DESIGN PATTERN

**设计模式项目文档**

天猫购物节

**小组成员名单**

组长: 1953617 李睿捷

成员: 1853629 席宇琛

1852461 莫海洋

1950698 陆天宇

1952522 张峰

1952815 余林旭

1953069 闫书玮

1953081 张福泰

1953910 李林洲

文档整合与排版: 李林洲

完成日期: 2021-11-05

目录

**Table of contents**

[项目概要 7](#_Toc86941219)

[设计模式列表 8](#_Toc86941220)

[UI子系统 9](#_Toc86941221)

[模块模式 9](#_Toc86941222)

[类图 9](#_Toc86941223)

[详述 9](#_Toc86941224)

[核心代码 9](#_Toc86941225)

[出处 12](#_Toc86941226)

[顾客子系统 13](#_Toc86941227)

[建造者模式 13](#_Toc86941228)

[类图 13](#_Toc86941229)

[详述 13](#_Toc86941230)

[代码实现 14](#_Toc86941231)

[观察者模式 15](#_Toc86941232)

[类图 15](#_Toc86941233)

[详述 16](#_Toc86941234)

[代码实现 16](#_Toc86941235)

[中介者模式 17](#_Toc86941236)

[类图 18](#_Toc86941237)

[详述 18](#_Toc86941238)

[代码实现 18](#_Toc86941239)

[备忘录模式 18](#_Toc86941240)

[类图 19](#_Toc86941241)

[详述 19](#_Toc86941242)

[代码实现 19](#_Toc86941243)

[模板方法 21](#_Toc86941244)

[类图 21](#_Toc86941245)

[详述 21](#_Toc86941246)

[单例模式 21](#_Toc86941247)

[类图 22](#_Toc86941248)

[详述 22](#_Toc86941249)

[代码实现 22](#_Toc86941250)

[店铺子系统 24](#_Toc86941251)

[抽象工厂模式 24](#_Toc86941252)

[类图 24](#_Toc86941253)

[详述 24](#_Toc86941254)

[工厂模式 25](#_Toc86941255)

[类图 25](#_Toc86941256)

[详述 25](#_Toc86941257)

[核心代码 25](#_Toc86941258)

[实现API 26](#_Toc86941259)

[空对象模式 26](#_Toc86941260)

[类图 27](#_Toc86941261)

[详述 27](#_Toc86941262)

[核心代码 27](#_Toc86941263)

[实现API 28](#_Toc86941264)

[出处 28](#_Toc86941265)

[访问者模式 28](#_Toc86941266)

[类图 28](#_Toc86941267)

[详述 29](#_Toc86941268)

[核心代码 29](#_Toc86941269)

[过滤器模式 29](#_Toc86941270)

[类图 30](#_Toc86941271)

[详述 30](#_Toc86941272)

[核心代码 30](#_Toc86941273)

[出处 31](#_Toc86941274)

[代理模式 31](#_Toc86941275)

[类图 32](#_Toc86941276)

[详述 32](#_Toc86941277)

[核心代码 32](#_Toc86941278)

[适配器模式 33](#_Toc86941279)

[类图 33](#_Toc86941280)

[详述 33](#_Toc86941281)

[核心代码 34](#_Toc86941282)

[装饰器模式 34](#_Toc86941283)

[类图 35](#_Toc86941284)

[详述 35](#_Toc86941285)

[核心代码 35](#_Toc86941286)

[责任链模式 36](#_Toc86941287)

[类图 37](#_Toc86941288)

[详述 37](#_Toc86941289)

[核心代码 37](#_Toc86941290)

[原型模式 38](#_Toc86941291)

[类图 39](#_Toc86941292)

[详述 39](#_Toc86941293)

[核心代码 39](#_Toc86941294)

[依赖注入模式 40](#_Toc86941295)

[类图 41](#_Toc86941296)

[详述 41](#_Toc86941297)

[核心代码 41](#_Toc86941298)

[出处 42](#_Toc86941299)

[商品子系统 44](#_Toc86941300)

[桥接模式 44](#_Toc86941301)

[类图 44](#_Toc86941302)

[详述 44](#_Toc86941303)

[实现API 45](#_Toc86941304)

[组合模式 45](#_Toc86941305)

[类图 46](#_Toc86941306)

[详述 46](#_Toc86941307)

[实现API 46](#_Toc86941308)

[迭代器模式 47](#_Toc86941309)

[类图 47](#_Toc86941310)

[详述 47](#_Toc86941311)

[实现API 47](#_Toc86941312)

[享元模式 48](#_Toc86941313)

[类图 48](#_Toc86941314)

[关键代码 48](#_Toc86941315)

[详述 49](#_Toc86941316)

[实现API 49](#_Toc86941317)

[命令模式 49](#_Toc86941318)

[类图 50](#_Toc86941319)

[详述 50](#_Toc86941320)

[实现API 51](#_Toc86941321)

[活动与广告子系统 52](#_Toc86941322)

[功能与实现 52](#_Toc86941323)

[广告子系统 52](#_Toc86941324)

[活动子系统 53](#_Toc86941325)

[子系统API 57](#_Toc86941326)

[解释器模式 58](#_Toc86941327)

[类图 58](#_Toc86941328)

[详述 58](#_Toc86941329)

[迭代器模式 61](#_Toc86941330)

[详述 61](#_Toc86941331)

[单例模式 61](#_Toc86941332)

[类图 61](#_Toc86941333)

[详述 61](#_Toc86941334)

[访问者模式 62](#_Toc86941335)

[类图 63](#_Toc86941336)

[详述 63](#_Toc86941337)

[策略模式 64](#_Toc86941338)

[类图 65](#_Toc86941339)

[详述 65](#_Toc86941340)

[外观模式 66](#_Toc86941341)

[类图 66](#_Toc86941342)

[详述 66](#_Toc86941343)

[购物车子系统 68](#_Toc86941344)

[功能与实现 68](#_Toc86941345)

[命令模式 70](#_Toc86941346)

[类图 71](#_Toc86941347)

[实现过程 71](#_Toc86941348)

[实现API 72](#_Toc86941349)

[关键代码 74](#_Toc86941350)

[观察者模式 74](#_Toc86941351)

[类图 75](#_Toc86941352)

[实现过程 75](#_Toc86941353)

[实现API 75](#_Toc86941354)

[代码样例 76](#_Toc86941355)

[订单子系统 77](#_Toc86941356)

[状态模式 77](#_Toc86941357)

[类图 77](#_Toc86941358)

[详述 77](#_Toc86941359)

[实现API 77](#_Toc86941360)

[小组成员贡献 79](#_Toc86941361)

[图表索引 80](#_Toc86941362)

[参考文献 82](#_Toc86941363)

# 项目概要

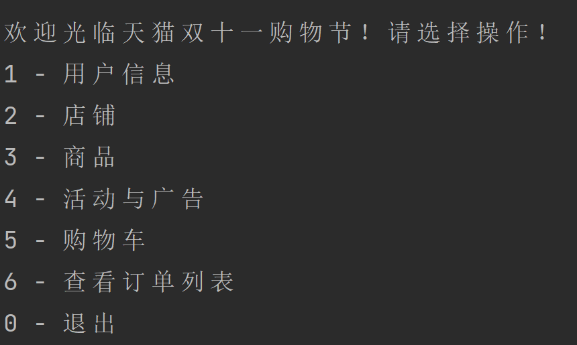
**Abstract**

**简介**

本项目选题为天猫购物节，实现语言为C++，共实现了27种不同的设计模式。可满足用户修改信息、浏览天猫购物节店铺/商品、浏览天猫购物街活动/广告、对天猫购物街店铺/商品操作并使用相应活动、生成订单并支付等相应操作。

**操作流程**

用户进入后，先设置用户信息，然后可以浏览店铺、商品、活动与广告，之后在购物车内进行添加和删除的操作后，生成订单并支付即可。



1 - 用户信息：用户在其中可以进行与个人信息相关的操作，包括显示当前用户信息、新建用户、退出当前用户、切换用户、修改当前用户信息、回滚当前用户信息、退出用户系统。

设计模式：建造者模式、观察者模式、中介者模式、备忘录模式、模板方法、单例模式

2 - 店铺：用户在其中可以进行与用户相关的操作，包括显示查看所有店铺、评价指定ID店铺、根据评分筛选店铺、查看指定店铺详细信息、查看所有店铺商铺列表、查看指定ID店铺商品列表。

