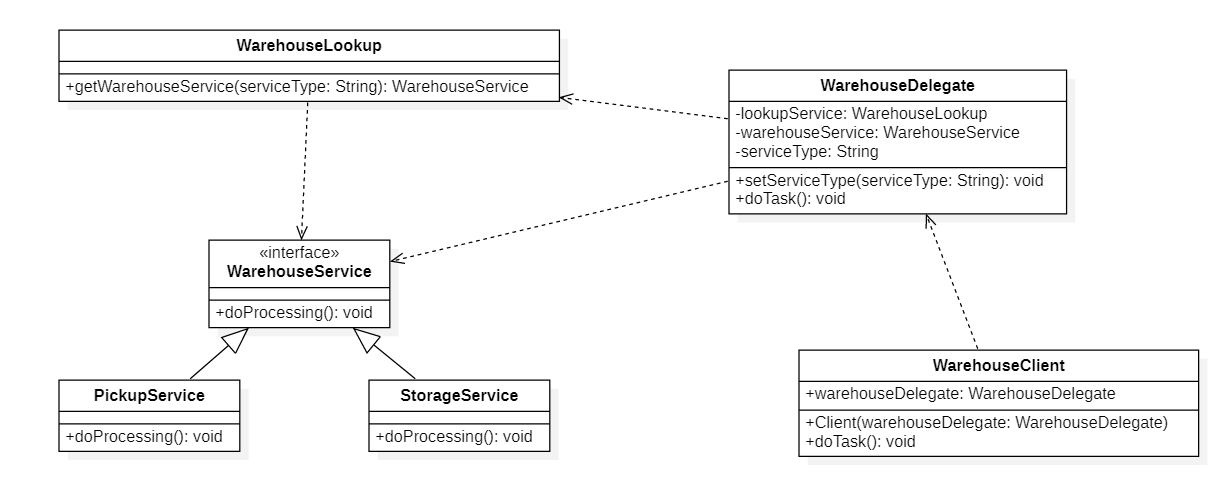
在巧克力工厂中，我们设计了工厂的智能仓库，员工不需要亲自进入仓库取货、存货，只需要通过一个仓库智能终端（WarehouseClient）即可完成工作。仓库提供多种服务，如取货服务（PickupService）、存货服务（StorageService），我们设计了相应的类来实现该服务，所有服务类都需实现仓库服务接口（WarehouseService）。

在仓库终端和仓库服务之间，我们设计了为仓库终端实体提供的入口类（WarehouseDelegate），它提供了对所有仓库服务的访问。访问具体服务前，需要通过仓库服务查询类（WarehouseLookup）来获取相关的服务实现。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| WarehouseDelegate | setServiceType(serviceType: String): void | 设置具体仓库服务的类型 参数：serviceType 仓库服务名称 |
| WarehouseDelegate | doTask(rawMaterial: RawMaterial): void | 业务代表让仓库执行任务，具体的任务由WarehouseDelegate对象中的服务类型决定 参数： rawMaterial 进行操作的原料 |
| WarehouseLookup | getWarehouseService(serviceType: String): WarehouseService | 通过查询服务查询智能仓库是否含有该服务，如果有则返回该服务的实体，否则返回空值 |
| WarehouseService | doProcessing(warehouse: Warehouse, rawMaterial: RawMaterial): void | 任何实现该接口的类都需要实现此方法，用于执行具体的仓库服务 |

### 3.24.3类图



### 3.24.4参考资料

<https://en.wikipedia.org/wiki/Business_delegate_pattern>

<https://www.tutorialspoint.com/design_pattern/business_delegate_pattern.htm>

## 3.25 Callback回调模式

### 3.25.1设计模式简述

回调（Callback）模式是一个小巧且被广泛使用的设计模式。其主要目的是为了暴露一些接口供用户在指定的时间点调用自定义方法来处理，或是对于有相互调用关系的众多类进行解耦合，抑或是满足OCP原则，对未来可能的扩展方法或根本无法提前预知的方法提供接口。

回调模式的核心在于将函数指针（引用）作为参数传入方法。在方法的执行过程的特定时间点调用该函数，传入方法产生的数据，并可选地获取返回值并进一步处理。其与Strategy模式最大的区别在于它不需要类具有继承关系，不依赖于虚函数机制。

### 3.25.2 SceneSelector实现API

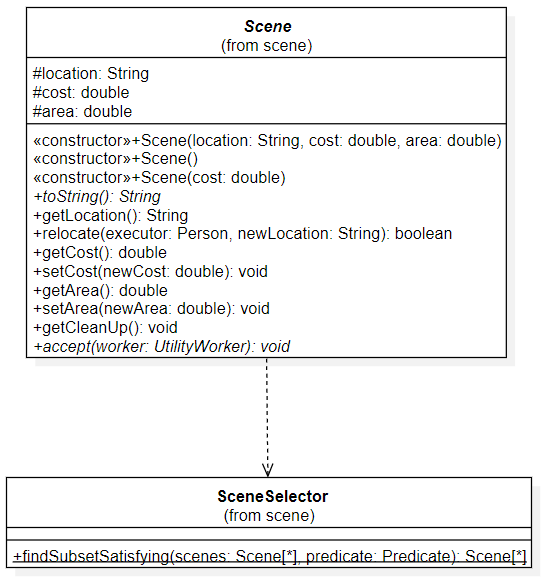
考虑到工厂管理人员常常要查找各类满足一定条件的场景，例如：寻找面积足够大的活动空间，寻找位置在二楼的所有办公室，寻找所有的厂房。而这些要求往往是复杂且难以预知的，故不能写作任何一个类的方法。这里使用Callback模式实现一个筛选器解决这种问题。

该类主要实现一个在Scene类数组上的筛选器。筛选器的方法传入两个参数：一个是待筛选的Scene类数组，一个是回调函数predicate。筛选器返回Scene中所有使predicate返回true的元素构成的子组。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| SceneSelector | findSubsetSatisfying(scenes: Scene[\*], predicate: Predicate): Scene[\*] | 上文所描述的筛选器。 参数：scenes 一个Scene构成的List，predicate 一个输入为Scene，返回boolean的**lambda函数**。（或者Predicate类型实例） |

### 3.25.3类图



### 3.25.4参考资料

<https://en.wikipedia.org/wiki/Callback_(computer_programming>

## 3.26 Converter转换器模式

### 3.26.1设计模式简述

在日常开发的时候，需要在对象之间进行值的 copy，如 POJO，DTO，VO，对象之间有相同的属性，想把一个对象的值 copy 到另一个对象中去，如 从数据库中查询出我们的 POJO 对象的数据，又有个对象是对 POJO 进行包装DTO，现在想把查询出来的 POJO 的值 copy 到 DTO 中相应的属性中去，之后再扩展其属性，对此，可以通过Converter来解决。

转换器设计模式，为相应类型之间的双向转换（如DTO和逻辑同构类型的域表示）提供了一种通用的方式，允许类型无需彼此了解的简洁的实现。此外，转换器设计模式引入了双向收集映射，将样板代码减少到最小。

### 3.26.2 VisitorConverter实现API

在巧克力工厂中，具有不同的人员，如经理，普通员工和参观人员等。我们往往需要存储这些不同人员的信息，并将其信息进行传输。在设计人员类时，我们不仅需要设计其属性，还需要设计其行为，但在存储其基本信息时，往往不需要其行为而只需其属性，因此为了存取的方便，往往需要进行新的对象的创建和信息的复制。

