罐头加工厂

目录

[1罐头生产 5](#_Toc86870218)

[1.1外观模式 5](#_Toc86870219)

[1.1.1模式介绍 5](#_Toc86870220)

[1.1.2项目实例 5](#_Toc86870221)

[1.1.3模式接口 5](#_Toc86870222)

[1.1.4模式类图 6](#_Toc86870223)

[1.2解释器模式 6](#_Toc86870224)

[1.2.1模式介绍 6](#_Toc86870225)

[1.2.2项目实例 6](#_Toc86870226)

[1.2.3模式接口 6](#_Toc86870227)

[1.2.4模式类图 7](#_Toc86870228)

[1.3适配器模式 7](#_Toc86870229)

[1.3.1模式介绍 7](#_Toc86870230)

[1.3.2项目实例 7](#_Toc86870231)

[1.3.3模式接口 8](#_Toc86870232)

[1.3.4模式类图 8](#_Toc86870233)

[1.4延迟初始化模式 9](#_Toc86870234)

[1.4.1模式介绍 9](#_Toc86870235)

[1.4.2项目实例 9](#_Toc86870236)

[1.4.3模式接口 9](#_Toc86870237)

[1.4.4模式类图 9](#_Toc86870238)

[1.5黑板模式 10](#_Toc86870239)

[1.5.1模式介绍 10](#_Toc86870240)

[1.5.2项目实例 10](#_Toc86870241)

[1.5.3模式接口 10](#_Toc86870242)

[1.5.4模式类图 11](#_Toc86870243)

[1.6装饰器模式 11](#_Toc86870244)

[1.6.1模式介绍 11](#_Toc86870245)

[1.6.2项目实例 11](#_Toc86870246)

[1.6.3模式接口 12](#_Toc86870247)

[1.6.4模式类图 12](#_Toc86870248)

[1.7惰性初始模式 12](#_Toc86870249)

[1.7.1模式介绍 12](#_Toc86870250)

[1.7.2项目实例 13](#_Toc86870251)

[1.7.3模式接口 13](#_Toc86870252)

[1.7.4模式类图 14](#_Toc86870253)

[1.8双胞胎模式 14](#_Toc86870254)

[1.8.1模式介绍 14](#_Toc86870255)

[1.8.2项目实例 14](#_Toc86870256)

[1.8.3模式接口 15](#_Toc86870257)

[1.8.4模式类图 15](#_Toc86870258)

[1.9原型模式 15](#_Toc86870259)

[1.9.1模式介绍 15](#_Toc86870260)

[1.9.2项目实例 16](#_Toc86870261)

[1.9.3模式接口 16](#_Toc86870262)

[1.9.4模式类 17](#_Toc86870263)

[1.10空对象模式 17](#_Toc86870264)

[1.10.1模式介绍 17](#_Toc86870265)

[1.10.2项目实例 17](#_Toc86870266)

[1.10.3模式接口 18](#_Toc86870267)

[1.10.4模式类图 18](#_Toc86870268)

[1.11享元模式 19](#_Toc86870269)

[1.11.1模式介绍 19](#_Toc86870270)

[1.11.2项目实例 19](#_Toc86870271)

[1.11.3模式接口 19](#_Toc86870272)

[1.11.4模式类图 20](#_Toc86870273)

[1.12桥梁模式 20](#_Toc86870274)

[1.12.1模式介绍 20](#_Toc86870275)

[1.12.2项目实例 20](#_Toc86870276)

[1.12.3模式接口 21](#_Toc86870277)

[1.12.4模式类图 21](#_Toc86870278)

[1.13观察者模式 22](#_Toc86870279)

[1.13.1模式介绍 22](#_Toc86870280)

[1.13.2项目实例 22](#_Toc86870281)

[1.13.3模式接口 22](#_Toc86870282)

[1.13.4模式类图 23](#_Toc86870283)

[1.14命令模式 23](#_Toc86870284)

[1.14.1模式介绍 23](#_Toc86870285)

[1.14.2项目实例 23](#_Toc86870286)

[1.14.3模式接口 24](#_Toc86870287)

[1.14.4模式类图 24](#_Toc86870288)

[1.15抽象工厂模式 25](#_Toc86870289)

[1.15.1模式介绍 25](#_Toc86870290)

[1.15.2项目实例 25](#_Toc86870291)

[1.15.3模式接口 25](#_Toc86870292)

[1.15.4模式类图 26](#_Toc86870293)

[1.16工厂模式 26](#_Toc86870294)

[1.16.1模式介绍 26](#_Toc86870295)

[1.16.2项目实例 26](#_Toc86870296)

[1.16.3模式接口 27](#_Toc86870297)

[1.16.4模式类图 27](#_Toc86870298)

[1.17单例模式 28](#_Toc86870299)

[1.17.1模式介绍 28](#_Toc86870300)

[1.17.2项目实例 28](#_Toc86870301)

[2订单销售 28](#_Toc86870302)

[2.1中介模式 28](#_Toc86870303)

[2.1.1模式介绍 28](#_Toc86870304)

[2.1.2项目实例 28](#_Toc86870305)

[2.1.3模式接口 29](#_Toc86870306)

[2.1.4模式类图 29](#_Toc86870307)

[2.2状态模式 30](#_Toc86870308)

[2.2.1模式介绍 30](#_Toc86870309)

[2.2.2项目实例 30](#_Toc86870310)

[2.2.3模式接口 30](#_Toc86870311)

[2.2.4模式类图 31](#_Toc86870312)

[2.3迭代器模式 31](#_Toc86870313)

[2.3.1模式介绍 31](#_Toc86870314)

[2.3.2项目实例 32](#_Toc86870315)

[2.3.3模式接口 32](#_Toc86870316)

[2.3.4模式类图 33](#_Toc86870317)

[2.4组合模式 33](#_Toc86870318)

[2.4.1模式介绍 33](#_Toc86870319)

[2.4.2项目实例 33](#_Toc86870320)

[2.4.3模式接口 34](#_Toc86870321)

[2.4.4模式类图 34](#_Toc86870322)

[2.5建造者模式 34](#_Toc86870323)

[2.5.1模式介绍 34](#_Toc86870324)

[2.5.2项目实例 35](#_Toc86870325)

[2.5.3模式接口 35](#_Toc86870326)

[2.5.4模式类图 35](#_Toc86870327)

[2.6策略模式 36](#_Toc86870328)

[2.6.1模式介绍 36](#_Toc86870329)

[2.6.2项目实例 36](#_Toc86870330)

[2.6.3模式接口 36](#_Toc86870331)

[2.6.4模式类图 37](#_Toc86870332)

[2.7状态模式 37](#_Toc86870333)

[2.7.1模式介绍 37](#_Toc86870334)

[2.7.2项目实例 37](#_Toc86870335)

[2.7.3模式接口 37](#_Toc86870336)

[2.7.4模式类图 38](#_Toc86870337)

[2.8适配器模式 38](#_Toc86870338)

[2.8.1模式介绍 38](#_Toc86870339)

[2.8.2项目实例 38](#_Toc86870340)

[2.8.3模式接口 39](#_Toc86870341)

[2.8.4模式类图 40](#_Toc86870342)

[3人员管理 40](#_Toc86870343)

[3.1责任链模式 40](#_Toc86870344)

[3.1.1模式介绍 40](#_Toc86870345)

[3.1.2项目实例 41](#_Toc86870346)

[3.1.3模式接口 41](#_Toc86870347)

[3.1.4模式类图 42](#_Toc86870348)

[4设备管理 42](#_Toc86870349)

[4.1状态模式 42](#_Toc86870350)

[4.1.1模式介绍 42](#_Toc86870351)

[4.1.2项目实例 43](#_Toc86870352)

[4.1.3模式接口 43](#_Toc86870353)

[4.2访问者模式 43](#_Toc86870354)

[4.2.1模式介绍 43](#_Toc86870355)

[4.2.2项目示例 44](#_Toc86870356)

[4.2.3模式接口 44](#_Toc86870357)

[4.2.4模式类图 45](#_Toc86870358)

[5财务管理 45](#_Toc86870359)

[5.1代理模式 45](#_Toc86870360)

[5.1.1模式介绍 45](#_Toc86870361)

[5.1.2项目实例 46](#_Toc86870362)

[5.1.3模式接口 46](#_Toc86870363)

[5.1.4模式类图 47](#_Toc86870364)

[5.2备忘录模式 47](#_Toc86870365)

[5.2.1模式介绍 47](#_Toc86870366)

[5.2.2项目实例 48](#_Toc86870367)

[5.2.3模式接口 48](#_Toc86870368)

[5.2.4模式类图 49](#_Toc86870369)

# 1 项目简介

本项目采用c++代码编写，实现了对一个罐头工厂采购原材料、生产罐头、销售罐头的整个过程的模拟，还涵盖了整个罐头工厂的财务管理系统、人员管理系统和设备管理系统，将一个罐头工厂能够顺利运行所需要的所有要素都囊括在内，共实现了28个设计模式和3个板块的功能。

为了尽可能模拟真实的生产场景，我们设立了四种罐头：大肉罐头，小肉罐头，大水果罐头，小水果罐头，同时还为各种罐头进行定价并设定了生产周期，让整个罐头生产更有现实意义。

对于设计模式的实现，我们将采购原材料和生产罐头合并到一个功能点内，在这一功能点内实现了抽象工厂模式、工厂模式、命令模式、观察者模式、单例模式、包装模式、惰性工厂模式、双胞胎模式、享元模式、桥梁模式、原型模式、空对象模式、黑板模式、外观模式、适配器模式、解释器模式、延迟初始化模式；在销售罐头这一功能点中，我们实现了策略模式、组合模式、适配器模式、建造者模式、状态模式、迭代器模式、中介者模式；在财务管理系统这一功能点中，我们实现了备忘录模式和代理模式；在人员管理功能点中，我们实现了责任链模式；在设备管理功能点中，我们实现了状态模式和访问者模式。而模板模式通过代码中的随处可见的继承就得到了体现。至此，我们的项目实现了所有要求的设计模式外加5个额外的设计模式。

同时，我们的项目基本实现了可扩展性，如果未来出现了新的种类的罐头，在生产罐头这一功能点内利用工厂模式给出了相关处理方法，在后续销售功能点内也可以通过添加子类的方式进行更新。

我们的各个设计模式之间有着有机的结合，多个设计模式交织在一起，彼此融合，相辅相成，通过高效、简洁的方式模拟设计模式的执行以及实现罐头加工厂的功能。比如：在罐头类的构造函数中我采用了适配器，适配器模式一般需要有一个其他类的成员变量，而我将这一成员变量声明为一个父类的指针，在适配器的实际使用时对其赋子类的实例，从而实现在适配器模式中嵌套策略模式，而这样的嵌套使用设计模式在我们的项目中还有很多体现，让整个项目更加有合理性。

项目我们进行了完整的架构和文件组织，在vs中的项目列表中，将各种同一设计模式的文件整合到了一起，方便对于设计模式的学习和讨论，大量的设计模式采用了cpp和h文件分离的方式进行组织，使得代码的可读性增强，方便错误检查。

在项目中，我们贯彻编写注释的操作，每一个函数、每一个类的前面都写出了对应的注解，帮助阅读者理解我们的代码，在代码的关键部分，我们也补充了帮助理解的提示，注释出了函数名称、参数及作用，增强可读性。

同时整个项目的制作周期内，我们采取了一部分敏捷管理思想，在分组的时候采取自组队的方式，平时更加熟悉的成员会分配到一组，并且保证“结对编程”，保证编写代码时可以互相监督互相提供建议，对于工作量较大的罐头生产这一功能点我们分配了五个人共同完成，而其他几个功能点由两个二人小组完成，每周会开会，各个小组分享各自的项目进展，这样的分配方式使得我们的程序编写效率和准确度大大提升；同时我们在整个项目开始之前进行了类图、程序框架和接口等的设计，做到了“planning”，使得我们的项目在整个编写过程中都有所依据有所参考，提前确定各自的类名和接口名也可以防止某个功能点当需要调用另一个功能点的函数时由于不知道类名和函数名而停滞；在整个项目结束之后交叉检查各自的代码并进行二次改进，做到了“review”，有助于查漏补缺，正所谓旁观者清，可以更加有效找出程序中不合逻辑之处。

# 2 模式汇总表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | **设计模式名称** | **文件** | **功能/函数体现** | **备注说明** |
| 1 | Abstract Factory Pattern |  |  |  |
| 2 | Factory Pattern |  |  |  |
| 3 | Command Pattern |  |  |  |
| 4 | Observer Pattern |  |  |  |
| 5 | Singleton Pattern |  |  |  |
| 6 | Composite Pattern |  |  |  |
| 7 | Decorator Pattern |  |  |  |
| 8 | Flyweight Pattern |  |  |  |
| 9 | Bridge |  |  |  |
| 10 | Prototype Pattern |  |  |  |
| 11 | Façade Pattern |  |  |  |
| 12 | Adapter Pattern |  |  |  |
| 13 | Interpreter Pattern |  |  |  |
| 14 | Builder Pattern |  |  |  |
| 15 | Strategy Pattern |  |  |  |
| 16 | State Pattern |  |  |  |
| 17 | Iterator Pattern |  |  |  |
| 18 | Mediator Pattern |  |  |  |
| 19 | Memento Pattern |  |  |  |
| 20 | Chain of Responsibility Pattern |  |  |  |
| 21 | Visitor Pattern |  |  |  |
| 22 | Proxy Pattern |  |  |  |
| 23 | Template Pattern |  |  |  |
| 24 | NullObject Pattern |  |  |  |
| 25 | Lazy Initialization Pattern |  |  |  |
| 26 | Blackboard Pattern |  |  |  |
| 27 |  |  |  |  |
| 28 |  |  |  |  |

# 3 罐头生产

## 1.1外观模式

### 1.1.1模式介绍

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。

### 1.1.2项目实例

在生产设备子系统中，外观模式用于在生产设备启动时的控制。客户与系统之间加一个外观层，外观层处理系统的调用关系、依赖关系等。以下实例以生产设备的启动过程为例，客户端只关心电脑开机的、关机的过程，并不需要了解生产设备内部子系统的启动过程。

### 1.1.3模式接口

**Equipment**

|  |  |
| --- | --- |
| **函数** | **功能** |
| Equipment() | Equipment罐头生产设备构造函数 |
| void start() | 生产设备启动 |
| void shutdown() | 生产设备关闭 |

**Control**

|  |  |
| --- | --- |
| virtual void start() = 0 | 纯虚函数启动接口，由子类实现 |
| virtual void shutdown() = 0 | 纯虚函数关闭接口，由子类实现 |

