**设计模式项目文档**

**——查理的巧克力工厂**

1850250 赵浠明

1751044 李申羽

1753307 蔡方俊妍

1850013 颜泽皓

1851202 雷泓

1851343 季潇熠

1851348 李添一

1851400 郑雨晴

1852137 张艺腾

1853829 杨雨辰

2020年11月28日

目录

[1.项目简介 6](#_Toc57488715)

[2.Design Pattern汇总表 6](#_Toc57488716)

[3.设计模式详述 7](#_Toc57488717)

[3.1 Abstract Factory抽象工厂模式 7](#_Toc57488718)

[3.1.1设计模式简述 7](#_Toc57488719)

[3.1.2 DarkWorkshop和WhiteWorkshop实现API 8](#_Toc57488720)

[3.1.3类图 9](#_Toc57488721)

[3.2 Adapter适配器模式 9](#_Toc57488722)

[3.2.1设计模式简述 9](#_Toc57488723)

[3.2.2 UAV实现API 9](#_Toc57488724)

[3.2.3类图 10](#_Toc57488725)

[3.3 Bridge桥接模式 10](#_Toc57488726)

[3.3.1设计模式简述 10](#_Toc57488727)

[3.3.2 Bridge实现API 11](#_Toc57488728)

[3.3.3类图 11](#_Toc57488729)

[3.4 Builder建造者模式 12](#_Toc57488730)

[3.4.1设计模式简述 12](#_Toc57488731)

[3.4.2 MilkChocController实现API 12](#_Toc57488732)

[3.4.3类图 13](#_Toc57488733)

[3.5 Chain of Responsibility责任链模式 13](#_Toc57488734)

[3.5.1设计模式简述 13](#_Toc57488735)

[3.5.2 FixWorker实现API 14](#_Toc57488736)

[3.5.3类图 14](#_Toc57488737)

[3.6 Command命令模式 15](#_Toc57488738)

[3.6.1设计模式简述 15](#_Toc57488739)

[3.6.2 Product实现API 15](#_Toc57488740)

[3.6.3类图 16](#_Toc57488741)

[3.6.4时序图 16](#_Toc57488742)

[3.7 Composite组合模式 17](#_Toc57488743)

[3.7.1设计模式简述 17](#_Toc57488744)

[3.7.2 QualityTestSystem实现API 17](#_Toc57488745)

[3.7.3类图 18](#_Toc57488746)

[3.8 Decorator装饰模式 18](#_Toc57488747)

[3.8.1设计模式简述 18](#_Toc57488748)

[3.8.2 ResidenceDecorator实现API 18](#_Toc57488749)

[3.8.3类图 20](#_Toc57488750)

[3.9 Facade外观模式 20](#_Toc57488751)

[3.9.1设计模式简述 20](#_Toc57488752)

[3.9.2 SecurityCheckTask实现API 20](#_Toc57488753)

[3.9.3类图 21](#_Toc57488754)

[3.10 Factory Method工厂模式 21](#_Toc57488755)

[3.10.1设计模式简述 21](#_Toc57488756)

[3.10.2 ChocolatesMachineAssemblyLine实现API 22](#_Toc57488757)

[3.10.3类图 22](#_Toc57488758)

[3.11 Flyweight享元模式 23](#_Toc57488759)

[3.11.1设计模式简述 23](#_Toc57488760)

[3.11.2 ChocolateMix实现API 23](#_Toc57488761)

[3.11.3类图 24](#_Toc57488762)

[3.12 Interpreter解释器模式 24](#_Toc57488763)

[3.12.1设计模式简述 24](#_Toc57488764)

[3.12.2 Expression实现API 25](#_Toc57488765)

[3.12.3类图 26](#_Toc57488766)

[3.13 Iterator迭代器模式 26](#_Toc57488767)

[3.13.1设计模式简述 26](#_Toc57488768)

[3.13.2 WorkerListIterator实现API 26](#_Toc57488769)

[3.13.3类图 27](#_Toc57488770)

[3.14 Mediator中介者模式 27](#_Toc57488771)

[3.14.1设计模式简述 27](#_Toc57488772)

[3.14.2 Product实现API 28](#_Toc57488773)

[3.14.3类图 29](#_Toc57488774)

[3.14.4时序图 29](#_Toc57488775)

[3.15 Memento备忘录模式 30](#_Toc57488776)

[3.15.1设计模式简述 30](#_Toc57488777)

[3.15.2 ProductMemento实现API 30](#_Toc57488778)

[3.15.3类图 31](#_Toc57488779)

[3.16 Observer观察者模式 31](#_Toc57488780)

[3.16.1设计模式简述 31](#_Toc57488781)

[3.16.2 Payroll实现API 31](#_Toc57488782)

[3.16.3类图 32](#_Toc57488783)

[3.17 Prototype原型模式 32](#_Toc57488784)

[3.17.1设计模式简述 32](#_Toc57488785)

[3.17.2 RawMaterial实现API 32](#_Toc57488786)

[3.17.3类图 33](#_Toc57488787)

[3.18 Proxy代理模式 33](#_Toc57488788)

[3.18.1设计模式简述 33](#_Toc57488789)

[3.18.2 PortableBattery实现API 33](#_Toc57488790)

[3.18.3类图 34](#_Toc57488791)

[3.19 Singleton单例模式 34](#_Toc57488792)

[3.19.1设计模式简述 34](#_Toc57488793)

[3.19.2 GeneralManager实现API 34](#_Toc57488794)

[3.19.3类图 35](#_Toc57488795)

[3.20 State状态模式 35](#_Toc57488796)

[3.20.1设计模式简述 35](#_Toc57488797)

[3.20.2 ProductState实现API 36](#_Toc57488798)

[3.20.3类图 36](#_Toc57488799)

[3.21 Strategy策略模式 37](#_Toc57488800)

[3.21.1设计模式简述 37](#_Toc57488801)

[3.21.2 WrapStrategy实现API 37](#_Toc57488802)

[3.21.3类图 38](#_Toc57488803)

[3.22 Template Method模板方法模式 38](#_Toc57488804)

[3.22.1设计模式简述 38](#_Toc57488805)

[3.22.2 TransportMachine实现API 38](#_Toc57488806)

[3.22.3类图 39](#_Toc57488807)

[3.23 Visitor访问者模式 40](#_Toc57488808)

[3.23.1设计模式简述 40](#_Toc57488809)

[3.23.2 UtilityWorker实现API 40](#_Toc57488810)

[3.23.3类图 41](#_Toc57488811)

[3.24 Business Delegate业务代表模式 41](#_Toc57488812)

[3.24.1设计模式简述 41](#_Toc57488813)

[3.24.2 WarehouseDelegate实现API 42](#_Toc57488814)

[3.24.3类图 43](#_Toc57488815)

[3.24.4参考资料 43](#_Toc57488816)

[3.25 Callback回调模式 43](#_Toc57488817)

[3.25.1设计模式简述 43](#_Toc57488818)

[3.25.2 SceneSelector实现API 43](#_Toc57488819)

[3.25.3类图 44](#_Toc57488820)

[3.25.4参考资料 44](#_Toc57488821)

[3.26 Converter转换器模式 45](#_Toc57488822)

[3.26.1设计模式简述 45](#_Toc57488823)

[3.26.2 VisitorConverter实现API 45](#_Toc57488824)

[3.26.3类图 46](#_Toc57488825)

[3.26.4参考资料 46](#_Toc57488826)

[3.27 COW写时复制模式 46](#_Toc57488827)

[3.27.1设计模式简述 46](#_Toc57488828)

[3.27.2 HandmadeTool实现API 47](#_Toc57488829)

[3.27.3类图 48](#_Toc57488830)

[3.27.4参考资料 49](#_Toc57488831)

[3.28 Data Access Object数据访问对象模式 49](#_Toc57488832)

[3.28.1设计模式简述 49](#_Toc57488833)

[3.28.2 StaffDaoImpl类实现API 49](#_Toc57488834)

[3.28.3类图 50](#_Toc57488835)

[3.28.4参考资料 50](#_Toc57488836)

[3.29 Dirty Flag脏标记模式 51](#_Toc57488837)

[3.29.1设计模式简述 51](#_Toc57488838)

[3.29.2 CharlieFactory/TimeSystem实现API 51](#_Toc57488839)

[3.29.3类图 52](#_Toc57488840)

[3.29.4参考资料 52](#_Toc57488841)

[3.30 Extension Objects扩展对象模式 52](#_Toc57488842)

[3.30.1设计模式简述 52](#_Toc57488843)

[3.30.2 AugProductMachine实现API 52](#_Toc57488844)

[3.30.3类图 53](#_Toc57488845)

[3.30.4参考资料 53](#_Toc57488846)

[3.31 Filter拦截过滤器模式 53](#_Toc57488847)

[3.31.1设计模式简述 53](#_Toc57488848)

[3.31.2 FilterManager实现API 54](#_Toc57488849)

[3.31.3类图 54](#_Toc57488850)

[3.31.4参考资料 55](#_Toc57488851)

[3.32 Immutable不变模式 55](#_Toc57488852)

[3.32.1设计模式简述 55](#_Toc57488853)

[3.32.2 AbstractReceipt实现API 55](#_Toc57488854)

[3.32.3类图 56](#_Toc57488855)

[3.32.4参考资料 56](#_Toc57488856)

[3.33 IOC控制反转模式 56](#_Toc57488857)

[3.33.1设计模式简述 56](#_Toc57488858)

[3.33.2 WorkerAddingController实现API 57](#_Toc57488859)

[3.33.3类图 57](#_Toc57488860)

[3.33.4参考资料 57](#_Toc57488861)

[3.34 Monostate单一状态模式 58](#_Toc57488862)

[3.34.1设计模式简述 58](#_Toc57488863)

[3.34.2 Screen 实现API 58](#_Toc57488864)

[3.34.3类图 58](#_Toc57488865)

[3.34.4参考资料 58](#_Toc57488866)

[3.35 Multition多例模式 59](#_Toc57488867)

[3.35.1设计模式简述 59](#_Toc57488868)

[3.35.2 Product实现API 59](#_Toc57488869)

[3.35.3类图 60](#_Toc57488870)

[3.35.4参考资料 60](#_Toc57488871)

[3.36 Null Object空对象模式 60](#_Toc57488872)

[3.36.1设计模式简述 60](#_Toc57488873)

[3.36.2 EmptyStaff实现API 60](#_Toc57488874)

[3.36.3类图 61](#_Toc57488875)

[3.36.4参考资料 61](#_Toc57488876)

[3.37 Value Object值对象模式 61](#_Toc57488877)

[3.37.1设计模式简述 61](#_Toc57488878)

[3.37.2 MilkChocolate类实现API 61](#_Toc57488879)

[3.37.3类图 62](#_Toc57488880)

[3.37.4参考资料 62](#_Toc57488881)

[3.38 Transfer Object传输对象模式 62](#_Toc57488882)

[3.38.1设计模式简述 62](#_Toc57488883)

[3.38.2 Product 实现API 62](#_Toc57488884)

[3.38.3类图 64](#_Toc57488885)

[3.38.4参考资料 64](#_Toc57488886)

# 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。

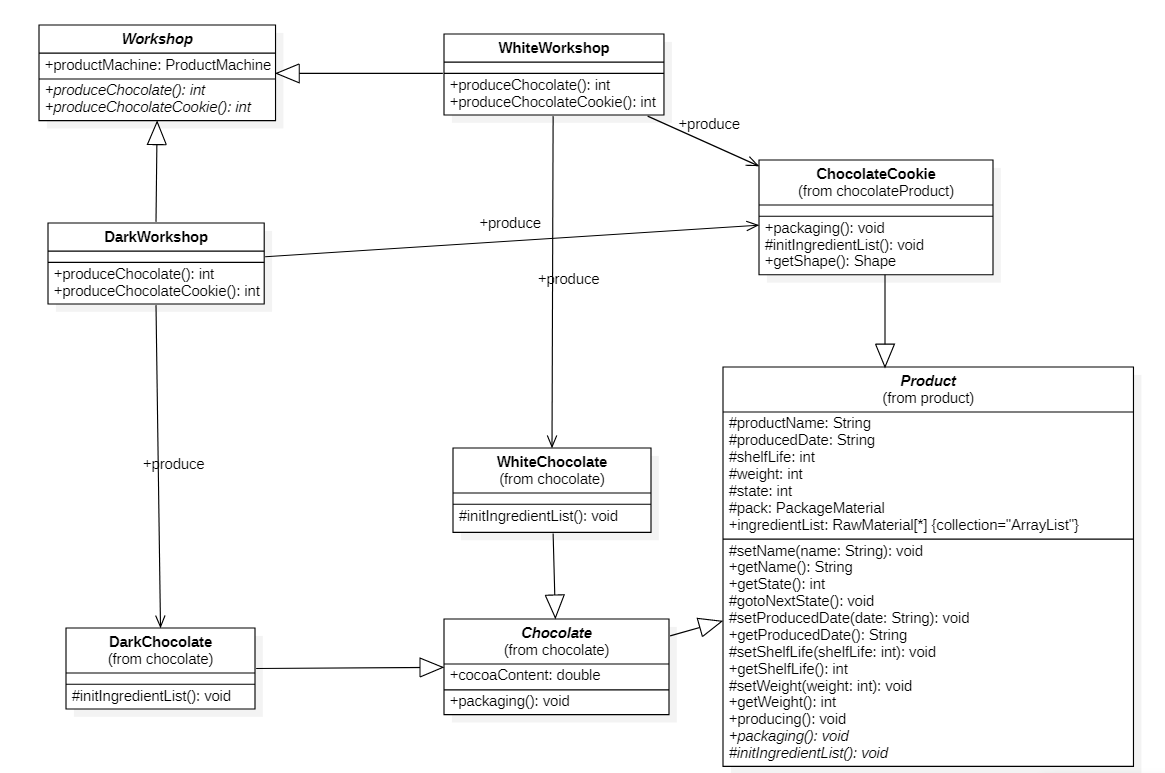
在抽象工厂模式中，通过继承抽象车间类和抽象产品类，可以实现在一个车间（Workshop）里生产一系列同族产品。黑巧克力车间（DarkWorkshop）生产黑巧克力和有黑巧克力夹心的小猫形状曲奇，白巧克力车间（WhiteWorkshop）生产白巧克力和有白巧克力夹心的小狗形状曲奇。

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



## 3.2 Adapter适配器模式

### 3.2.1设计模式简述

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

### 3.2.2 UAV实现API

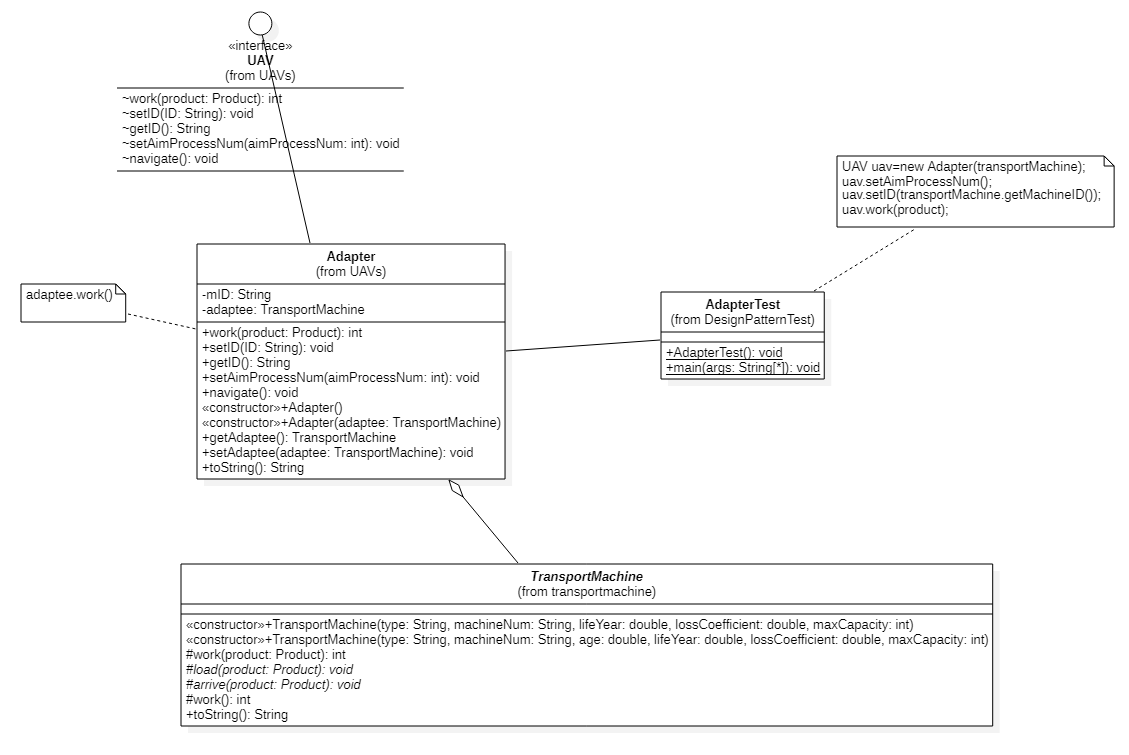
UAV全称为unmanned aerial vehicle，即无人驾驶飞机，它负责在无人操控的情况下将产品空运到对应的生产线或目的地上。因此，其具有运输机器具有的功能，如工作work(),修理fix()。但UAV工作过程又与普通的TransMachine不同，具体表现在UAV需要自动导航navigate()，它和TransMachine是有区别的。若重新开发work功能又将产生新的成本（假设work开发成本很高），且如果TransMachine后续进行维护，UAV无法与其保持一致。

此时选择使用适配器模式，定义一个适配器类来实现当前无人机的业务接口，同时又继承现有TransMachine类已有功能，这样减少了开发成本，还降低了UAV和TransMachine之间的耦合性。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Adapter | work():int | Adapter类里的work函数是对里函数的实现，在函数中，调用适配者（即TransMachine对象）的work函数。 |
| Adapter | getID():String | 获取Adapter适配的UAV的MachineID |
| Adapter | setID(String name):void | 设置Adapter适配的Watchdog的MachineID |
| Adapter | navigate():void | 适配Adapter独有的自动导航功能 |