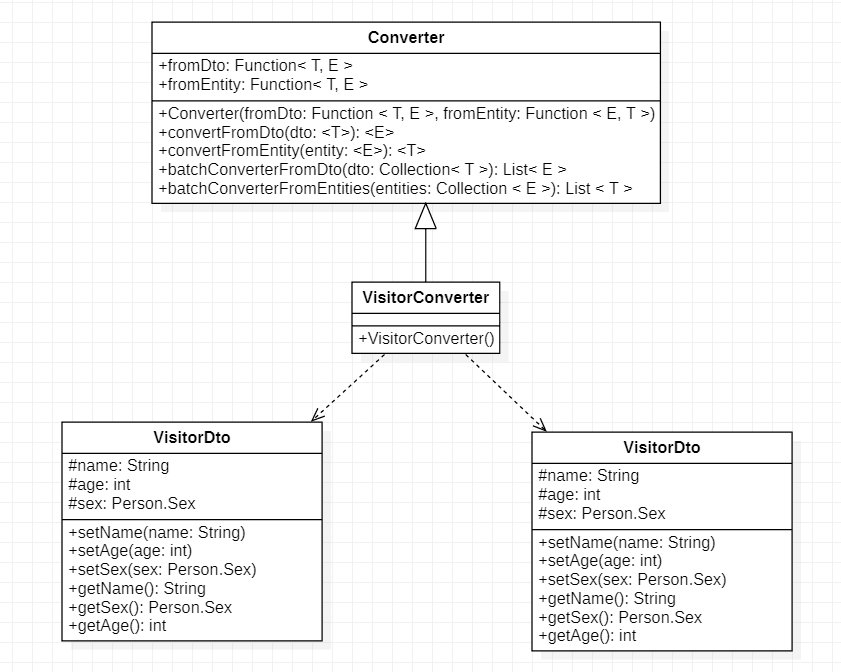
显然的，这种工作是较为单一繁琐的，容易引起代码的冗余和浪费。因此，我们使用转化器设计模式，使类型无需彼此了解即可简洁实现，可扩展性增强。

* Converter在项目中的实现
  + 抽象转化器（Aggregate）：定义了公共的转换方法。
  + 参观者转化器（VisitorConverter）：具体转化器，实现抽象转化器功能，完成visitorDto和visitor这两个不同的类所创建对象之间的相互转化。
  + 参观者 （Visitor）：可以通过VisitorConverter与VisitorDto进行相互转化。
  + VisitorConverte：可以通过VisitorConverter与Visitor进行相互转化。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| StaffArea | +convertFromDto(): Iterator | 获取⼀个根据工作地点的聚合类创建出来的员工的迭代器 |
| StaffArea | +first(): Worker | 获取第⼀个合理的员工对象 |
| StaffArea | +next()：Worker | 获取下⼀个合理的员工对象 |
| StaffArea | +hasNext()：boolean | 判断是否还有下⼀个合理的员工对象 |

### 3.26.3类图



### 3.26.4参考资料

<https://www.sourcecodeexamples.net/2018/03/converter-pattern.html>

<https://www.boldare.com/blog/converter-pattern-in-java-8/>

## 3.27 COW写时复制模式

### 3.27.1设计模式简述

写时复制（Copy on Write, COW）模式是一种节约内存的方式。其思想核心在于尽可能的延后实例构建的时间。和Flyweight模式类似，当对象需要被复制或新建时，在没有被写入时，其与初始对象共享同一个引用。当对象被写入时，才真正的复制一个对象，并修改对象引用。这个设计模式在进程协作，内存资源管理都有运用。

COW模式的设计核心在于引用计数和共享。共享是指在实例被两个请求者同时请求时，在没有写入发生时引用同一个实例。引用计数是指共享过程中记录该实例被引用的个数。随着请求者请求对象和修改对象，该计数相应的增加或减少。

### 3.27.2 HandmadeTool实现API

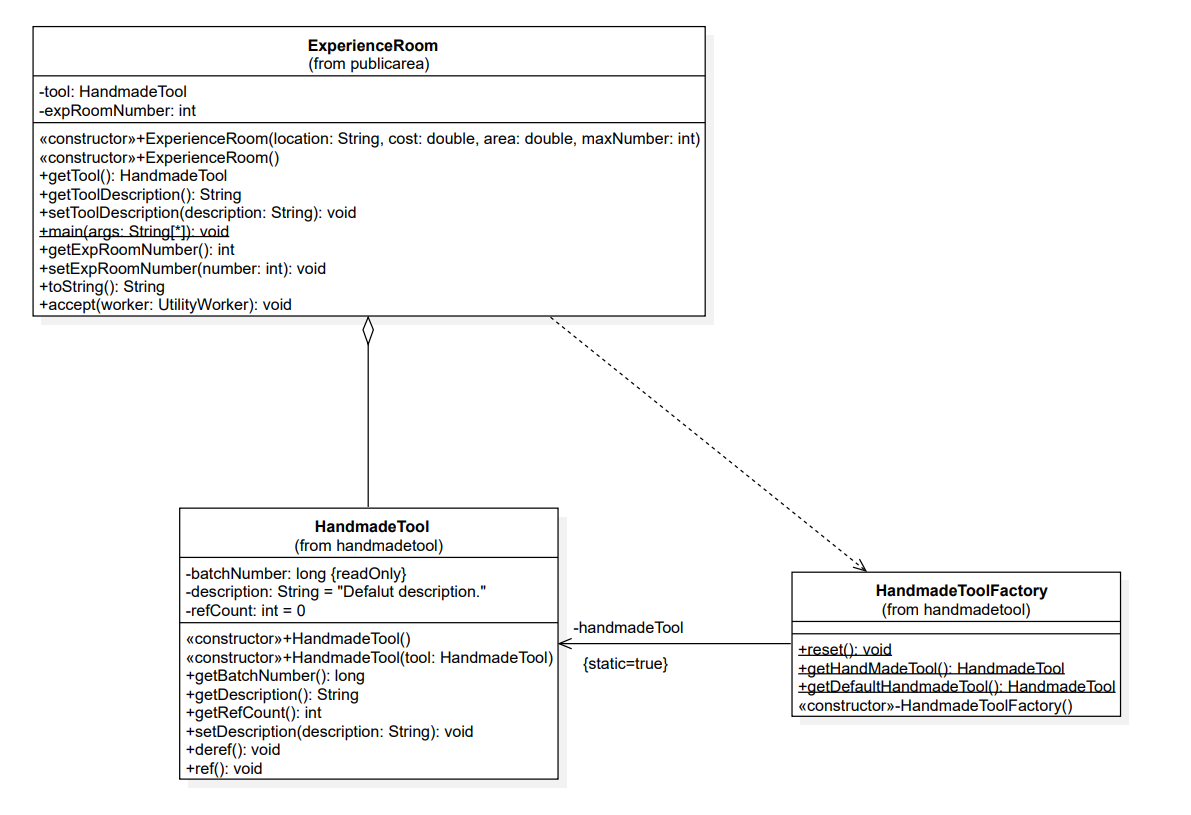
在每个体验室（ ExperienceRoom ）中，都拥有一个手工制作巧克力的工具（ HandmadeTool ）。该工具拥有一段描述和生产批号。在向工厂类 HandmadeToolFactory 请求 HandmadeTool 实例时，会得到默认工具的引用（且默认工具的引用计数自增）。此工具的描述是默认的（"Defalut description."）。 如果修改该工具实例的描述，则触发COW，实例被复制，体验室实例会得到新的引用，且默认实例的引用计数自减。生产批号在复制时不改变。