**Host**

|  |  |
| --- | --- |
| void start() | 主机子控件启动 |
| void shutdown() | 主机子控件关闭 |

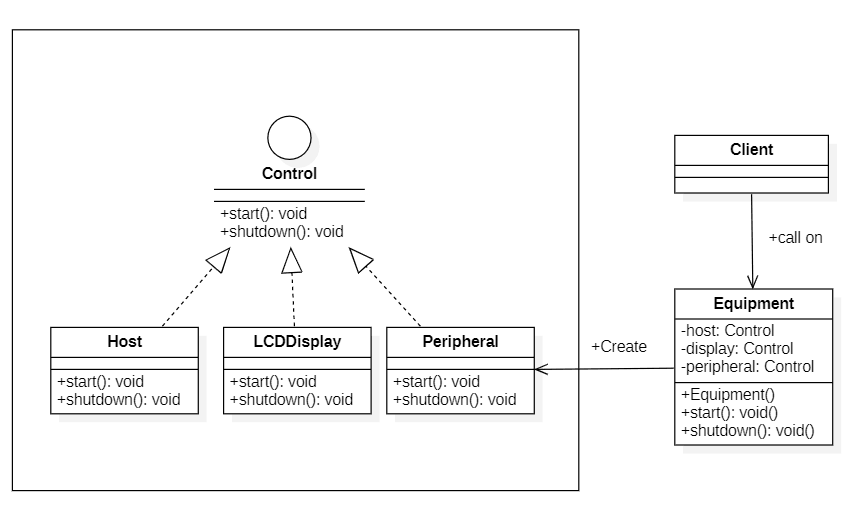
**LCDDisplay**

|  |  |
| --- | --- |
| void start() | 显示屏子控件启动 |
| void shutdown() | 显示屏子控件关闭 |

**Peripheral**

|  |  |
| --- | --- |
| void start() | 外部设备子控件启动 |
| void shutdown() | 外部设备子控件关闭 |

### 1.1.4模式类图



## 1.2解释器模式

### 1.2.1模式介绍

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

### 1.2.2项目实例

在解释器模式中，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。通过解释器模式，验证生产设备是否存在，对其存在性进行判断。

### 1.2.3模式接口

**ContextInterpreter**

|  |  |
| --- | --- |
| **函数** | **功能** |
| ContextInterpreter() | 上下文全局信息类构造函数 |
| void IsFree(const std::string& info) | 验证生产设备是否可用 |

**Expression**

|  |  |
| --- | --- |
| virtual bool Interpret(const std::string& info) = 0 | 抽象表达式类纯虚函数，解析表达式 |

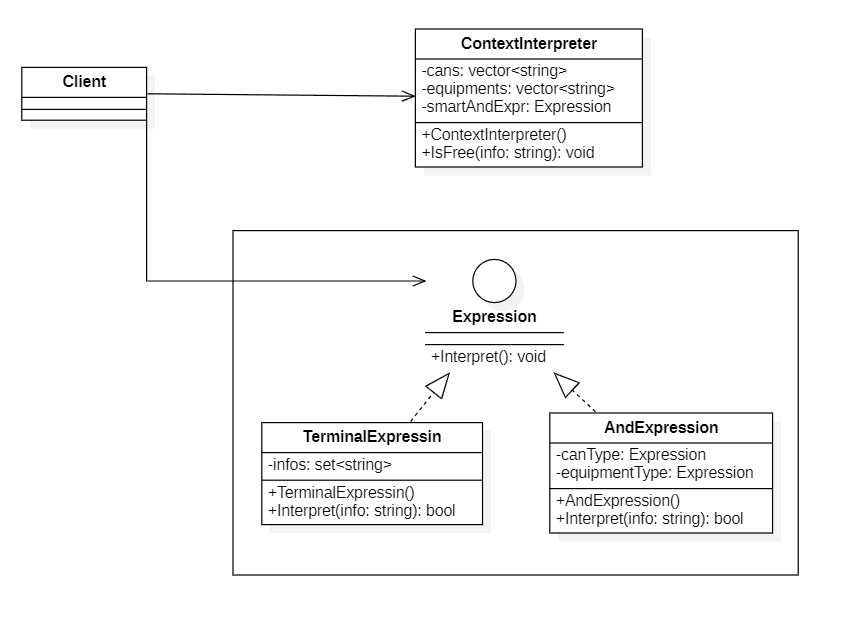
**TerminalExpression**

|  |  |
| --- | --- |
| TerminalExpressin(const std::vector<std::string> datas) | 终结符表达式类构造函数 |
| bool Interpret(const std::string& info) | 终结符表达式类解释功能具体实现，判断是否在终结符数据集合中 |

**AndExpression**

|  |  |
| --- | --- |
| AndExpression(std::shared\_ptr<Expression> city, std::shared\_ptr<Expression> person) | 非终结符表达式类构造函数 |
| bool Interpret(const std::string& info) | 非终结符表达式类解释功能具体实现，接收需要解析的表达式，并添加AND处理逻辑 |

### 1.2.4模式类图



## 1.3适配器模式

### 1.3.1模式介绍

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。

### 1.3.2项目实例

适配器模式通过将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。适配器模式分为类结构型模式和对象结构型模式两种，前者类之间的耦合度比后者高。在本项目中，适配器模式通过适配器使得生产设备能够处理客户端的特殊请求。

### 1.3.3模式接口

**Target**

|  |  |
| --- | --- |
| **函数** | **功能** |
| virtual void request() = 0 | 纯虚函数接口，由子类实现接收请求功能 |

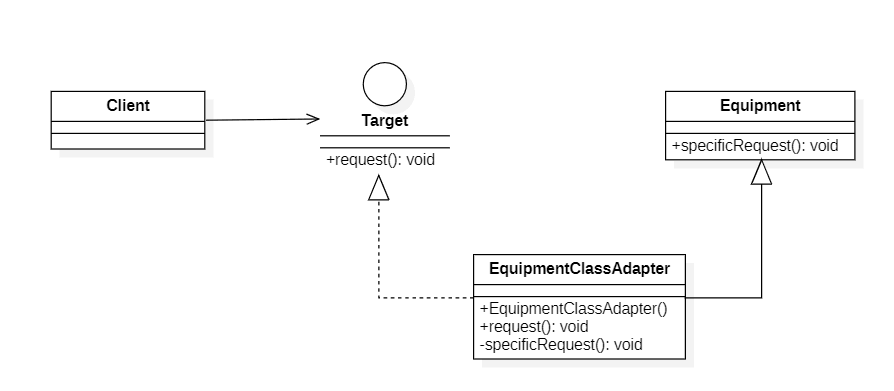
**EquipmentClassAdapter**

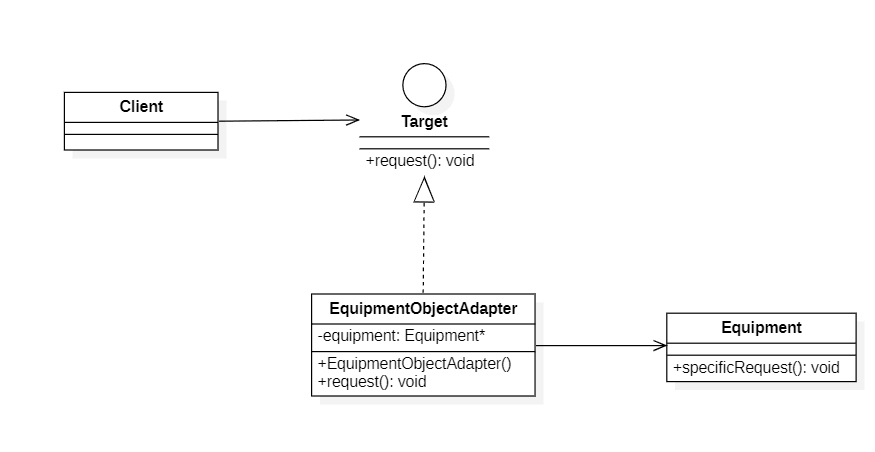
|  |  |
| --- | --- |
| EquipmentClassAdapter() | EquipmentClassAdapter构造类 |
| void request() | 类适配器接收请求，并通过调用specificRequest()实现对Equipment的适配 |
| void specificRequest() | 私有继承Equipment的specificRequest()函数，调用此函数以实现适配功能 |

**Equipment**

|  |  |
| --- | --- |
| void specificRequest() | 生产设备对接收到的请求进行特例处理 |

### 1.3.4模式类图





## 1.4延迟初始化模式

### 1.4.1模式介绍

在计算机编程中，延迟初始化是一种策略，它将对象的创建、值的计算或其他一些代价高昂的进程延迟到第一次需要它的时候。它是一种惰性计算，专门指对象或其他资源的实例化。

这通常是通过增加访问器方法(或属性getter)来检查充当缓存的私有成员是否已经初始化。如果有，就会立即退回。如果不是，则创建一个新实例，放在成员变量中，并在第一次使用时及时返回给调用者。

如果对象有很少使用的属性，这可以提高启动速度。意味着平均程序性能可能稍差的记忆(条件变量)和执行周期(检查),但对象实例化的影响在传播时间(“分期”),而不是集中在系统的启动阶段,因此可以大大提高平均响应时间。

在多线程代码中，对惰性初始化对象/状态的访问必须同步，以防止出现竞争条件。

### 1.4.2项目实例

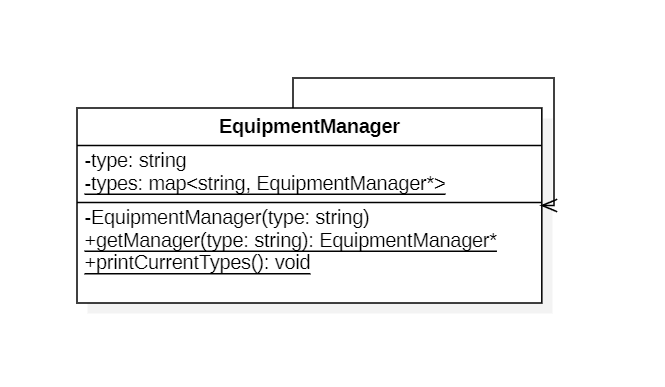
一个对象的延迟初始化意味着该对象的创建将会延迟至第一次使用该对象时。延迟初始化主要用于提高性能，避免浪费计算，并减少程序内存要求。 在本项目中，延迟初始化被用于设备管理员的生成中，当多次生成设备管理员时，会返回第一次生成的实例。

### 1.4.3模式接口

**EquipmentManager**

|  |  |
| --- | --- |
| **函数** | **功能** |
| EquipmentManager(const std::string& type) | EquipmentManager私有构造函数 |
| static EquipmentManager\* getManager(const std::string& type) | EquipmentManager延迟初始化函数，通过map初始化并存储不同类型的设备管理员 |
| static void printCurrentTypes() | 查看已初始化的设备管理员 |

### 1.4.4模式类图



## 1.5黑板模式

### 1.5.1模式介绍

“黑板”（Blackboard）模式，基于一种很直观的概念，就是一群人为了解决一个问题，在黑板前聚集。每个人都可以发表自己的意见，然后在黑板上写下自己的看法，也可以基于别人记录在黑板上的看法，来发表和更新自己的看法，在这样不断的意见交换，看法更新的过程中，越来越趋向于对于问题的最终解答。  
 一开始的黑板模式就是这样一个由多个子系统来共同协作的人工智能解决方案。

黑板模式是一种常用的架构模式，应用中的多种不同数据处理逻辑相互影响和协同来完成数据分析处理。就好像多位不同的专家在同一黑板上交流思想，每个专家都可以获得别的专家写在黑板上的信息，同时也可以用自己的分析去更新黑板上的信息，从而影响其它专家。

### 1.5.2项目实例

黑板模式是观察者模式的一个扩展,知名度并不高,但使用的范围却非常广. 其意图为: 允许消息的读写同时进行, 广泛的交互消息.

简单的说, 黑板模式允许多个消息读写者同时存在, 消息的生产者和消费者完全分开. 这就像一个黑板, 任何一个老师(消息的生产者)都可以在上面书写消息, 任何一个学生(消息的消费者)都可以从黑板上读取消息, 两者在空间和时间上可以解耦, 互不干扰.

在本项目实例中，黑板模式为生产设备添加维修信息，黑板模式中存储生产设备是否待维修，使得生产设备管理员能够查看生产设备的维修信息，并进行处理。

### 1.5.3模式接口

**BlackBoard**

|  |  |
| --- | --- |
| **函数** | **功能** |
| access () | 获取黑板中的数据 |
| update () | 更新黑板中的数据 |
| void setValue(std::string key, bool value) | 在黑板中设置bool值 |
| void setValue(std::string key, bool value, float expiredTime, bool expiredValue) | 在黑板中设置bool值，添加额外条件 |
| bool getBool(std::string key) | 在黑板中获取bool值 |
| std::string parseBool(bool value) | 解析黑板中获得的bool值 |
| void showBlackBoard() | 展示黑板中所写的数据 |

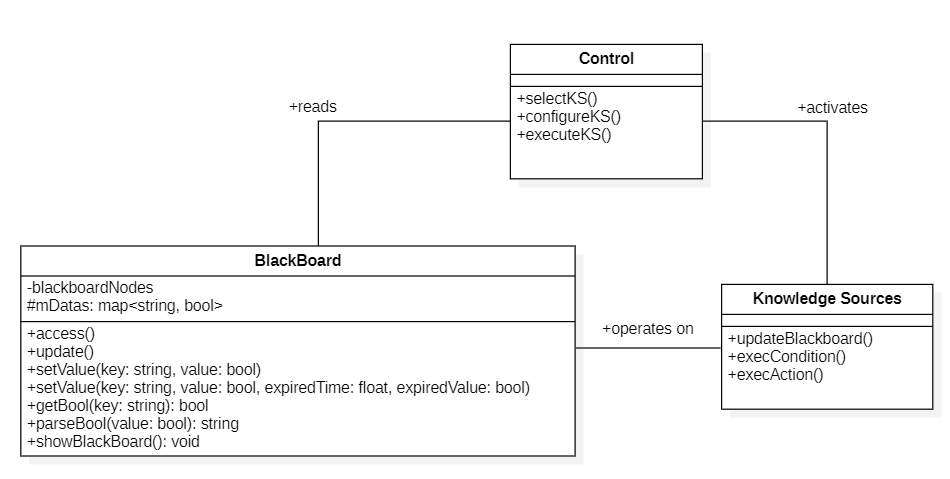
**Control**

|  |  |
| --- | --- |
| selectKS () | 搜索Knowledge Sources数据 |
| configureKS () | 配置Knowledge Sources数据 |
| executeKS () | 执行Knowledge Sources数据 |

**Knowledge Sources**

|  |  |
| --- | --- |
| updateBlackboard () | 更新黑板中的数据 |
| execCondition () | 添加Knowledge Sources数据执行条件 |
| execAction () | 添加Knowledge Sources数据执行行为 |

### 1.5.4模式类图



## 1.6装饰器模式

### 1.6.1模式介绍

装饰器模式主要是允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。通过使用装饰器模式，可以在运行时扩充一个类的功能。这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

代码的具体实现主要是：增加一个修饰类包裹原来的类，包裹的方式是在修饰类的构造函数中将原来的类以参数的形式传入。装饰类实现新的功能，但是，在不需要用到新功能的地方，它可以直接调用原来的类中的方法。修饰类必须和原来的类有相同的接口。

### 1.6.2项目实例

在罐头加工实例中，通过装饰器模式实现了对于不同种类的罐头的加工和包装。其中，我们将一个种类的罐头包装成大、小两种不同的规格，同时不会改变罐头的种类。 罐头类（Can）类充当抽象角色，没有具体实现。 包装类（Can\_pack）引用和继承 Can类，具体扩展类（小罐头包装类、大罐头包装类）重写父类（Can\_pack类）方法。将罐头类的具体实现（肉罐头类、水果罐头类）作为参数传入到Can\_pack（包装类）的子类，该类调用类成员方法对罐头类进行包装，分装成大罐头和小罐头。该操作实现了装饰器模式的基本功能，符合装饰器模式的设计意图。