### 3.2.3类图



## 3.3 Bridge桥接模式

### 3.3.1设计模式简述

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

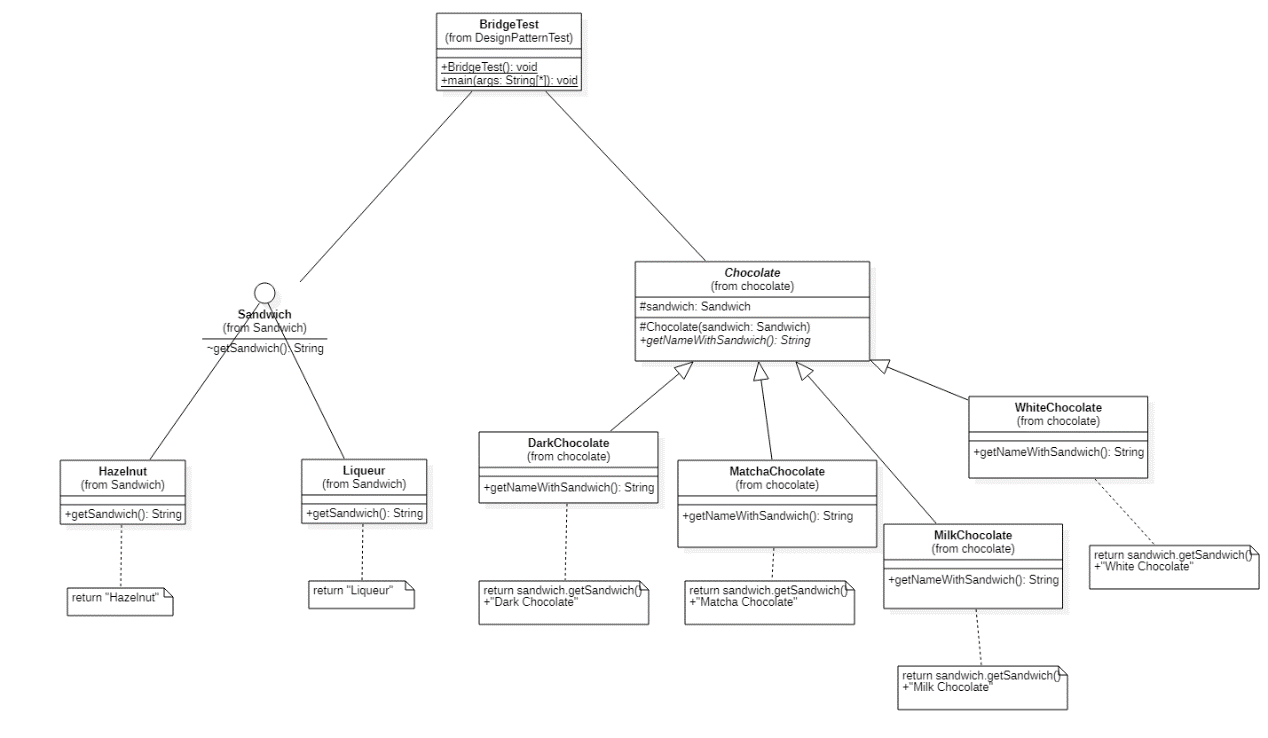
### 3.3.2 Bridge实现API

巧克力（Chocolate）分为黑巧克力、白巧克力、牛奶巧克力等等不同可可豆含量的巧克力类型，且有可能有不同夹心。当把巧克力类型和夹心互相组合的时候，将每个组合都开发成一个新的子类显然开发量很大，而且很难进行后续的开发。创建一个夹心（Sandwich）类，具体有Pure（无夹心）、Liqueur（酒心）、Hazelnut（榛仁），用不同的子类与Chocolate的子类的组合关系来代替单纯开发Chocolate的子类，可以降低两种属性的耦合性，后续也可以开发其他类型和其他夹心。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Sandwich | getSandwich():String | 继承自Sandwich类，获取当前夹心类型的字符串 |
| Chocolate | Chocolate(Sandwich sandwich) | 用夹心类型作为参数来构造一个巧克力对象，实现夹心与巧克力本身的解耦 |
| Chocolate | getNameWithSandwich():String | 返回Chocolate的类型和夹心 |

### 3.3.3类图



## 3.4 Builder建造者模式

### 3.4.1设计模式简述

建造者（Builder）模式的定义是指将一个复杂对象的构造与它的表示分离，使同样的构建过程可以创建不同的表示，这样的设计模式被称为建造者模式。它是将一个复杂的对象分解为多个简单的对象，然后一步一步构建而成。它将变与不变相分离，即产品的组成部分是不变的，但每一部分是可以灵活选择的。

该模式的主要优点如下：

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

其缺点如下：

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

### 3.4.2 MilkChocController实现API

生产牛奶巧克力时用到的原料有可可豆、糖和牛奶，在生产时加入不同数量的原料可生产出不同口味的牛奶巧克力。这就需要用到建造者（Builder）模式，分步骤加入组成原料，再进行生产。

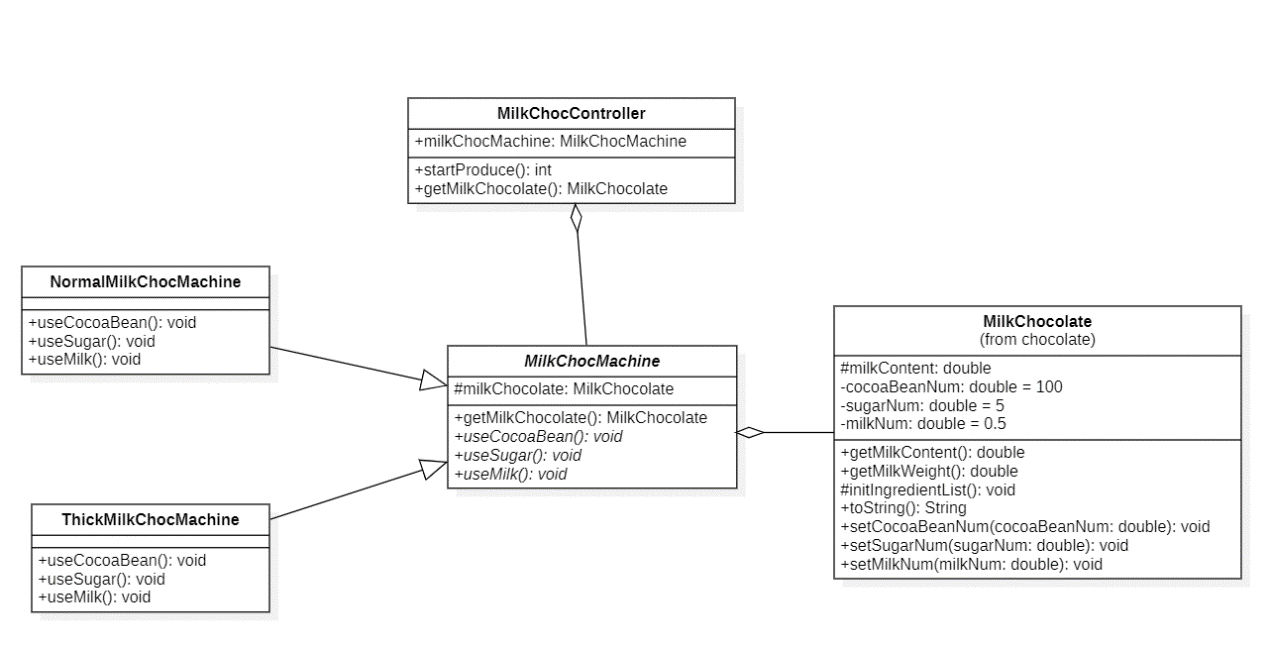
MilkChocMachine是继承自BasicProductMachine的抽象类，定义了生产牛奶巧克力的抽象方法，它的子类NormalMilkChocMachine和ThickMilkChocMachine决定了具体的处理过程。NormalMilkChocMachine生产中使用的可可豆数量更多，最后产出较苦的牛奶巧克力。ThickMilkChocMachine生产中使用的糖数量更多，最后产出较甜的牛奶巧克力。

MilkChocController是生产机器的控制器，给外部提供API，通过控制器来启动生产、加入原料、获得产品。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| MilkChocController | startProduce(): int | 说明：生产不同口味的牛奶巧克力 参数：无 返回值：完成一次生产的产品产出量 |
| MilkChocController | getMilkChocolate(): MilkChocolate | 说明：获得不同口味的牛奶巧克力 参数：无 返回值：生产出的牛奶巧克力 |

### 3.4.3类图



## 3.5 Chain of Responsibility责任链模式

### 3.5.1设计模式简述

* 定义：为了避免请求发送者与多个请求处理者耦合在一起，于是将所有请求的处理者通过前一对象记住其下一个对象的引用而连成一条链；当有请求发生时，可将请求沿着这条链传递，直到有对象处理它为止。
* 主要优点
  + 降低了对象之间的耦合度。该模式使得一个对象无须知道到底是哪一个对象处理其请求以及链的结构，发送者和接收者也无须拥有对方的明确信息。
  + 增强了系统的可扩展性。可以根据需要增加新的请求处理类，满足开闭原则。
  + 增强了给对象指派职责的灵活性。当工作流程发生变化，可以动态地改变链内的成员或者调动它们的次序，也可动态地新增或者删除责任。
  + 责任链简化了对象之间的连接。每个对象只需保持一个指向其后继者的引用，不需保持其他所有处理者的引用，这避免了使用众多的 if 或者 if···else 语句。
  + 责任分担。每个类只需要处理自己该处理的工作，不该处理的传递给下一个对象完成，明确各类的责任范围，符合类的单一职责原则。
* 主要缺点
  + 不能保证每个请求一定被处理。由于一个请求没有明确的接收者，所以不能保证它一定会被处理，该请求可能一直传到链的末端都得不到处理。
  + 对比较长的职责链，请求的处理可能涉及多个处理对象，系统性能将受到一定影响。
  + 职责链建立的合理性要靠客户端来保证，增加了客户端的复杂性，可能会由于职责链的错误设置而导致系统出错，如可能会造成循环调用。

### 3.5.2 FixWorker实现API

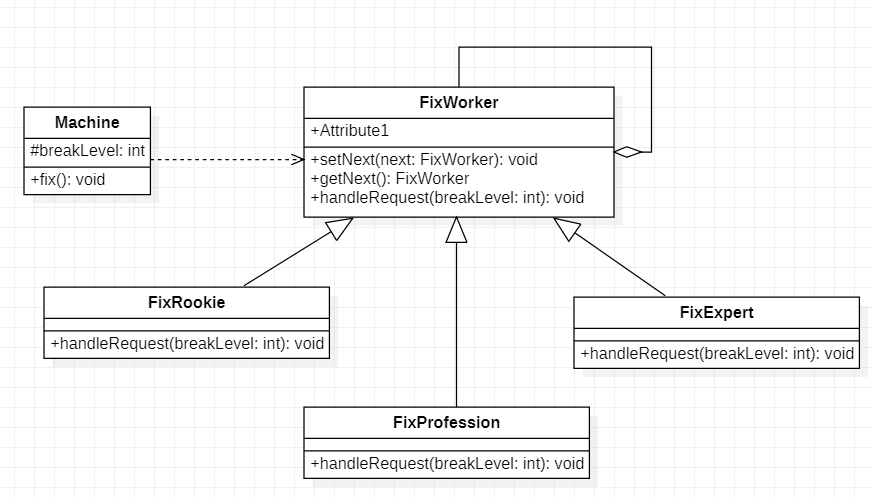
在巧克力工厂中，具有很多不同的机器，当员工机器发现机器出现故障，可以通过调用Machine的notifyBroke()方法进行通知，该方法内会调用fix()方法对机器进行维修。针对不同等级的故障，需要不同的修理工人来处理，在这里我们使用了Chain of Responsibility来实现。

* Chain of Responsibility 在项目中的实现
  + 修理工（FixWorker）是抽象处理者。
    - 修理工的各种子类（FixRookie、FixProfession、FixExpert）是具体处理者。
  + 机器（Machine）是客户类。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| FixWorker | +setNext(next:FixWorker)：void | 创建后继连接 |
| FixWorker | +getNext()：FixWorker | 获取后继连接 |
| FixWorker | +handleRequest(breakLevel:int): boolean | 具体的处理请求过程，机器可以修理则修理，不能修理传递给后继连接，没有后续连接进行提示 参数：breakLevel，机器故障等级 |
| Machine | +fix(): void | 创建修理工处理链，并向链头的具体处理者对象提交请求 |

### 3.5.3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。

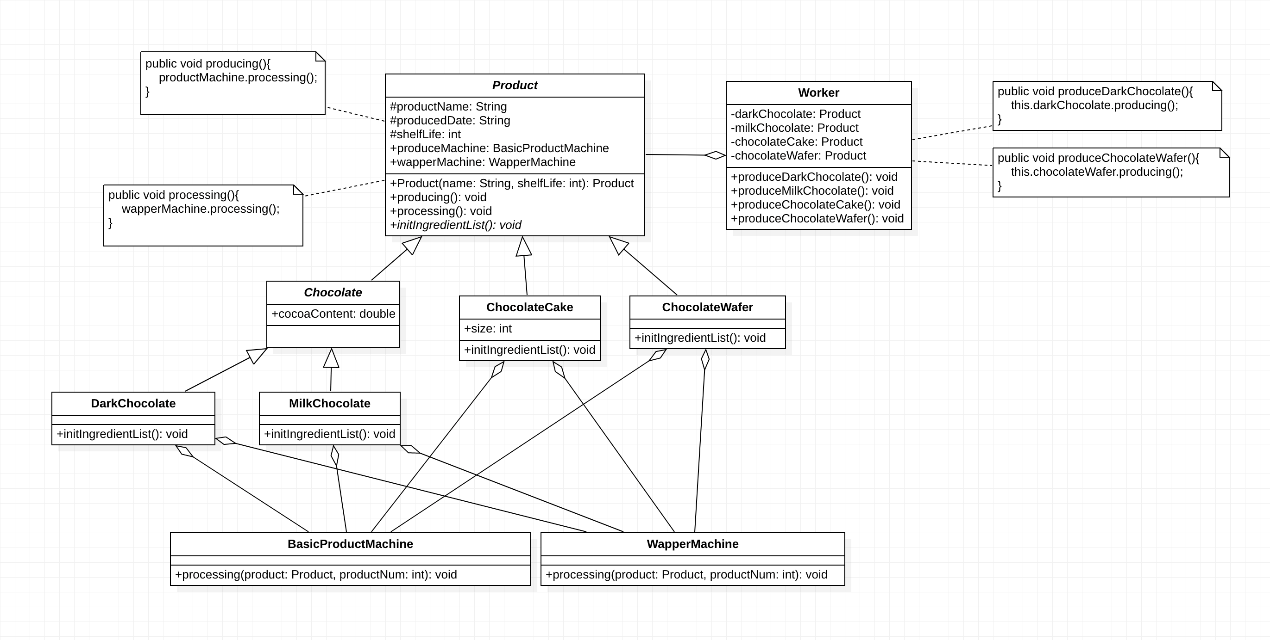
### 3.6.2 Product实现API

* 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**

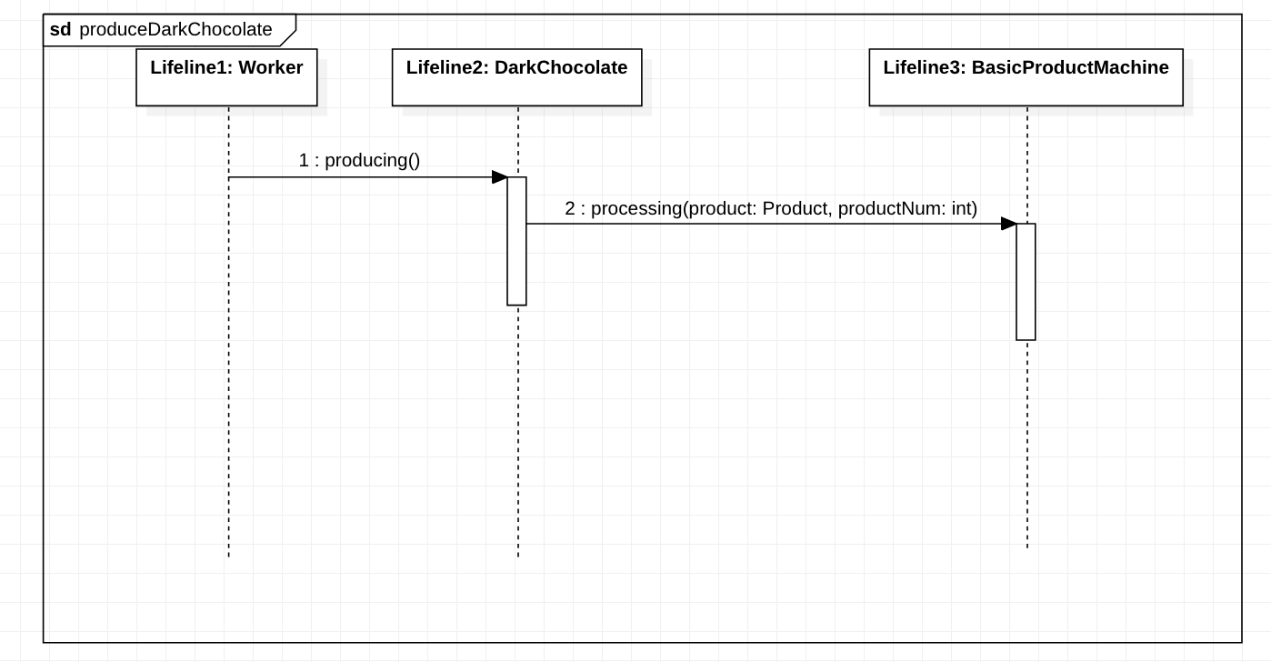
|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Worker | +produce*Something*(): void (something可以换成任意产品类名，如DarkChocolate) | 下达生产某产品的命令，通过调用具体的命令类的producing()方法向接收方机器发送生产该产品的指令。 |
| Product | +producing(): void | 生产该产品的具体命令，调用相应接收方机器的processing()方法，传递命令，使之开始执行指令生产该产品。具体是哪个机器作为接收者需要在子类中确定。如下面的类图所示。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