注：如果按照COW的默认实现，有体验室修改唯一默认实例的描述时，接下来的请求都会复制修改后描述。为了避免这个问题，本程序在对COW实现中，规定所有对工具实例的请求都会获得默认实例的引用。且所有修改都会触发COW，即使默认实例的引用为1。在默认实例引用计数为0时假装销毁（没有体 验室实例拥有该引用，但是没有实际销毁默认实例）。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| HandmadeTool | HandmadeTool() | 默认构造函数。 输出：被调用时输出"New Handmade Tool!" |
| HandmadeTool | HandmadeTool(tool: HandmadeTool) | 复制构造函数。 输出：被调用时输出"Copy Handmade Tool!" |
| HandmadeTool | getBatchNumber(): long | 获得生产批号，即实例被生成的时间 戳。（由于COW的关系，以及该类没有开放修改生产批号的接口，在 HandmadeToolFactory.reset()没有被调用之前，所有实例的生产批号应当是一致的。） |
| HandmadeTool | getDescription(): String | 获得该实例的描述。 |
| HandmadeTool | setDescription(description: String): void | 不建议测试时直接调用。建议通过 ExperienceRoom类的 setToolDescription()调用。设置该实例 的描述。该方法会触发COW。 输出：由于该函数依赖 HandmadeTool的默认构造函数，故会输出"Copy Handmade Tool!"。 |
| HandmadeTool | getRefCount(): int | 测试用API。获取引用计数。 |
| HandmadeTool | deref(): void | 不建议测试时调用。减少引用计数。 |
| HandmadeTool | ref(): void | 不建议测试时调用。增加引用计数。 |
| HandmadeTool  Factory | getHandMadeTool(): HandmadeTool | 获取一个HandmadeTool实例。自动处理默认实例引用计数。 |
| HandmadeTool  Factory | getDefaultHandmadeTool(): HandmadeTool | 测试用API。获取默认HandmadeTool 实例。不处理默认实例引用计数。 |
| HandmadeTool  Factory | reset(): void | 测试用API。默认情况下不使用。重新生成默认实例（包括生产批号）。输出：由于该函数依赖 HandmadeTool()，故调用时输出"New Handmade Tool!"。 |
| ExperienceRoom | getTool(): HandmadeTool | 获得当前房间的工具实例。 |
| ExperienceRoom | getToolDescription(): String | 获得当前房间工具实例的描述。与 getTool().getDescription()等价。 |
| ExperienceRoom | setToolDescription(description: String): void | 设置当前房间工具实例的描述。与 getTool().setDescription()等 价。该方法会触发COW。 输出：由于该函数依赖 HandmadeTool的默认构造函数，故会输出"Copy Handmade Tool!"。 |
| ExperienceRoom | main(args: String[\*]): void | 对于COW模式的样例测试代码，建议参考。该函数中包括对写入之前是否为同一个实例的引用以及是否做到写时复制的测试。 |

### 3.27.3类图



### 3.27.4参考资料

<https://en.wikipedia.org/wiki/Copy-on-write>

## 3.28 Data Access Object数据访问对象模式

### 3.28.1设计模式简述

DAO模式用于把低级的数据访问API或操作从高级的业务服务中分离出来，通过新增DAO层将底层数据访问和上层业务逻辑解耦。DAO是一个对象，负责提供抽象的接口与特定形式的数据库进行通信。通过映射方法，程序可以通过DAO提供的某种类型的数据操作（如GET，DELETE，PUT，INSERT等）调用持久层，而无需公开数据库实际包含的内容，这种解耦能够支持单一责任原则。

DAO模式适用于许多编程语言，以及具有持久化需求的相同类型的软件和大型数据库。数据访问的对象使用时相对简单，其代表了应用程序两个部分之间的分离（业务逻辑和持久性逻辑）。这两部分能够但不应该相互了解，业务逻辑的更改往往取决于DAO接口，而在接口正确实现之前，对持久性逻辑的更改也不会影响DAO客户端。使用DAO的应用程序其余部分无权访问存储的全部详细信息，代码人员也无需修改整个应用程序即可对持久性机制进行更改，从而减少了业务逻辑和持久层逻辑之间的耦合。

最后，由于DAO模式是基于接口的，因此它还推广了面向对象的设计原则——“接口编程胜于实现”，能够产生灵活而高质量的代码。

### 3.28.2 StaffDaoImpl类实现API

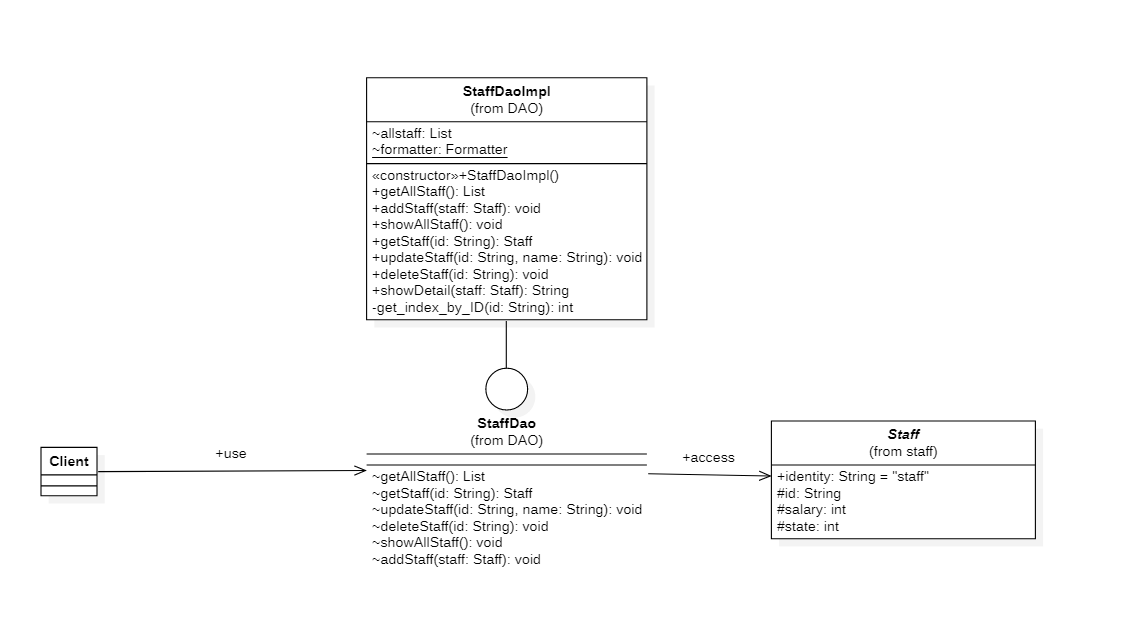
在巧克力工厂中，为了更好地管理员工，通常将员工的信息以通过存入数据库的方式实现数据持久化。在本例中，我们使用一个元素为Staff对象的List模拟为存储员工信息表的数据库，设计模式的主要核心体现在数据库操作的业务逻辑和持久性逻辑的解耦上。本例中的StaffDao接口定义对应Staff类型数据对象的操作函数，包括数据库常用的增删查改操作，StaffDaoImpl则作为StaffDao的实现类读取存储Staff对象的List，具体实现数据库相关的模拟操作。若将来有Staff类型数据存入数据库，只需新增StaffDao的子类从数据库中读取Staff对象，无需改动原有代码。Staff对象主要存储了员工ID、员工姓名、员工性别、员工年龄、员工薪水等信息，设置员工ID为主键，进行查询或删除操作时都以员工ID方式进行获取。