### 1.6.3模式接口

|  |  |
| --- | --- |
| **Can** | |
| show() | 用于展示罐头的种类 |

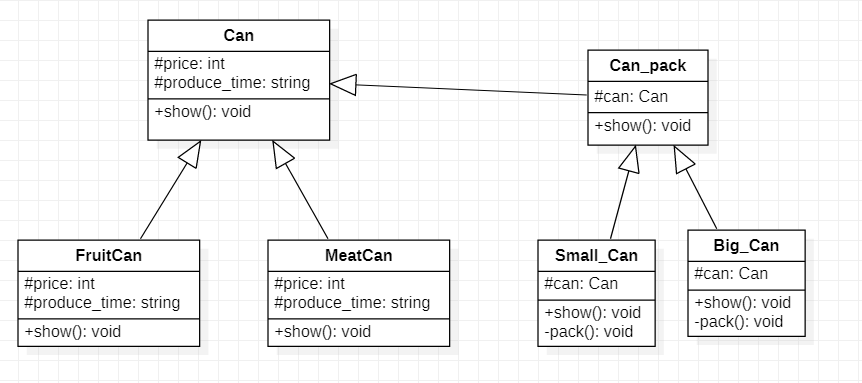
|  |  |
| --- | --- |
| **Can\_pack** | |
| Can\_pack(\*Can) | 构造函数，对某一个种类的罐头进行包装 |
| show() | 用于展示包装过后的罐头的种类 |

|  |  |
| --- | --- |
| **Small\_Can** | |
| Small\_Can(\*Can) | 构造函数，对某一个种类的罐头进行包装 |
| show() | 用于展示包装过后的罐头的种类 |
| pack() | 用于将罐头包装成小罐头 |

|  |  |
| --- | --- |
| **Big\_Can** | |
| Big\_Can(\*Can) | 构造函数，对某一个种类的罐头进行包装 |
| show() | 用于展示包装过后的罐头的种类 |
| pack() | 用于将罐头包装成大罐头 |

|  |  |
| --- | --- |
| **FruitCan** | |
| show() | 展示水果罐头种类 |

### 1.6.4模式类图



## 1.7惰性初始模式

### 1.7.1模式介绍

在程式设计中, 惰性初始是一种拖延战术。在第一次需求出现以前，先延迟创建物件、计算值或其它昂贵程序。

这通常是以一个旗号来实现，用旗号来标示是否完成其程式。每次请求对象时，会先测试此旗号。如果已完成，直接传回，否则当场执行。对指令式语言，这个模式可能潜藏着危险，尤其是使用共享状态的程式习惯。

以设计模式的观点来说，惰性初始通常会和工厂方法模式合作，这结合了三种构想：

使用一个工厂去得到一个类别的实例（工厂方法模式）。

2）将实例存在一个集合中，所以下次要求一个实例却有相同参数时，可以得到同一个实例（可和单例模式来做比较）。

3）在第一次时，使用惰性初始来实例化物件（惰性初始模式）

### 1.7.2项目实例

在罐头加工实例中，通过惰性工厂模式实现了对于不同种类的罐头的进一步细分种类。其中，我们先用MeatCan、FruitCan和VegCan实现Can接口，罐头（Can）类充当抽象角色，没有具体实现。再用KindOfXXXCan分别实现以上三个接口。每个类中设置两个私有成员变量type（string）和types（map），分别用于存储罐头的具体细化种类和细化种类表，私有构造函数KindOfXXXCan（kind）用于得到具体种类的罐头。当调用getXXXCan（kind）获取某种类的罐头时，调用构造函数，对该罐头的具体种类进行检测，如果存在，则生成实例，如果不存在，则创建该类型的罐头，并将其加入到types的map中，以此实现惰性初始化。该操作实现了惰性初始模式的基本功能，符合装惰性初始模式的设计意图。

### 1.7.3模式接口

|  |  |
| --- | --- |
| **Can** | |
| show() | 用于展示罐头的种类 |

|  |  |
| --- | --- |
| **MeatCan** | |
| show() | 用于展示罐头的种类 |

|  |  |
| --- | --- |
| **FruitCan** | |
| show() | 用于展示罐头的种类 |

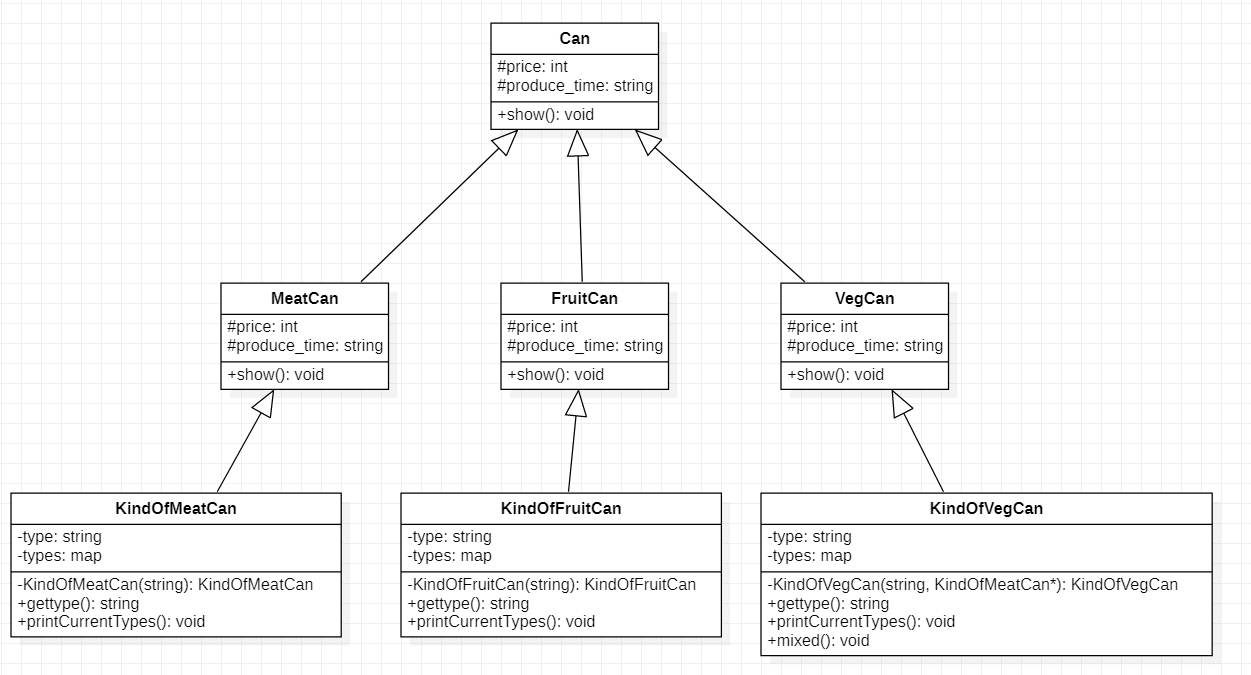
|  |  |
| --- | --- |
| **VegCan** | |
| show() | 用于展示罐头的种类 |

|  |  |
| --- | --- |
| **KindOfMeatCan** | |
| KindOfMeatCan(string) | 私有构造函数，创建某种类型肉罐头 |
| gettype() | 获取罐头的类型 |
| printCurrentTypes() | 输出当前罐头的所有类型 |

|  |  |
| --- | --- |
| **KindOfFruitCan** | |
| KindOfFruitCan(string) | 私有构造函数，创建某种类型的水果罐头 |
| gettype() | 获取罐头的类型 |
| printCurrentTypes() | 输出当前罐头的所有类型 |

|  |  |
| --- | --- |
| **KindOfVegCan** | |
| KindOfVegCan(string) | 私有构造函数，创建某种类型蔬菜罐头 |
| gettype() | 获取罐头的类型 |
| printCurrentTypes() | 输出当前罐头的所有类型 |

### 1.7.4模式类图



## 1.8双胞胎模式

### 1.8.1模式介绍

在软件工程中，Twin模式是一种软件设计模式，它允许开发人员在不支持多重继承的编程语言中对多重继承建模。这种模式避免了多重继承的许多问题

代码的具体实现主要是：将有两个或更多的父类用于继承。有多个子类，每个子类都是从一个父类派生出来的。子类通过字段相互链接，每个子类都可以覆盖从父类继承的方法。新方法和字段通常在一个子类中声明。

### 1.8.2项目实例

在罐头加工实例中，通过双胞胎模式实现了对于不同种类罐头的组合（如牛肉罐头和竹笋罐头组成了混合类型的罐头）。有两个父类MeatCan和VegCan，KindOfMeatCan和KindOfVegCan分别继承两个父类。在KindOfVegCan中，将KindOfMeatCan的指针声明为成员变量，通过构造函数为KindOfVegCan和KindOfMeatCan赋值，使其获得具体细化的种类。通过KindOfVegCan的成员函数mixed（）将两种罐头进行混合，生产出组合类型的罐头。该操作实现了双胞胎模式的基本功能，符合双胞胎模式的设计意图。

### 1.8.3模式接口

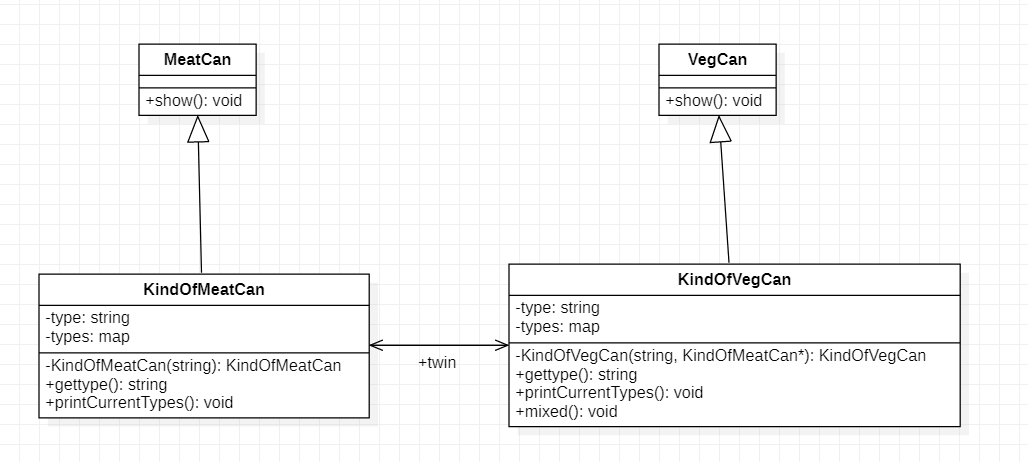
|  |  |
| --- | --- |
| **MeatCan** | |
| show() | 用于展示罐头的种类 |

|  |  |
| --- | --- |
| **VegCan** | |
| show() | 用于展示罐头的种类 |

|  |  |
| --- | --- |
| **KindOfMeatCan** | |
| KindOfMeatCan(string) | 私有构造函数，创建某种类型肉罐头 |
| gettype() | 获取罐头的类型 |
| printCurrentTypes() | 输出当前罐头的所有类型 |

|  |  |
| --- | --- |
| **KindOfVegCan** | |
| KindOfVegCan(string，KindOfMeatCan\*) | 私有构造函数，创建某种类型蔬菜罐头 |
| gettype() | 获取罐头的类型 |
| printCurrentTypes() | 输出当前罐头的所有类型 |
| mixed() | 将不同类型的罐头进行组合 |

### 1.8.4模式类图



## 1.9原型模式

### 1.9.1模式介绍

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。

### 1.9.2项目实例

在本项目中，我们在生产了足够多的原料和罐头外壳之后需要对它们进行加工包装。由于这种工作是大量且重复的，需要创建大量的相同对象，同时也需要经常调整这些对象的属性（如罐头种类及大小），因此采用原型模式能够大大减少创建对象的代价。

相应的具体代码表现为：设定好罐头包装的属性有大小以及重诶，packAbsPrototype为抽象父类，packConcPrototype继承了这个父类，每次创建调用构造函数，当需要创建大量重复对象时可以使用clone函数。此外，我们也可以使用setSize或setType函数调整罐头包装的属性以更改包装方案。

### 1.9.3模式接口

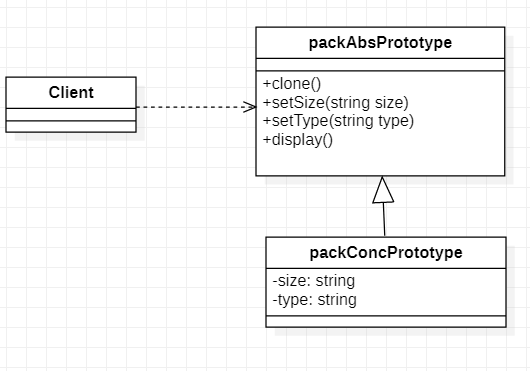
**packAbsPrototype**

|  |  |
| --- | --- |
| **函数** | **功能** |
| packAbsPrototype() | 构造函数 |
| ~packAbsPrototype | 析构函数 |
| virtual packAbsPrototype\* clone() | 虚函数，用于创建已有对象的克隆 |
| virtual void setSize(string size) | 虚函数，设置包装罐头的大小 |
| virtual void setType(string type) | 虚函数，设置包装罐头的种类 |
| virtual void display() | 虚函数，每包装好一个罐头以展示 |

**packConcPrototype**

|  |  |
| --- | --- |
| **函数** | **功能** |
| packAbsPrototype() | 构造函数 |
| ~ packAbsPrototype | 析构函数 |
| packAbsPrototype\* clone() | 核心函数，用于创建已有对象的克隆 |
| void setSize(string size) | 设置包装罐头的大小 |
| void setType(string type) | 设置包装罐头的种类 |
| void display() | 展示函数，每包装好一个罐头以输出 |

### 1.9.4模式类



## 1.10空对象模式

### 1.10.1模式介绍

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

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

### 1.10.2项目实例

在本项目中，罐头生产加工完毕后需要向指定的采购商供货，因此我们需要检查采购商清单并将货物送往这些地方。这里我们就可以使用空对象模式。使得那些不在我们清单上的采购商也能够有一些默认的操作。

具体代码实现为：先使用BuyerFactory存储一个采购商清单，可以使用getBuyer函数检验某个采购商是否在清单上。我们的采购商由RealBuyer和NullBuyer组成，属于哪一种取决于它们的名称是否在BuyerFactory所产生的清单上，若经过getBuyer检验不在，则isNull返回true，确认其为空对象，此时我们采取默认操作如将其加入清单或要求其预约等。

### 1.10.3模式接口

**BuyerFactory**

|  |  |
| --- | --- |
| **函数** | **功能** |
| BuyerFactory() | 构造函数，构造一个采购商工厂 |
| ~BuyerFactory() | 析构函数 |
| AbstractBuyer\* getBuyer() | 检测并选择返回采购商类型（存在与否） |

**AbstractBuyer**

|  |  |
| --- | --- |
| **函数** | **功能** |
| AbstractBuyer() | 构造函数，构造一个抽象采购商 |
| virtual ~AbstractBuyer() | 虚函数，析构函数 |
| virtual bool isNull() | 虚函数，检测采购商（是否存在清单上） |
| virtual string getName() | 虚函数，输出采购商名或提示信息 |