设计模式：抽象工厂模式、工厂模式、空对象模式、访问者模式、过滤器模式、代理模式、适配器模式、装饰器模式、责任链模式、原型模式、依赖注入模式

3 - 商品：用户可以查看本次购物节内提供的所有商品，进入后程序会输出所有商品信息。

设计模式：桥接模式、组合模式、迭代器模式、享元模式、命令模式

4 - 活动与广告：用户在其中可以进行与活动、广告相关的操作。进入后程序会输出所有活动和广告信息以供用户参考。

设计模式：解释器模式、迭代器模式、单例模式、访问者模式、策略模式、外观模式

5 - 购物车：用户在其中可以进行与购物车相关的操作，包括查看购物车、将指定商品加入购物车、删除购物车中指定商品、减少购物车内指定商品、结算购物车。

设计模式：命令模式、观察者模式

6 - 查看订单列表：用户在其中可以进行与购物车相关的操作，包括查看进行中的订单、支付全部未支付的订单、查看所有订单。

订单子系统：状态模式

# 设计模式列表

**List of Design Patterns**

表 1设计模式列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 编号 | 设计模式 | 文件路径 | 功能或函数 | 备注 |
| 1 | 模块模式 | Welcome\welcome.h | 将所有子系统的命令提示系统打包成模块全局使用 | 非经典设计模式 |
| 2 | 建造者模式 | Customer/Customer.h | 产生不同级别的用户 |  |
| 3 | 观察者模式 | Customer/Customers.h | 触发用户事件，发送双十一提醒消息 |  |
| Cart/CartCommand.h | 借助宏命令ExecuteCommands的广播，修改每个CartCommand的内容 |  |
| 4 | 中介者模式 | Customer/Customer.h | 实现公共聊天室 |  |
| 5 | 备忘录模式 | Customer/Information.h | 回退用户信息 |  |
| 6 | 模板方法 | Customer/ Information.h | 用户信息拷贝 |  |
| 7 | 单例模式 | Customer/Customers.h | 实现多用户系统 |  |
| AC\_ADSystem/ActivitySystem.h AC\_ADSystem/AdtivitySystem.h | 确保广告与活动子系统类只有一个实例 |  |
| 8 | 抽象工厂 模式 | Shop/shopInterface.h | 制造生产商品的工厂 |  |
| 9 | 工厂模式 | Shop/shopInterface.h | 工厂制造生产商品 |  |
| 10 | 空对象模式 | Shop/shop.h | 用户输入参数为空时返回对象 | 非经典设计模式 |
| 11 | 访问者模式 | Shop/shop.h | 不需要直接对数据进行操作即可访问数据 |  |
| AC\_ADSystem/AcVisitor.h AC\_ADSystem/AdVisitor.h | 用于后续灵活添加对广告与活动的操作 |  |
| 12 | 过滤器模式 | Shop/shopFilter.h | 根据评分筛选店铺 | 非经典设计模式 |
| 13 | 代理模式 | Shop/shop.h | 对代理店铺操作等效于对店铺操作 |  |
| 14 | 适配器模式 | Welcome/welcome.h | 扩展全局命令提示系统，可以查看有哪些命令 |  |
| 15 | 装饰器模式 | Shop/shop.h | 无需修改商店类，即可修改商店评分为高分 |  |
| 16 | 责任链模式 | ChainOfResponsibility /ChainOfResponsibility.h | 在每个操作执行完后输出系统日志显示系统情况 |  |
| 17 | 原型模式 | Shop/shopInterface.h | 初始化商店时，预先加载已有商店到缓存中，生成时已有的话直接加载 |  |
| 18 | 依赖注入 模式 | Shop/shop.h | 将类当作参数传入其他类，实现客户端-服务器模式 | 非经典设计模式 |
| 19 | 桥接模式 | Commodity/Commodity.h | 解耦商品实体类与商品功能类 |  |
| 20 | 组合模式 | Commodity/Commodity.h | 实现单一商品与组合商品 |  |
| 21 | 迭代器模式 | Commodity/Commodity.h | 遍历组合商品中的单一商品 |  |
| AC\_ADSystem/AcVisitor.cpp  AC\_ADSystem/AdVisitor.cpp | 遍历所有活动或广告 |  |
| 22 | 享元模式 | Commodity/CommodityFactory.cpp | CommodityFactory::getSingleCommodity() |  |
| 23 | 命令模式 | Commodity/Commodity.h | CommoditySale::execute() |  |
| Cart/CartCommand.h | 借助CartCommand类的派生类间接对购物车进行操作 |  |
| 24 | 解释器模式 | AC\_ADSystem/Interpreter.h | 用解释器解释计算最优价格函数输出的特殊字符 |  |
| 25 | 策略模式 | AC\_ADSystem/Facade.cpp | 将一个优惠活动的多种选择封装以便使用 |  |
| 26 | 外观模式 | AC\_ADSystem/Facade.h | 为广告与活动子系统对外借口提供统一界面方便使用 |  |
| 27 | 状态模式 | Order/Order.h | 借助状态类代表（改变）订单的支付状态 |  |

# UI子系统

User Interface Subsystem

## 模块模式

Module Pattern

模式场景：在全局使用的类中添加私有属性和私有方法，减少全局变量的使用

### 类图

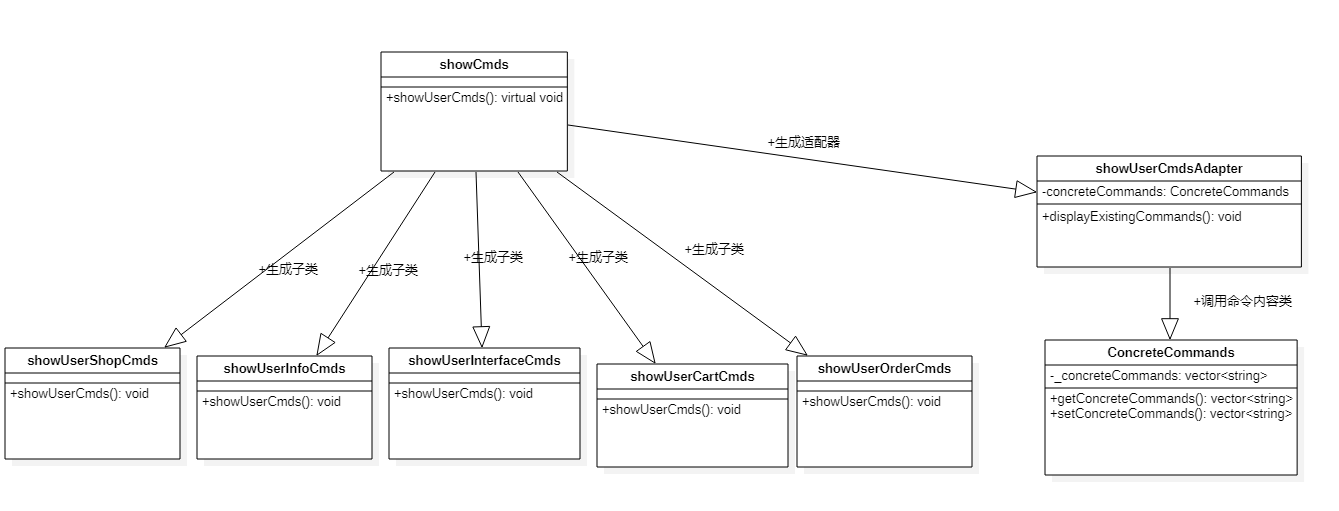


图 1模块模式（UI）

### 详述

模块模式Module Pattern用于将全局中要使用到的成员如类、变量、函数等打包成一个“模块”。在本例的情况中，使用模块模式将所有在交互界面给用户提供的命令提示打包成模块，组成全局的用户命令提示UI，在主函数中调用。showCmds为一个全局的命令提示类，其中的各个派生子类分别代表了面对不同子系统的命令，函数用来输出其对应的命令。

这样做的目的和好处有二：1、可以减少重复的代码量，加强代码的整洁度和可读性。2、打包所有要用到的零碎命令，加强输出信息的结构性和系统性。

### 核心代码

/\* 设计模式 模块模式 Module Pattern   
 \* 模块模式一般用于集成、打包全局要使用的相关的类、方法、函数   
 \* 这个类集成了全局中主程序与用户交互时给用户输出的信息。   
 \* showCmds为基类，其中有一个函数showUserCmds()以及适配器等   
 \* 该函数在具体派生类中会根据具体情况向用户输出不同的交互信息。   
 \* 目前有用户命令、店铺命令、商品命令、购物车命令、订单命令等。   
 \* 上述函数只负责输出命令提示，用户可以根据上述提示在主函数进行多种操作   
 \*/   
class showCmds {   
public:   
 virtual void showUserCmds() = 0;   
};   
   
class showUserShopCmds : public showCmds {   
public:   
 void showUserCmds();   
};   
   
class showUserInfoCmds : public showCmds {   
public:   
 void showUserCmds();   
};   
   
class showUserInterfaceCmds : public showCmds {   
public:   
 void showUserCmds();   
};   
   
class showUserCartCmds : public showCmds {   
public:   
 void showUserCmds();   
};   
   
class showUserOrderCmds : public showCmds {   
public:   
 void showUserCmds();   
};   
   
void Initial();   
   
void showUserShopCmds::showUserCmds() {   
 cout << endl;   
 cout << "欢迎您使用天猫双十一购物节店铺板块功能！" << endl;   
 cout << "1 - 查看所有店铺" << endl;   
 cout << "2 - 评价指定ID店铺" << endl;   
 cout << "3 - 根据评分筛选店铺" << endl;   
 cout << "4 - 查看指定店铺详细信息" << endl;   
 cout << "5 - 查看所有店铺商品列表" << endl;   
 cout << "6 - 查看指定ID店铺商品列表" << endl;   
 cout << "0 - 退出" << endl;   
}   
   
void showUserInfoCmds::showUserCmds() {   
 cout << endl;   
 cout << "您正在查看您的用户信息！" << endl;   
 cout << "1 - 设置用户信息" << endl;   
 cout << "2 - 查看用户信息" << endl;   
 cout << "3 - 修改用户信息" << endl;   
 cout << "0 - 退出" << endl;   
}   
   
void showUserCartCmds::showUserCmds() {   
 cout << endl;   
 cout << "您正在查看您的购物车信息！" << endl;   
 cout << "1 - 查看购物车" << endl;   
 cout << "2 - 将指定商品加入购物车" << endl;   
 cout << "3 - 删除购物车中指定商品" << endl;   
 cout << "4 - 减少购物车中指定商品" << endl;   
 cout << "5 - 结算购物车" << endl;   
 cout << "0 - 退出" << endl;   
}   
   
void showUserInterfaceCmds::showUserCmds() {   
 cout << endl;   
 cout << "欢迎光临天猫双十一购物节！请选择操作！" << endl;   
 cout << "1 - 用户信息" << endl;   
 cout << "2 - 店铺" << endl;   
 cout << "3 - 商品" << endl;   
 cout << "4 - 活动与广告" << endl;   
 cout << "5 - 购物车" << endl;   
 cout << "6 - 查看订单列表" << endl;   
 cout << "0 - 退出" << endl;   
}   
   
void showUserOrderCmds::showUserCmds() {   
 cout << endl;   
 cout << "您正在查看您的订单信息！" << endl;   
 cout << "1 - 查看您进行中的订单" << endl;   
 cout << "2 - 支付全部未支付的订单" << endl;   
 cout << "3 - 查看所有订单" << endl;   
 cout << "0 - 退出" << endl;   
}   
   
void showUserCmdsAdapter::displayExistingCommands() {   
 vector<string> \_AdapterCommands = {"用户命令", "店铺命令", "商品命令", "购物车命令", "订单命令"};   
 concreteCommands.setConcreteCommands(\_AdapterCommands);   
 cout << "已有的用户可以使用的命令提示集合: " << endl;   
 for (string \_commands:concreteCommands.getConcreteCommands()) {   
 cout << \_commands << endl;   
 }   
}   
   
void Initial() {   
 shopInterface shopInterface;   
 shopInterface.initialize();   
}

### 出处

|  |
| --- |
| We can use functions and closure to make modules. A module is a function or object that presents an interface but that hides its state and implementation. By using functions to produce modules, we can almost completely eliminate our use of global variables.  The general pattern of a module is a function that defines private variables and functions; creates privileged functions which, through closure, will have access to the private variables and functions; and that returns the privileged functions or stores them in an accessible place.  Use of the module pattern can eliminate the use of global variables. It promotes information hiding and other good design practices. It is very effective in encapsulating applications and other singletons.[1] |