我们将包装的数据访问对象设计为StaffDao接口类、StaffDaoImpl实现类及其子类，下表具体说明实现类中的函数用法。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| StaffDaoImpl | StaffDaoImpl() | 构造函数，新建List列表存储staff信息并add已存在员工的信息。 |
| StaffDaoImpl | getAllStaff()): List | 获取员工信息表 |
| StaffDaoImpl | showAllStaff(): void | 模拟数据库表格规范化的形式输出当前员工表。 |
| StaffDaoImpl | addStaff(Staff staff): void | 实现数据库中增加操作，新增一个员工到表中。 参数：Staff对象 |
| StaffDaoImpl | getStaff(String id):Staff | 实现数据库中查询操作，根据员工ID（primary key）来进行查询。 参数：员工ID |
| StaffDaoImpl | updateStaff(String id,String name)：void | 实现数据库中的修改操作，对有特定ID的员工进行重命名操作（因权限问题，测试代码中只允许修改员工姓名）。 参数：员工ID，新命名Name |
| StaffDaoImpl | deleteStaff(String id)：void | 实现数据库中的删除操作，删除指定ID员工的所有信息。 参数：员工ID |

### 3.28.3类图



### 3.28.4参考资料

<https://www.dineshonjava.com/data-access-object/>

<https://www.javaguides.net/2018/08/data-access-object-pattern-in-java.html>

## 3.29 Dirty Flag脏标记模式

### 3.29.1设计模式简述

Dirty Flag模式（脏标记模式）定义了一种方法，对于某些不需要每次进行的操作（通常这类操作对时间或空间消耗较大），为其设置一个标记来指明其是否需要立即执行，在每次可能执行的情况下进行判断，如果标记为真则执行，标记为假则跳过本次执行，等待下一次检查时确定是否执行。

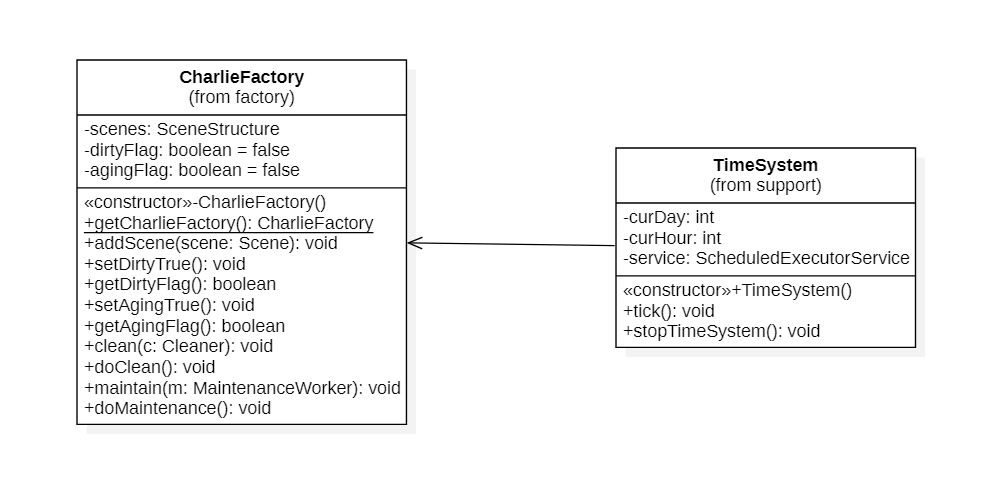
### 3.29.2 CharlieFactory/TimeSystem实现API

在巧克力工厂中，需要定期对工厂内的场景进行清理，也需要对工厂里的各个场景执行不同的保养工作，清理与保养操作的具体操作与细节已经通过Visitor模式实现，而其中定期这一功能则使用Dirty Flag模式实现。在实现过程中，工厂类设置了脏标记与设备老化标记，并设置了修改与获取的方法，然后实现了一个时间系统类，内置一个定时器，每隔一段时间进行一次标记检查，并依据标记状况调用工厂类的清理与保养方法：

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| TimeSystem | tick(): void | 类内设置了一个定时器，每隔一段时间检查一下清理与保养操作的标记，如果为真则执行对应操作，如果为假则此次不执行该操作 |
| CharlieFactory | setDirtyTrue(): void  setAgingTrue(): void | 用于修改清理与保养操作的标记为真 |
| CharlieFactory | getDirtyFlag(): boolean  getAgingFlag(): boolean | 用于获取清理与保养操作的标记 |

### 3.29.3类图



### 3.29.4参考资料

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

<https://gameprogrammingpatterns.com/dirty-flag.html>

## 3.30 Extension Objects扩展对象模式

### 3.30.1设计模式简述

我们希望改进生产机器的功能，使之能连续生产给定次数，同时又不希望改变原有的接口，这就要用到扩展对象（Extension objects）模式。

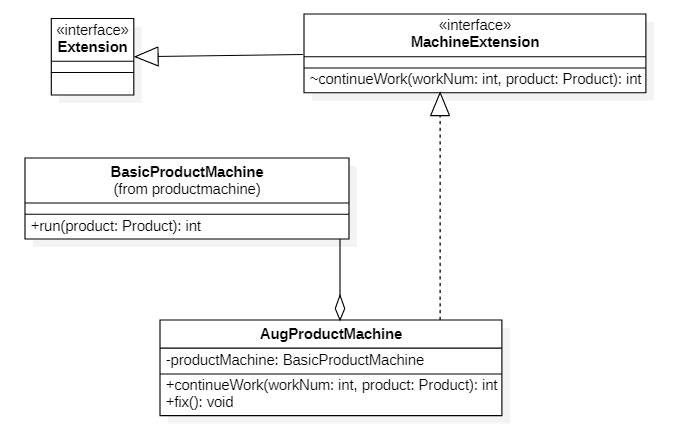
MachineExtension接口扩展自Extension接口，AugProductMachine类实现这一接口，它有一个私有成员变量productMachine，在AugProductMachine类的continueWork方法中根据给定的次数循环调用productMachine的run方法，实现连续生产。

### 3.30.2 AugProductMachine实现API

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| AugProductMachine | continueWork(workNum: int, product: Product): int | 说明：连续生产给定次数 参数：workNum代表要求的生产次数，Product代表要生产的产品类型 返回值：完成多次生产的产品产出量 |
| AugProductMachine | fix(): void | 说明：提供机器维修接口以防故障 参数：无 返回值：无 |

### 3.30.3类图



### 3.30.4参考资料

<https://ecs.syr.edu/faculty/fawcett/handouts/cse776/PatternPDFs/ExtensionObject.pdf>