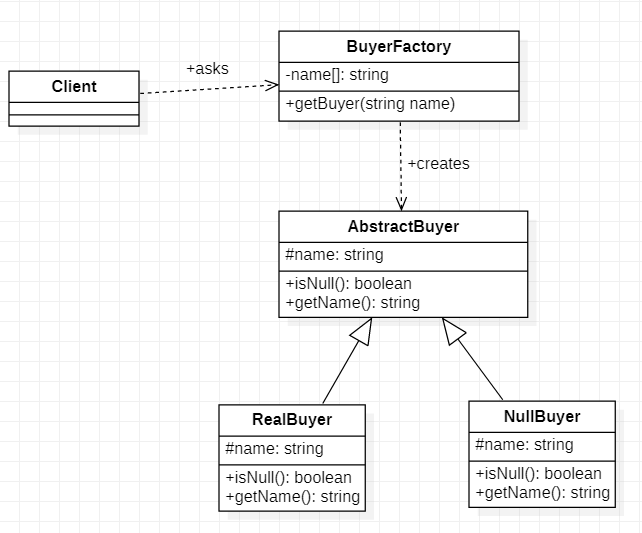
**RealBuyer**

|  |  |
| --- | --- |
| **函数** | **功能** |
| RealBuyer() | 构造函数，构造一个真实采购商 |
| ~RealBuyer() | 析构函数 |
| bool isNull() | 返回True，表示在清单上 |
| string getName() | 输出采购商名称表示在清单上 |

**NullBuyer**

|  |  |
| --- | --- |
| **函数** | **功能** |
| NullBuyer() | 构造函数，构造一个不存在的采购商 |
| ~NullBuyer() | 析构函数 |
| bool isNull() | 返回False，表示不在清单上 |
| string getName() | 输出提示信息表示不在清单上，此处也可进行一些空对象默认操作如添入采购商清单等。 |

### 1.10.4模式类图



## 1.11享元模式

### ****1.11.1模式介绍****

享元模式（flyweight pattern）使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件；它适合用于当大量物件只是重复因而导致无法令人接受的使用大量内存。通常物件中的部分状态是可以分享。常见做法是把它们放在外部数据结构，当需要使用时再将它们传递给享元。其关键在于减少内存，共享物件，外部数据。

代码的具体实现主要是有一个FlyweightFactory类，一个Flyweight类以及一个继承于Flyweight类的具体子类。享元工厂用来创建并管理Flyweight对象，它主要是用来确保合理地共享Flyweight，当用户请求一个Flyweight时，FlyweightFactory对象提供一个已创建的实例或者创建一个（如果不存在的话），Flyweight类则为所有具体享元类的超类或者接口，通过这个接口，Flyweight可以接受并作用于外部状态。Flyweight的具体子类则是实现Flyweight接口并为其内部状态增加存储空间。

### 1.11.2项目实例

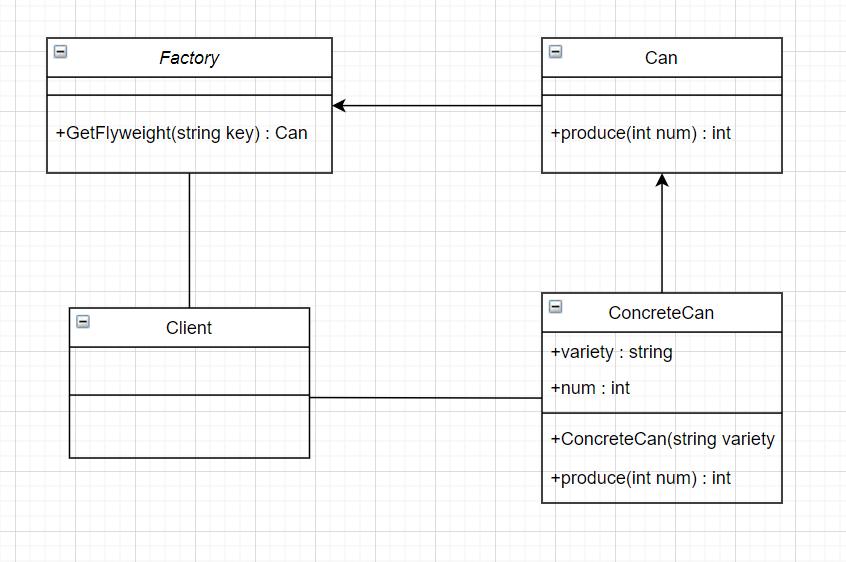
在罐头工厂项目中，享元模式用于最初生产不同种类的罐头。Factory中可以生产不同菜品种类不同大小的罐头。一旦需求某种罐头，Factory就通过GetFlyweight会去匹配相应种类的罐头，如果此种罐头不存在，则会生产出此种罐头；若此种罐头已经存在，则会返回这个已经存在的罐头。本项目利用享元模式生产相似的罐头产品，共享了物件，实现了享元的基本功能，符合享元模式的设计意图。

### 1.11.3模式接口

**Factory**

|  |  |
| --- | --- |
| Factory() | 构造函数，生产对应种类的罐头 |
| Can\* GetFlyweights(string key) | 应用哈希表，key为所需生产罐头的种类，通过键值对应调用相应的ConcreteCan实现生产所需罐头的行为 |

### 1.11.4模式类图



## 1.12桥梁模式

### 1.12.1模式介绍

桥梁模式（bridge pattern）将抽象部分和它的实现部分分离，使它们都可以独立的变化。简单粗暴的说，就是抽象对外提供调用的接口；对外隐瞒实现部分，在抽象中引用实现部分，从而实现抽象对实现部分的调用，而抽象中引用的实现部分可以在今后的开发过程中，切换成别的实现部分。

即桥梁模式的意图为：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。

其适用性为：你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时Bridge 模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

代码的具体实现主要为Abstraction类定义了抽象类的接口，并且维护一个指向Implementor实现类的指针；RefineAbstraction类扩充了Abstraction类的接口；Implementor类定义了实现类的接口，这个接口不一定要与Abstraction的接口完全一致；实际上，这两个接口可以完全不同；ConcreteImplementor类实现了Implementor定义的接口。

### 1.12.2项目实例

在罐头工厂项目中，桥梁模式主要实现生产新增新的种类以及新的大小的罐头。有两个新增的维度，分别为大小以及菜品种类。在Factory中可以新增MiddleCan中等型号罐头，在新增完罐头之后可以在Kind的具体子类中，如PorkKind,BeefKind中生产具体的对应的菜品种类的罐头。

### 1.12.3模式接口

**MiddleCan**

|  |  |
| --- | --- |
| Void produceSize(Kind \*kind) | 实现上层基类的纯虚函数，获取所需生产罐头的菜品样式，调用Kind中的produceKind()函数 |

**Kind**

|  |  |
| --- | --- |
| Void produceKind() | 纯虚函数，由子类实现不同的动作，用于Bridge pattern |

**FishKind**

|  |  |
| --- | --- |
| Void produceKind() | 实现上层基类的纯虚函数，根据Can的所需size，生产对应大小的鱼罐头 |

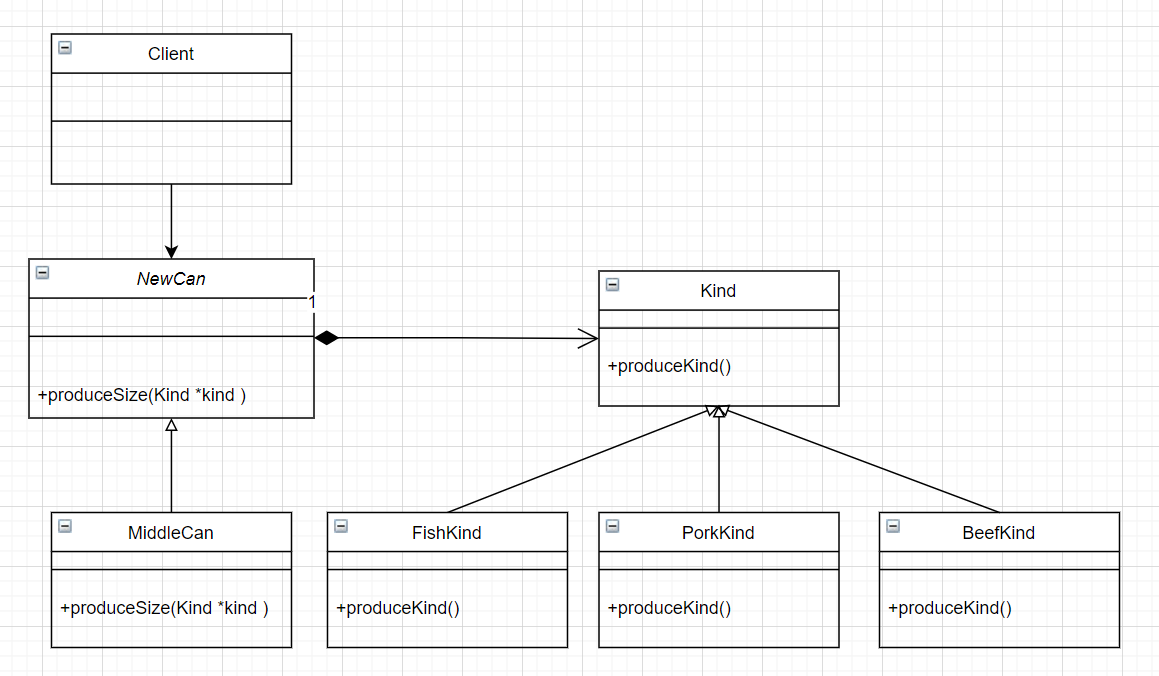
**PorkKind**

|  |  |
| --- | --- |
| Void produceKind() | 实现上层基类的纯虚函数，根据Can的所需size，生产对应大小的猪肉罐头 |

**BeefKind**

|  |  |
| --- | --- |
| Void produceKind() | 实现上层基类的纯虚函数，根据Can的所需size，生产对应大小的牛肉罐头 |

### 1.12.4模式类图



## 1.13观察者模式

### 1.13.1模式介绍

观察者模式主要用于对象间存在一对多关系的情况，当一个对象被修改时，与它有关系的对象都会需要做出对应的应答，这时就需要用到观察者模式，它是一种行为型模式。

通俗的讲，观察者模式就是将对象间的依赖关系通过隐性的类的设计表现出来，将逻辑上的依赖关系转变成物理上的依赖关系，并在程序的运行过程中得到体现。

代码的具体实现主要是有一个subject类、一个observe类以及继承于observe类的具体的子类。Observe的父子类都含有一个update函数，当这个函数被调用时就会执行对应的observe类的功能，也就是执行有关对象的应答，observe父类作为抽象类出现，用于在subject类中方便的对所有不同类型的observe进行统一的管理。subject就是被观察者，它的动作影响着所有与它连接的observe的动作。其中含有attach函数用于与observe作连接，含有dettach函数用来取消连接，含有notify函数用于调用所有已连接的observe的update。

### 1.13.2项目实例

在罐头加工厂项目中，观察者模式用于对于工厂购买原料时的订单处理，subject中可以创建订单，订单一旦被创建，就会调用notify函数从而与观察者进行交互。项目中对应的观察者是存储原料的仓库，当订单被创建时，就会进行购买，也就是向原料仓库中添加对应订单中物品数量的货物。这一过程用户需要操作的只是创建订单这一步，而后续的操作都是程序自身逻辑下发生的，实现了观察者的基本功能，符合观察者模式的设计意图。

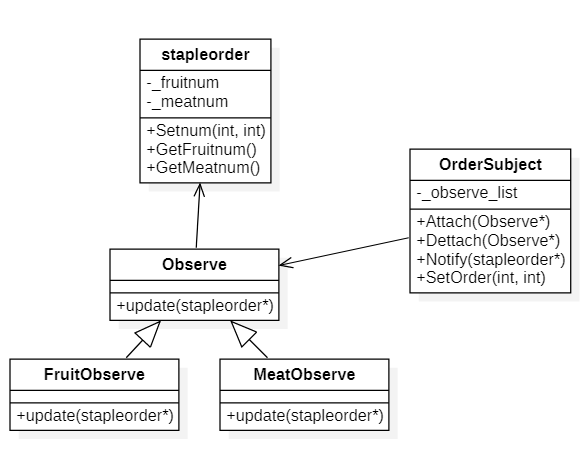
### 1.13.3模式接口

|  |  |
| --- | --- |
| **Observe** | |
| Update（stapleorder\* ） | 观察者进行更新，并执行其中的动作 |

|  |  |
| --- | --- |
| **OrderSubject** | |
| Attach（Observe\* addone） | 将一个新的observe添加到observe列表中 |
| Dettach（Observe\*delone） | 从observe列表中删除一个特定的列表 |
| Notify（stapleorder\* order） | 将order传给observe列表中的每一个observe，并且使之进行对应的update |
| SetOrder（int fnum，int mnum） | 生成订单，其中水果和肉的数量会被传给后续操作 |

|  |  |
| --- | --- |
| **FruitObserve** | |
| Update（stapleorder\* order） | 更新所有的值 |

### 1.13.4模式类图



## 1.14命令模式

### 1.14.1模式介绍

命令模式可以将一系列请求封装起来，不显性的调用执行者提供的方法，可以实现对于指令的退回操作、也易于拓展指令的功能模块。首先需要一个命令之间的继承关系，一个父类的command类统领所有的命令，作为抽象类，command产生子类命名为concretecommand，子类可以有很多个，每一个子类都有一个execute方法，可以执行对应处理类的有关方法。同时含有一个invoke类用来操作命令模式的执行，其中含有一个命令列表，其中存放着此时需要执行的所有命令，当调用invoke中的call方法时，所有的命令会依次执行，而当需要执行undo操作时，就可以通过设立多个不同的指令组合从而使得系统可以回退到之前的某个状态。

### 1.14.2项目实例

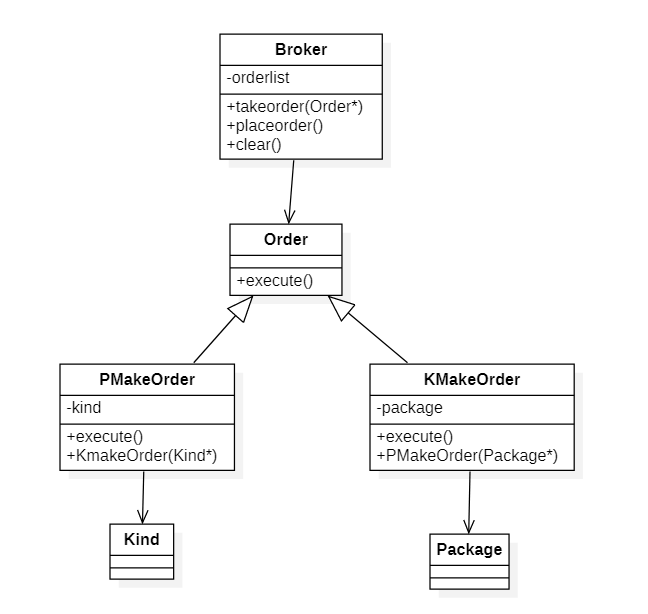
在罐头加工厂中，命令模式用于对于罐头的组件的生产，组件分为水果果肉、肉糜这些罐头的内容物，以及大包装和小包装这种罐头的外包装。这里命令模式的使用主要是因为内容物和外包装的生产是同步的，任意一个罐头都同时需要这两种东西，因此可以用命令模式一步将他们一同生成。当发现当前的生成是无用的时，通过命令模式也可以便捷的将所有的操作都消除。