### 3.6.4时序图

以黑巧克力的生产为例，其时序图如下



## 3.7 Composite组合模式

### 3.7.1设计模式简述

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

### 3.7.2 QualityTestSystem实现API

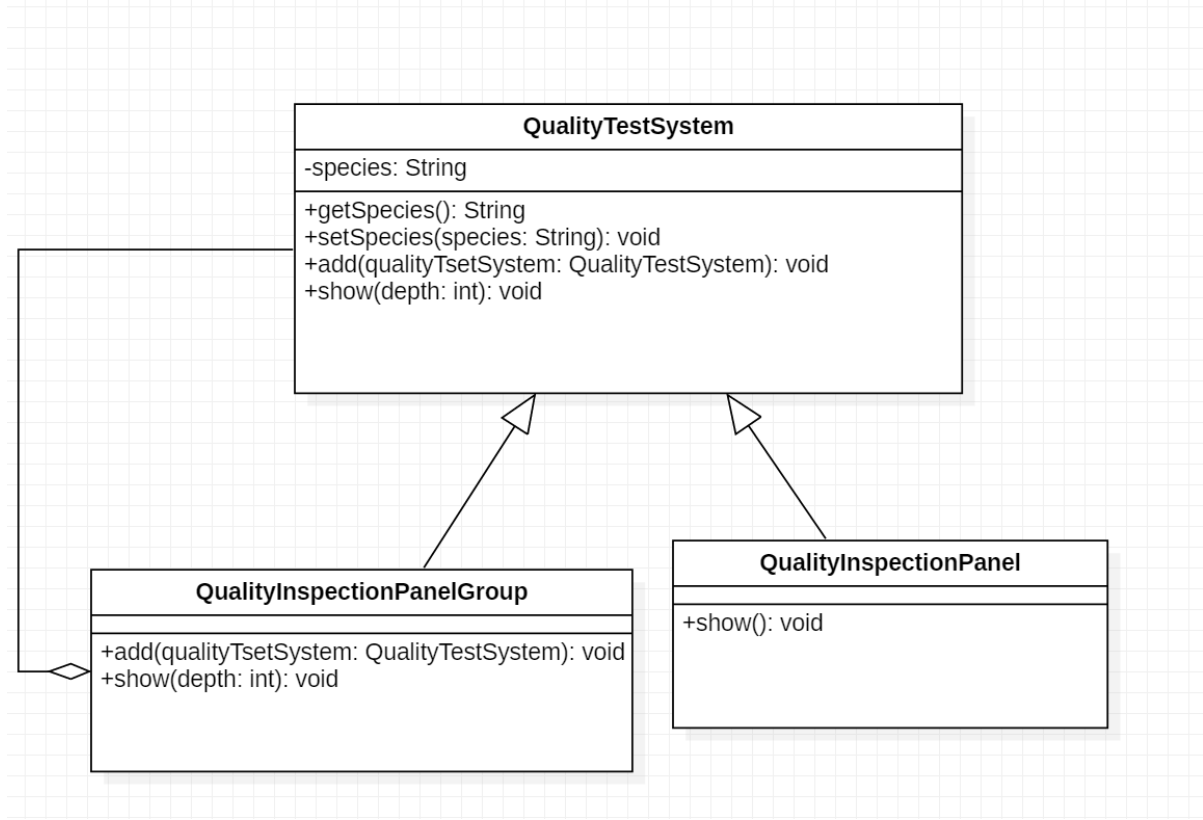
对于巧克力原材料可可豆而言，质量的检查往往是有一个质检系统完成的，而质检系统既可以包括一组筛选口大小不一的质检盘集合而成的质检盘组，也可以仅仅是一个单个的质检盘。将质检系统作为一个抽象父类QualityTestSystem，将质检盘组和单个质检盘分别作为相当于整体与部分的子类QualityInspectionPanelGroup和QualityInspectionPanel。这样就能把质检盘组和单个质检盘同等的对待进行操作。

对于一个质检系统中，可以包含数个质检盘组和数个单体质检盘，而任意一个质检盘组中又能够包含数个质检盘组和单个质检盘。作为整体的子类的质检盘组QualityInspectionPanelGroup具有add函数能添加内容，同时QualityInspectionPanelGroup和QualityInspectionPanel都有show函数来查看其内部的元素名称。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 方法 |
| QualityTestSystem | add(in qualityTestSystem: QualityTestSystem): void | 提供往作为整体集合的子类QualityInspectionPanelGroup中添加元QualityTestSystem的方法 |
| QualityTestSystem | show(in depth:int): void | 显示以当前元素为根结点的全部叶子元素。depth默认值为0，作用是为了体现输出的规整性。 |

### 3.7.3类图

****

## 3.8 Decorator装饰模式

### 3.8.1设计模式简述

装饰模式又名包装模式，它以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案。运用该设计模式可以动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

### 3.8.2 ResidenceDecorator实现API

Residence 住宅抽象类有函数 getDescription() 和 getCost()，它们分别返回住宅的简介和造价。

在最初的设计中，我们默认每种住宅都有固定的价格：豪宅 50000 元 / 栋，接待中心 30000 元 / 栋，员工宿舍 20000 元 / 栋。但在后续开发过程中，我们想给住宅添加一些附加功能：消防栓、监控器和中央空调，而增加这些附加功能会相应地增加住宅的造价。如何设计出一个造价动态改变的住宅类呢？这是个问题。

一种传统的做法是创建若干新的子类，例如带消防栓的住宅、带监控器的接待中心、同时带中央空调和红外报警器的豪宅等等。但这样做的缺点非常明显。假设我们有三种住宅（豪宅、接待中心和员工宿舍）以及三种附加功能（红外报警器、监控器和中央空调），因住宅的种类和附加功能可以任意排列组合，我们需要创建大量的子类，共计 3 \* P(3) = 18 种，这是十分繁琐的。

还有一种传统的做法是给住宅类新增3个布尔类的成员变量，用于表示是否安装消防栓、是否安装监控器和是否安装中央空调。这种做法仍有缺陷：如果以后又有新的附加功能产生，例如太阳能电池板，那我们修改住宅类的设计，给它添加新的布尔类成员变量，用于表示是否安装太阳能电池板。如果我们以后不再支持某个附加功能，我们还得删掉住宅类的相关布尔变量。这违反了开放封闭原则。

为了解决上述缺陷，我们采用装饰模式。首先设计一个附加功能抽象类（ResidenceDecorator)，其继承自住宅类（Residence）。再设计三个附加功能实体类 （Monitor,FireHydrant 和 CentralAirCondition），它们继承自ResidenceDecorator。这样附加功能类和三种住宅实体类都同时继承自Residence类。每一个附加功能类内都有一个指向 Residence 类的指针，并且它们也同样具有 getDescription() 和 getCost() 函数。每当我们想要给Residence对象新增某个附加功能时，就创建这个附加功能类并让其成员指针指向Residence对象。这种添加附加功能的操作被称作“装饰”，而附加功能被称作装饰器decorator，原本的三种住宅被称作decorated。

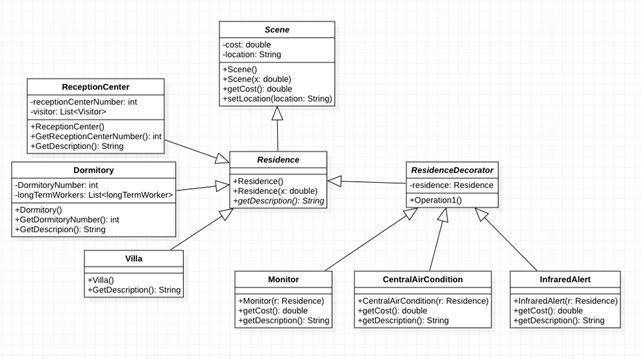
一个decorated在经过各种decorator的装饰后，它们会呈链表状。链表的尾节点为decorated，而其余节点都为decorator。前一个节点的 getDescription() 函数会打印出自己的描述并调用后一个节点的 getDescription() 函数，前一个节点的 getCost() 函数会调用后一个节点的 getCost() 函数并将其返回值与自己的价钱相加。故每当我们想知道被装饰的住宅的简介和总造价时，可以迭代地调用这个链表中每个节点地 getDescription() 和 getCost()。

运用装饰模式，我们的程序变得易于扩展，且满足开放封闭原则。每当我们需要给系统新增一种附加功能时，只需要新增一个附加功能实体类，而无需改变住宅实体类本身。

**API 描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Residence | Dormitory(), ReceptionCenter(), Villa() | 三个住宅实体类的构造函数。 |
| ResidenceDecorator | Monitor(Residence r), FireHydrant(Residence r), CentralAirCondition(Residence r) | 三个附加功能实体类的构造函数，在调用过程中会将被其装饰的 Residence 对象传给他的成员指针 residence。 |
| Scene | double getCost() | 返回住宅的总造价（它可能被装饰也可能未被装饰） |
| Residence | String getDescription() | 返回住宅的描述 |

### 3.8.3类图



## 3.9 Facade外观模式

### 3.9.1设计模式简述

通过对外观的包装，使客户端只能看到外观对象，而不会看到具体的细节对象，这样降低了应用程序的复杂度，并且提高了程序的可维护性。换言之，在外观模式中，子系统的外部调用者必须通过一个统一的 Facade 对象才能和子系统内部通信。

### 3.9.2 SecurityCheckTask实现API

在巧克力工厂的员工当中有一种员工类型为guard（保安/警卫），警卫每天都要执行的工作是对公共场地和机器设备进行例行的安全检查，最后记录检查情况并汇报给主管。为了实现以上功能，一种比较传统的做法是在警卫的类里设计多个工作函数以实现以上功能，但是假设工作函数非常复杂，且不希望警卫与机器设备类或场地类的耦合度增加，我们可以封装警卫每日例行巡检的工作细节，简化工作函数设计，降低警卫类与两种实体类的耦合度。

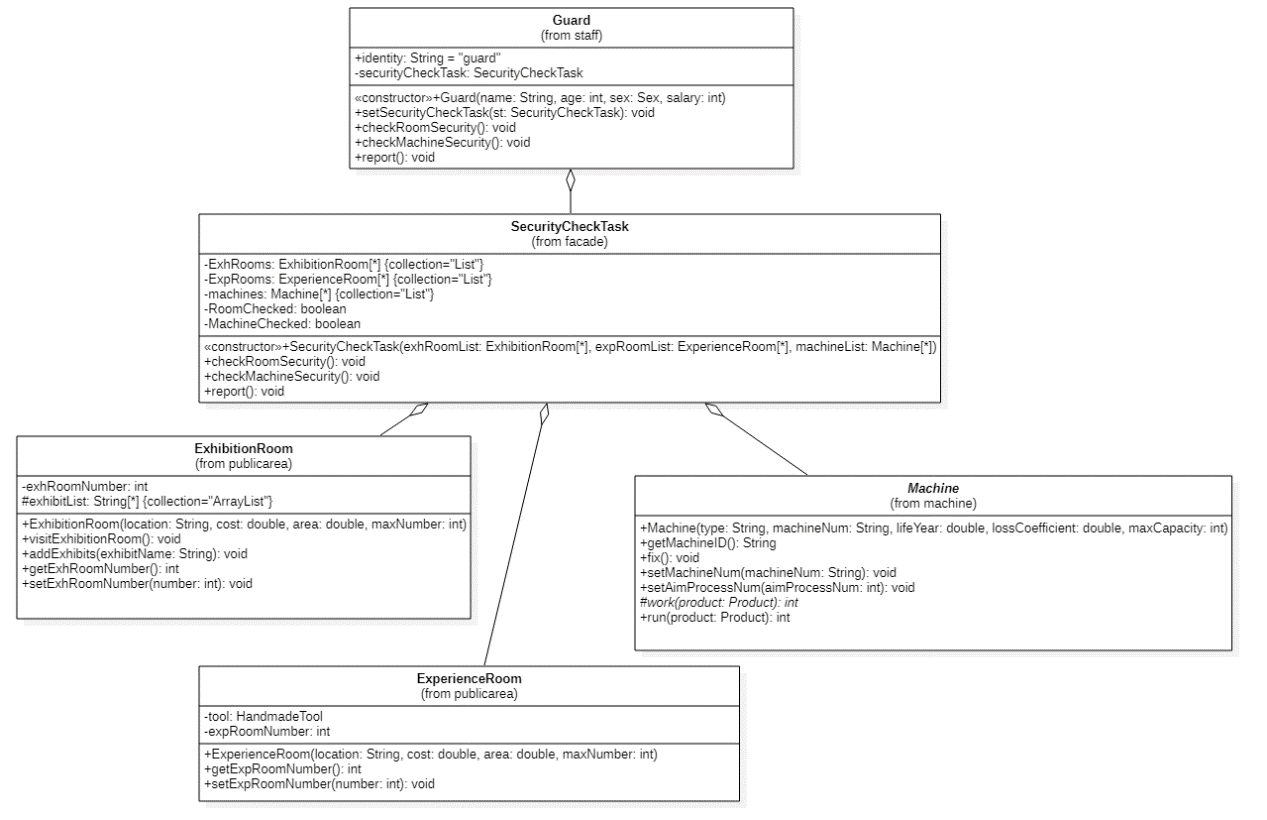
具体实现方法为：新增SecurityCheckTask类，即安检任务类。这个类将存储需要检查的展览厅和实践区两个列表、一个机器设备列表，以及两个bool变量分别用于判断当日检查工作是否完成。它将警卫巡检的工作封装为三个函数，分别为checkRoomSecurity()，checkMachineSecurity()，report()，让这些函数去执行庞杂的工作，例如checkRoomSecurity()要依次遍历每一个展览厅，并记录展厅中每一个展品的情况；checkMachineSecurity()则需要具体检查机器是否存在损坏情况，若发现机器有损坏还要进行报修等待维修人员进行修理；report()需要将具体检查信息通过表格进行汇报。

当需要警卫进行日常巡检工作时，他们只需要调用自己的SecurityCheckTask对象中的三个函数即可，且它们并不知道这三个函数的具体实现细节。这个SecurityCheckTask被称为工作类的外观（Facade），当需要修改安检工作的细节时，只需要对SecurityCheckTask进行改动即可，并不需要过多修改警卫类本身。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| SecurityCheckTask | SecurityCheckTask(List<ExhibitionRoom> exhRoomList,List<ExperienceRoom> expRoomList,List<Machine> machineList) | SecurityCheckTask 构造函数 |
| SecurityCheckTask | Guard. checkRoomSecurity() | 对公共场所进行日常巡检 |
| SecurityCheckTask | Guard. checkMachineSecurity() | 对机器设备进行日常巡检 |
| SecurityCheckTask | Guard. report() | 向总管汇报当天巡检状况 |

### 3.9.3类图



## 3.10 Factory Method工厂模式

### 3.10.1设计模式简述

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

### 3.10.2 ChocolatesMachineAssemblyLine实现API

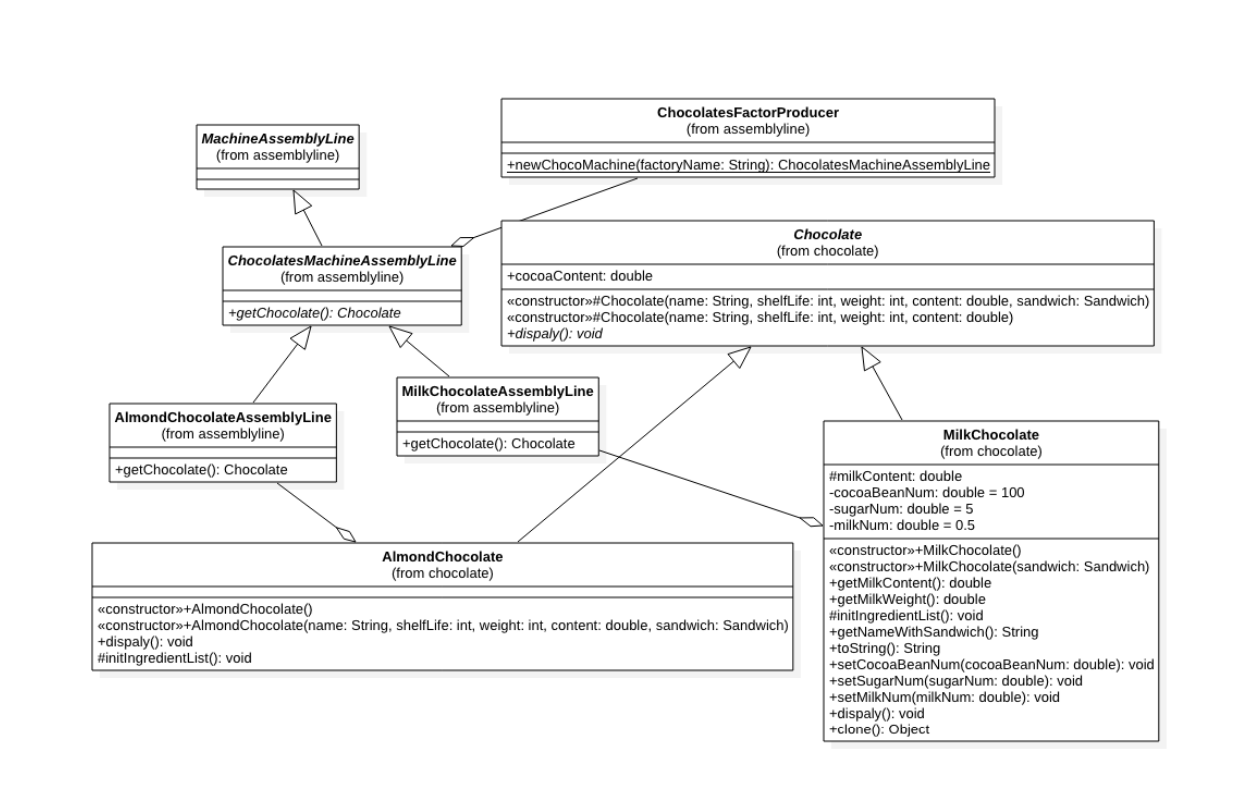
通过工厂模式，调用(ChocolatesFactorProcucer)创建巧克力的工厂(MilkChocolateFactory)，并通过巧克力的工厂，来创建我们的实际的巧克力产品(AlmondChocolate、MilkChocolate)。