<http://www.design-nation.net/en/archives/000489.php>

## 3.31 Filter拦截过滤器模式

### 3.31.1设计模式简述

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

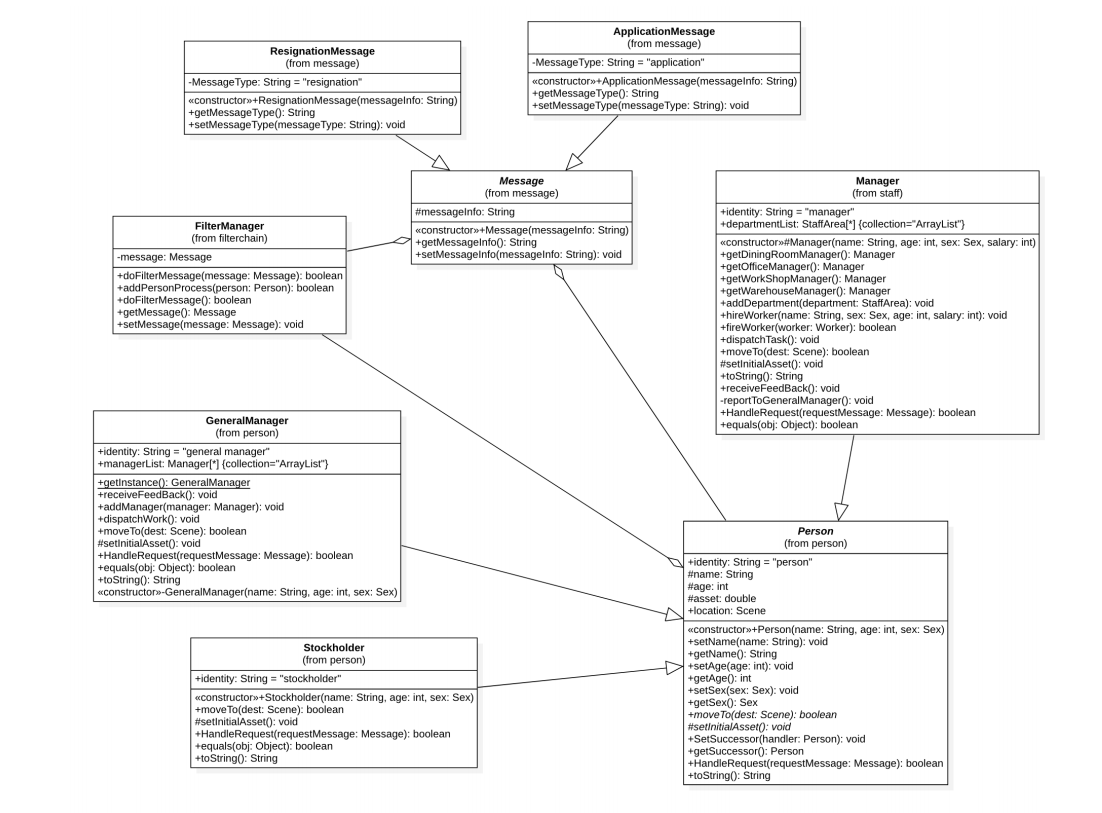
### 3.31.2 FilterManager实现API

FilterManager提供管理过滤器链的一些列⽅法，并且可以执行过滤器链并返回结果。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| FilterManager | public boolean doFilterMessage(Message message) | 传入Message信息，执行拦截过滤链，要是成功通过，返回true，要是失败返回 false。 |
| FilterManager | public boolean addPersonProcess(Person person) | 传入审批⼈，加入到过滤器链中参与审核。 |
| FilterManager | public boolean doFilterMessage() | 无参的执行过滤链的函数，要是成功通过，返回true，要是失败返回false。 |
| Person | public void SetSuccessor(Person handler) | 设置下一个审批⼈ |
| Person | public abstract boolean HandleRequest(Message requestMessage); | 接收Message，返回处理结果 |

### 3.31.3类图



### 3.31.4参考资料

<https://www.tutorialspoint.com/design_pattern/intercepting_filter_pattern.htm>

## 3.32 Immutable不变模式

### 3.32.1设计模式简述

immutable就是不变的，不发生改变的。Immutable模式中存在着确保实例状态不发生变化改变的类。这些实例不需要互斥处理。String就是一个Immutable类，String实例所表示的字符串的内容不会变化。

### 3.32.2 AbstractReceipt实现API

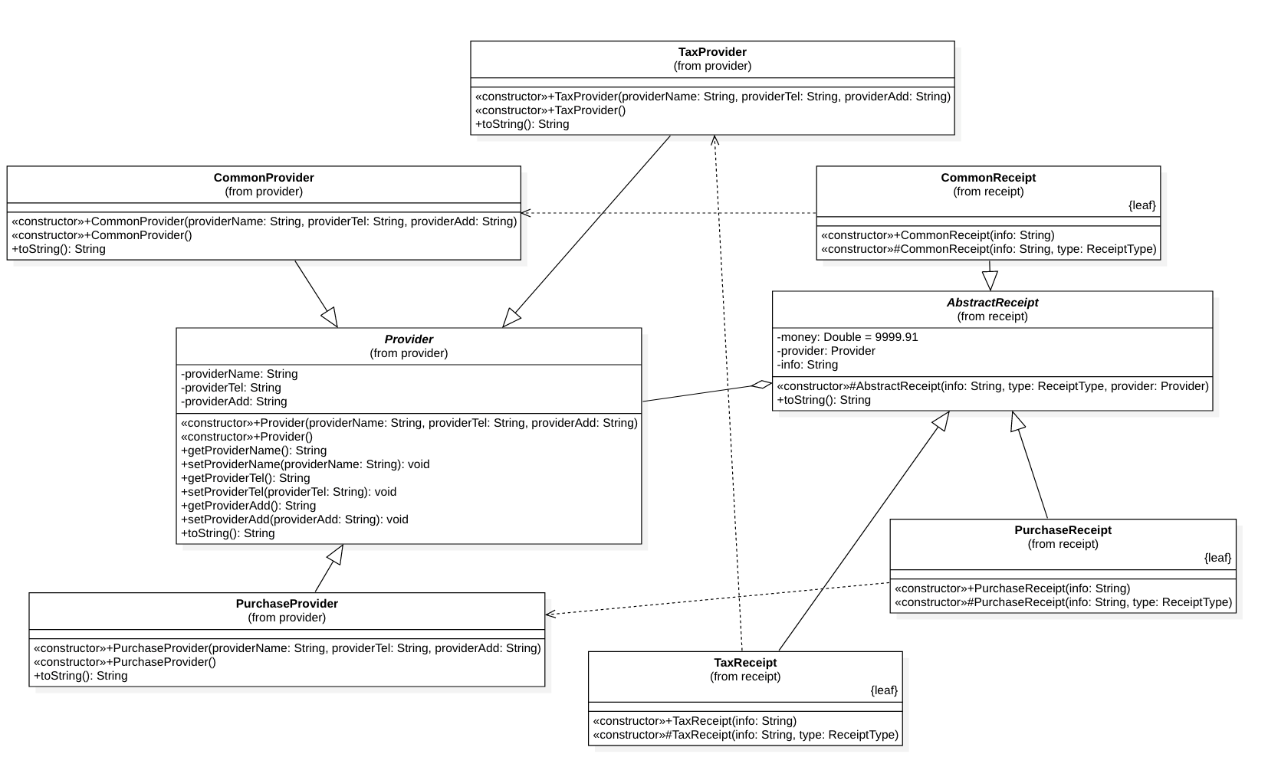
在巧克力工厂整个大环境下，我们每天都会产生大量的资金流动，资金流动就会产生发票，由于金额的敏感性，我们的发票对象一旦初始化，就不可以改变，系统使用了三种发票，分别是税务发票（TaxReceipt），普通发票（CommonReceipt），购买发票（PurchaseReceipt），他们都使用了final修饰，也就是对象不能被改变。