# 顾客子系统

**Customer Subsystem**

## 建造者模式

**Builder**

模式场景：在创建不同级别的复杂用户类型的代码中应用建造者模式

### 类图

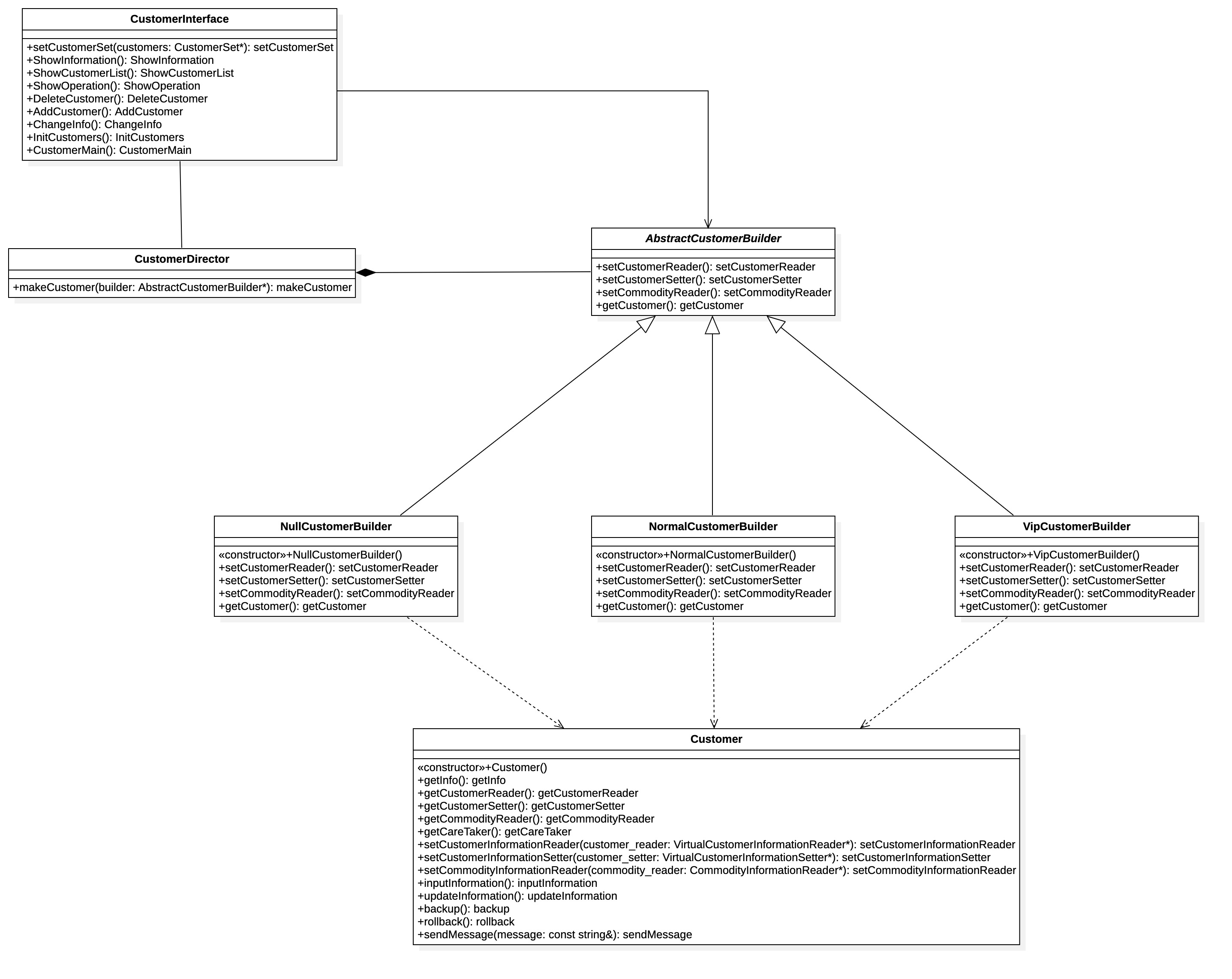


图 2建造者模式（Customer）

### 详述

**AbstractCustomerBuilder**

定义了创建Customer对象的过程，提供了构建不同组成部分的接口。setCustomerReader setCustomerSetter setCommodityReader是对Customer对象不同部分的构建函数接口，由AbstractCustomerBuilder的派生类具体实现。

**NullCustomerBuilder**

具体实现了面向未登录用户的的构建接口的具体实现。

**NormalCustomerBuilder**

具体实现了面向普通用户的的构建接口的具体实现。

**VipCustomerBuilder**

具体实现了面向vip用户的的构建接口的具体实现。

**CustomerDirector::makeCustomer**

CustomerDirector::makeCustomer固定了各个组成部分的装配方式，具体怎样装配由AbstractCustomerBuilder的派生类实现。

### 代码实现

class AbstractCustomerBuilder{  
public:  
 virtual void setCustomerReader()=0;//构成CustomerReader的接口  
 virtual void setCustomerSetter()=0;//构成CustomerSetter的接口  
 virtual void setCommodityReader()=0;//构成CommodityReader的接口  
 virtual Customer\* getCustomer()=0;//返回Customer的接口  
};  
//面向未登录用户的的构建接口的具体实现  
class NullCustomerBuilder:public AbstractCustomerBuilder{  
private:  
 Customer \*\_customer;  
public:  
 NullCustomerBuilder(){  
 \_customer=new Customer();  
 };  
 void setCustomerReader() override;  
 void setCustomerSetter() override;  
 void setCommodityReader() override;  
 Customer\* getCustomer() override;  
};  
//面向普通用户的的构建接口的具体实现  
class NormalCustomerBuilder:public AbstractCustomerBuilder{  
private:  
 Customer \*\_customer;  
public:  
 NormalCustomerBuilder(){  
 \_customer=new Customer();  
 };  
 void setCustomerReader() override;  
 void setCustomerSetter() override;  
 void setCommodityReader() override;  
 Customer\* getCustomer() override;  
};  
//面向vip用户的的构建接口的具体实现  
class VipCustomerBuilder:public AbstractCustomerBuilder{  
private:  
 Customer \*\_customer;  
public:  
 VipCustomerBuilder(){  
 \_customer=new Customer();  
 };  
 void setCustomerReader() override;  
 void setCustomerSetter() override;  
 void setCommodityReader() override;  
 Customer\* getCustomer() override;  
};  
  
class CustomerDirector{  
public:  
 //固定了各个组成部分的装配方式，具体怎样装配由AbstractCustomerBuilder的派生类实现  
 static void makeCustomer(AbstractCustomerBuilder\* builder){  
 builder->setCommodityReader();  
 builder->setCustomerReader();  
 builder->setCustomerSetter();  
 }  
};

## 观察者模式

**Observer**

模式场景：对双十一活动开始和结束进行监听的代码中应用观察者模式

### 类图

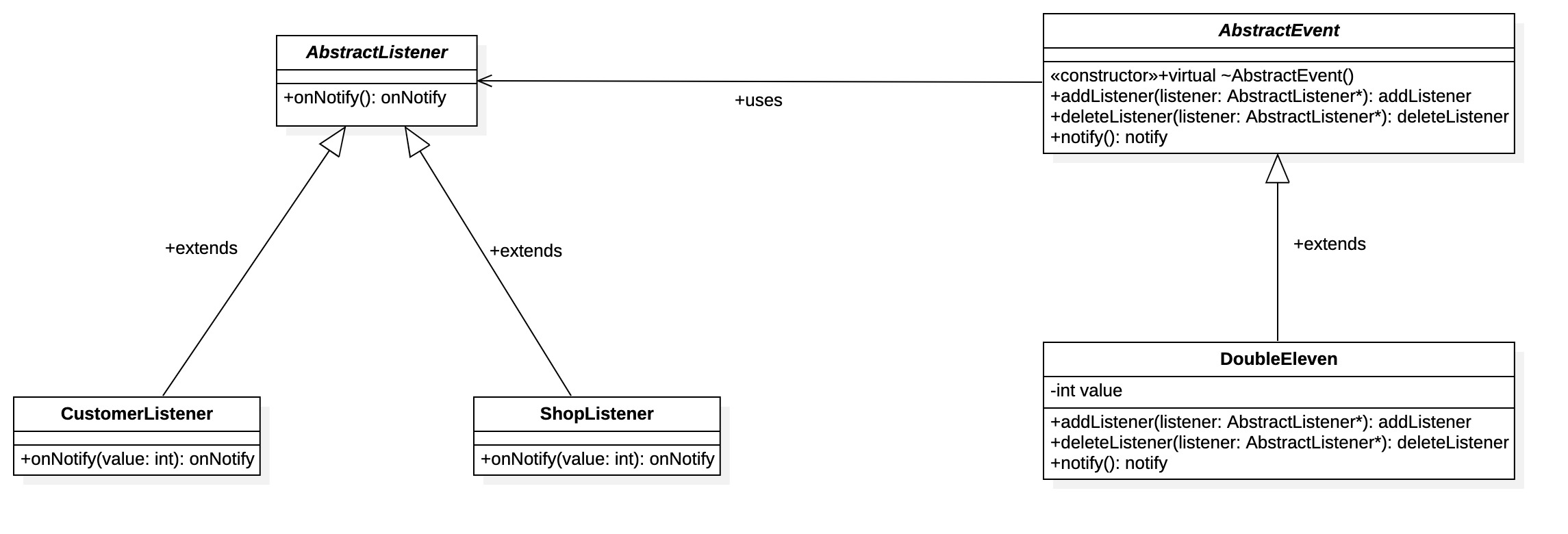


图 3观察者模式（Customer）

### 详述

**AbstractEvent**

定义了抽象的事件类和监听的接口，具体的监听事件由实体类实现。

**DoubleEleven**

抽象事件类的实体类，表示双十一活动的开始和结束事件，定义了监听者的vector容器 ，和表示监听情况的int类型value，当int值是偶数表示双十一的开始，奇数表示双十一结束，每次监听事件发生之后切换状态。

**AbstractListener**

监听者（观察者）的抽象类，定义了接收到监听信号之后的行为，具体实现行为由实体类实现。

**CustomerListener**

表示顾客客户端的监听，收到监听信号之后发送一条消息提醒消费者。

**ShopListener**

表示商店的监听，收到监听信号之后发送一条消息提醒消费者。

### 代码实现

class AbstractListener;   
class AbstractEvent {   
public:   
 virtual ~AbstractEvent()=default;   
 virtual void addListener(AbstractListener\* listener)=0;   
 virtual void deleteListener(AbstractListener\* listener)=0;   
 virtual void notify()=0;   
};   
   
class DoubleEleven:public AbstractEvent{   
public:   
 void addListener(AbstractListener\* listener);   
 void deleteListener(AbstractListener\* listener);   
 void notify();   
 ~DoubleEleven();   
private:   
 vector<AbstractListener\*> listeners;   
 int value=0;   
};   
   
class AbstractListener{   
public:   
 virtual void onNotify(int)=0;   
};   
   
class CustomerListener:public AbstractListener{   
public:   
 void onNotify(int value){   
 if(value%2==0){   
 cout<<"[系统]：双十一活动已经开始！！！开始选购吧"<<endl;   
 }else{   
 cout<<"[系统]：双十一活动即将结束！！！抓紧时间选购吧"<<endl;   
 }   
 }   
};   
   
class ShopListener:public AbstractListener{   
public:   
 void onNotify(int value){   
 if(value%2==0){   
 cout<<"[\*\*店铺]：双十一活动已经开始！！！欢迎前来本店选购"<<endl;   
 }else{   
 cout<<"[\*\*店铺]：双十一活动即将结束！！！不来本店看看嘛"<<endl;   
 }   
 }   
};