**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 |
| ChocolatesMachineAssemblyLine | public abstract Chocolates getChocolate(); | 通过getChocolate返回我们实际的巧克力产品。 |
| ChocolatesFactoryProducer | public static ChocolatesMachineFactory newChocoMachine(String factoryName) | 传入巧克力工厂的名称，实例化我们的巧克力工厂。 |
| MilkChocolateAssemblyLine | public Chocolates getChocolate() | 特定的牛奶巧克力工厂，实现了抽象的巧克力工厂，返回特定的巧克力(MilkChocolate) |
| AlmondChocolateAssemblyLine | public Chocolates getChocolate() | 特定的牛奶巧克力工厂，实现了抽象的巧克力工厂，返回特定的巧克力(AlmondChocolate) |

### 3.10.3类图



## 3.11 Flyweight享元模式

### 3.11.1设计模式简述

该模式运用共享技术有效地支持大量细粒度的对象，意在避免大量相同的示例的创建而带来的内存消耗。通过将具有相同属性的示例合并为一个示例来实现内存的节省。在其他类需要该类的工厂类创建一个类示例的时候，该工厂类并不会直接创建一个实例，而是在对象池中查看是否有对应要求的实例。如果有要求的实例则返回该实例的引用，如果没有则创建一个实例。这样，所有相同属性的实例的引用都指向一个对象，从而避免内存消耗。

### 3.11.2 ChocolateMix实现API

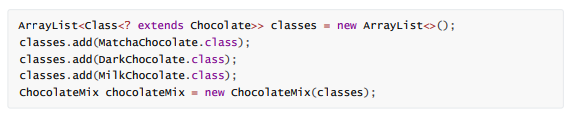
巧克力工厂的生产产品中有一种是“什锦巧克力”。该种产品是将多种巧克力（ Chocolate 的子类，如MatchaChocolate ）包装在一起而形成。由于同一种类的巧克力在原料、重量、保质期等方面具有完全相同的属性，所以可以用同样的一个实例代表。

在实现过程中使用 MixChocolateFactory 工厂类来为 ChocolateMix 返回实例。该类使用哈希表来保存所有现有的 Chocolate 子类的实例。当 ChocolateMix 实例向 MixChocolateFactory 要求一个Chocolate 子类的实例时，该工厂类就会在哈希表中寻找实例。如果找到就立即返回该实例，如果没有就新建一个实例、插入哈希表中再返回。

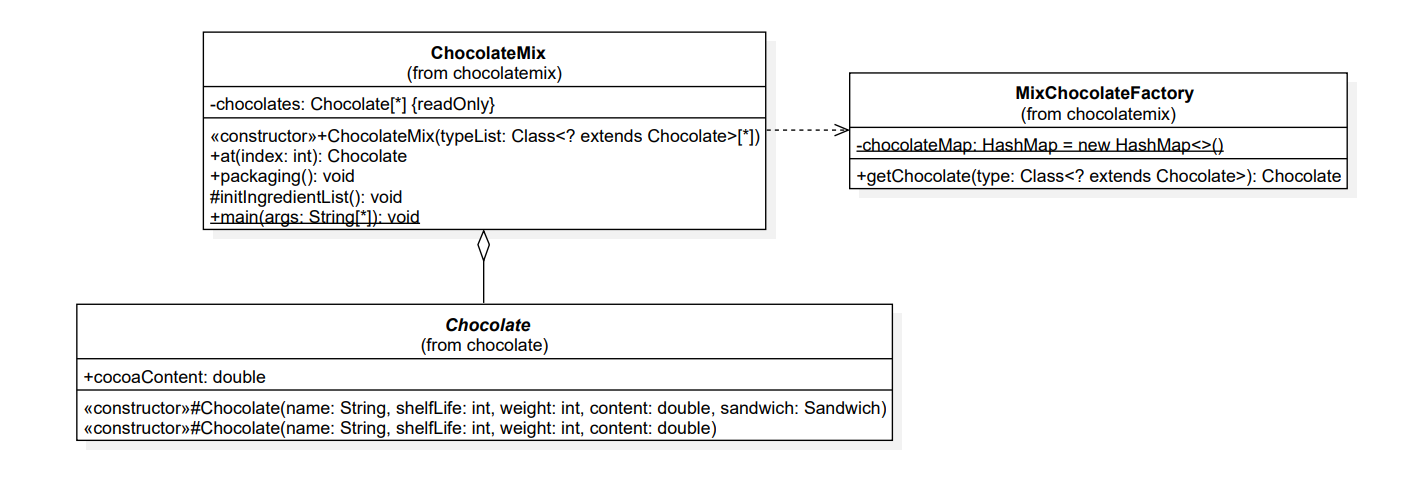
**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| MixChocolateFactory | getChocolate(type:Class<? extendsChocolate>):Chocolate | 获取Chocolate子类的实例。ChocolateMix类初始化时依赖于该方法。  参数：需要获取的Chocolate子类的类型，如：MatchaChocolate.class。  输出：在需要新建实例时输出"Creating chocolate of type: %s"。%s是新建实例的类名。不管新建对象是否成功都输出。 |
| ChocolateMix | ChocolateMix(typeList:Class<? extendsChocolate>[\*]) | ChocolateMix的构造函数。该函数封装了总重量和保质期的计算，以及原料列表的初始化。  参数：一个Chocolate子类类型的列表List<Class<? extends Chocolate>>。（可以为ArrayList<...>或LinkList<...>）  输出：由于该函数依赖MixChocolateFactory.getChocolate()，所以在MixChocolateFactory类新建实例时会有对应的输出。 |
| ChocolateMix | at(index: int): Chocolate | 返回该什锦巧克力实例中第i个巧克力对象。若下标i不合法则返回null。（下标与传入构造函数列表的下标一致） |
| ChocolateMix | packaging(): void | 对Product中同名虚函数的实现。 |
| ChocolateMix | main(args: String[\*]): void | 对Flyweight模式样例测试代码，建议参考。 该函数中包含有ChocolateMix的构造方法样 例以及测试Flyweight模式中相同类的引用是否指向同一个对象的测试。 |

ChocolateMix构造方法示例：



### 3.11.3类图



## 3.12 Interpreter解释器模式

### 3.12.1设计模式简述

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

### 3.12.2 Expression实现API

在巧克力工厂中，具有不同的区域和不同的角色，角色中包括参观人员（Visitor），可以参观工厂的公共区域（PublicArea），而不可以进入其它区域。因此，对于游客要进入某区域的行为，需要进行判断。

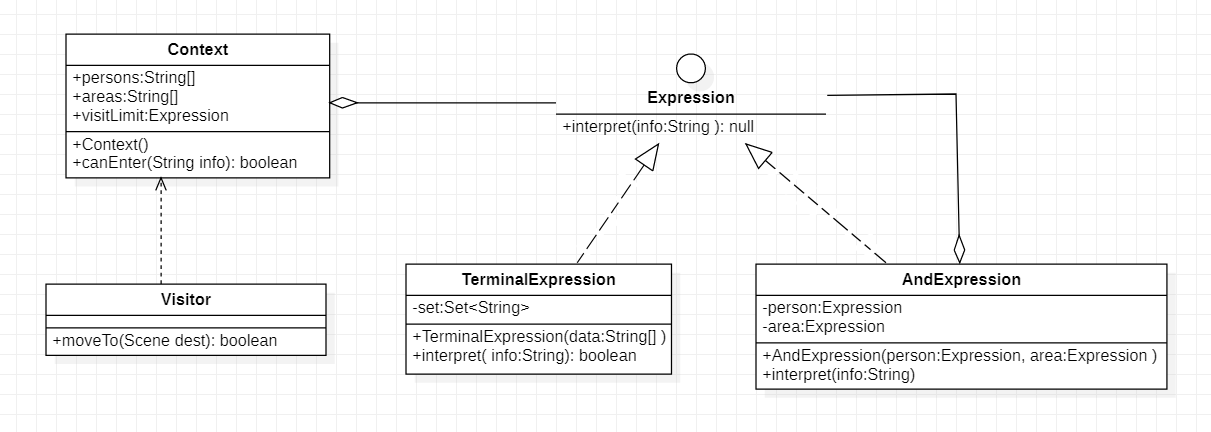
显然，使用大量的if语句来进行这种判断是低效且扩展性差的，因此，在这里选择使用解释器模式来完成这种判断，使得逻辑易于理解，且增强了可扩展性，降低了耦合度。

* Interpreter 在项目中的实现
  + 抽象表达式（Abstract Expression）：主要包含解释方法 interpret()。
    - 终结符表达式（Terminal Expression）：抽象表达式的子类，用来实现文法中与终结符相关的操作。
    - 非终结符表达式（AndTerminal Expression）：抽象表达式的子类，用来实现文法中与非终结符相关的操作。
  + 环境（Context）：包含各个解释器需要的数据和公共的功能。
  + 游客（Visitor）：客户类，此处通过环境角色间接访问解释器的解释方法判断能否进入某区域。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Terminal Expression | +interpret(info:String)：boolean | 判断被分析的字符串是否是集合中的终结符。 参数：info，被分析的字符串，此处为”sb. enter somewhere” |
| AndExpression | +interpret(info:String)：boolean | 判断被分析的字符串是否是满足条件的身份和满足条件的区域。 参数：info，被分析的字符串，此处为”sb. enter somewhere” |
| Context | +canEnter(info:String)：boolean | 调用AndExpression的interpret方法来对被分析的字符串进行解释。 参数：info，被分析的字符串，此处为”sb. enter somewhere” |
| Visitor | +moveTo(dest：Scene)：boolean | 内部创建一个Context对象，通过该对象间接访问解释器的解释方法判断人员能否进入某区域。 参数：dest，想要进入的地点 |

### 3.12.3类图



## 3.13 Iterator迭代器模式

### 3.13.1设计模式简述

* 定义：提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。迭代器模式是一种对象行为型模式，其主要优点如下。
* 主要优点
  + 访问一个聚合对象的内容而无须暴露它的内部表示。
  + 遍历任务交由迭代器完成，这简化了聚合类。
  + 它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。
  + 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
  + 封装性良好，为遍历不同的聚合结构提供一个统一的接口。
* 主要缺点
  + 增加了类的个数，这在一定程度上增加了系统的复杂性。

### 3.13.2 WorkerListIterator实现API

在巧克力工厂中，具有不同的办公部门，每个部门都有很多员工，我们经常需要去查看部门员工数量以及员工信息，而且有时候我们会希望仅仅查看员工工资，有时候会希望仅仅查看年龄，或者会希望去查看男员工或女员工……我们有如此之多的对于员工的查看方式，所以仅仅将遍方式固定为某一种是不合理的。

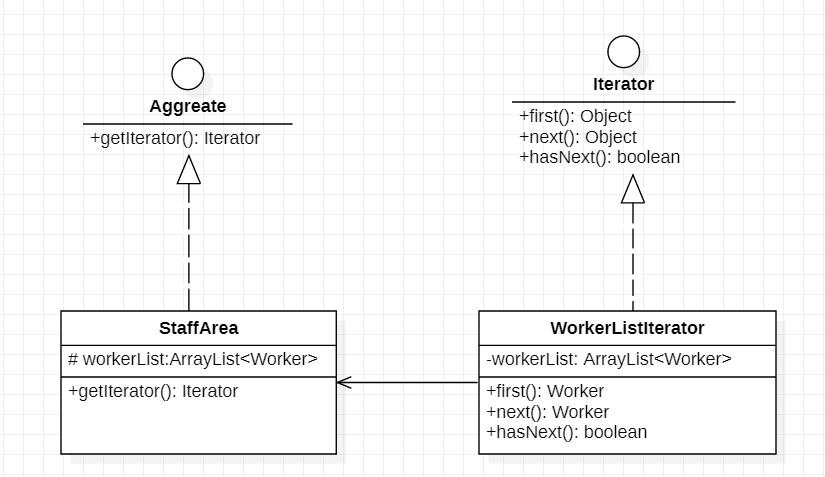
显然的，将这么多遍历方式都实现一遍并写入仓库管理类中是不合理的，这严重违反了单一职责原则，引起代码的冗余和浪费。因此，我们使用迭代器模式，使得程序逻辑易于理解，可扩展性增强。

* Iterator在项目中的实现
  + 抽象聚合（Aggregate）：定义创建迭代器对象的接口。
    - 员工区域（StaffArea）：具体聚合（ConcreteAggregate），实现抽象聚合接口，返回一个具体迭代器的实例。
  + 抽象迭代器（Iterator）：定义访问和遍历聚合元素的接口，包含hasNext()、first()、next() 等方法。
    - WorkerListIterator：具体迭代器（ConcreteIterator）：实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| StaffArea | +getIterator(): 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 |
| WorkerListIterator | +first(): Worker | 获取第⼀个合理的员工对象 |
| WorkerListIterator | +next()：Worker | 获取下⼀个合理的员工对象 |
| WorkerListIterator | +hasNext()：boolean | 判断是否还有下⼀个合理的员工对象 |

### 3.13.3类图



## 3.14 Mediator中介者模式

### 3.14.1设计模式简述

* 定义：定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。
* 主要优点
  + 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。
* 主要缺点
  + 当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护。

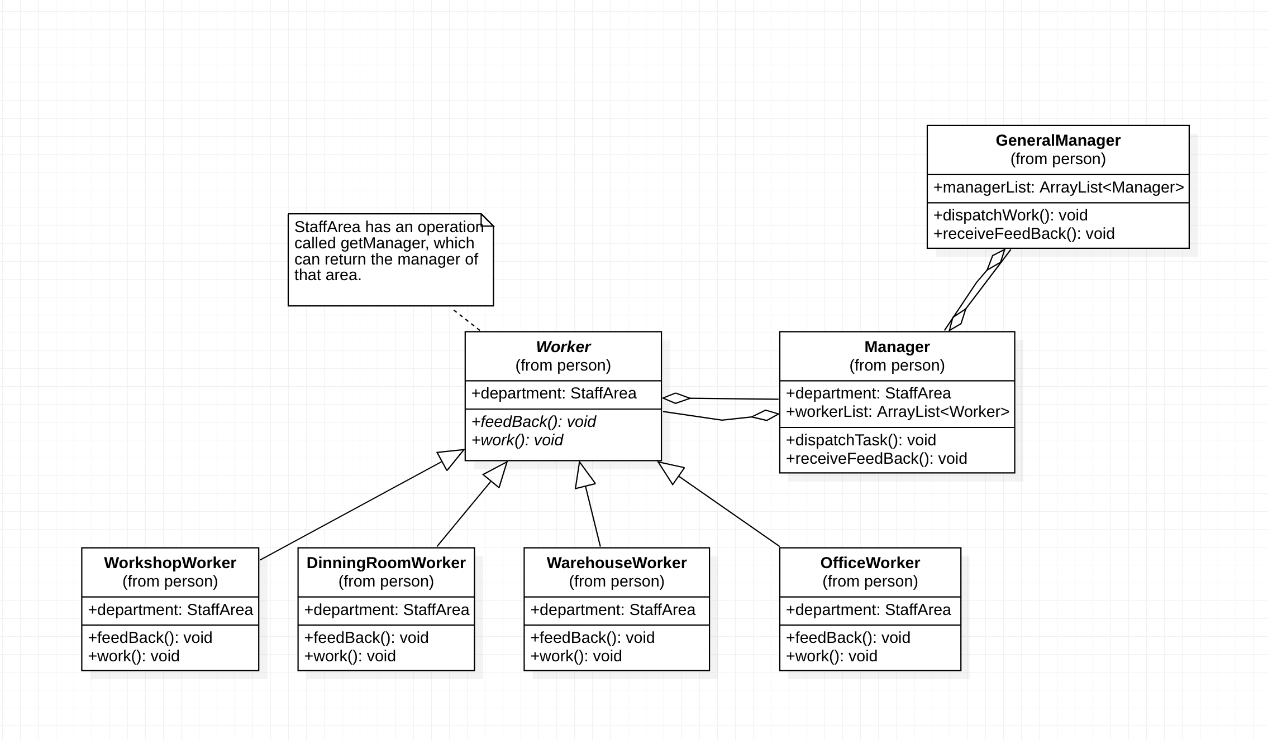
### 3.14.2 Product实现API

* 本项目中的实现
  + 经理（Manager）作为中介者
  + 工人（Worker）和总经理（GeneralManager）作为被中介者关联的对象
    - 总经理通过经理将不同车间的任务派发给具体的工人
    - 工人通过自己车间的经理将意见反馈给总经理

**API描述**

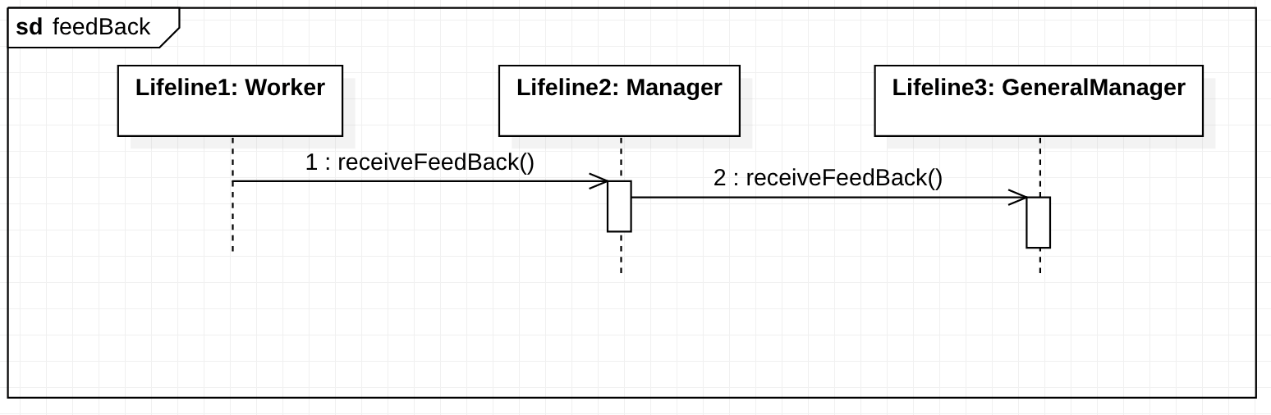
|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Manager | +dispatchTask(): void | 向管辖车间的工人发布任务，使之开始工作 |
| Manager | +receiveFeedBack(): void | 接收所管辖车间的工人反馈信息，并将汇总信息传递给总经理 |
| Worker | +work(): void | 接收车间经理的任务，开始工作 |
| Worker | +feedBack(): void | 向车间经理反馈意见 |
| GeneralManager | +receiveFeedBack(): void | 接收各车间的经理反馈的意见并做出响应 |
| GeneralManager | +dispatchWork(): void | 向各个车间的经理派发工作，由他们将具体任务分配给每个具体的工人 |