**API描述**

巧克力工厂实现这个抽象类，返回特定巧克力产品。

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| AbstractReceipt | protected AbstractReceipt(String info, ReceiptType type) | 给子类调用的构造函数。 |
| CommonReceipt | public CommonReceipt(String info) | 初始化普通的发票 |
| PurchaseReceipt | public PurchaseReceipt(String info) | 初始化购买发票 |
| TaxReceipt | public TaxReceipt(String info) | 初始化税务发票 |

### 3.32.3类图



### 3.32.4参考资料

<https://en.m.wikipedia.org/wiki/Immutable_interface>

## 3.33 IOC控制反转模式

### 3.33.1设计模式简述

IOC模式（控制反转模式）定义了一种方法，用来削减计算机程序的耦合问题。控制反转也被称作“依赖注入”，其遵循了面向对象程序设计的重要法则之一的“依赖倒置原则”。

IOC模式适合解决这样的高耦合问题：在某个类（用ClassA指代）中用确定的代码生成另一个类（用ClassB指代）的实例对象，则这两个类因此高度耦合，如果ClassB修改定义，则ClassA因此受到影响，也需要修改代码，这违背了OCP原则。

为解决这种问题，可以使用IOC模式，有两种实现方式：第一种是创建一个专用于实例ClassB操作的控制类ClassC，将实例对象的控制权反转给ClassC，这样ClassC直接将实例好的对象注入ClassA，从而将ClassA与ClassB之间的高度耦合关系消除；一种实现方法是将实例化ClassB的相关内容写入一个配置文件中（一般来说使用xml文件作为配置文件），在ClassA中实例ClassB时通过读取配置文件中的内容来完成，这样ClassB修改定义时只需要修改配置文件而不需要再修改ClassA中内容了，从而实现了ClassA与ClassB的解耦。

在本项目中使用第一种方法实现IOC模式。

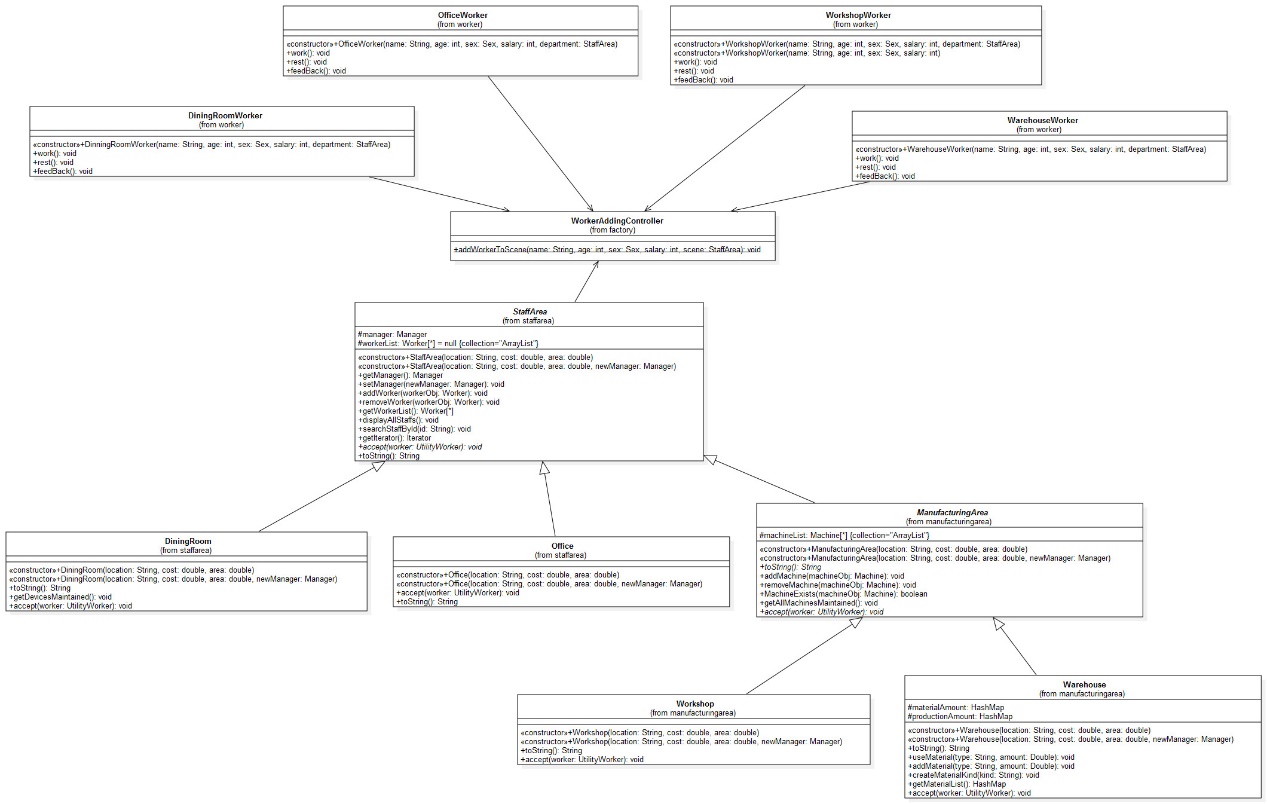
### 3.33.2 WorkerAddingController实现API

在巧克力工厂中，每个部门均对应一个工种，若不使用IOC模式，在对应部门的场景类中创建对应工种的对象，则各场景类与各工人类紧紧耦合，若修改工人的属性，则需要修改全部场景类，违背了OCP原则。考虑到场景类基本稳定，因此为了使工人类的修改不影响场景类，遵照依赖倒置原则，使用IOC模式，创建一个工人类创建与添加至场景操作的控制器类来进行这一操作，将创建好的工人对象直接注入场景类，实现各工人类与各场景类的解耦：

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| WorkerAddingController | addWorkerToScene(in name:String, in age:int, in sex:Sex, in salary:int, in scene:StaffArea): void | WorkerAddingController类为创建并将工人添加至场景这一功能的控制类，调用addWorkerToScene方法并传入工人信息以及所在部门来创建工人对象并添加至该部门 |

### 3.33.3类图



### 3.33.4参考资料

<https://en.wikipedia.org/wiki/Inversion_of_control>

## 3.34 Monostate单一状态模式

### 3.34.1设计模式简述

强制执行一种行为，例如在所有实例之间共享相同的状态。

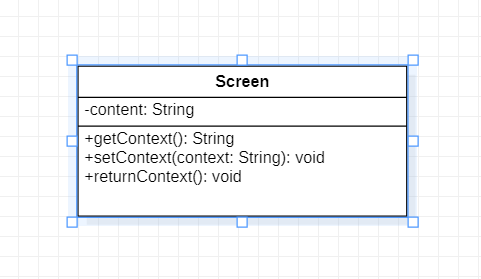
### 3.34.2 Screen 实现API

对于巡逻守卫队员Guard来说，每当一个人发现了危险情况并且在自己报警器的屏幕上输入危险信息alert之后，所有guard成员报警器上的屏幕理应强制替换为该预警信息alert，所以对于Screen类中的属性context进行static处理，这样就可以保证，当任何一个实例中context属性被改变时，其他的实例拥有相同的context属性。Screen类中使用setContext函数来达到发布预警信息的作用，使用getContext函数来获取和展示预警信息。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Screen | public void setContent(String content) { Screen.content = content; } | 提供发布预警信息的方法 |
| Screen | public String getContent() | 提供获取预警信息的方法 |