### 1.14.3模式接口

|  |  |
| --- | --- |
| **Broker** | |
| Takeorder（Order\* order） | 将一个命令加入到命令的列表中 |
| Placeorder（） | 执行命令列表中所有的命令 |
| Clear（） | 去除命令列表中所有的命令 |
| Undo（） | 撤销命令列表中所有命令造成的后果 |

|  |  |
| --- | --- |
| **Order** | |
| Execute（） | 执行命令所对应的动作 |
| Unexecute（） | 撤销命令对应的动作 |

### 1.14.4模式类图



## 1.15抽象工厂模式

### 1.15.1模式介绍

围绕一个工厂建立其他工厂，这个工厂又叫做超级工厂，它的作用就是能够产生其他的工厂，而那些工厂又可以创造具体的类以解决问题。它包含一个工厂生成器类，叫做FactoryProducer，这个类可以产生工厂，其中含有一个方法名叫getFactory，这个方法的返回类型是AbstractFactory，也就是所有可被超级工厂生成的工厂的共同父类，这里体现了composite的设计模式。AbstractFactory扩展出各种各样不同的工厂，可以实现不同的功能。Get函数中会通过参数以及特定的判断算法返回有效的工厂。

### 1.15.2项目实例

在罐头加工厂中，抽象工厂模式用于产生内容物加工工厂和包装加工工厂这两种工厂，producer会根据外界传进来的参数判断应该返回哪种工厂，这两种工厂用于产生与罐头有关的一些组件的生产，这两个工厂会用于后续其他设计模式的产生。

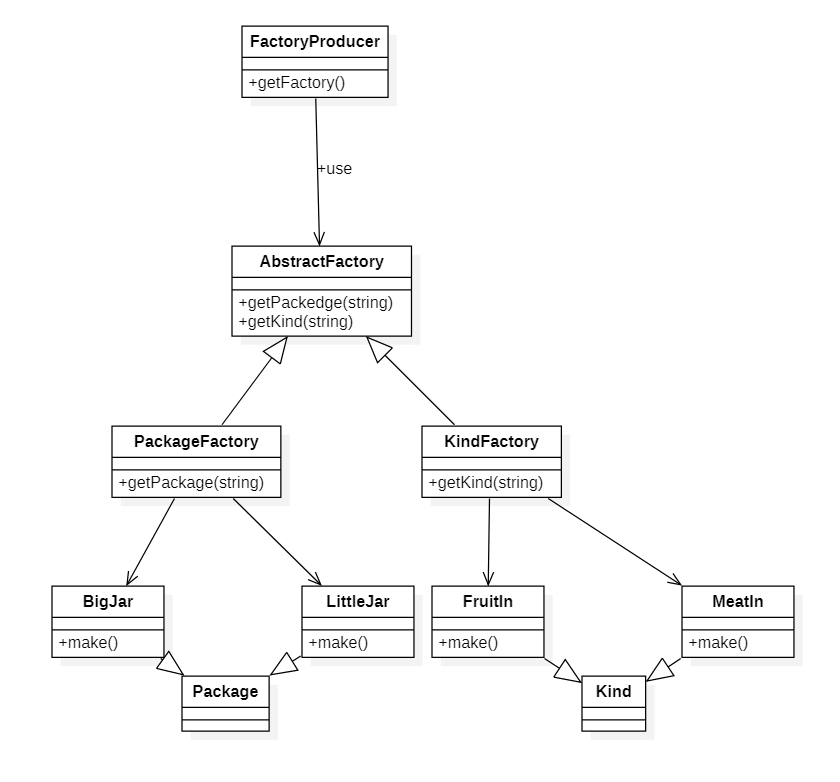
### 1.15.3模式接口

|  |  |
| --- | --- |
| **FactoryProducer** | |
| getFactory（string fac） | 获取fac所对应的工厂，分为包装工厂和种类工厂两类 |

|  |  |
| --- | --- |
| **AbstractFactory** | |
| getPackedge(string) | 获取对应的包装 |
| getKind(string) | 获取对应的种类 |

|  |  |
| --- | --- |
| **KindFactory** | |
| getKind（string a） | 获取对应的种类产品 |

### 1.15.4模式类图



## 1.16工厂模式

### 1.16.1模式介绍

提供一种创建对象的方式，在创建对象时不会暴露逻辑，而是直接通过调用函数返回指针的方式将对象创建完成，而且在工厂类中还含有既定的算法用来帮助用户自动的选择正确的对象去创建。包含一个所有具体类的父类，父类中有一个getconcrete函数，这个函数通过参数的内容返回用户需要的类的指针，对应的类在函数中被创建，并且被返回给用户使用。

### 1.16.2项目实例

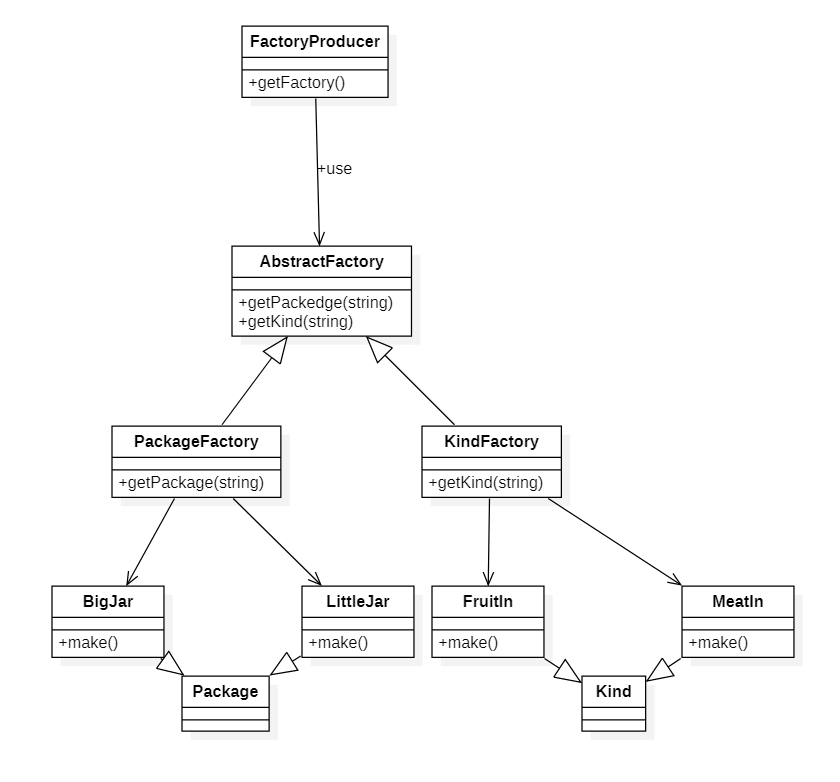
在罐头加工厂中，共有两个明显的工厂模式，工厂通过抽象工厂模式创建，分别是Package和Kind factory，这两个工厂分别可以根据输入的参数判断用户需要什么样的类，并将对应类的指针返回给用户。

### 1.16.3模式接口

|  |  |
| --- | --- |
| **KindFactory** | |
| getKind（string a） | 获取对应的种类产品 |

|  |  |
| --- | --- |
| **BigJar** | |
| Make（） | 生产对应的产品 |

### 1.16.4模式类图



## 1.17单例模式

### 1.17.1模式介绍

用于生成一个类，它在整个项目中只会有一个实例，通过一些限制可以保证这点的实现，主要用到了static的限制符。在类中含有一个静态的本类类型的指针，同时将构造函数设定为private的类型，还含有一个静态函数getinstance，用户可以通过类名：：getinstance的方式获得这个唯一的实例，如果该类已有一个实例，则会返回该实例，如果没有，则会创建一个实例。

### 1.17.2项目实例

在罐头加工厂中，所有的仓库都被设计为单例，比如：原料仓库、罐头仓库，这样可以保证项目的整体一致性。

# 4 订单销售

## 2.1中介模式

### 2.1.1模式介绍

用一个中介对象（中介者）来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。中介者模式又称为调停者模式，它是一种对象行为型模式。优点：（1）减少对象之间直接交互产生的错误，多个对象之间的交互（多对多），变成了通过中介者，形成了一对错的模式。（2）将各个同事时间接解耦，形成独立的类，使系统变得松耦合，可灵活的增加新的中介者以及具体同事类。缺点：（1）中介者承担角色任务重，中介者出现问题，导致系统崩溃。（2）增加新的具体同事类，需更改具体中介者类，违背了开闭原则，但是可以观察者和状态者模式可以更改该问题。适用环境：（1）多个对象之间进行复杂的相互通信。（2）一个对象和多个对象直接通信，那一复用该对象（给大家发相同消息，确不得不一个个的发送的时候）。（3）想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

### 2.1.2项目实例

中介模式在我们这一项目之中是为了实现罐头购买的消息的传递，通过抽象同事类实现两个具体同事类，一个是顾客，一个是罐头售卖者。抽象中介者中有声明了一个接口来发送消息，在具体中介者类之中实现。然后具体同事类先与中介者通信，然后中介者完成通信。

### 2.1.3模式接口

**Mediator**

|  |  |
| --- | --- |
| **函数** | **功能** |
| virtual void send(std::string msg, Colleague\* p) = 0 | 发送消息的全虚函数 |

**Colleague**

|  |  |
| --- | --- |
| Colleague(Mediator\* p) | 构造函数 |
| virtual void send(std::string msg) = 0 | 发送消息的纯虚函数 |
| virtual void notify(std::string msg) = 0 | 纯虚函数，由子类实现 |

**ConcreteColleague\_0**

|  |  |
| --- | --- |
| ConcreteColleague\_0(Mediator\* p) | 构造函数 |
| void send(std::string msg) | 实现上层基类的纯虚函数，发送消息 |
| void notify(std::string msg) | 实现上层基类的纯虚函数，接收消息 |

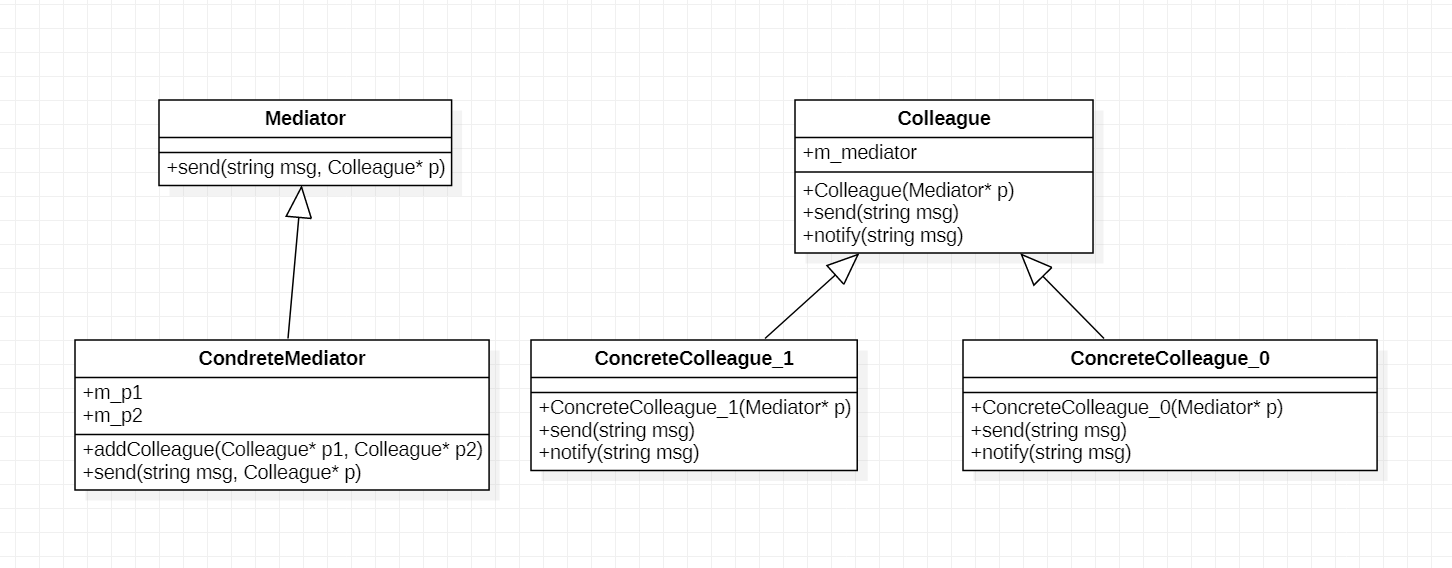
**ConcreteColleague\_1**

|  |  |
| --- | --- |
| ConcreteColleague\_1(Mediator\* p) | 构造函数 |
| void send(std::string msg) | 实现上层基类的纯虚函数，发送消息 |
| void notify(std::string msg) | 实现上层基类的纯虚函数，接受消息 |

**ConcreteMediator**

|  |  |
| --- | --- |
| void addColleague(Colleague\* p1, Colleague\* p2) | 添加同事类信息交流 |
| void send(std::string msg, Colleague\* p) | 发送消息 |

### 2.1.4模式类图



## 2.2状态模式

### 2.2.1模式介绍

这一模式允许对象在内部状态改变的时候改变他的行为，这个对象看起来就像改变了他的类。使用这个模式的优点有<1>状态的改变转化不是单纯的依靠if-else或者switch语句来实现，而是通过封装在具体的状态类或者环境类的代码来实现;<2>每一个具体的状态可以具有自己独特的功能，同时可以动态且方便地增加状态。缺点：当具体的状态类太多的时候会造成系统开销过大。

### 2.2.2项目实例

在我们的这一项目当中，我们将这一模式使用在了订单的状态转化上，随着时间的变化，订单的状态会随之变化，订单的状态输出的功能函数也会发生变化。

### 2.2.3模式接口

**State**

|  |  |
| --- | --- |
| virtual Creation() | 虚函数，由子类实现订单创建 |
| virtual Inprogress() | 虚函数，由子类表示订单正在进行的状态 |
| virtual void End() | 虚函数，由子类表示订单结束的状态 |
| virtual void CurrentState(Order\* order) | 虚函数，由子类表示订单创建初始的状态 |

**Order**

|  |  |
| --- | --- |
| Order(State\* state,int now\_day) | 构造函数 |
| ~Order() | 析构函数 |
| int SumDays() | 计算订单执行需要的总时间 |
| int GetDay() | 获取private变量day的值 |
| int GetID() | 返回订单的id |
| int GetCurrentay() | 获取private变量currentday的值 |
| void SetID(int id) | 设置id |
| void SetCurrentay(int current\_day) | 设置currentday的值 |
| void SetState(State\* state) | 设置当前状态 |
| void GetState() | 对于当前状态进行更新 |
| void ShowState() | 输出当前状态 |

**EndState**

|  |  |
| --- | --- |
| void End(Order\* order) | 订单结束，修改状态值 |
| void CurrentState(Order\* order) | 修改当前状态为结束 |