### 3.14.3类图

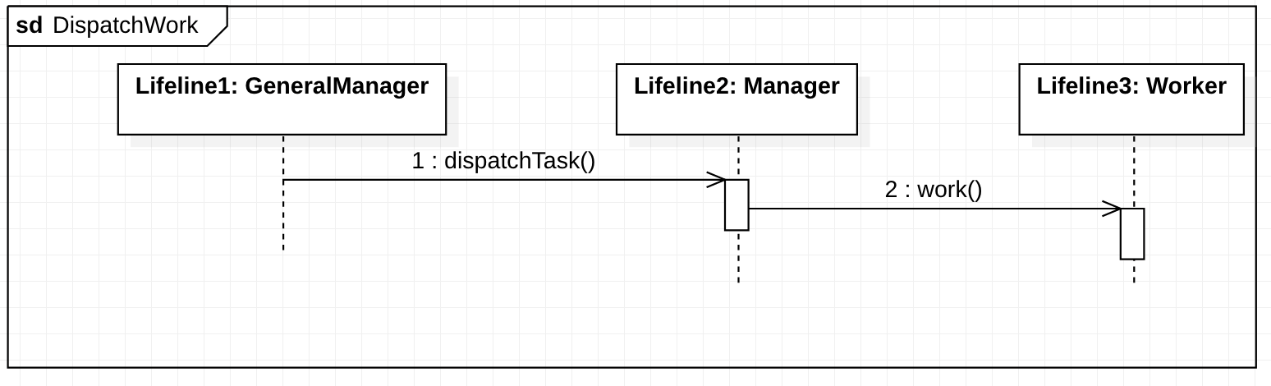


### 3.14.4时序图

以向上级反馈为例，其时序图如下



以向下级发布任务为例，其时序图如下



## 3.15 Memento备忘录模式

### 3.15.1设计模式简述

该模式的定义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。该模式又叫快照模式。

备忘录模式是一种对象行为型模式，其主要优点如下。

* 提供了一种可以恢复状态的机制。当用户需要时能够比较方便地将数据恢复到某个历史的状态。
* 实现了内部状态的封装。除了创建它的发起人之外，其他对象都不能够访问这些状态信息。
* 简化了发起人类。发起人不需要管理和保存其内部状态的各个备份，所有状态信息都保存在备忘录中，并由管理者进行管理，这符合单一职责原则。

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

### 3.15.2 ProductMemento实现API

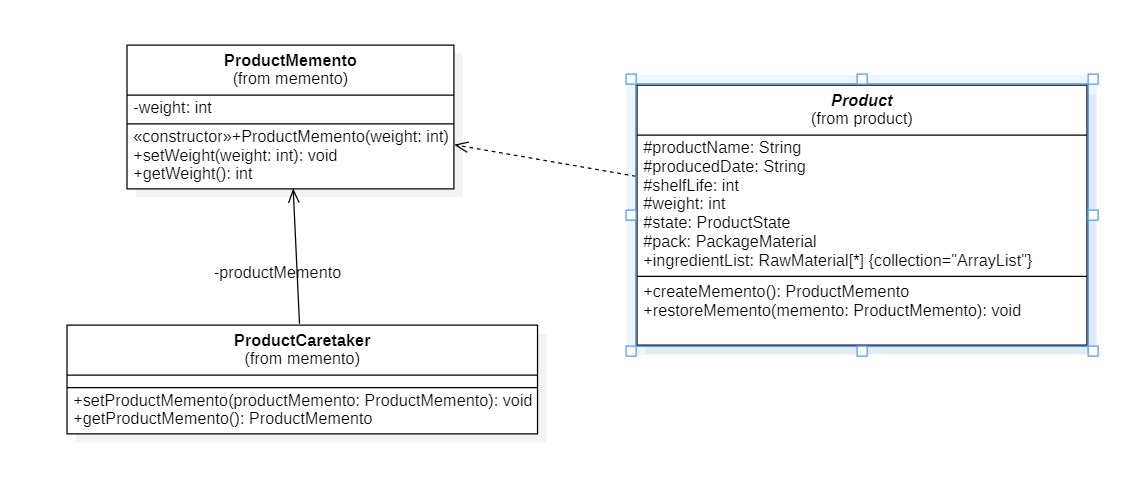
在工厂的实际生产过程中，对一些历史记录进行存档是很普遍的现象。于是我们选择使用备忘录模式，来记录本项目中的产品数据。

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 |
| ProductMemento | ProductMemento(weight:int) | ProductMemento的构造函数 |
| ProductMemento | setWeight(weight:int):void | 存储weight数据 |
| ProductMemento | getWeight():int | 查询记录中的weight数据 |
| Product | createMemento():ProductMemento | 创建备忘录 |
| Product | restoreMemento(memento: ProductMemento):void | 从备忘录中获取历史数据以恢复当前数据 |
| ProductCaretaker | setProductMemento(productMemento: ProductMemento):void | 保存备忘录 |
| ProductCaretaker | getProductMemento():ProductMemento | 获取备忘录 |

### 3.15.3类图



## 3.16 Observer观察者模式

### 3.16.1设计模式简述

观察者模式是指多个对象间存在一对多的依赖关系，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。

* 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。
* 缺点
  + 目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。
  + 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。

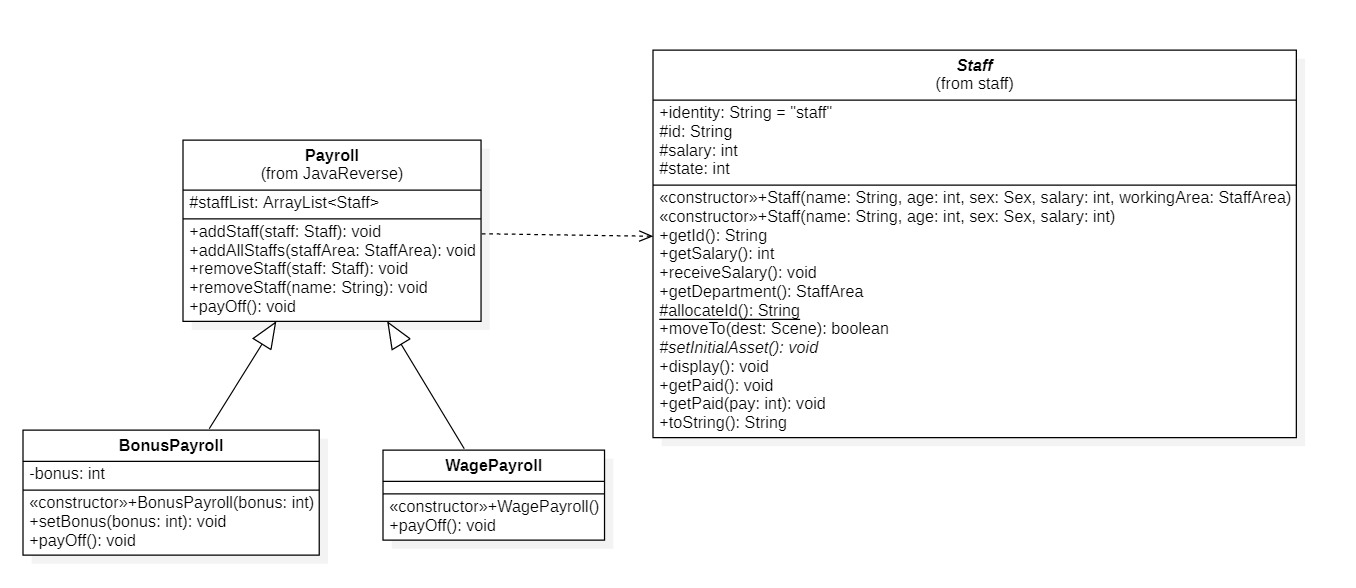
### 3.16.2 Payroll实现API

观察者模式通常用于通知一些群体性事件，本项目中我们通过工资发放这一事件来实现该设计模式。定义Payroll为抽象主题角色，它保存了需要发放工资的员工列表，并且提供增加和删除员工的方法，以及通知所有员工发工资的抽象方法。BonusPayroll和WagePayroll是两个具体主题角色，分别实现奖金和基础工资的发放。Staff类在这里即为观察者角色，提供了一个增加工资的接口，当Payroll中发工资的事件发生的时候，通过调用这一接口，改变其salary属性的值，实现员工工资的增加。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Payroll | addStaff(staff:Staff):void | 向列表中添加需要发放工资的员工 |
| Payroll | addAllStaffs(staffArea:StaffArea):void | 将所有员工添加到需要发工资的列表中 |
| Payroll | removeStaff(staff:Staff):void  removeStaff(name:String):void | 从列表中移除员工 |
| Payroll | payoff():void | 通知员工发工资的抽象方法 |
| Staff | getPaid():void  getPaid(pay:int):void | 实现员工工资的增加 |

### 3.16.3类图



## 3.17 Prototype原型模式

### 3.17.1设计模式简述

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。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。这种模式是实现了一个原型接口，该接口用于创建当前对象的克隆。当直接创建对象的代价比较大时，则采用这种模式。例如，一个对象需要在一个高代价的数据库操作之后被创建。我们可以缓存该对象，在下一个请求时返回它的克隆，在需要的时候更新数据库，以此来减少数据库调用。

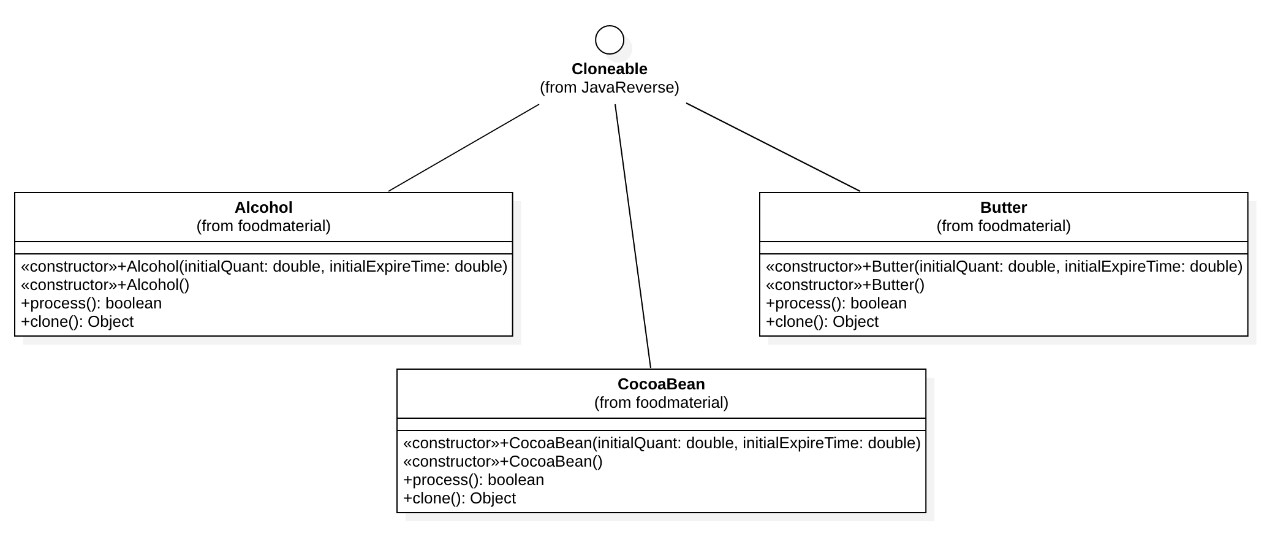
### 3.17.2 **RawMaterial实现API**

所有的raw material类实现了Cloneable接口，重写了clone方法，手动实现了深拷贝（手动对引用进行了一个赋值的操作）。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| Alcohol | public Object clone() | 通过实现了Object类的方法，实现了深拷贝。 |
| Buttter | public Object clone() | 通过实现了Object类的方法，实现了深拷贝。 |
| CocoaBean | public Object clone() | 调用Object类的clone方法，实现了浅拷贝。 |

### 3.17.3类图



## 3.18 Proxy代理模式

### 3.18.1设计模式简述

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

### 3.18.2 PortableBattery实现API

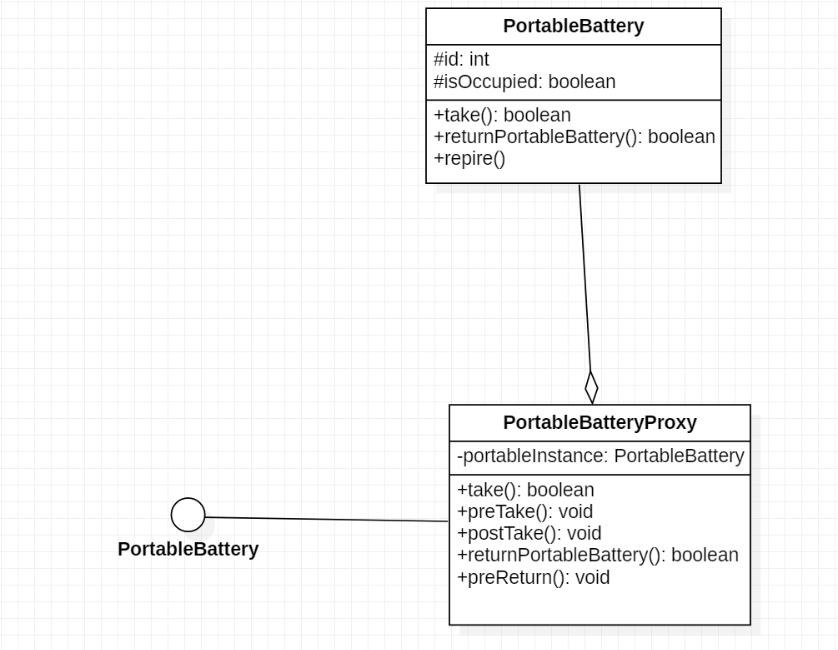
为portablebattery的借用做一个代理，人们通过与代理的交互进行充电宝的获取与归还。

抽象主题（Subject）类：通过接口实现portablebattery的取用和归还两个功能  
真实主题（Real Subject）类：即portablebattery类本身  
代理（Proxy）类：提供了和portablebattery本身相同的方法，有所扩展

**API 描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| PortableBatteryProxy | boolean take() | 取用该充电宝 |
| PortableBatteryProxy | boolean returnPortableBattery () | 归还该充电宝 |
| PortableBatteryProxy | void preTake() | 取用充电宝的前置处理 |
| PortableBatteryProxy | void preReturn() | 归还充电宝的前置处理 |
| PortableBatteryProxy | void postTake() | 取用充电宝的后置处理 |
| PortableBatteryProxy | void PostReturn() | 归还充电宝的后置处理 |

### 3.18.3类图



## 3.19 Singleton单例模式

### 3.19.1设计模式简述

单例模式（Singleton）指一个类只有一个实例，且该类能自行创建这个实例的一种模式。单例模式可以保证内存里只有一个实例，减少内存的开销，也可以避免对资源的多重占用。单例模式有 3 个特点：

1. 单例类只有一个实例对象；
2. 该单例对象必须由单例类自行创建；
3. 单例类对外提供一个访问该单例的全局访问点。

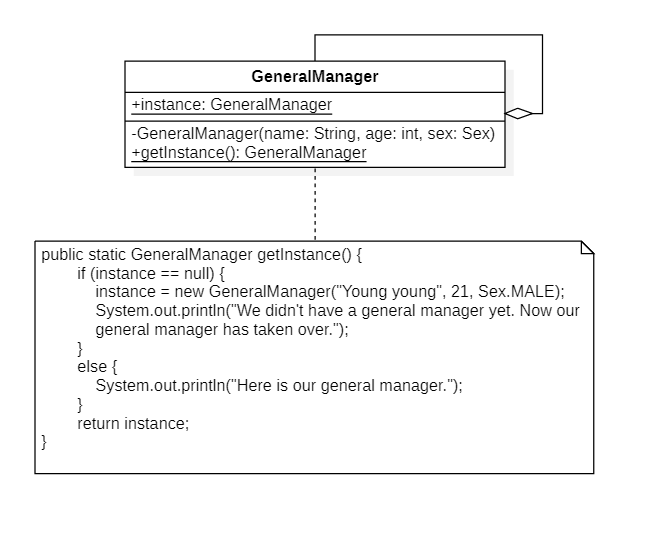
### 3.19.2 GeneralManager实现API

考虑到整个巧克力工厂由一个总经理进行管理，因此GeneralManager类采用Singleton实现。使用getInstance()方法获取GeneralManager的唯一实例对象，只有在第一次调用时，会通过私有的构造函数构造GeneralManager的唯一实例对象。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| GeneralManager | getInstance(): GeneralManager | 静态方法，获取唯一实例对象，在第一次调用时创建对象 |
| GeneralManager | GeneralManager() | 私有构造函数 |

### 3.19.3类图



## 3.20 State状态模式

### 3.20.1设计模式简述

* 定义：对有状态的对象，把复杂的“判断逻辑”提取到不同的状态对象中，允许状态对象在其内部状态发生改变时改变其行为。
* 主要优点：
  + 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”。
  + 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。
  + 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。
* 主要缺点：
  + 会增加系统的类与对象的个数。
  + 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。
  + 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。