## 中介者模式

**Mediator**

模式场景：实现一个所有人的聊天室，多个用户可以向聊天室发送消息，聊天室向所有用户显示消息。

### 类图

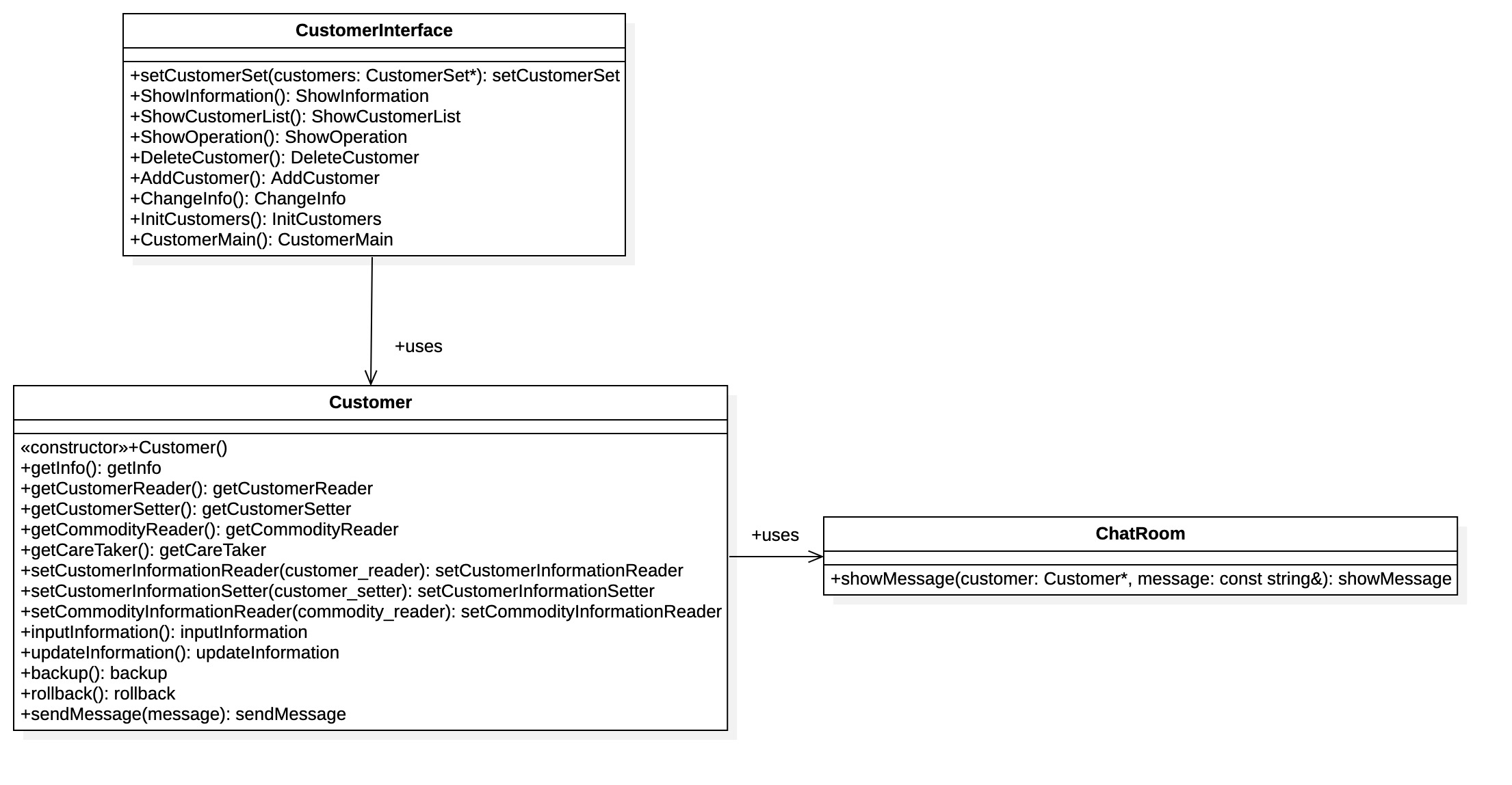


图 4中介者模式（Customer）

### 详述

将Customer之间的通信封装到一个类中单独处理，设计了Customer和ChatRoom，Customer用ChatRoom分享他们的消息。CustoemrInterface是演示类。

### 代码实现

//Customer.h   
//聊天室   
class ChatRoom {   
public:   
 static void showMessage(Customer\* customer,const string& message);   
};   
   
//Customer.cpp   
//Customer发送消息   
void Customer::sendMessage(const string &message) {   
 ChatRoom::showMessage(this,message);   
}

## 备忘录模式

**Memento**

模式场景：用户的信息修改之后后悔，在回滚信息的的代码中应用备忘录模式

### 类图

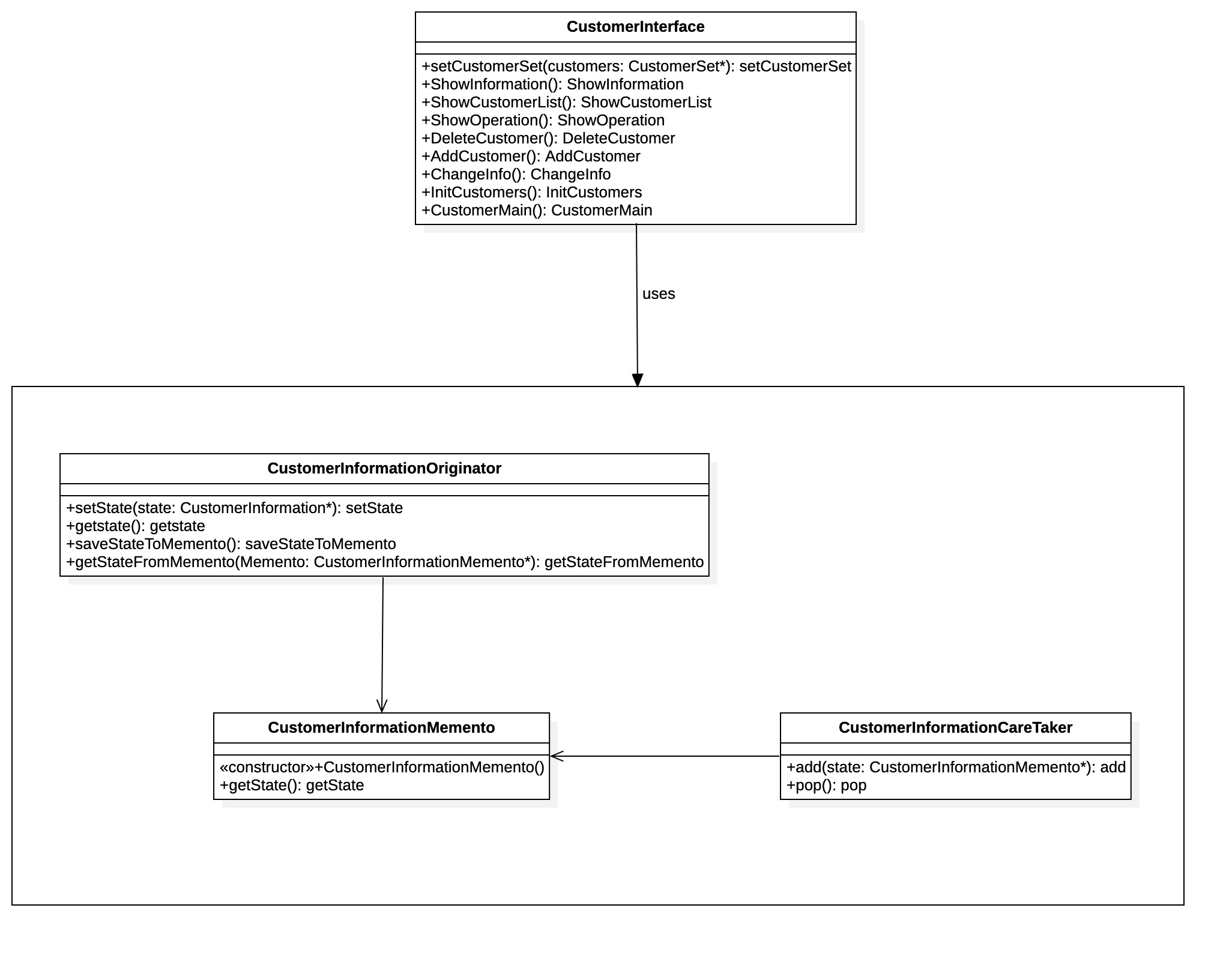


图 5备忘录模式（Customer）

### 详述

**CustomerInformationOriginator**

用于创建并在CustomerInformationMemento对象中存储状态。

**CustomerInformationMemento**

CustomerInformationMemento用于包含要被恢复的CustomerInformation的状态。

**CustomerInformationCareTaker**

CustomerInformationCareTaker 对象负责从 CustomerInformationMemento中恢复对象的状态。

### 代码实现

//用户信息回退, Memento Pattern   
class CustomerInformationMemento{   
private:   
 CustomerInformation\* state;   
public:   
 explicit CustomerInformationMemento(CustomerInformation\* state){   
 this->state=state;   
 }   
 CustomerInformation\* getState(){   
 return state;   
 }   
};   
class CustomerInformationOriginator{   
private:   
 CustomerInformation\* \_state;   
public:   
 void setState(CustomerInformation\* state){   
 this->\_state=state;   
 }   
 CustomerInformation\* getstate(){   
 return \_state;   
 }   
 CustomerInformationMemento\* saveStateToMemento(){   
 return new CustomerInformationMemento(\_state);   
 }   
 void getStateFromMemento(CustomerInformationMemento\* Memento){   
 \_state=Memento->getState();   
 }   
};   
class CustomerInformationCareTaker{   
private:   
 stack<CustomerInformationMemento\*> mementoList;   
public:   
 void add(CustomerInformationMemento\* state){   
 mementoList.push(state);   
 }   
 ~CustomerInformationCareTaker(){   
 while(!mementoList.empty()){   
 delete mementoList.top();   
 mementoList.pop();   
 }   
 }   
 CustomerInformationMemento\* pop(){   
 if(mementoList.size()>1){   
 delete mementoList.top();   
 mementoList.pop();   
 }   
 return mementoList.top();   
 }   
};

## 模板方法

**Template**

模式场景：用户信息的阅读器，整体的显示的顺序是固定的，可以应用模板模式

### 类图

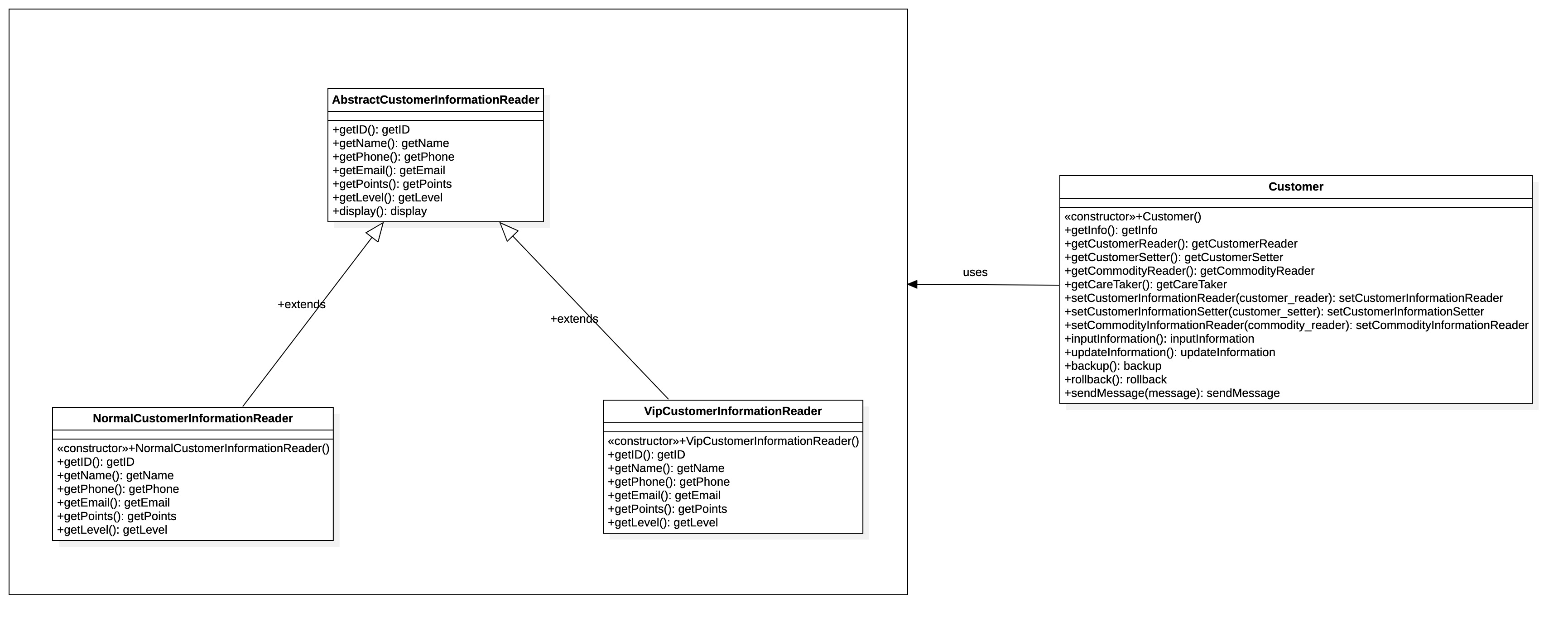


图 6模板方法（Customer）

### 详述

创建AbstractCustomerInformationReader抽象类，模板方法为display()，NormalCustomerInformationReader和VipCustomerInformationReader拓展了AbstractCustoemrInformationReader的实体类，重写了抽象类的方法。