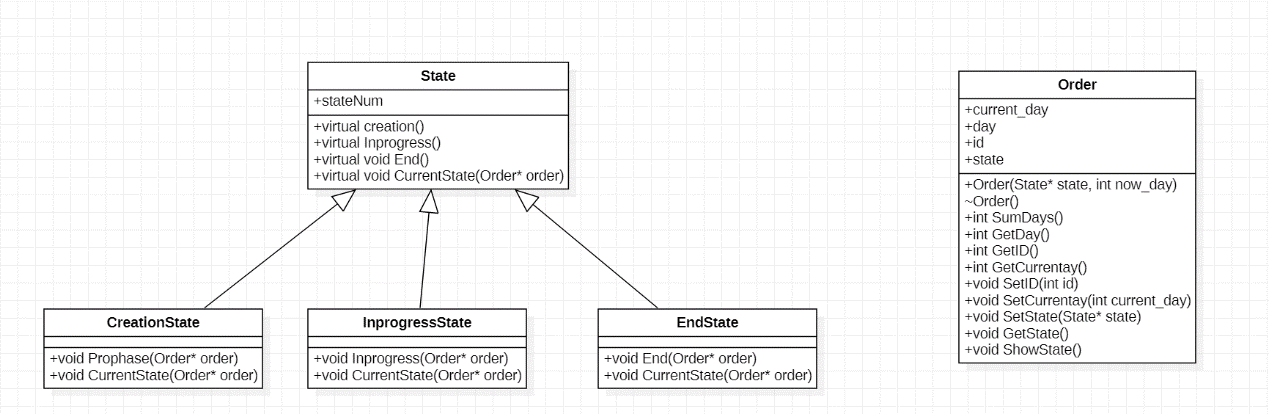
**InprogressState**

|  |  |
| --- | --- |
| void Inprogress(Order\* order) | 通过对于currentday和day的值的比较来判断当前订单状态并且修改 |
| void CurrentState(Order\* order) | 设置当前订单状态为进行 |

**CreationState**

|  |  |
| --- | --- |
| void Prophase(Order\* order) | 通过对于currentday和day的值的比较来判断当前订单状态并且修改 |
| void CurrentState(Order\* order) | 设置当前订单状态为开始 |

### 2.2.4模式类图



## 2.3迭代器模式

### 2.3.1模式介绍

迭代器模式是为了提供一种顺序访问聚合类内部的元素，而无需暴露聚合物内部描述，同时针对于不同的需求还能够以不同的方式来遍历聚合类。迭代器模式包含了抽象迭代器，抽象聚合类，具体聚合类，具体迭代器。在聚合类之中实现创建对应的具体的迭代器，使用迭代器来顺序遍历或者逆序遍历聚合类即可。

### 2.3.2项目实例

在我们的项目当中，我们使用迭代器模式来收集关注订单的状态，具体聚合类可以对应地添加订单以及删除订单，而具体的迭代器可以有多种方式对应地遍历所有的订单状态。

### 2.3.3模式接口

**Iterator**

|  |  |
| --- | --- |
| Iterator() | 构造函数 |
| virtual ~Iterator() | 析构函数 |
| virtual void first() | 虚函数，子类实现找到聚合类第一个元素 |
| virtual void next() | 虚函数，子类实现找到聚合类的下一个元素 |

**Aggregate**

|  |  |
| --- | --- |
| Aggregate() | 构造函数 |
| ~Aggregate() | 析构函数 |

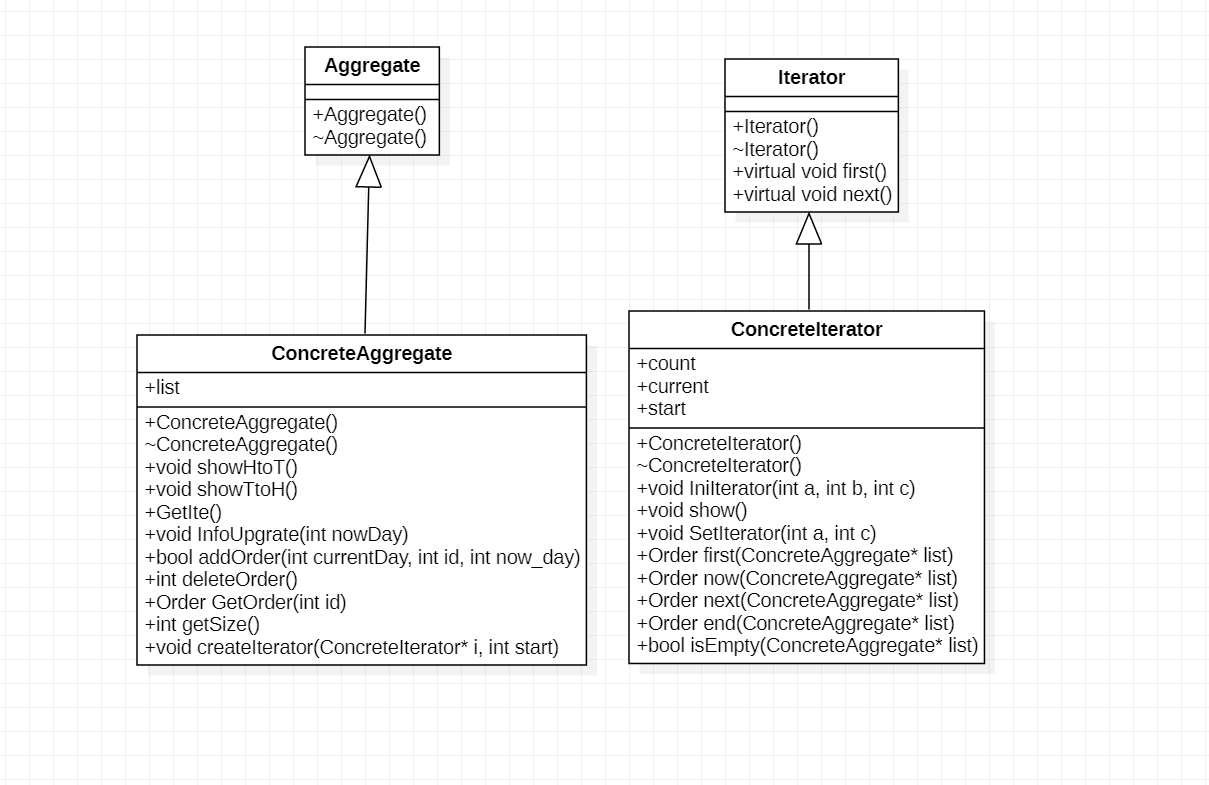
**ConcreteIterator**

|  |  |
| --- | --- |
| ConcreteIterator() | 构造函数 |
| ~ConcreteIterator() | 析构函数 |
| void IniIterator(int a, int b,int c) | 具体迭代器赋值 |
| void show() | 输出迭代器内部参数值 |
| void SetIterator(int a,int c) | 迭代器赋值 |
| Order first(ConcreteAggregate\* list) | 得到聚合类的第一个元素 |
| Order now(ConcreteAggregate\* list) | 得到指针指向的聚合类当前元素 |
| Order next(ConcreteAggregate\* list) | 得到指针指向的聚合类的下一个元素 |
| Order end(ConcreteAggregate\* list) | 得到聚合类的最后一个元素 |
| bool isEmpty(ConcreteAggregate\* list) | 判断聚合类是否为空 |

**ConcreteAggregate**

|  |  |
| --- | --- |
| ConcreteAggregate(vector<Order>\* v) | 构造函数 |
| ~ConcreteAggregate() | 析构函数 |
| void showHtoT() | 从头到尾遍历聚合类 |
| void showTtoH() | 从尾到头遍历聚合类 |
| vector<Order> GetIte() | 返回聚合类 |
| void InfoUpgrate(int nowDay) | 订单的时间信息更新 |
| bool addOrder(int currentDay, int id,int now\_day) | 添加订单到聚合类 |
| int deleteOrder() | 删除指定的订单 |
| Order GetOrder(int id) | 获取指定的id的订单 |
| int getSize() | 获取聚合类的元素个数 |
| void createIterator(ConcreteIterator\* i,int start) | 创建具体的迭代器 |

### 2.3.4模式类图



## 2.4组合模式

### 2.4.1模式介绍

组合模式，又叫部分整体模式，是用于把一组相似的对象当作一个单一的对象。组合模式依据树形结构来组合对象，用来表示部分以及整体层次。这种类型的设计模式属于结构型模式，它创建了对象组的树形结构。模糊了简单元素和复杂元素的概念，客户程序可以像处理简单元素一样来处理复杂元素，从而使得客户程序与复杂元素的内部结构解耦。

代码的具体实现是在上级类中含有一个vector，其数据类型是下级类，从而实现将下级类聚合到上级类中。

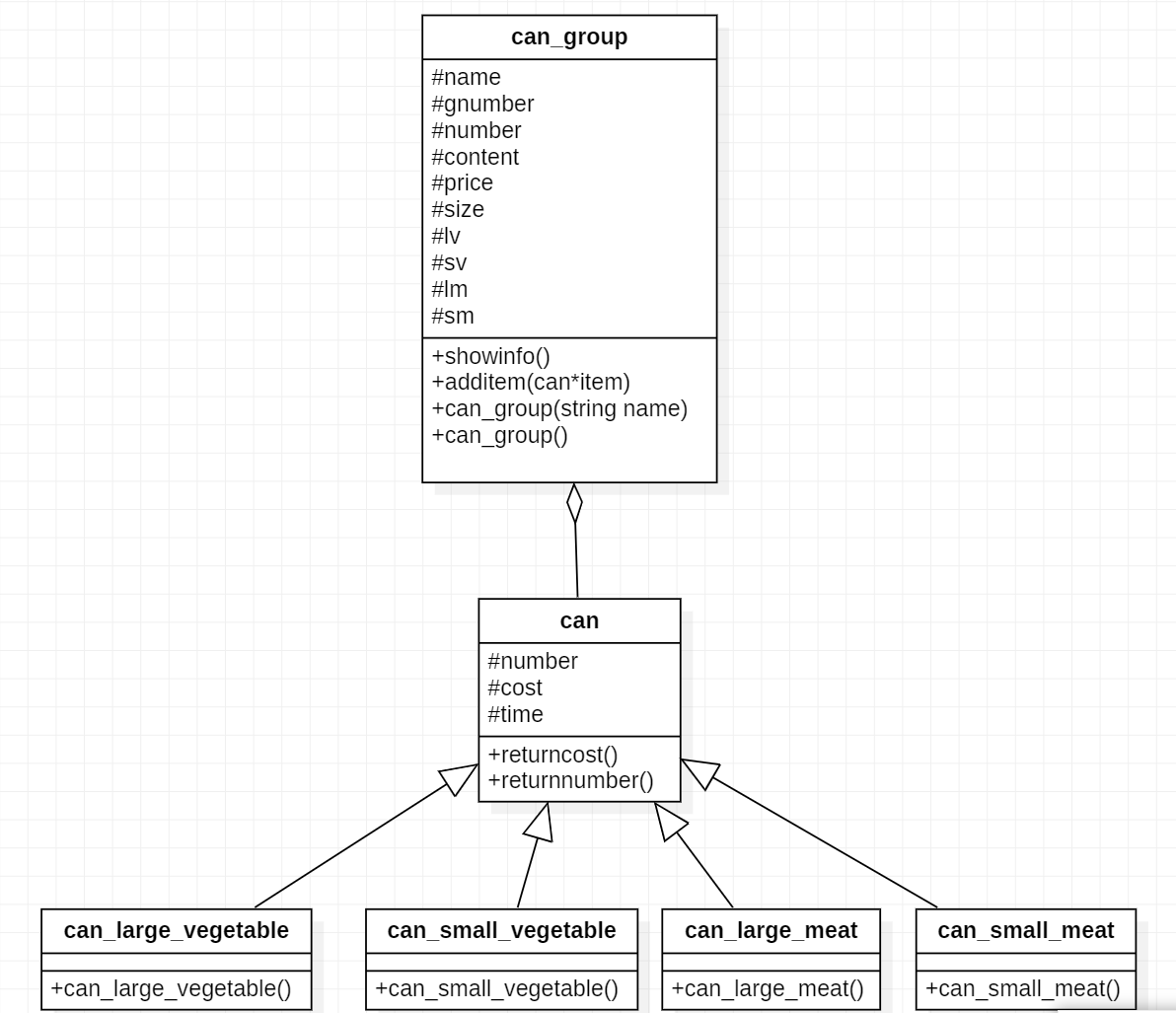
### 2.4.2项目实例

在罐头加工厂项目中，组合模式用于套餐类can\_group的组织。can\_group类应该由几个can类聚合而成，因此在can\_group类中声明vector<can> content，从而实现对can类的聚合

### 2.4.3模式接口

|  |  |
| --- | --- |
| **can\_group** | |
| showinfo() | 控制台显示套餐信息 |
| additem(can\* item) | 为套餐添加罐头 |
| **can** | |
| returncost() | 返回罐头成本 |
| returnnumber() | 返回罐头编号 |

### 2.4.4模式类图



## 2.5建造者模式

### 2.5.1模式介绍

建造者模式使用多个简单的对象一步一步构建成一个复杂的对象。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。主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

代码的具体实现主要是有一个builder类，其中有一个vector类来存储它所创建的实例，同时其有一个函数专门用于创建相应的实例

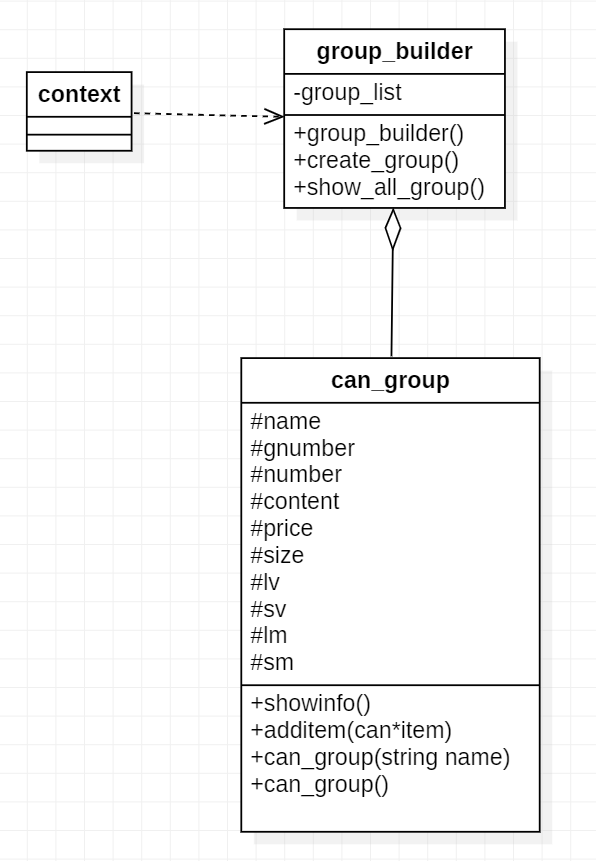
### 2.5.2项目实例

在罐头加工厂项目中，建造者模式用于创建套餐。首先会有一个套餐类can\_group，而在建造者类group\_builder中创建create\_group()函数用于创建can\_group类的实例，并储存在vector类group\_list中

### 2.5.3模式接口

|  |  |
| --- | --- |
| **can\_group** | |
| showinfo() | 控制台显示套餐信息 |
| additem(can\* item) | 为套餐添加罐头 |
| **group\_builder** | |
| create\_group() | 创建套餐 |
| show\_all\_group() | 显示所有套餐 |

### 2.5.4模式类图



## 2.6策略模式

### 2.6.1模式介绍

在策略模式中，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。一个系统有许多许多类，而区分它们的只是他们直接的行为时会使用。