### 3.20.2 ProductState实现API

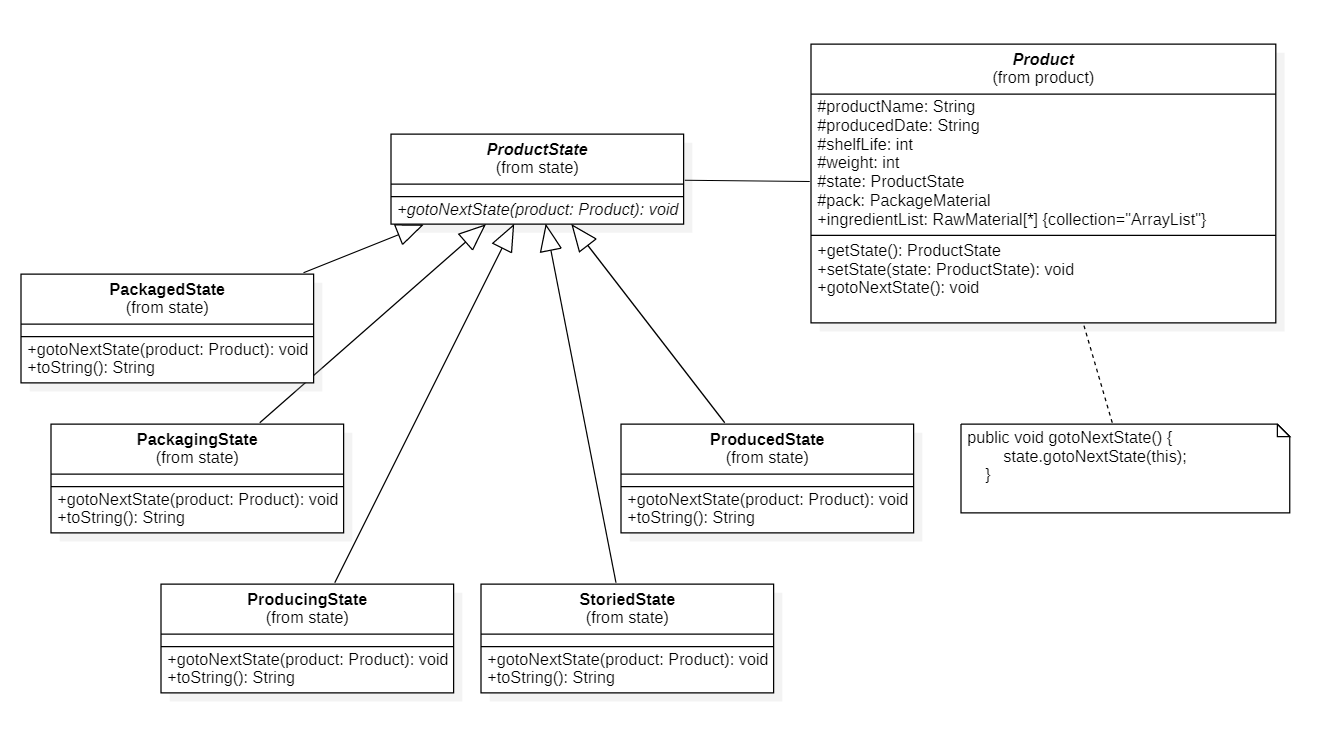
状态模式主要包含环境类、抽象状态和具体状态三个角色。

在我们的项目中，定义了一个ProductState类为抽象状态角色，通过一个接口封装了产品状态。环境类角色定义为Product类，通过getState定义了获取产品状态的接口，内部维护当前的产品状态，并通过gotoNextState实现具体状态的切换。而ProducingState、ProducedState、PackagingState、PakagedState和StoriedState即为具体状态角色，它们定义了每一个具体的产品状态，并进行状态切换。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| ProductState | gotoNextState(product:Product):void | 切换状态的抽象方法 |
| Product | getState():ProductState | 获取产品状态 |
| Product | setState(state:ProductState):void | 更改产品状态 |
| Product | gotoNextState():void | 实现产品状态的切换 |

### 3.20.3类图



## 3.21 Strategy策略模式

### 3.21.1设计模式简述

该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

Strategy模式可以避免使用多重条件语句，可以提供相同行为的不同实现，客户可以根据不同时间或空间要求选择不同的策略。同时，策略模式提供了对OCP原则的完美支持，可以在不修改源代码的情况下，灵活增加新算法。

### 3.21.2 WrapStrategy实现API

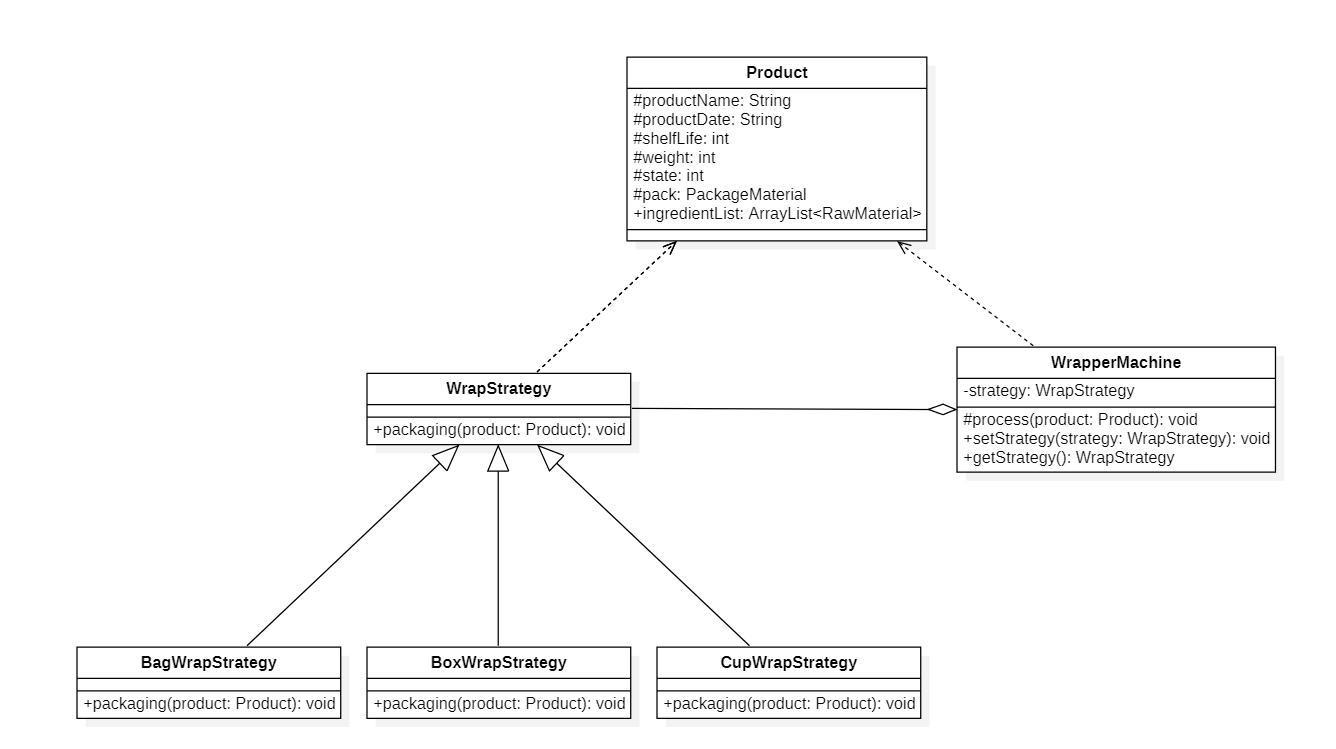
在巧克力工厂中，不同的机器负责执行不同的工作，其中包装机器WrapperMachine负责进行对巧克力制品的包装。员工可以通过调用WrapperMachine的process()方法获取需要包装的产品并进行包装，而包装的方法有许多，如袋装、盒装、杯装，在这里我们使用了Strategy模式来进行实现。

我们将包装的策略设计为WrapStrategy类及其子类，在WrapperMachine类中增加WrapStrategy类型的成员变量，并将具体的包装过程封装在WrapStrategy类及其子类，process()方法只需要调用WrapStrategy类中的接口就能完成包装。同时，如果要增加一个新的包装方式，只需新创建一个WrapStrategy类的子类即可。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| WrapStrategy | packaging(product: Product): void | 封装了包装策略的具体操作，WrapperMachine类中的加工操作只需调用此方法即可进行。每一个WrapStrategy的子类都需要实现该方法 参数：product 需要进行包装的产品 |
| WrapperMachine | process(product: Product): void | 包装机器的包装操作 参数：product 需要进行包装的产品 |
| WrapperMachine | setStrategy(strategy: Strategy): void | 设置包装机器的包装策略 |
| WrapperMachine | getStrategy(): WrapStrategy | 获取包装机器的包装策略 |

### 3.21.3类图



## 3.22 Template Method模板方法模式

### 3.22.1设计模式简述

模板方法（Template Method）是一种行为型模式，它定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。

模板方法的主要优点有：

1. 封装了不变部分，扩展可变部分。将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
2. 在父类中提取了公共的部分代码，便于代码复用
3. 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OCP原则

同时模板方法也有相应的缺点：

1. 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
2. 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度
3. 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

### 3.22.2 TransportMachine实现API

巧克力工厂中，所有机器的基类是机器类（Machine），而继承自机器类（Machine）的交通运输机器类（TransportMachine）又是所有运输机器的基类，形成了两层的继承关系。

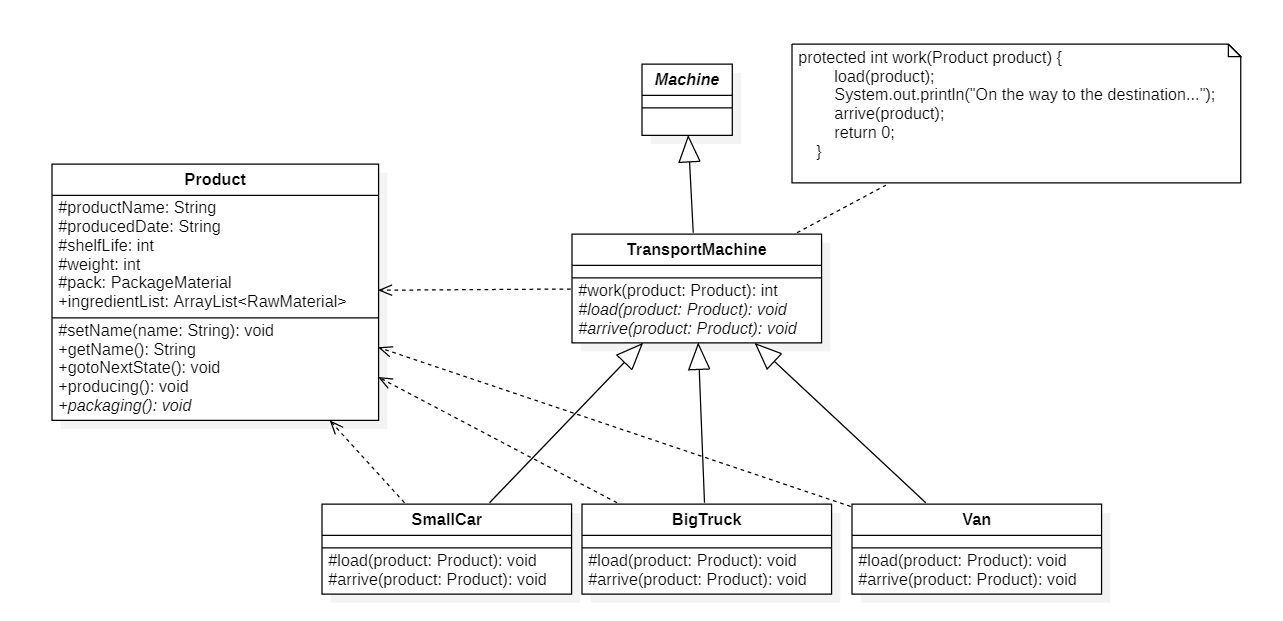
交通运输机器类（TransportMachine）含有方法work()，用于执行运输机器的运输工作。work()方法中有三个步骤：装载、运输、到达并卸货。对于每种具体的运输机器来说，运输的过程是相同的，而装载和卸货的过程则有所不同，我们采用模板方法的思想来实现这一关系。

在交通运输机器类（TransportMachine）的方法work()中，我们实现了运输的过程，而将装载和到达的过程封装在两个抽象方法中，并在work()中调用，具体的实现由交通运输机器类（TransportMachine）的子类来实现。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| TransportMachine | #work(product: Product): int | 使用了模板方法的思想，实现交通运输机器的运输工作，封装了运输的过程，可变的装载和到达过程调用load()和arrive()执行，具体的实现封装在子类的方法中 参数 product：具体运输的产品 |
| TransportMachine | #load(product: Product): void | 抽象方法，交通运输机器的装载过程 |
| TransportMachine | #arrive(product: Product): void | 抽象方法，交通运输机器的到达、卸货过程 |
| SmallCar | #load(product: Product): void | 装载过程在SmallCar类的具体实现 |
| SmallCar | #arrive(product: Product): void | 到达、卸货过程在SmallCar类的具体实现 |

### 3.22.3类图



## 3.23 Visitor访问者模式

### 3.23.1设计模式简述

Visitor模式（访问者模式）定义了一种方法，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。

使用Visitor模式可以将对数据的操作与数据结构相分离，实现数据操作者与数据的解耦，可以在不影响 数据对象结构的基础上增添对数据元素的操作。Visitor模式具有良好的扩展性，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，而且Visitor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，这遵循了单一职责原则。

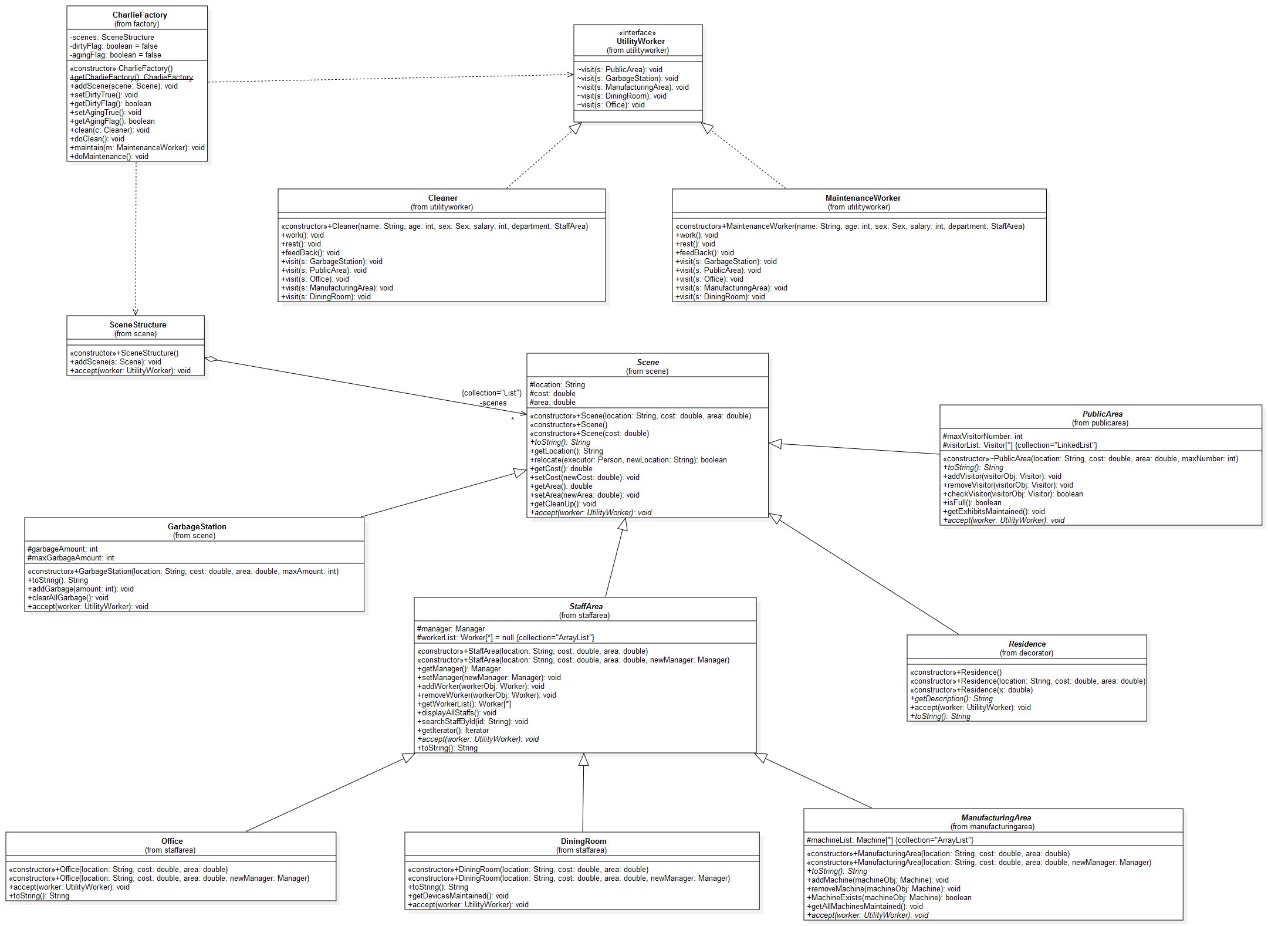
### 3.23.2 UtilityWorker实现API

在巧克力工厂中，有很多不同的场景（比如车间、仓库、办公室、参观区域等），场景需要日常保养与维护（比如清洁卫生、保养机器等），负责这些工作的不同工作人员关注的操作不同（比如清洁工关注卫生状况，而机器保养工人负责机器的磨损与故障状况），同样是保养对于不同的场景有不同的行为（比如保养与检查工作，餐厅要进行厨具的更换，车间要进行机器的保养，而办公室则不需要这部分工作），而且日常维护的工作随着情况可能会有增加，但是场景是保持基本稳定的。

**API描述**

|  |  |  |
| --- | --- | --- |
| 类名 | 函数名 | 作用 |
| UtilityWorker  Cleaner  MaintenanceWorker | visit(in s:PublicArea): void  visit(in s:GarbageStation): void  visit(in s:ManufacturingArea): void  visit(in s:DiningRoom): void  visit(in s:Office): void | UtilityWorker类为接口，里面包含了很多重载方法的定义，这些方法由类Cleaner和类 MaintenanceWorker实现，每个方法内定义了具体的工作对于某种具体场景所进行的操作 |
| CharlieFactory | clean(in c:Cleaner): void, maintenance(in m:MaintenanceWorker): void | PeriodicMaintenance类为Visitor模式的中心类，是外部调用本模式内方法的入口，通过调用clean方法和checkAndMaintenance方法进行清扫和维护工作 |
| SceneStructure | accept(in worker:UtilityWorker): void | SceneStructure类中包含一个由所有场景对象组成的列表，需要进行清扫或维护工作时，调用 accept方法完成每一个元素（场景）对工人（访问者）的接受，在Visitor模式中，任何清扫或维护工作均调用此接口进行进而实现对全部场景的维护。本方法中调用每个具体场景对象的accept 方法，场景对象不需要关注具体操作是什么 |
| Scene  GarbageStation  PublicArea  StaffArea  Office  DiningRoom  ManufacturingArea  Residence | accept(in worker:UtilityWorker): void | Scene类为场景基类，其余类为场景系统中的子类，accept方法为该元素（场景）对工人（访问者）的接受。本方法中调用工人具体的操作方法实现相关操作，场景对象不需要关注具体操作是什么 |