## 单例模式

**Singleton**

模式场景：多用户登录管理每个应用程序只有一个，可以应用单例模式

### 类图

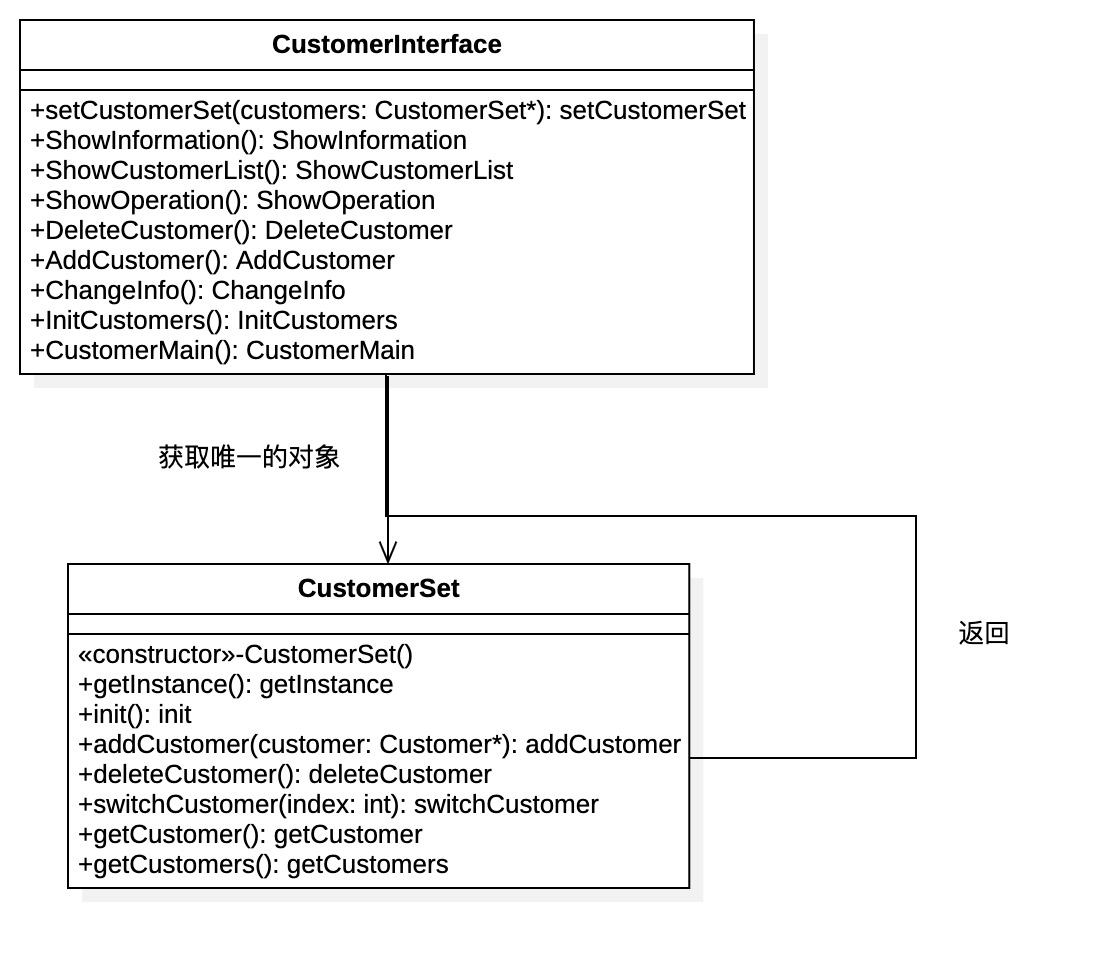


图 7单例模式（Customer）

### 详述

创建CustomerSet类。CustomerSet类有它的私有构造函数和本身的一个静态实例。CustomerSet类提供了一个静态方法，供外界获取它的静态实例。

### 代码实现

class CustomerSet{   
private:   
 vector<Customer\*> \_customers;   
 int \_index;   
 static CustomerSet\* \_instance;   
 CustomerSet(){};   
public:   
 static CustomerSet\* getInstance(){   
 if(\_instance==nullptr)   
 \_instance=new CustomerSet;   
 return \_instance;   
 }   
 void init();   
 void addCustomer(Customer\* customer);   
 void deleteCustomer();   
 bool switchCustomer(int index);   
 Customer\* getCustomer();   
 vector<Customer\*> getCustomers();   
};

# 店铺子系统

**Shop Subsystem**

**子系统的作用与功能**

1. 浏览所有店铺概要信息
2. 对店铺进行评价
3. 根据评分筛选店铺
4. 查看店铺的详细信息
5. 查看所有店铺、指定店铺的商品列表

## 抽象工厂模式

**Abstract Factory**

模式场景：围绕一个超级工厂创建其他工厂

### 类图

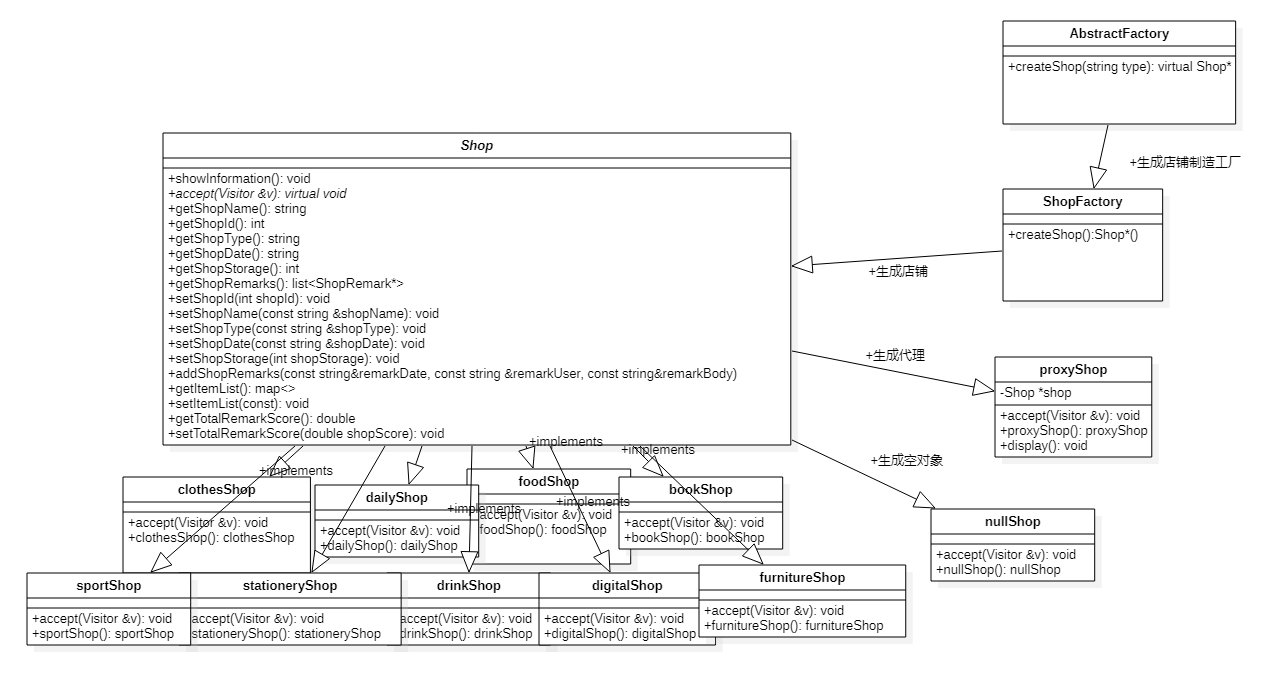


图 8抽象工厂模式（Shop）

### 详述

AbstractFactory为各个Factory的生产工厂，即由该抽象工厂先生产对应的工厂，后续再由各工厂生产其所对应的产品。其中生产Shop的工厂的shopFactory的函数createShop()对应声明为虚函数，以供工厂后续实际使用时重新定义和具体实现。

## 工厂模式

**Factory**

模式场景：创建对象

### 类图

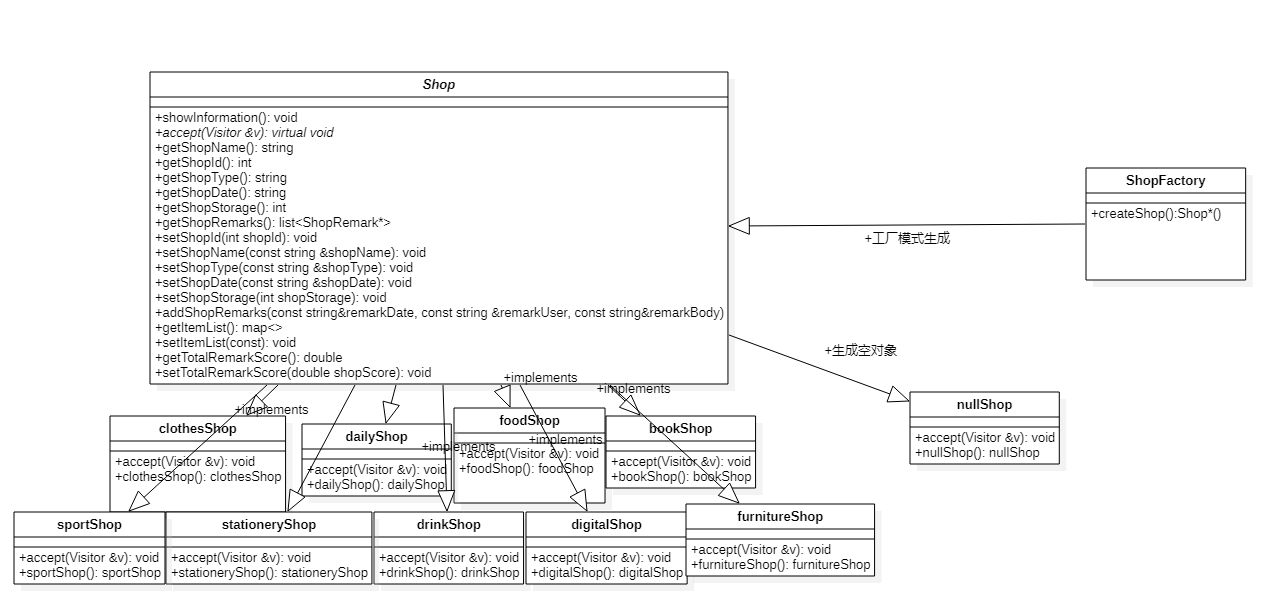


图 9工厂模式（Shop）

### 详述

accept()函数被声明为虚函数，根据基类所派生的子类的不同情况和访问者模式所派生的不同子类，决定具体该怎么执行。

实例化一个shopFactory工厂，调用工厂中的createShop()函数，该函数会根据用户所提供的类型type返回一个店铺。

### 核心代码

Shop \*ShopFactory::createShop(string type) {   
 ...   
   