### 3.34.3类图



### 3.34.4参考资料

<https://developer.aliyun.com/article/227516>

<http://ddrv.cn/a/123655>

## 3.35 Multition多例模式

### 3.35.1设计模式简述

* 定义：作为对象的创建模式，多例模式中的多例类可以有多个实例，而且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。
* 主要优点
  + 需要频繁创建和销毁的对象时可以提高系统的性能。
  + 在某些场景下比单例模式更具有伸缩性和适应性（如一个公司不同部门各有经理）
* 主要缺点
  + 当多例模式的类创建的static对象太多时，系统会比较庞大，复杂

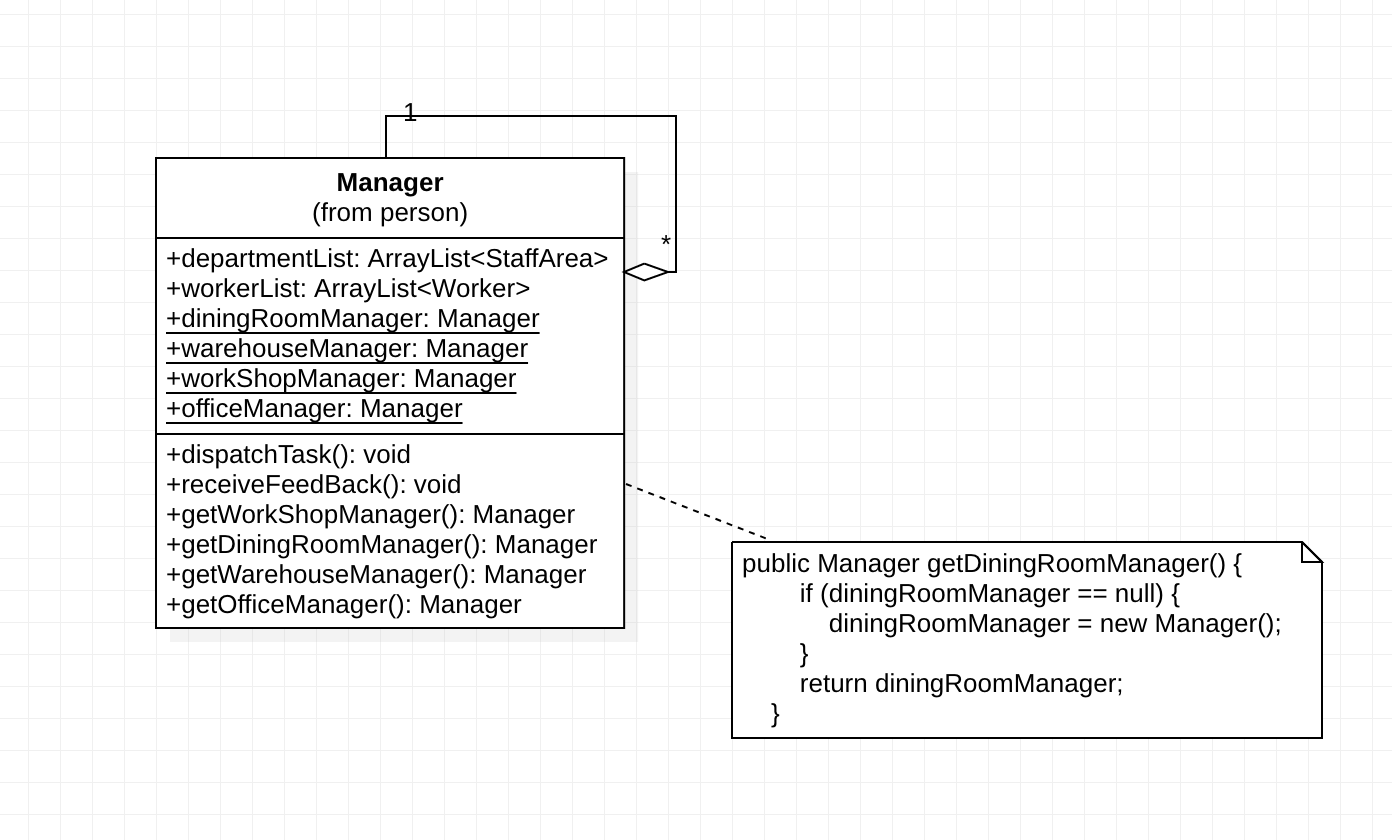
### 3.35.2 Product实现API

* 本项目中的实现
  + 经理（Manager）作为多例模式的主体类

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Manager | +getWorkShopManager(): Manager | 获取WorkShop的经理 |
| Manager | +getDiningRoomManager(): Manager | 获取DiningRoom的经理 |
| Manager | +getWarehouseManager(): Manager | 获取Warehouse的经理 |
| Manager | +getOfficeManager(): Manager | 获取Office的经理 |

### 3.35.3类图



### 3.35.4参考资料

<https://en.wikipedia.org/wiki/Multiton_pattern>

## 3.36 Null Object空对象模式

### 3.36.1设计模式简述

Null Object模式（空对象模式）定义了一种方法，用一种用户定义的对象来代替null，使得在进行某些操作时不需要检测是否为空，可以直接按照正常情况下的语句进行。

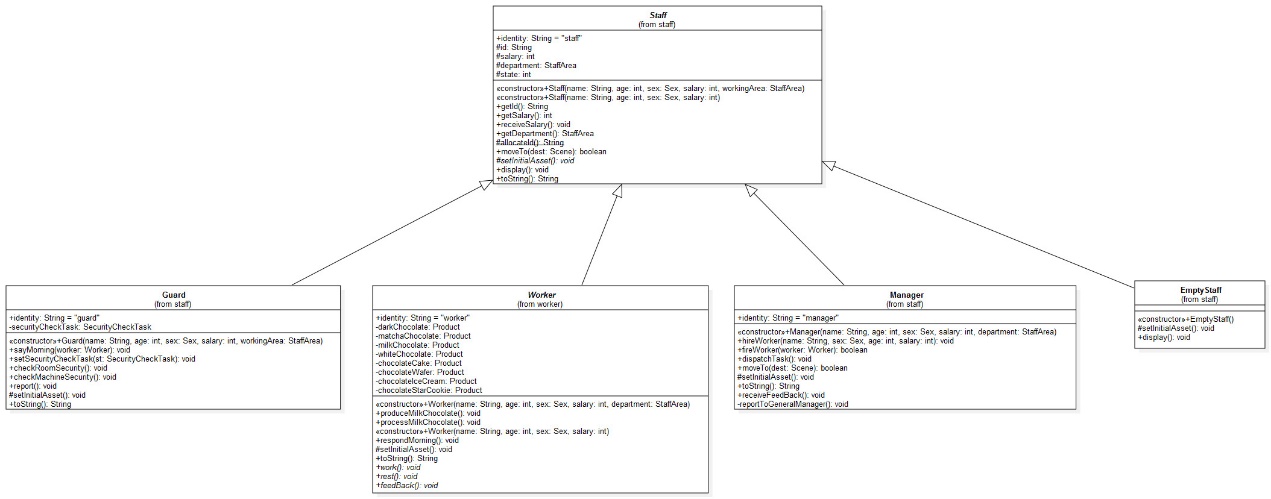
### 3.36.2 EmptyStaff实现API

在巧克力工厂中，每个职工都有ID、姓名、部门等信息，有时需要根据条件查询某个对象，当查询失败时使用Null Object模式返回一个空对象，这样在展示结果时可以无需判断结果是否为null直接调用display方法即可，在空对象的diaplay方法中会输出所求对象不存在的信息。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| EmptyStaff | display(): void | EmptyStaff为空职工对象类，其中display方法调用时  输出显示本对象代表查询没找到满足目标的对象。 |