代码的具体实现主要是利用多态性，将父类指针作为函数的变量，而在实际执行时令父指针指向一个子类的实体并进行调用，从而实现策略模式。

### 2.6.2项目实例

在罐头加工厂项目中，策略模式用于创建can类时进行初始化数据，是嵌套在一个适配器中的。根据我们的设计要求，需要利用Can类对can类进行初始化，因此我们创建了适配器can\_set\_adapter，并在其中声明父类Can指针Tool，在对can及其子类进行初始化时通过适配器调用Can类指针Tool中的函数来赋值，而在实际的can类子类构造函数中通过令Tool指向Can类的子类实例从而实现多态性，实现策略模式

### 2.6.3模式接口

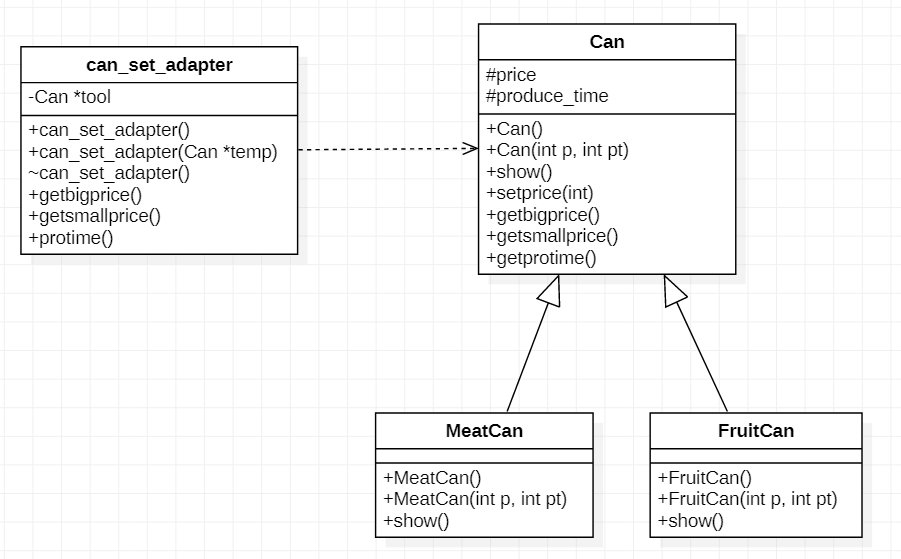
|  |  |
| --- | --- |
| **can\_set\_adapter** | |
| getbigprice() | 获取某种罐头大包装的成本 |
| getsmallprice() | 获取某种罐头小包装的成本 |
| protime() | 获取某种罐头的生产周期 |

|  |  |
| --- | --- |
| **FruitCan** | |
| show() | 展示罐头信息 |

|  |  |
| --- | --- |
| **MeatCan** | |
| show() | 展示罐头信息 |

|  |  |
| --- | --- |
| **can** | |
| returncost() | 返回罐头成本 |
| returnnumber() | 返回罐头编号 |

### 2.6.4模式类图



## 2.7状态模式

### 2.7.1模式介绍

在状态模式中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。

代码的具体实现主要是创建一个状态变量state，根据系统的状态改变state的值，而相关函数则根据state的值动态改变自己要进行的操作。

### 2.7.2项目实例

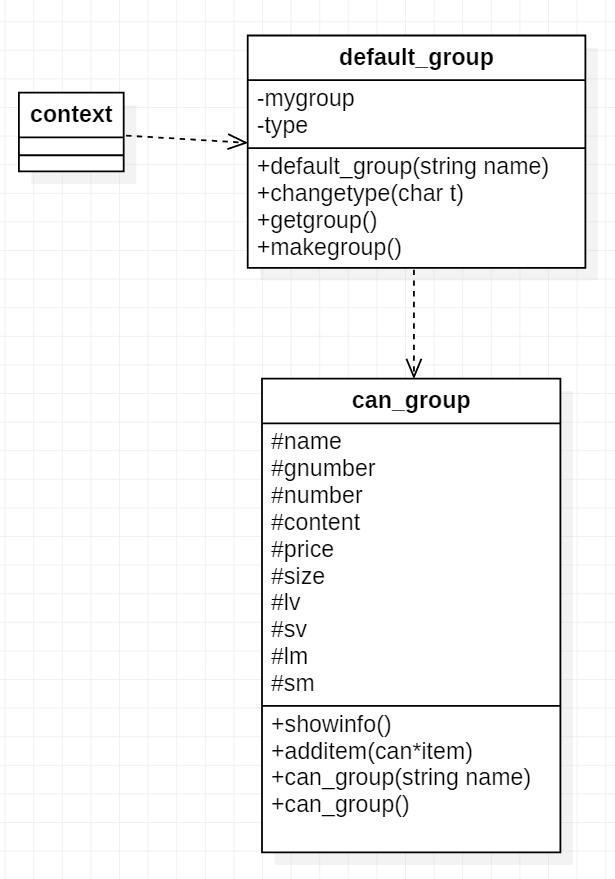
在罐头加工厂项目中，状态模式用于默认套餐建造者类default\_group的创建。首先定义状态变量type，并定义changetype(char t)函数来改变状态变量的值，并在函数makegroup()中依据状态变量的值创建相应的实例

### 2.7.3模式接口

|  |  |
| --- | --- |
| **default\_group** | |
| changetype(char t) | 改变套餐操作类型 |
| getgroup() | 获取套餐 |
| makegroup() | 根据操作类型创建套餐 |

|  |  |
| --- | --- |
| **can\_group** | |
| showinfo() | 控制台显示套餐信息 |
| additem(can\* item) | 为套餐添加罐头 |

### 2.7.4模式类图



## 2.8适配器模式

### 2.8.1模式介绍

适配器模式是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。这种模式涉及到一个单一的类，该类负责加入独立的或不兼容的接口功能。主要解决在软件系统中，常常要将一些"现存的对象"放到新的环境中，而新环境要求的接口是现对象不能满足的。

代码的具体实现这里采用了对象的适配器模式，即在适配器中创建一个目标类的对象的指针，在适配器函数中调用该对象指针中的函数，并在具体的其他类中通过调用适配器实例及函数来调用目标类中的函数。

### 2.8.2项目实例

在罐头加工厂项目中，适配器模式用于创建can类时进行初始化数据。相当于can类需要调用Can类中的接口，因此创建适配器类can\_set\_adapter，并在其中声明Can类型的指针，并在getbigprice()，getsmallprice()和protime()中调用Can中的函数。而在创建can类时，在can的子类构造函数中创建can\_set\_adapter实体并调用getbigprice()，getsmallprice()和protime()，从而实现调用Can类中的函数。

### 2.8.3模式接口

|  |  |
| --- | --- |
| **can** | |
| returncost() | 返回罐头成本 |
| returnnumber() | 返回罐头编号 |

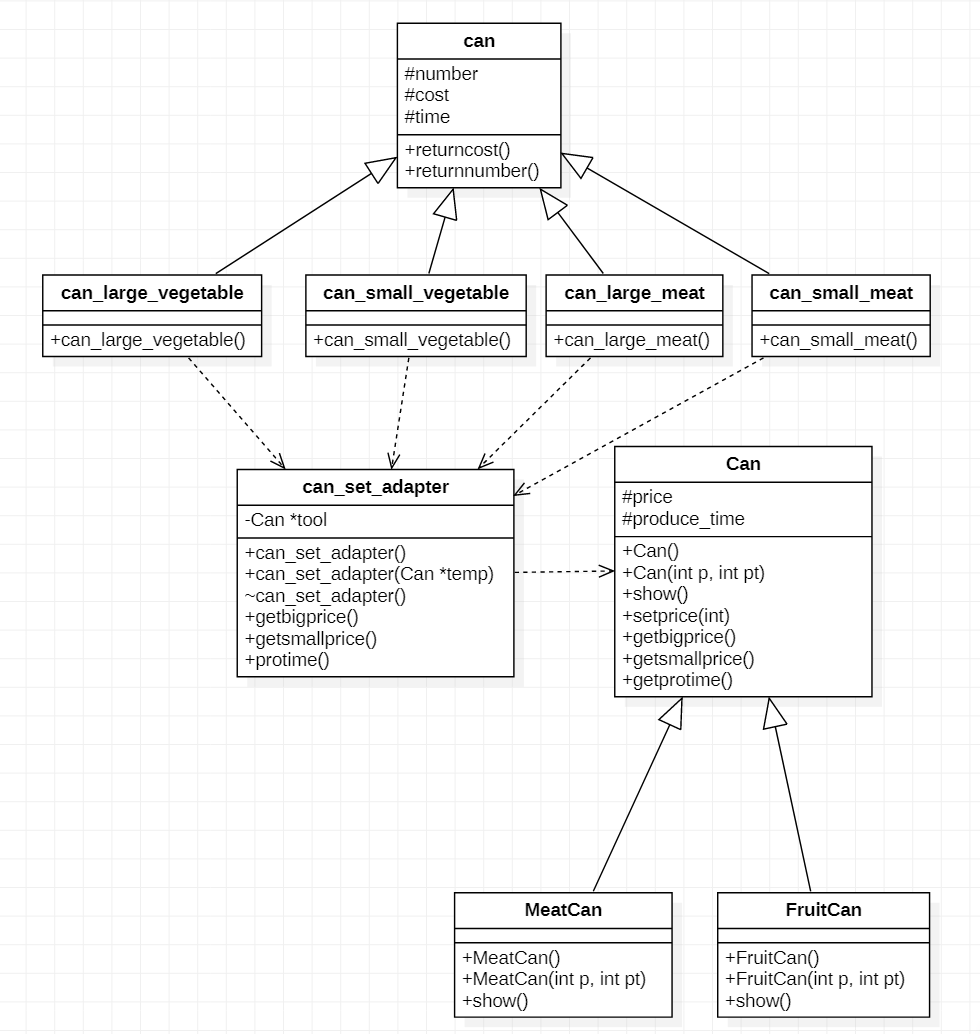
|  |  |
| --- | --- |
| **can\_set\_adapter** | |
| getbigprice() | 获取某种罐头大包装的成本 |
| getsmallprice() | 获取某种罐头小包装的成本 |
| protime() | 获取某种罐头的生产周期 |

|  |  |
| --- | --- |
| **Can** | |
| show() | 展示罐头信息 |
| setprice(int) | 设置罐头成本 |
| getbigprice() | 获取大包装罐头成本 |
| getsmallprice() | 获取小包装罐头成本 |
| getprotime() | 获取罐头生产周期 |

|  |  |
| --- | --- |
| **FruitCan** | |
| show() | 展示罐头信息 |

|  |  |
| --- | --- |
| **MeatCan** | |
| show() | 展示罐头信息 |

### 2.8.4模式类图



# 5 人员管理

## 3.1责任链模式

### 3.1.1模式介绍

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。责任链模式是一种行为型模式。

在责任链模式中，客户只需要将请求发送到责任链上即可，无须关心请求的处理细节和请求的传递过程，请求会自动进行传递。所以责任链实现了请求发送者和请求处理者的解耦。

主要优点：

1. 每个类只需处理自己负责的工作，不该处理的交给下一级，符合单一职责原则
2. 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
3. 降低了对象之间的耦合度。用户无需关注具体的处理者是谁，只需关注处理的结果

缺点：

1. 不能保证所有请求都能被处理
2. 责任链过长会导致处理效率下降

### 3.1.2项目实例

在罐头加工厂项目中，责任链模式用于人员管理中处理员工的请求（如请短假、长假、申请加薪、升职、离职等）。实现方法是对每种请求都设置一个level，不同处理者处理不同level的请求。然后设计一个抽象处理者Handler类，由三个子类（具体处理者）Leader、Manager、Boss继承。Handler中有一个成员变量nextHandler，表示该处理者的下一级处理者。Handler类还提供一个虚函数handle()，在三个子类中实现，每个子类的handle()实现处理自己负责的level，超出自己能力范围的则交给下一级处理者。当员工创建一个请求时，将会交给第一级的处理者Leader处理，Leader若不能处理则交给Manager，Manager不能处理就交给Boss，最后将处理的结果返回给用户。

### 3.1.3模式接口

|  |  |
| --- | --- |
| **Request** | |
| Request(int, string) | 构造函数，设置请求的等级和内容 |
| int getLevel() | 获取该请求的等级 |

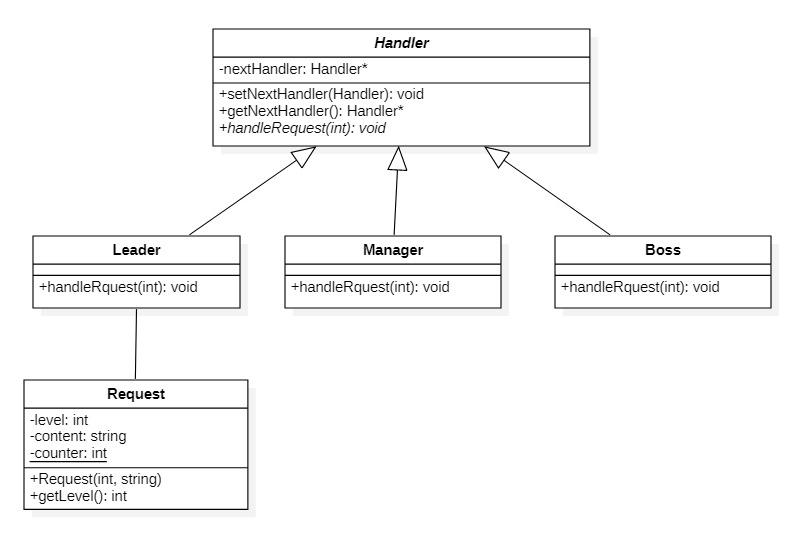
|  |  |
| --- | --- |
| **Handler** | |
| void setNextHandler(Handler\*) | 设置责任链中下一个handler |
| Handler\* getNextHandler() | 获取下一个handler |
| void handleRequest() | 纯虚函数，handler处理请求 |

|  |  |
| --- | --- |
| **Leader** | |
| void handleRequest() | 实现leader处理请求的操作 |

|  |  |
| --- | --- |
| **Manager** | |
| void handleRequest() | 实现manager处理请求的操作 |

|  |  |
| --- | --- |
| **Boss** | |
| void handleRequest() | 实现boss处理请求的操作 |

### 3.1.4模式类图



# 6 设备管理

## 4.1状态模式

### 4.1.1模式介绍

状态模式允许一个对象在其内部状态改变的时候改变其行为。这个对象看上去就像是改变了它的类一样。

用一句话来表述，状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。

代码的具体实现主要涉及到三个角色：

1. 环境(Context)角色，也成上下文：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境对象的现有状态。
2. 抽象状态(State)角色：定义一个接口，用以封装环境（Context）对象的一个特定的状态所对应的行为。
3. 具体状态(ConcreteState)角色：每一个具体状态类都实现了环境（Context）的一个状态所对应的行为。

### 4.1.2项目实例

环境类Context调用request()，在request()中调用某一个具体状态类的handle()。通过使用多态性原则，可以动态改变环境类Context的属性State的内容，使其从指向一个具体状态类变换到指向另一个具体状态类，从而使环境类的行为request()由不同的具体状态类来执行。因此在外部看来就是一个对象在其内部状态改变的时候，其行为也随之改变。