 else if (type == "food") // 不同的商铺种类   
 return new foodShop("newFoodshop", 1, "food", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 else if (type == "drink")   
 return new drinkShop("newDrinkshop", 2, "drink", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 else if (type == "digital")   
 return new digitalShop("newDigitalshop", 3, "digital", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 else if (type == "book")   
 return new bookShop("newBookshop", 4, "book", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 else if (type == "daily")   
 return new dailyShop("newDailyshop", 5, "daily", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 ...   
}

### 实现API

ShopFactory::createShop(string type):Shop\*，生成不同类型的商铺。

## 空对象模式

**Null Object**

模式场景：一个空对象取代 NULL 对象实例的检查

### 类图

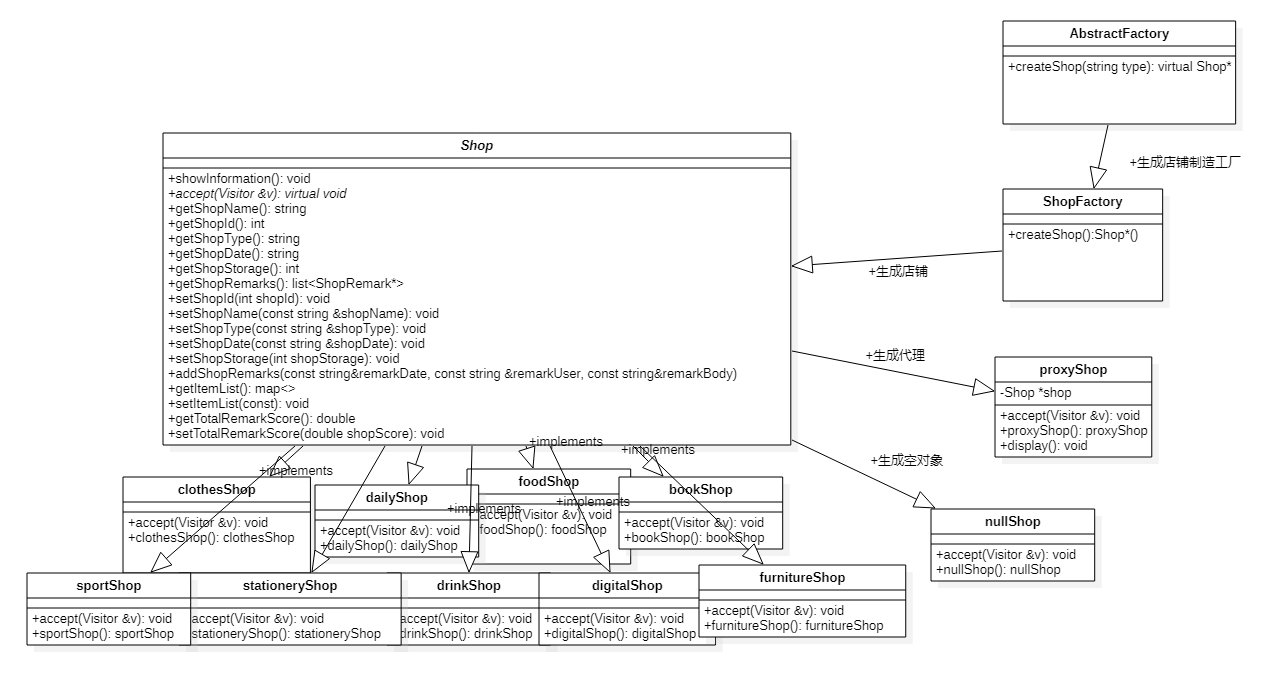


图 10空对象模式（Shop）

### 详述

实例化一个shopFactory工厂时，如果用户所提供的类型为null的话，一般的处理方法是输出一条错误信息然后返回空指针或空对象。但空对象模式支持用户未输入参数或输入参数为null类型的情况下仍然正确生成对象，符合OCP原则的同时增强了程序的工作能力和健壮性。

### 核心代码

Shop \*ShopFactory::createShop(string type) {   
 if (type == "")   
 return new nullShop("unknown", 0, "unknown", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 else if (type == "food")   
 return new foodShop("newFoodshop", 1, "food", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 ...   
   
 else if (type == "sport")   
 return new sportShop("newSportshop", 9, "sport", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
   
 return nullptr;   
}

### 实现API

ShopFactory::createShop(string type):Shop\*，生成不同类型的商铺。

### 出处

|  |
| --- |
| Those of us who have been using C-based languages for a long time have grown accustomed to functions that return null or 0 on some kind of failure. We presume that the return value from such functions needs to be tested. The NULL OBJECT pattern changes this. By using this pattern, we can ensure that functions always return valid objects, even when they fail. Those objects that represent failure do “nothing.”[2] |

## 访问者模式

**Visitor**

模式场景：使用了一个访问者类，改变元素类的执行算法

### 类图

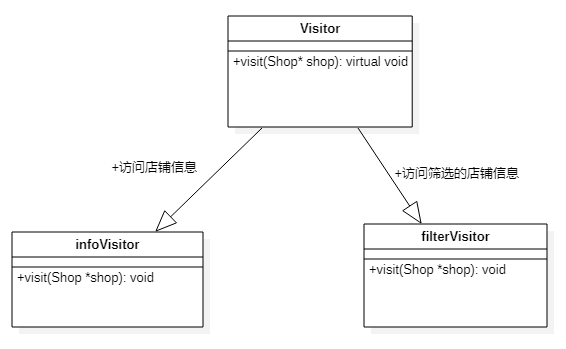


图 11观察者模式（Shop）

### 详述

Visitor作为访问者模式的基类，声明了纯虚函数visit()用来具体执行Shop类中展示对应信息的函数。

派生类infoVisitor会调用visit()来调用Shop中showShopInformation()函数以展示店铺的详细信息。

派生类filterVisitor会调用visit()来调用Shop中showShopInformation()函数以展示利用过滤器模式筛选之后店铺的详细信息。

### 核心代码

class Visitor {   
public:   
 virtual void visit(Shop \*shop) = 0;   
};   
   
class infoVisitor : public Visitor {   
public:   
 void visit(Shop \*shop) {   
 shop->showShopInformation();   
 }   
};   
   
class filterVisitor : public Visitor {   
public:   
 void visit(Shop \*shop) {   
 cout << shop->getShopId() << " - 店铺名称: " << shop->getShopName() << endl;   
 }   
};

## 过滤器模式

**Intercepting Filter**

模式场景：使用不同的标准来过滤一组对象，通过逻辑运算以解耦的方式把它们连接起来

### 类图

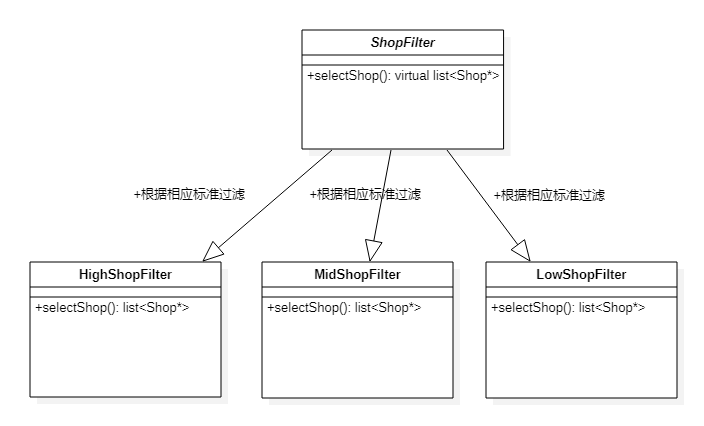


图 12过滤器模式（Shop）

### 详述

ShopFilter是店铺过滤器模式的基类，里面定义了根据相应标准过滤店铺的选择标准（店铺评分），符合对应标准的店铺会根据店铺评分被过滤到相应的过滤器HighShopFilter,MidShopFilter,LowShopFilter中，以此实现同一属性不同值的店铺的分离。

### 核心代码

类声明：

class ShopFilter {   
public:   
 //Define as an abstract class and functions.   
 virtual list<Shop \*> selectShop(list<Shop \*> ShopList) = 0;   
};   
   
class HighShopFilter:public ShopFilter{   
public:   
 list<Shop \*> selectShop(list<Shop \*> ShopList);   
};   
   
class MidShopFilter:public ShopFilter{   
public:   
 list<Shop \*> selectShop(list<Shop \*> ShopList);   
};   
   
class LowShopFilter:public ShopFilter{   
public:   
 list<Shop \*> selectShop(list<Shop \*> ShopList);   
};

实现：

list<Shop \*> HighShopFilter::selectShop(list<Shop \*> ShopList) {  
 list<Shop \*> HighShopList;  
 for (auto shop:ShopList) {  
 if (shop->getTotalRemarkScore() >= 7.50)  
 HighShopList.push\_back(shop);  
 }  
 return HighShopList;  
}  
  
list<Shop \*> MidShopFilter::selectShop(list<Shop \*> ShopList) {  
 list<Shop \*> MidShopList;  
 for (auto shop:ShopList) {  
 if (shop->getTotalRemarkScore() >= 5.00 && shop->getTotalRemarkScore() < 7.50)  
 MidShopList.push\_back(shop);  
 }  
 return MidShopList;  
}  
  
list<Shop \*> LowShopFilter::selectShop(list<Shop \*> ShopList) {  
 list<Shop \*> LowShopList;  
 for (auto shop:ShopList) {  
 if (shop->getTotalRemarkScore() >= 2.50 && shop->getTotalRemarkScore() < 5.00)  
 LowShopList.push\_back(shop);  
 }  
 return LowShopList;  
}

### 出处

|  |
| --- |
| *1. You want common processing to be centralized into reusable components.*  *2. The preprocessing and postprocessing components should be loosely coupled to the existing application code.*  *3. Apply common processing declaratively.*  Use an *intercepting filter* to apply reusable processing transparently before and after the actual request execution by the front and page controllers.[3] |