### 3.36.3类图



### 3.36.4参考资料

<https://en.wikipedia.org/wiki/Null_object_pattern>

## 3.37 Value Object值对象模式

### 3.37.1设计模式简述

值对象模式提供遵循值语义而不是引用语义的对象。这意味着值对象的相等性并非基于身份，当两个值对象具有相同的值时，它们是相等的，不一定是相同的对象。

### 3.37.2 MilkChocolate类实现API

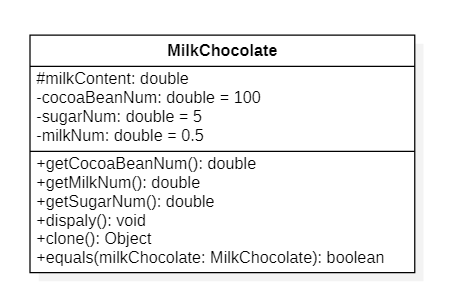
对于产品牛奶巧克力(MilkChocolate)而言，有相同原料及原料含量的牛奶巧克力可看作是同一产品，不同牛奶巧克力之间的区别仅仅是原料含量的不同，所以我们采用值对象(Value Object)模式，将具有相同原料及原料含量的牛奶巧克力看作是同一对象。

在Java中，因为Object.equals在核心库中定义，并且其他类库使用它来进行比较，所以重写equals函数即可实现值对象的比较。值对象应该是不可变的，假如要改变值的话可以通过创建新的对象来替代。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| MilkChocolate | equals(MilkChocolate milkChocolate): boolean | 说明：判断两牛奶巧克力对象值是否相等 参数：另一类型为牛奶巧克力的产品 返回值：是/否(Boolean) |

### 3.37.3类图



### 3.37.4参考资料

(部分网站需要vpn访问)：

<https://en.wikipedia.org/wiki/Value_object>

<https://dzone.com/articles/value-objects>

<https://medium.com/@hermesmonteiro1981/valueobject-pattern-when-to-use-identify-pattern-situation-e753292113c7>

## 3.38 Transfer Object传输对象模式

### 3.38.1设计模式简述

* 定义：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。
* 主要优点
  + 方便前端和数据库进行交互
  + 通过此传输者进行对数据库的读写，有助于保证数据库的安全性

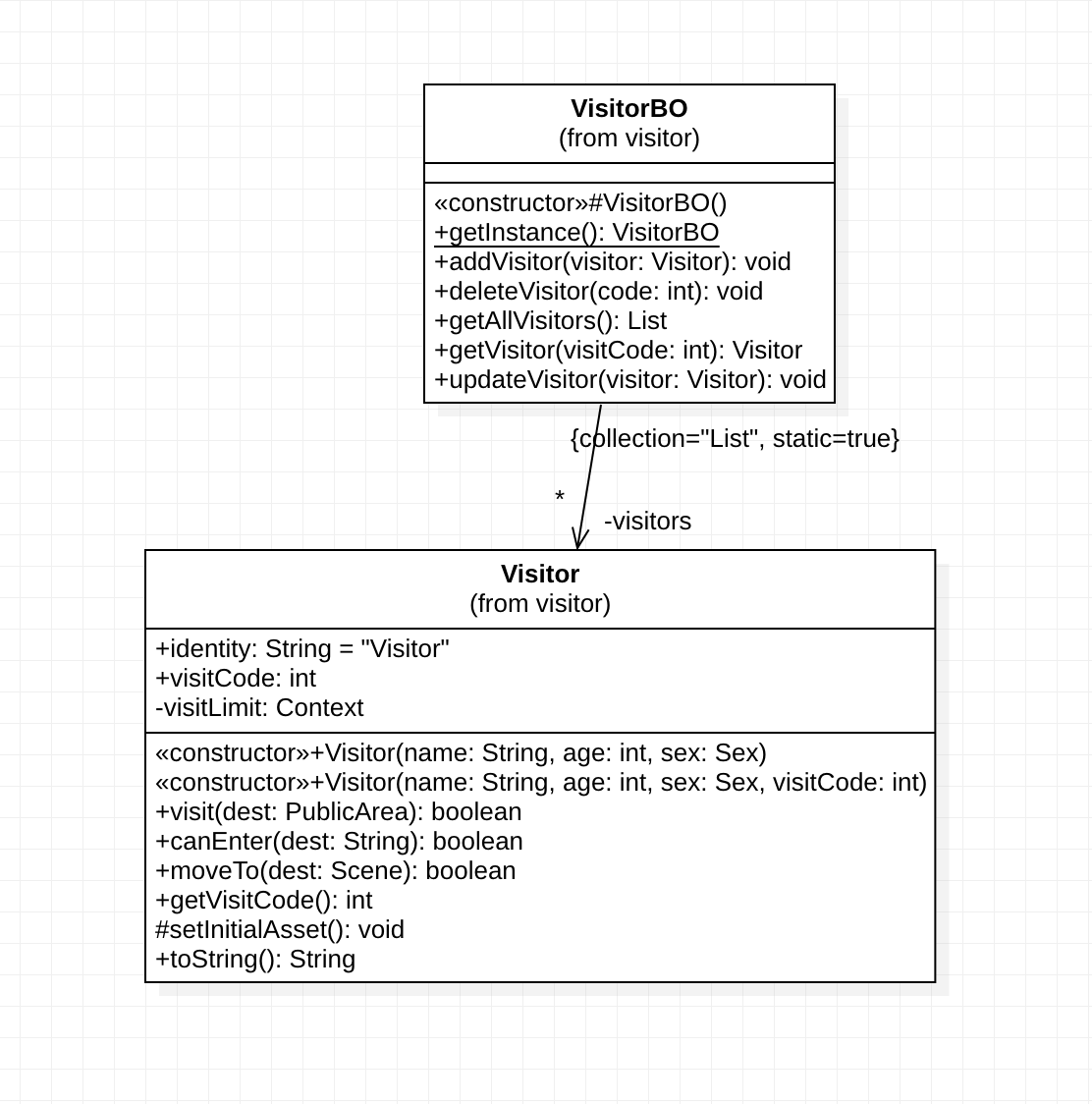
### 3.38.2 Product 实现API

* 本项目中的实现
  + 参观业务对象（VisitorBO） 为传输对象填充数据的业务服务。
  + 参观者（Visitor）作为被传输的对象，有设置/获取属性的方法。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| VisitorBO | +getInstance():VisitorBo | 获得单例的参观业务对象 |
| VisitorBO | +addVisitor(Visitor visitor):void | 向数据库中添加一个visitor |
| VisitorBO | +deleteVisitor(Visitor visitor):void | 删除指定的visitor |
| VisitorBO | +getAllVisitors(): List\ | 获得参观者列表 |
| VisitorBO | updateVisitor(Visitor visitor): void | 更新某个参观者的姓名 |
| Visitor | getName(): String | 获得参观者的姓名 |
| Visitor | getAge(): int | 获得参观者的年龄 |
| Visitor | getSex(): Sex | 获得参观者的性别 |
| Visitor | setName(): void | 获得参观者的姓名 |

### 3.38.3类图



### 3.38.4参考资料

<https://www.tutorialspoint.com/design_pattern/transfer_object_pattern.htm>

<https://www.runoob.com/design-pattern/transfer-object-pattern.html>