### 3.23.3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

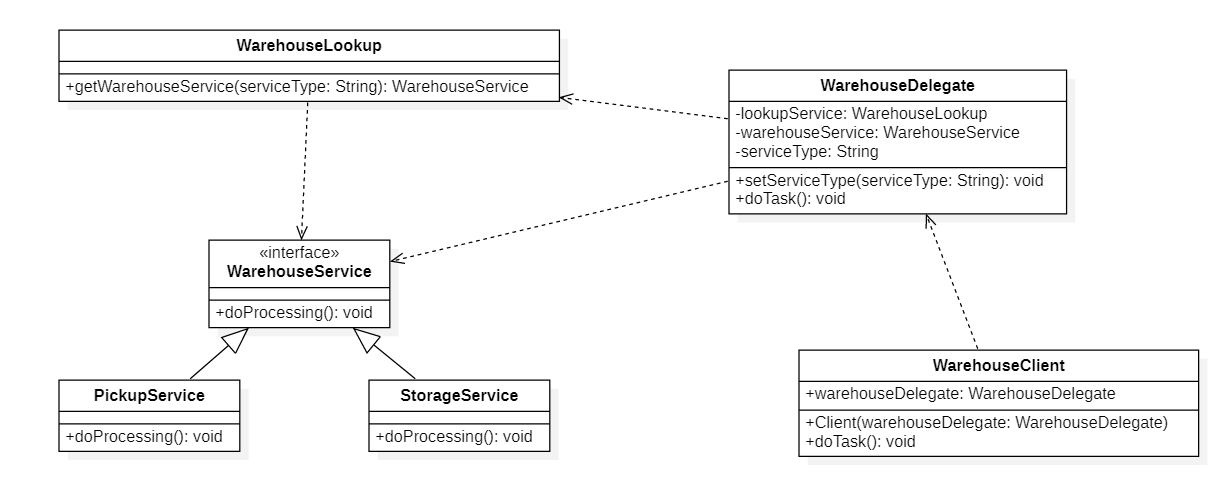
在巧克力工厂中，我们设计了工厂的智能仓库，员工不需要亲自进入仓库取货、存货，只需要通过一个仓库智能终端（WarehouseClient）即可完成工作。仓库提供多种服务，如取货服务（PickupService）、存货服务（StorageService），我们设计了相应的类来实现该服务，所有服务类都需实现仓库服务接口（WarehouseService）。

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

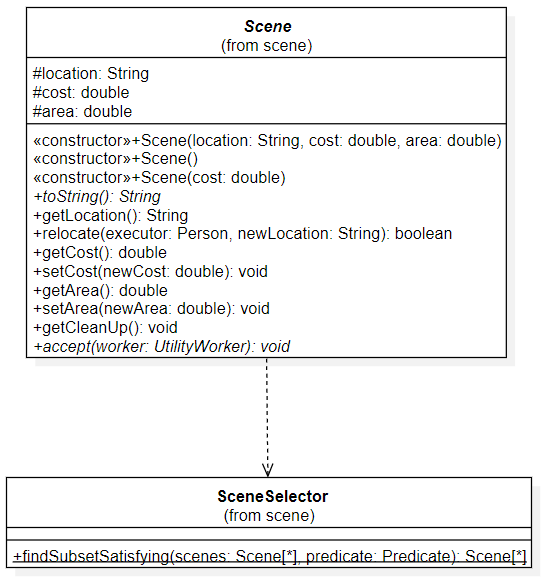
考虑到工厂管理人员常常要查找各类满足一定条件的场景，例如：寻找面积足够大的活动空间，寻找位置在二楼的所有办公室，寻找所有的厂房。而这些要求往往是复杂且难以预知的，故不能写作任何一个类的方法。这里使用Callback模式实现一个筛选器解决这种问题。

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。

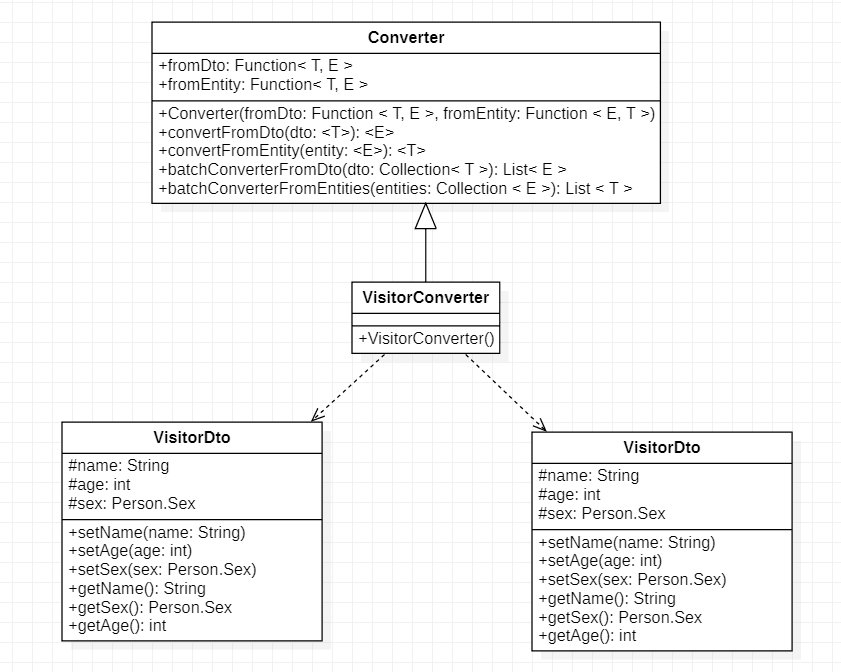
显然的，这种工作是较为单一繁琐的，容易引起代码的冗余和浪费。因此，我们使用转化器设计模式，使类型无需彼此了解即可简洁实现，可扩展性增强。

* 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

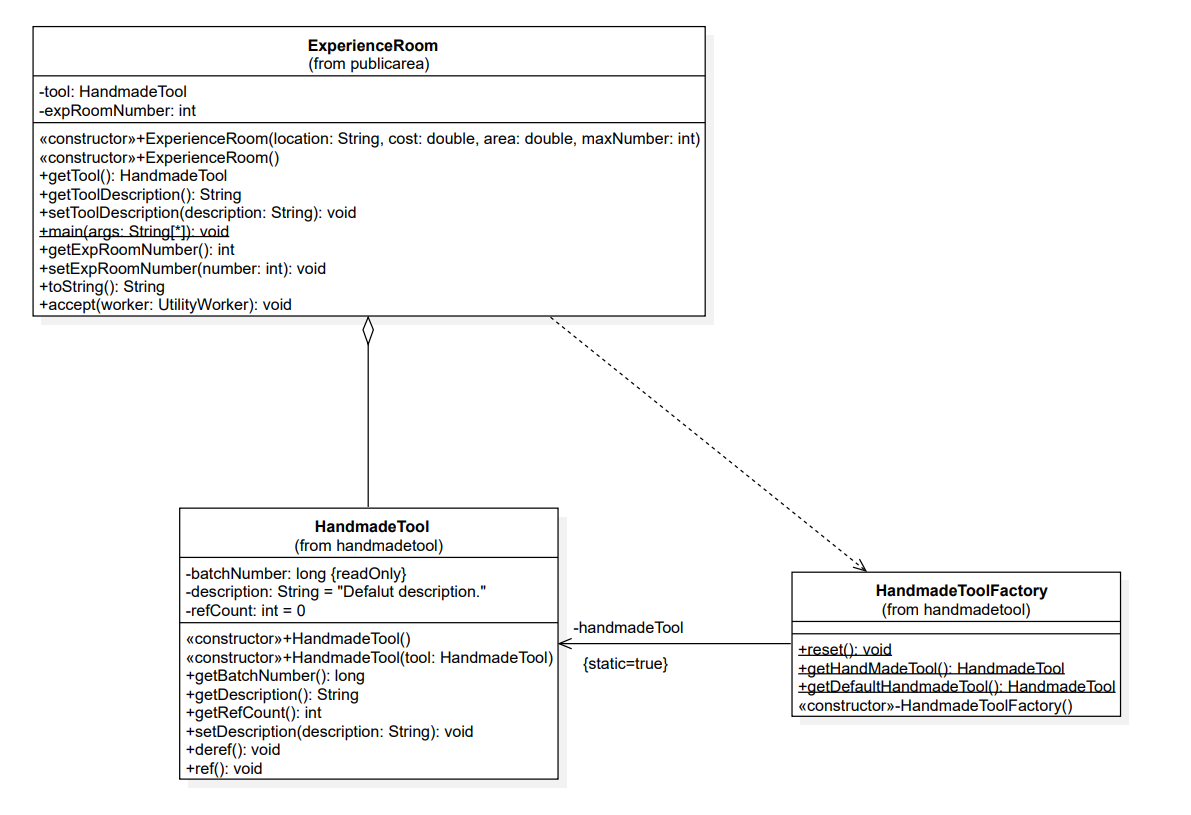
在每个体验室（ ExperienceRoom ）中，都拥有一个手工制作巧克力的工具（ HandmadeTool ）。该工具拥有一段描述和生产批号。在向工厂类 HandmadeToolFactory 请求 HandmadeTool 实例时，会得到默认工具的引用（且默认工具的引用计数自增）。此工具的描述是默认的（"Defalut description."）。 如果修改该工具实例的描述，则触发COW，实例被复制，体验室实例会得到新的引用，且默认实例的引用计数自减。生产批号在复制时不改变。

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

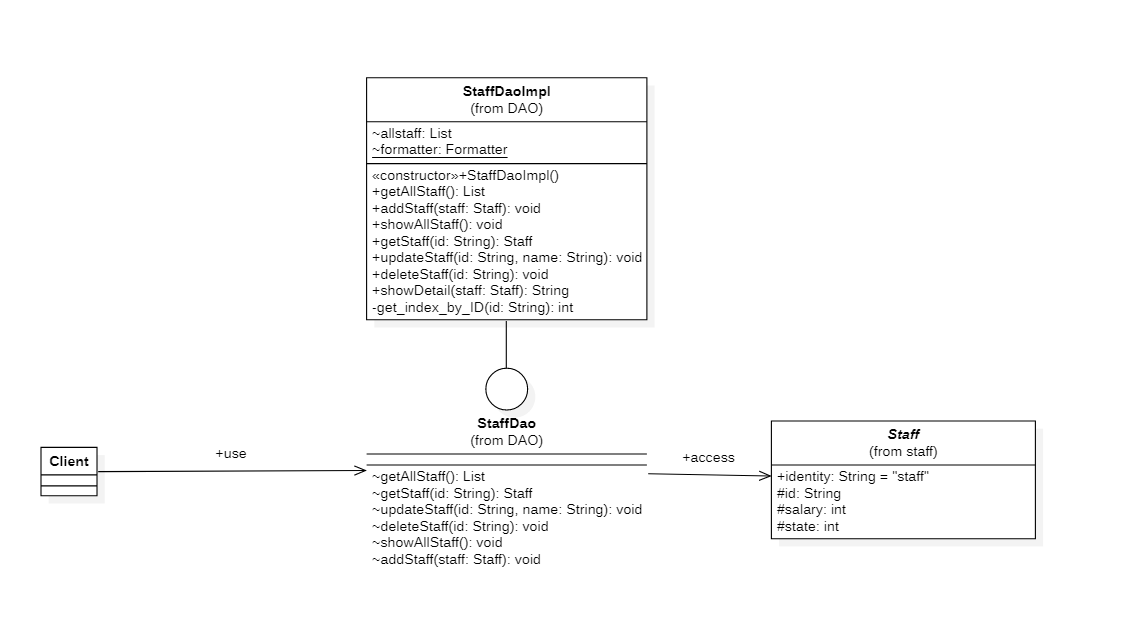
在巧克力工厂中，为了更好地管理员工，通常将员工的信息以通过存入数据库的方式实现数据持久化。在本例中，我们使用一个元素为Staff对象的List模拟为存储员工信息表的数据库，设计模式的主要核心体现在数据库操作的业务逻辑和持久性逻辑的解耦上。本例中的StaffDao接口定义对应Staff类型数据对象的操作函数，包括数据库常用的增删查改操作，StaffDaoImpl则作为StaffDao的实现类读取存储Staff对象的List，具体实现数据库相关的模拟操作。若将来有Staff类型数据存入数据库，只需新增StaffDao的子类从数据库中读取Staff对象，无需改动原有代码。Staff对象主要存储了员工ID、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

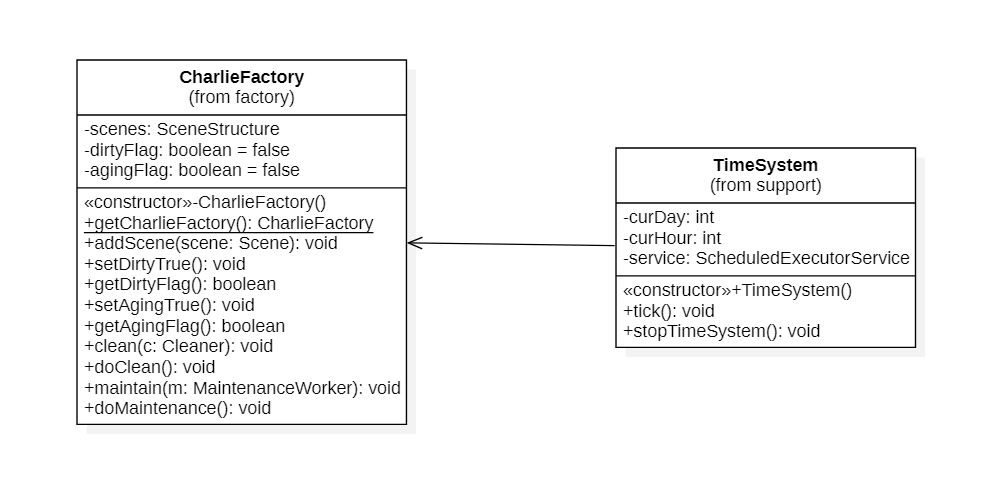
### 3.29.2 CharlieFactory/TimeSystem实现API

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。

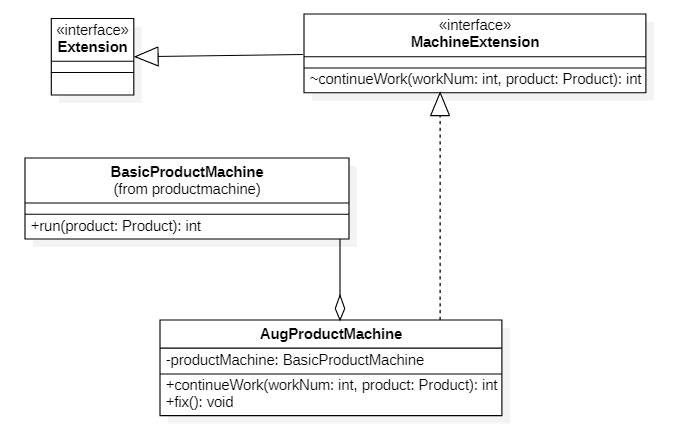
MachineExtension接口扩展自Extension接口，AugProductMachine类实现这一接口，它有一个私有成员变量productMachine，在AugProductMachine类的continueWork方法中根据给定的次数循环调用productMachine的run方法，实现连续生产。