## 代理模式

**Proxy**

模式场景：一个类代表另一个类的功能

### 类图

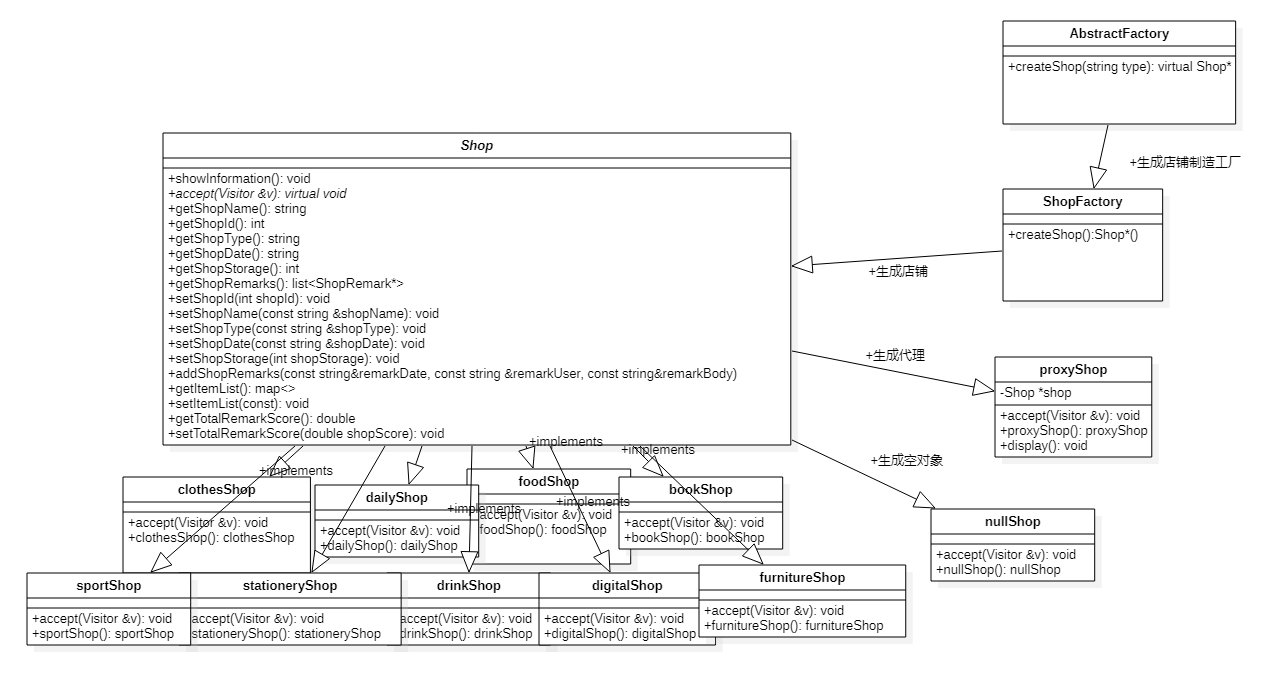


图 13代理模式（Shop）

### 详述

类proxyShop是店铺类Shop的一个子类，但是其中包含了一个Shop类型的对象，这正是代理模式的核心所在：代理对象与被代理对象属于同一个基类之下的派生子类，但是代理对象的声明中要包含被代理对象，这样，对代理对象的操作就可以转接到对被代理对象的操作，而非直接对被代理对象进行操作。在本子系统中，即通过proxyShop中的函数可以对上述其他9种Shop进行操作，实现代理功能。

### 核心代码

class proxyShop:public Shop{   
private:   
 Shop \*shop;   
public:   
 void accept(Visitor &v);   
   
 proxyShop(const string &shopName, int shopId, const string &shopType, const string &shopDate, int shopStorage,   
 double shopScore, const list<ShopRemark \*> &shopRemarks, const map<CommodityInformation \*, int> &itemList,   
 Shop \*shop);   
   
 void display();   
};   
void proxyShop::accept(Visitor &v) {   
 return v.visit(this);   
}   
   
proxyShop::proxyShop(const string &shopName, int shopId, const string &shopType, const string &shopDate,   
 int shopStorage, double shopScore, const list<ShopRemark \*> &shopRemarks,   
 const map<CommodityInformation \*, int> &itemList, Shop \*shop) : Shop(shopName, shopId, shopType, shopDate, shopStorage, shopScore, shopRemarks, itemList), shop(shop) {}   
   
void proxyShop::display() {   
 // 创建代理对象   
 if (shop == nullptr) {   
 shop = new nullShop("unknown", rand() % 10000 + 10, "unknown", "2021-01-01", 0, 0, list<ShopRemark \*>(),   
 map<CommodityInformation \*, int>());   
 }   
 shop->showShopInformation();   
}

## 适配器模式

**Adapter**

模式场景：作为两个不兼容的接口之间的桥梁

### 类图

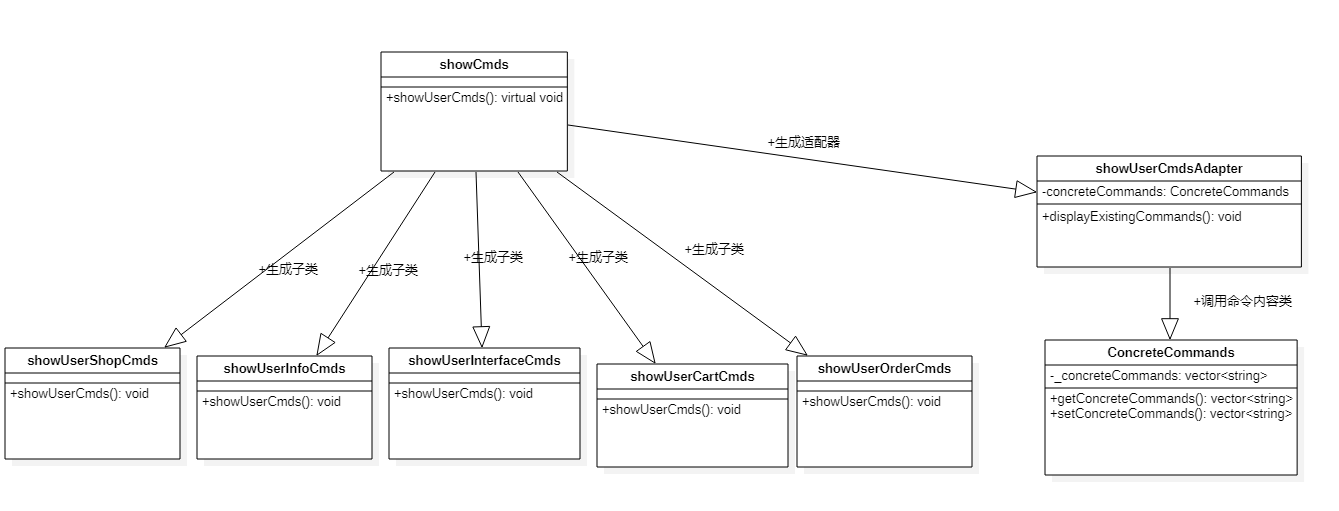


图 14适配器模式（Shop）

### 详述

在实现与用户交互界面的过程中，为了使代码更加简洁整齐以及使输出命令的语句可复用，子系统Basic其中的命令类showCmds专门用来实现与用户交互时所用到的命令和提示语句。其中虚函数showUserCmds()会根据实际上每个派生的命令类使所具体输出的语句各自不同。为了更方便地管理这些命令类，可以使用适配器模式。因为基类中没有对应功能的函数，而承载用户命令内容的类与命令类又不是同一种类，所以适配器模式可以很方便地将这两种类连接在一起。

首先使用类ConcreteCommands()读取已经存在的用户命令类中的内容；然后创建命令类的派生子类适配器showUserCmdsAdapter。调用其中的函数displayExistingCommands()即可读取到具体的内容。

### 核心代码

class ConcreteCommands {   
private:   
 vector<string> \_concreteCommands;   
public:   
 const vector<string> &getConcreteCommands() const {return \_concreteCommands;}   
   
 void setConcreteCommands(const vector<string> &concreteCommands) {\_concreteCommands = concreteCommands;}   
};   
/\*   
 \* 设计模式 - 适配器模式 Adapter Mode   
 \* 使用适配器模式可以在派生类中融入外部的其他对象和增加基类中没有的功能   
 \* 本适配器中，融入了外部一个包含所有命令集合的类，以及输出它们的新函数（基类没有）   
 \* 增强了类和对象的功能性。   
 \*/   
class showUserCmdsAdapter : public showCmds {   
private:   
 ConcreteCommands concreteCommands;   
public:   
 void displayExistingCommands();   
};

## 装饰器模式

Decorator

模式场景：向一个现有的对象添加新的功能，同时又不改变其结构

### 类图

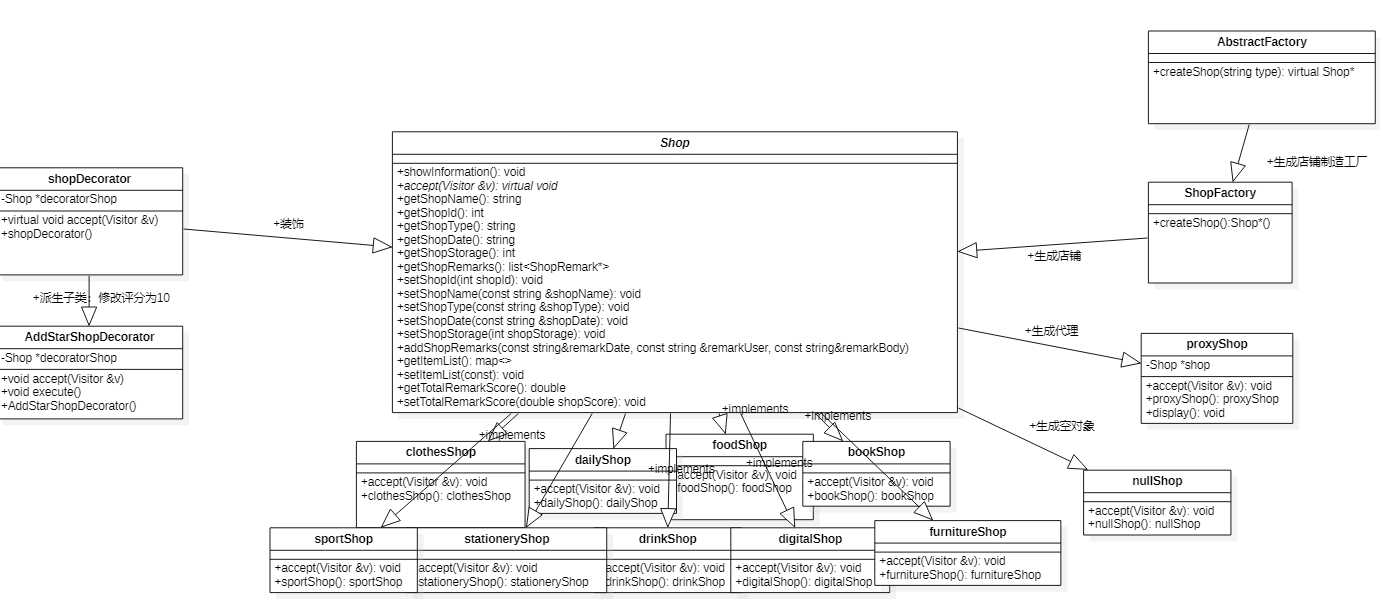


图 15装饰器模式（Shop）

### 详述

装饰器模式允许向一个现有的对象添加新的功能，同时又不改变其结构。这种类型的设计模式属于结构型模式，它是作为现有的类的一个包装。

这种模式创建了一个装饰类，用来包装原有的类，并在保持类方法签名完整性的前提下，提供了额外的功能。

shopDecorator作为店铺装饰器模式的基类，AddStarShopDecorator是在其基础上派生出的子类，其中execute()函数用来将某个特定店铺的评分置为10分。通过该装饰器以及其中的函数，可以实现基类中没有定义的修改店铺评分为10分的功能。