### 4.1.3模式接口

**Context**

|  |  |
| --- | --- |
| 函数 | 功能 |
| Context() | 构造函数 |
| ~ Context() | 析构函数，delete State指针 |
| void changeState(State\* st) | 改变设备的state |
| void request() | State模式，调用不同state的handle方法，使设备内部状态改变的时候，其行为也随之改变 |

## 4.2访问者模式

### 4.2.1模式介绍

访问者模式表示一个作用于某对象结构中的各元素的操作，我们可以在不改变各元素类的前提下定义作用于这些元素的新操作。

代码的具体实现主要涉及到三个角色：

1. Visitor 抽象访问者角色，为该对象结构中具体元素角色声明一个访问操作接口。该操作接口的名字和参数标识了发送访问请求给具体访问者的具体元素角色，这样访问者就可以通过该元素角色的特定接口直接访问它。
2. ConcreteVisitor.具体访问者角色，实现Visitor声明的接口。
3. Element 定义一个接受访问操作(accept())，它以一个访问者(Visitor)作为参数。
4. ConcreteElement 具体元素，实现了抽象元素(Element)所定义的接受操作接口。
5. ObjectStructure 结构对象角色，用于能枚举Element元素并提供一个接口以允许访问者访问它的元素。

ConcreteElement 具体元素实现了抽象父类中的accept()函数，accept()函数中又调用参数Visitor的visit()，对ConcreteElement进行访问。ObjectStructure 结构对象遍历Element元素调用其accept()方法使相对应的Visitor逐个对其访问。

### 4.2.2项目示例

在罐头加工厂中用于维护设备状态的部分，设备有闲置、运行、损坏三种状态，设备不同的状态会导致不同的行为，维护人员通过Visitor模式访问设备状态， State状态会使设备状态产生相对应的行为，因此如果外部环境因素导致设备状态发生改变时，设备的行为也会相对应的进行改变。

### 4.2.3模式接口

**State**

|  |  |
| --- | --- |
| State() | 构造函数 |
| void handle(Context& c) | 纯虚函数，由子类实现不同动作，用于实现State pattern |
| void accept(Visitor& visitor) | 纯虚函数，由子类实现不同动作，用于实现Visitor pattern |

**IdleState**

|  |  |
| --- | --- |
| IdleState() | 构造函数 |
| void handle(Context& c) | 实现上层基类的纯虚函数，实现对不同的设备状态调用不同的行为 |
| void accept(Visitor& visitor) | 实现上层基类的纯虚函数，调用visitor.visit(\*this)方法 |

**RunningState**

|  |  |
| --- | --- |
| RunningState () | 构造函数 |
| void handle(Context& c) | 实现上层基类的纯虚函数，实现对不同的设备状态调用不同的行为 |
| void accept(Visitor& visitor) | 实现上层基类的纯虚函数，调用visitor.visit(\*this)方法 |

**DamageState**

|  |  |
| --- | --- |
| DamageState() | 构造函数 |
| void handle(Context& c) | 实现上层基类的纯虚函数，实现对不同的设备状态调用不同的行为 |
| void accept(Visitor& visitor) | 实现上层基类的纯虚函数，调用visitor.visit(\*this)方法 |

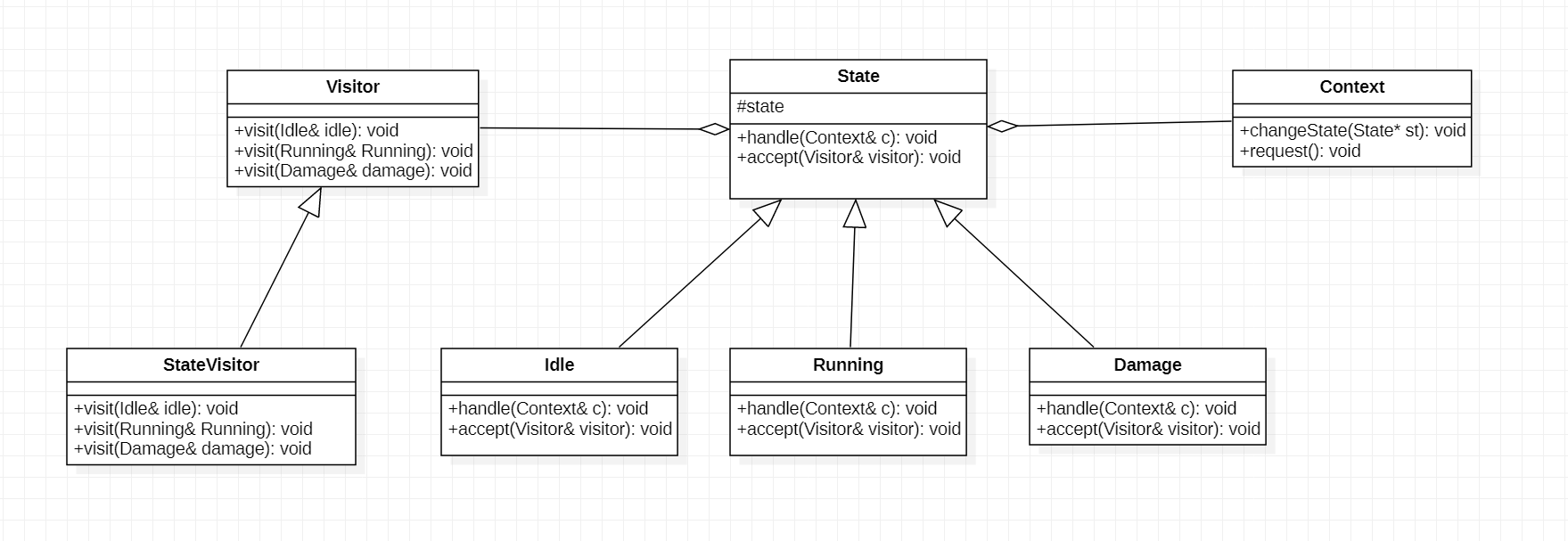
**Visitor**

|  |  |
| --- | --- |
| Visitor() | 构造函数 |
| void visit(IdleState& idle) | 纯虚函数，由具体的visitor去实现对IdleState的访问 |
| void visit(RunningState& RunningState) | 纯虚函数，由具体的visitor去实现对RunningState的访问 |
| void visit(DamageState& damage) | 纯虚函数，由具体的visitor去实现对DamageState的访问 |

**StateVisitor**

|  |  |
| --- | --- |
| StateVisitor() | 构造函数 |
| void visit(IdleState& idle) | 实现对IdleState的访问 |
| void visit(RunningState& RunningState) | 实现对RunningState的访问 |
| void visit(DamageState& damage) | 实现对DamageState的访问 |

### 4.2.4模式类图



# 7 财务管理

## 5.1代理模式

### 5.1.1模式介绍

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。

代理模式的实现结构比较简单，主要是通过定义一个继承抽象主题（Subject）的代理（Proxy）来包含真实主题（RealSubject），从而实现对真实主题的访问。其中父类Subject中有一个纯虚函数Request()，具体实现在子类RealSubject和Proxy中，Proxy的Request()中会调用RealSubject的Request()函数。这样客户端通过创建一个Proxy类对象，调用Proxy的Request函数，Request函数中创建一个RealSubject类对象，再调用RealSubject中的Request函数，实现对目标对象的访问。

主要优点：

1. 在客户端与目标对象之间起到一个中介作用和保护目标对象的作用；
2. 代理对象可以扩展目标对象的功能；
3. 代理模式能将客户端与目标对象分离，在一定程度上降低了系统的耦合度，增加了程序的可扩展性

缺点：

1. 请求处理速度变慢
2. 系统复杂度增加

### 5.1.2项目实例

在罐头加工厂项目中，代理模式用于财务管理中对账本记录的访问。Originator作为代理模式中的RealSubject，记录账本当前的状态信息。用户端想要查询账本信息，不能直接访问账本，而是通过会计（Accountant类）作为代理，Accountant在接到用户请求后，调用 Originator中的printInfo函数打印账本信息并呈现给用户。

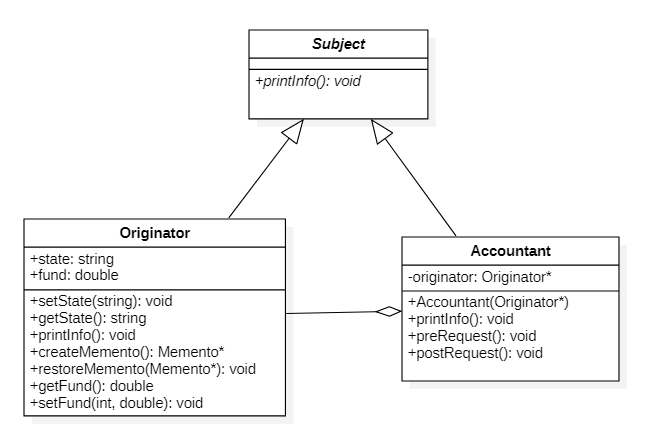
### 5.1.3模式接口

|  |  |
| --- | --- |
| **Subject** | |
| printInfo（） | Proxy中父类的纯虚函数 |

|  |  |
| --- | --- |
| **Originator** | |
| void setState(string) | 修改账本当前状态 |
| string getState() | 获取当前账本的状态信息 |
| void printInfo() | 纯虚函数，由子类实现不同动作 |
| Memento\* createMemento() | 将当前账本状态信息创建为备忘录 |
| void restoreMemento(Memento\*) | 将当前账本状态恢复到某一历史状态 |
| double getFund() | 获取当前企业的资金总数 |
| void setFund(int,double) | 实现对流动资金的修改操作 |

|  |  |
| --- | --- |
| **Accountant** | |
| Accountant(Originator\*) | 构造函数，初始化代理对象 |
| void preRequest() | 实现代理请求的预操作 |
| void printInfo() | 实现代理请求，调用Originator中的函数 |
| void postRequest() | 实现代理请求的后续操作 |

### 5.1.4模式类图



## 5.2备忘录模式

### 5.2.1模式介绍

备忘录模式是指：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便以后当需要时能将该对象恢复到原先保存的状态。

备忘录模式由3个类实现，分别是发起人（Originator）、备忘录（Memento）和管理者（Caretaker）。其中，Originator记录当前内部状态信息，提供创建备忘录和恢复备忘录数据的方法，并可以访问备忘录中的全部信息；Memento负责存储状态信息，在需要的时候提供给Originator；Caretaker对备忘录进行管理，提供保存与获取备忘录的功能，但其不能对备忘录的内容进行访问与修改。

主要优点：

1. 提供可恢复状态的机制。方便地将数据恢复到某个历史的状态。
2. 实现了内部状态的封装。除了创建它的发起人之外，其他对象不能够访问状态信息。
3. 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。

缺点：如果频繁备份记录，内存消耗会很大

### 5.2.2项目实例

在罐头加工厂项目中，备忘录模式用于财务管理中对账本记录的保存和恢复。Originator记录账本的当前状态信息，调用createMemento()创建一个Memento类的对象，它含有账本状态信息state。创建的所有备忘录通过Caretaker的addMemento函数存储到Caretaker类的list中，同时Caretake提供获取上一条Memento的方法，必要时提供给Originator，Originator再调用restoreMemento()恢复账本上一状态。

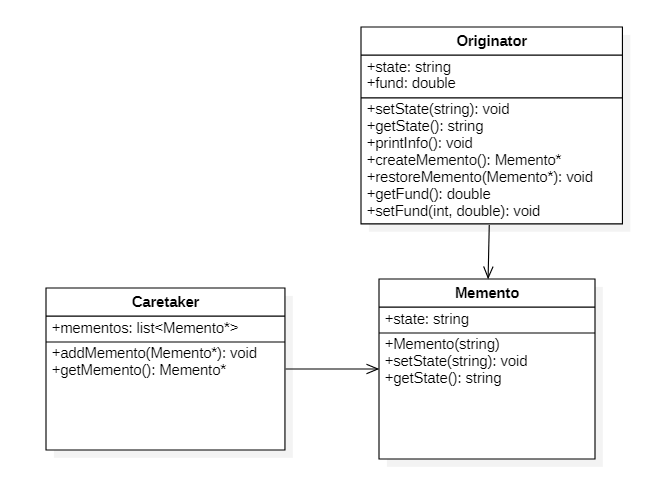
### 5.2.3模式接口

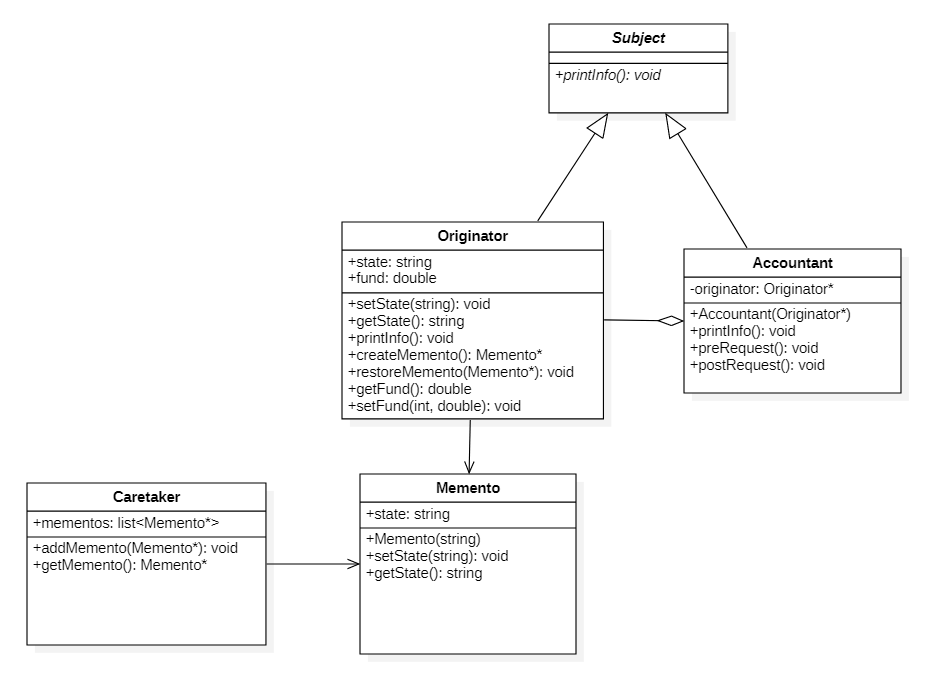
|  |  |
| --- | --- |
| **Originator** | |
| void setState(string) | 修改账本当前状态 |
| string getState() | 获取当前账本的状态信息 |
| void printInfo() | 纯虚函数，由子类实现不同动作 |
| Memento\* createMemento() | 将当前账本状态信息创建为备忘录 |
| void restoreMemento(Memento\*) | 将当前账本状态恢复到某一历史状态 |
| double getFund() | 获取当前企业的资金总数 |
| void setFund(int,double) | 实现对流动资金的修改操作 |

|  |  |
| --- | --- |
| **Memento** | |
| Memento(string) | 构造函数，初始化备忘录存储的状态信息 |
| void setState(string) | 实现对备忘录的状态信息的修改操作 |
| string getState() | 获取该条备忘录的状态信息 |

|  |  |
| --- | --- |
| **Caretaker** | |
| void addMemento(Memento\*) | 实现备忘录的添加和存储操作 |
| Mememto\* getMemento() | 获取最新一条备忘录信息 |

### 5.2.4模式类图





财务管理总类图