### 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。

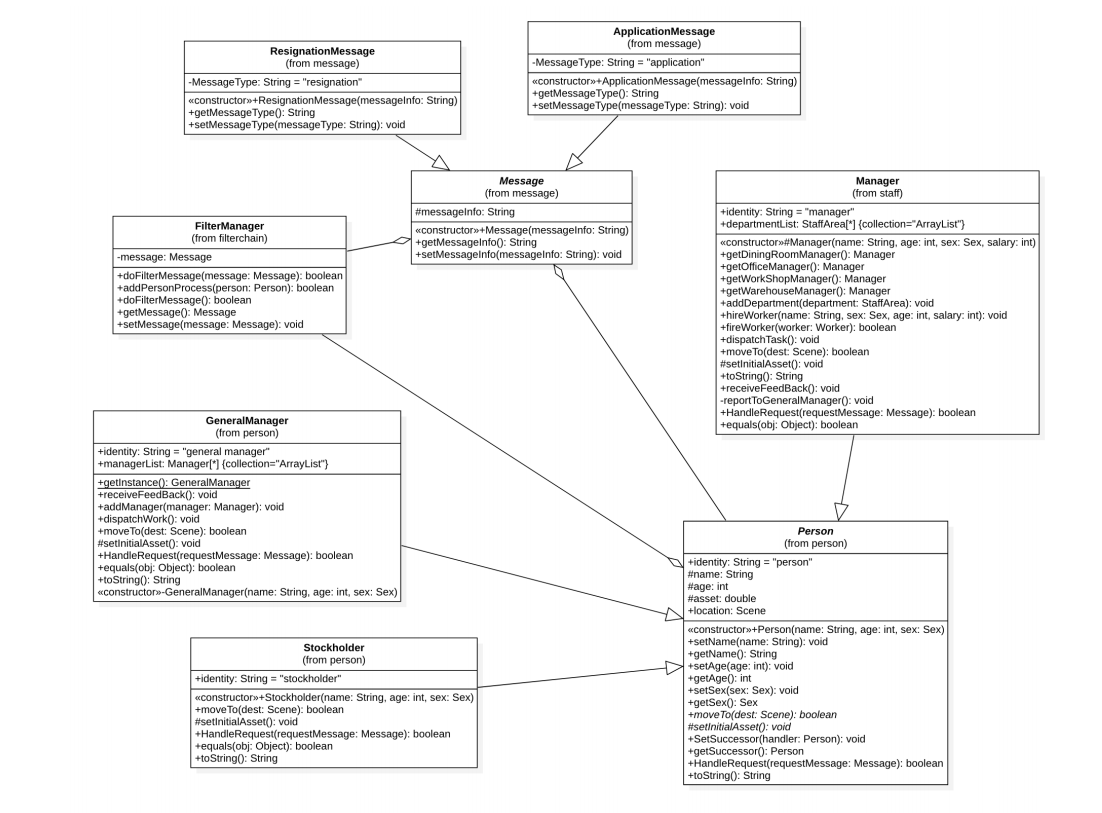
### 3.31.2 FilterManager实现API

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

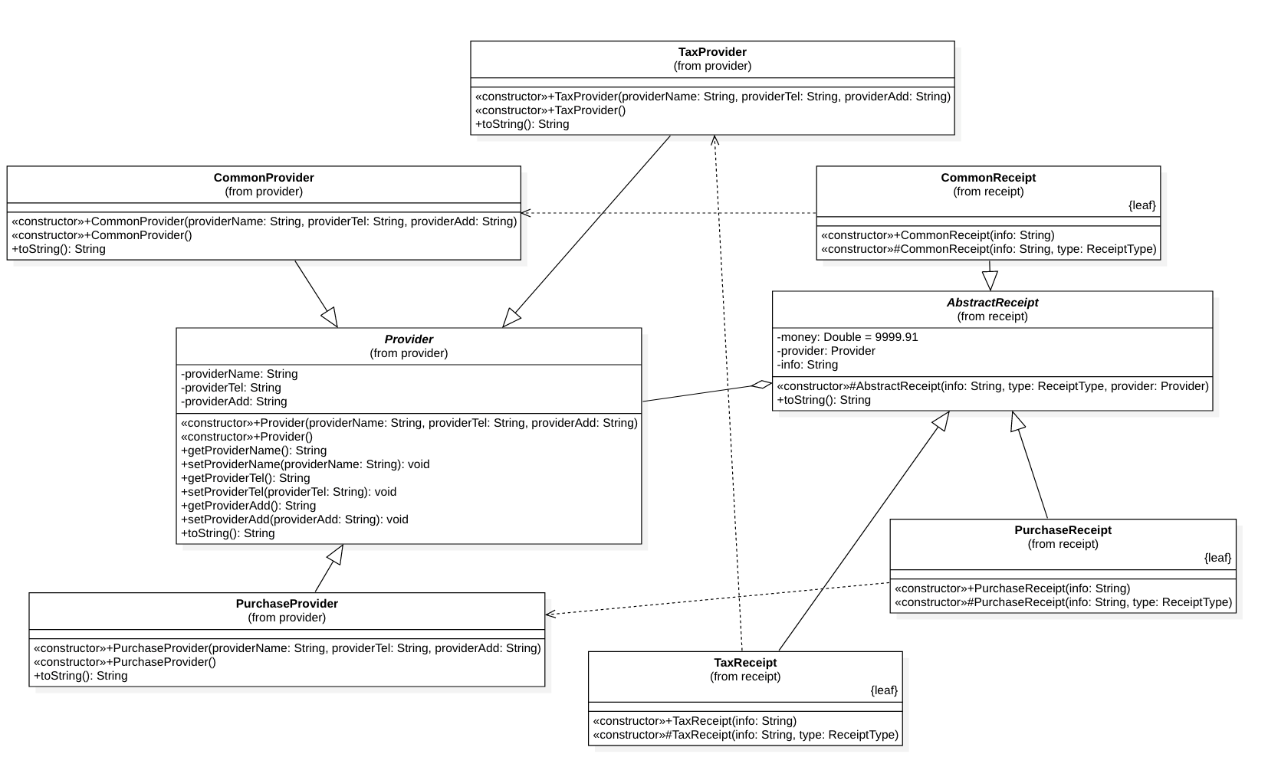
在巧克力工厂整个大环境下，我们每天都会产生大量的资金流动，资金流动就会产生发票，由于金额的敏感性，我们的发票对象一旦初始化，就不可以改变，系统使用了三种发票，分别是税务发票（TaxReceipt），普通发票（CommonReceipt），购买发票（PurchaseReceipt），他们都使用了final修饰，也就是对象不能被改变。

**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。

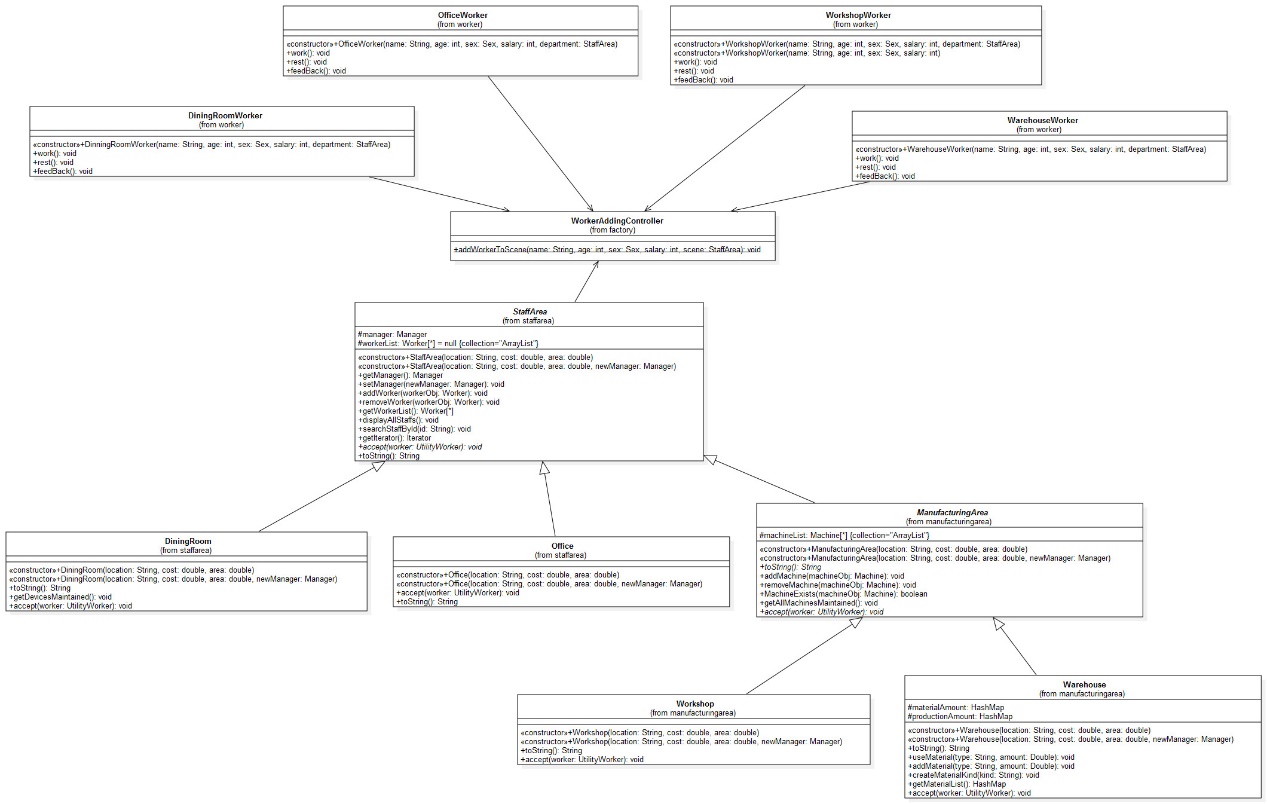
### 3.33.2 WorkerAddingController实现API

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。

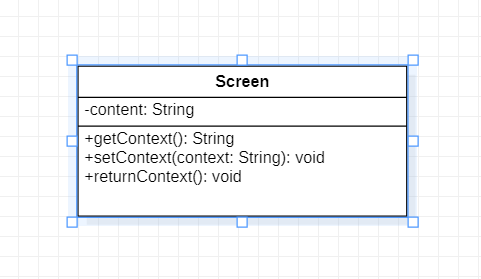
### 3.34.2 Screen 实现API

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

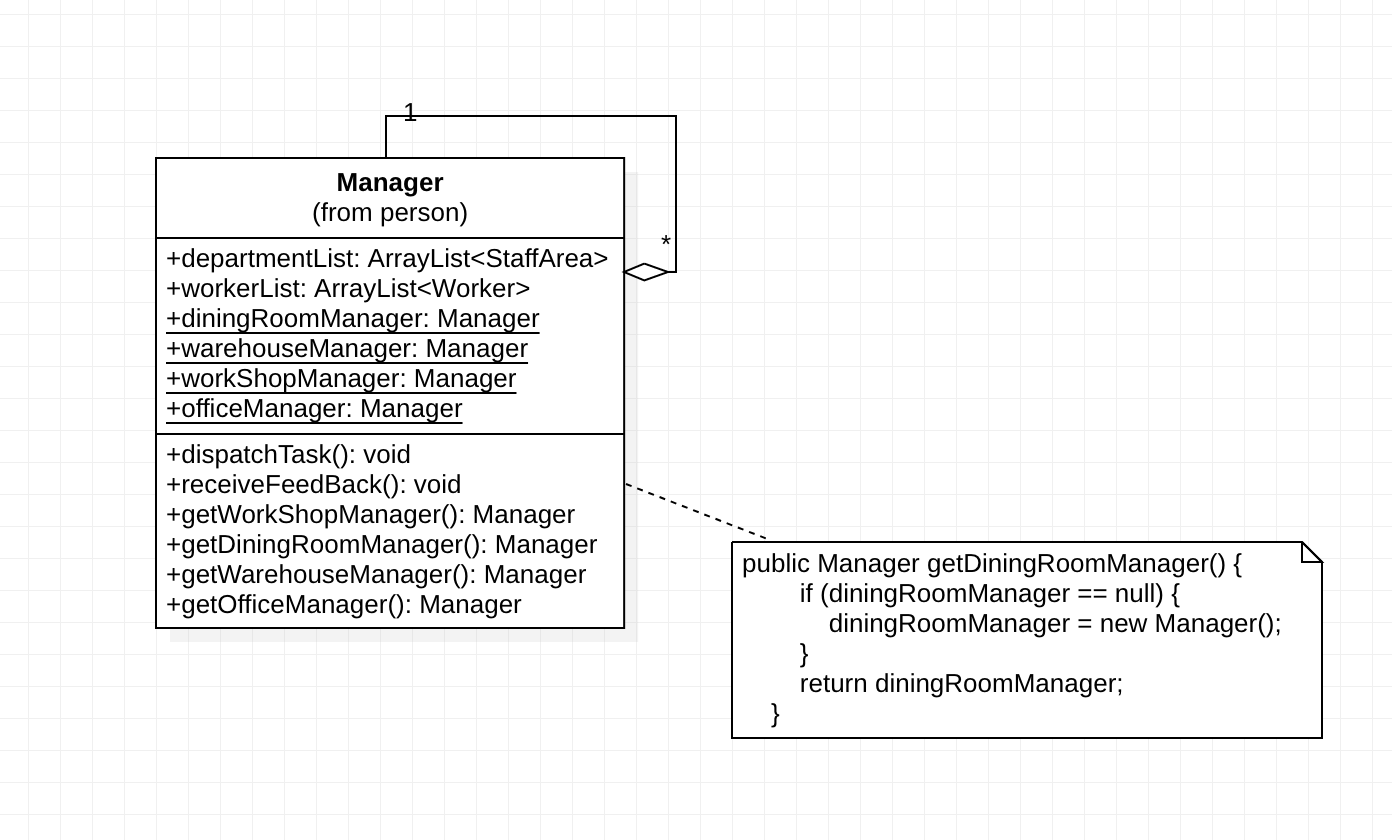
### 3.35.2 Product实现API

* 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。

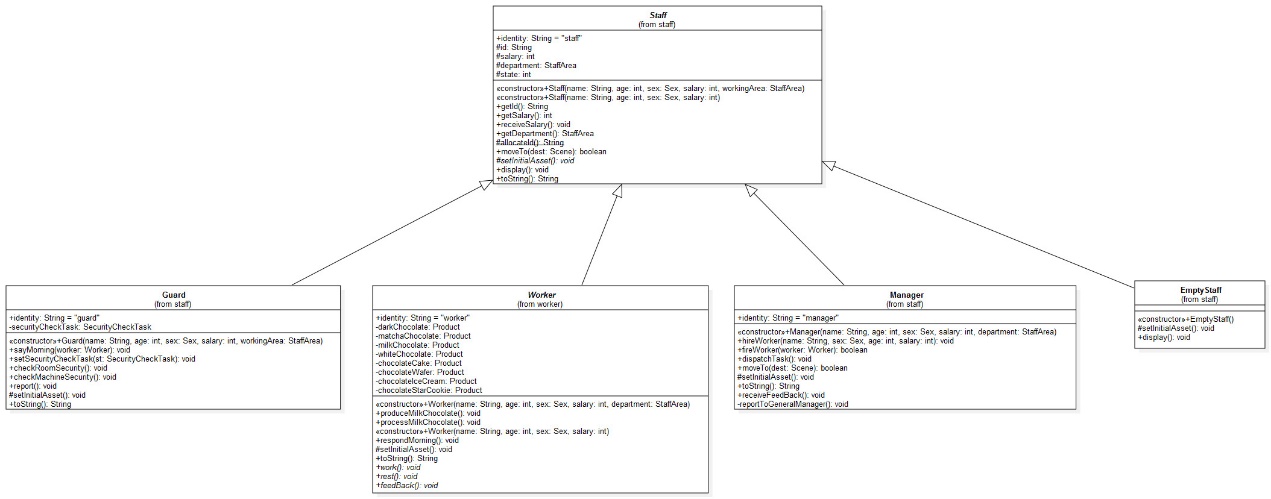
### 3.36.2 EmptyStaff实现API

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

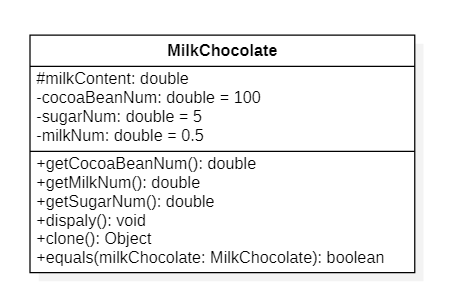
对于产品牛奶巧克力(MilkChocolate)而言，有相同原料及原料含量的牛奶巧克力可看作是同一产品，不同牛奶巧克力之间的区别仅仅是原料含量的不同，所以我们采用值对象(Value Object)模式，将具有相同原料及原料含量的牛奶巧克力看作是同一对象。

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

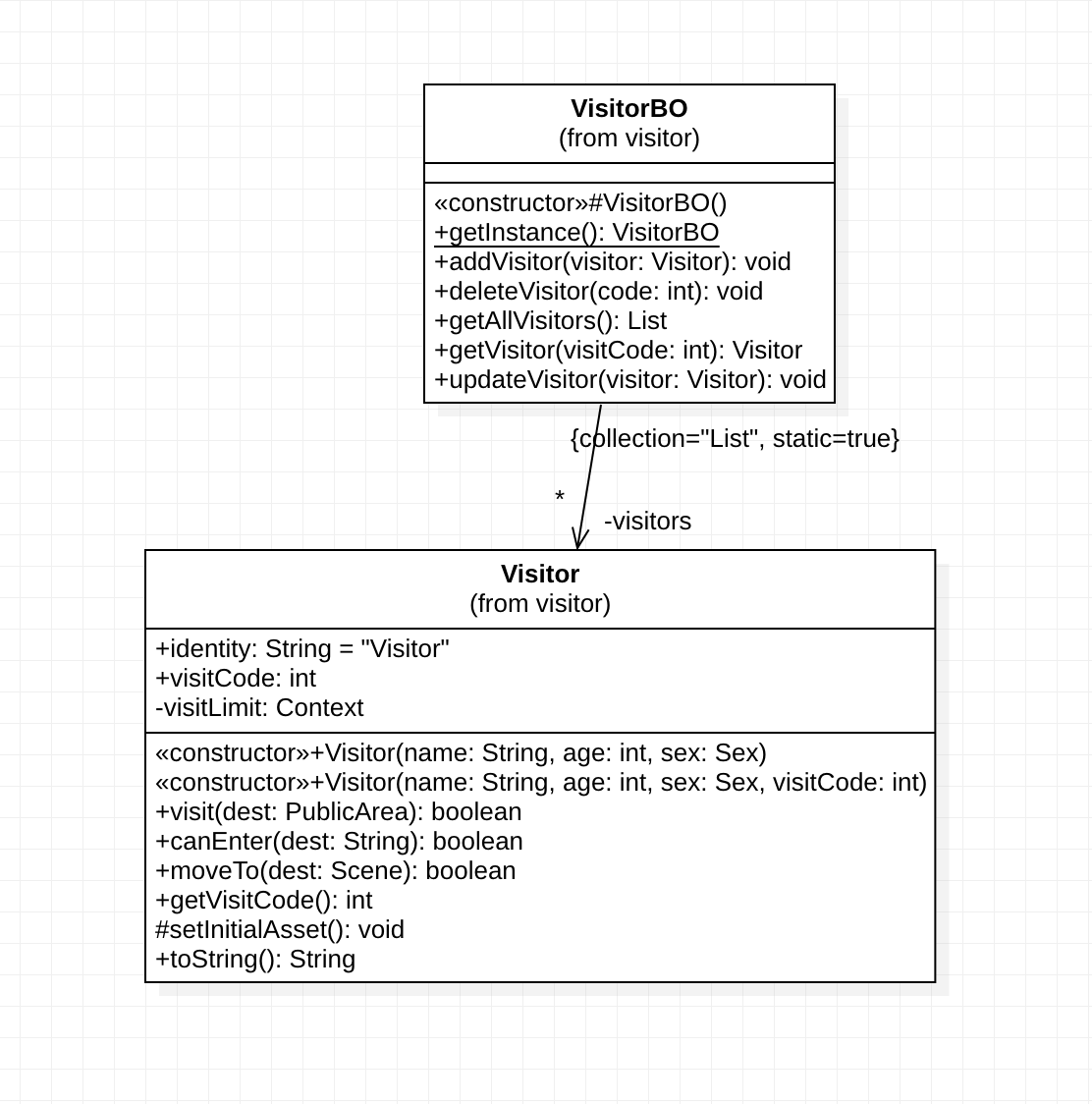
### 3.38.2 Product 实现API

* 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>