### 核心代码

// 商铺装饰器   
class shopDecorator : public Shop {   
protected:   
 Shop \*decoratedShop;   
public:   
 virtual void accept(Visitor &v);   
   
 shopDecorator(const string &shopName, int shopId, const string &shopType, const string &shopDate, int shopStorage, double shopScore, const list<ShopRemark \*> &shopRemarks, const map<CommodityInformation \*, int> &itemList, Shop \*decoratedShop);   
};   
   
// 商铺装饰器，向商铺添加星级   
class AddStarShopDecorator : public shopDecorator {   
public:   
   
 void accept(Visitor &v);   
   
 void execute() {   
 decoratedShop->setRemarkScore(10.00);   
 }   
   
 AddStarShopDecorator(const string &shopName, int shopId, const string &shopType, const string &shopDate, int shopStorage, double shopScore, const list<ShopRemark \*> &shopRemarks, const map<CommodityInformation \*, int> &itemList, Shop \*decoratedShop);   
};   
   
// 商铺装饰器构造函数   
shopDecorator::shopDecorator(const string &shopName, int shopId, const string &shopType, const string &shopDate, int shopStorage, double shopScore, const list<ShopRemark \*> &shopRemarks, const map<CommodityInformation \*, int> &itemList, Shop \*decoratedShop) : Shop(shopName, shopId, shopType, shopDate, shopStorage, shopScore,shopRemarks, itemList), decoratedShop(decoratedShop) {}   
   
void shopDecorator::accept(Visitor &v) {   
 return v.visit(this);   
}   
   
// 商铺星级装饰器构造函数   
AddStarShopDecorator::AddStarShopDecorator(const string &shopName, int shopId, const string &shopType, const string &shopDate, int shopStorage, double shopScore, const list<ShopRemark \*> &shopRemarks, const map<CommodityInformation \*, int> &itemList, Shop \*decoratedShop) : shopDecorator(shopName, shopId, shopType, shopDate, shopStorage, shopScore, shopRemarks, itemList, decoratedShop) {}   
   
void AddStarShopDecorator::accept(Visitor &v) {   
 setRemarkScore(10.00);   
 //return v.visit(this);   
}

## 责任链模式

Chain Of Responsibility

模式场景：对某一请求的发送者和接收者进行解耦

### 类图

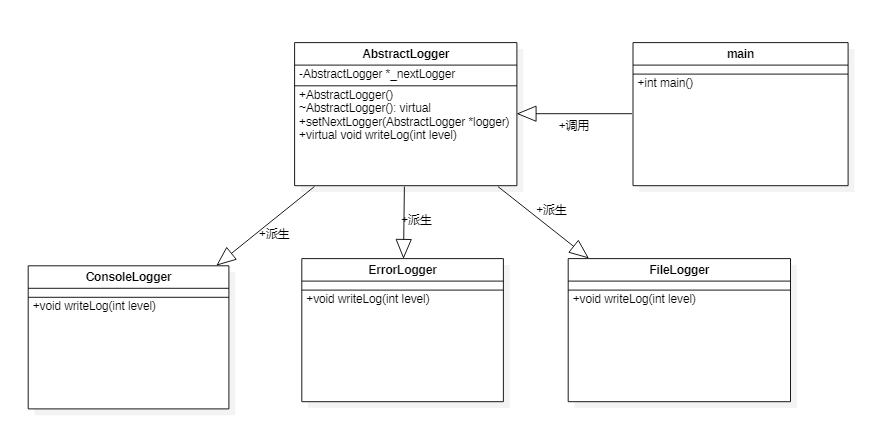


图 16责任链模式（Shop）

### 详述

责任链模式为请求创建了一个接收者对象的链。这种模式给予请求的类型，对请求的发送者和接收者进行解耦。这种类型的设计模式属于行为型模式。如果请求在本级责任链中无权执行，则传递到下一级责任节点进行判断。

责任链模式作为日志系统，在主界面内每个函数都执行完毕时均可以输出对应的信息。等级责任链按照等级1-2-3 正常-成功-失败来安排，逻辑为：如果为1则正常，不为1传递请求到下一级责任节点判断是否为2，如果为2则成功，不为2传递请求到下一级责任节点则判断是否为3，如果为3则为失败，并输出对应的日志语句。

### 核心代码

class AbstractLogger {   
protected:   
 AbstractLogger \*\_nextLogger;   
public:   
 AbstractLogger() { \_nextLogger = nullptr; }   
   
 virtual ~AbstractLogger() {}   
   
 void SetNextLogger(AbstractLogger \*logger) { \_nextLogger = logger;}   
   
 virtual void writeLog(int level) = 0;   
};   
   
class NormalLogger : public AbstractLogger {   
 virtual void writeLog(int level);   
};   
   
class ErrorLogger : public AbstractLogger {   
 virtual void writeLog(int level);   
};   
   
class SuccessLogger : public AbstractLogger {   
 virtual void writeLog(int level);   
};   
void NormalLogger::writeLog(int level) {   
 if (level == 1)   
 cout << "日志: 目前系统正常运行。" << endl;   
 else   
 \_nextLogger->writeLog(level);   
}   
   
void SuccessLogger::writeLog(int level) {   
 if (level == 2)   
 cout << "系统日志: 用户操作成功。" << endl;   
 else   
 \_nextLogger->writeLog(level);   
}   
   
void ErrorLogger::writeLog(int level) {   
 if (level == 3)   
 cout << "系统日志: 用户输入命令错误。" << endl;   
 else   
 \_nextLogger->writeLog(level);   
}

## 原型模式

Prototype

模式场景：用于创建重复的商铺对象

### 类图

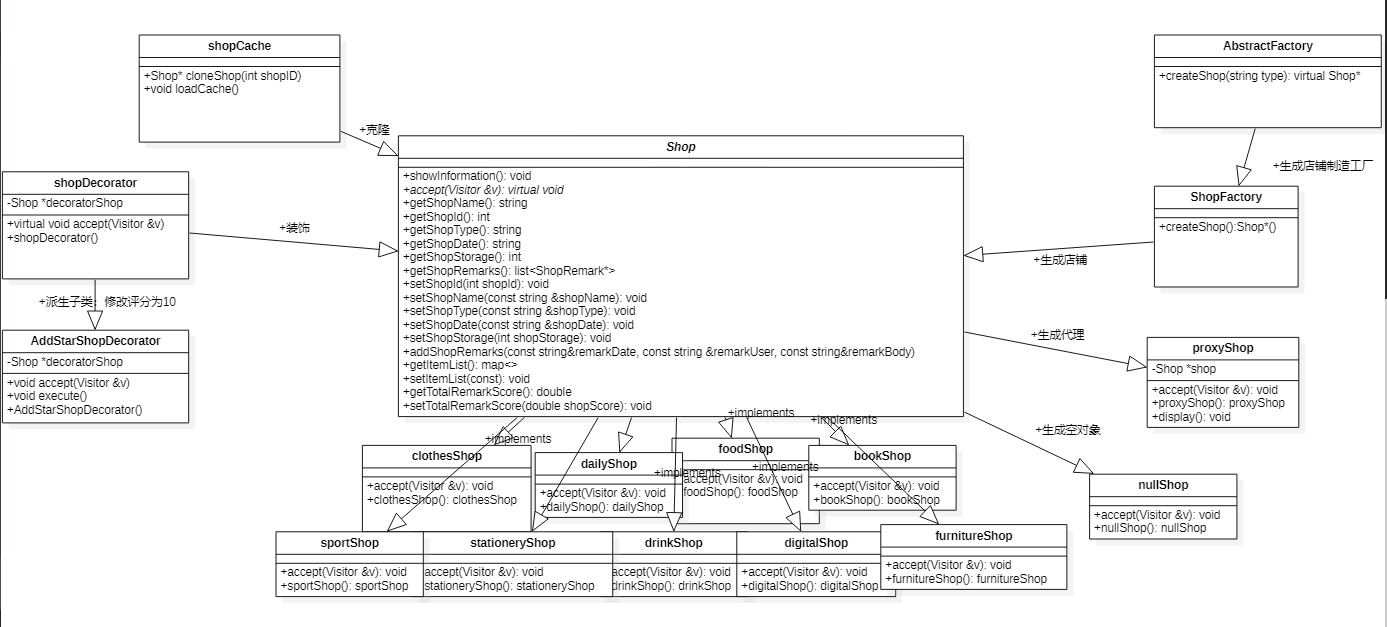


图 17原型模式（Shop）

### 详述

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。

在本项目中，初始化商店用到了原型模式。因为商店是将预先创建好的商店加入原型类中的数据缓存库，然后根据ID查找对应已经存在的商店，如果商店在数据缓存库已经有的话，直接利用已经实现的克隆函数快速完成拷贝克隆即可。

### 核心代码

/\*   
 \* 原型模式 Prototype   
 \*/   
class shopCache{   
private:   
 map<Shop\*,int> \_shopMap; //原型模式所需的商店缓存数据池   
public:   
 Shop\* cloneShop(int shopID);//获取数据池中所缓存的店铺   
 void loadCache();// 预先设置缓存的函数   
};   
/\*   
 \* 设计模式：原型模式 Prototype   
 \* 因为商店是初始化的，不需要大规模、频繁的变化，所以可以使用原型模式以完成浅拷贝   
 \* 现在原型模式自带的商店缓存中加入所有的已有商店，然后在实际初始化的时候直接使用已有的商店即可。   
 \* 确保已有的商店都是可以找到的，找不到返回空指针nullptr即可。   
 \*/   
void shopCache::loadCache() {   
 ShopRemark \*defaultRemark = new ShopRemark("2021-01-01", "Administrator", "Good.", 5);   
 list<ShopRemark \*> defaultRemarkList;   
 defaultRemarkList.push\_back(defaultRemark);   
 map<CommodityInformation \*, int> defaultGoodsList;   
 Shop \*shop1 = new foodShop("芜湖肉蛋葱鸡专营店", 1, "food", "2013-09-18", 0, 0, defaultRemarkList, defaultGoodsList);   
 \_shopMap[shop1] = 1;   
 Shop \*shop2 = new drinkShop("芜湖美汁汁儿专营店", 2, "drink", "2013-09-18", 0, 0, defaultRemarkList, defaultGoodsList);   
 \_shopMap[shop2] = 2;   
 Shop \*shop3 = new digitalShop("芜湖外设专营店", 3, "digital", "2013-09-18", 0, 0, defaultRemarkList, defaultGoodsList);   
 ... ...   
 \_shopMap[shop9] = 9;   
 Shop \*shop0 = new nullShop("芜湖空商店", 0, "", "", 0, 0, defaultRemarkList, defaultGoodsList);   
 \_shopMap[shop0] = 0;   
 Shop \*shop10 = new AddStarShopDecorator("芜湖装饰器高分空商店", 10, "", "", 0, 10, defaultRemarkList, defaultGoodsList,   
 new nullShop("", 0, "", "", 0, 0, defaultRemarkList, defaultGoodsList));   
 \_shopMap[shop10] = 10;   
}   
   
// 原型模式的寻找商店   
Shop \*shopCache::cloneShop(int shopID) {   
 for (auto \_map:\_shopMap) {   
 if (\_map.second == shopID)   
 return \_map.first;   
 }   
 return nullptr;   
}

## 依赖注入模式

Dependence Injection

模式场景：容器将某个类依赖的其他类注入到这个类中

### 类图

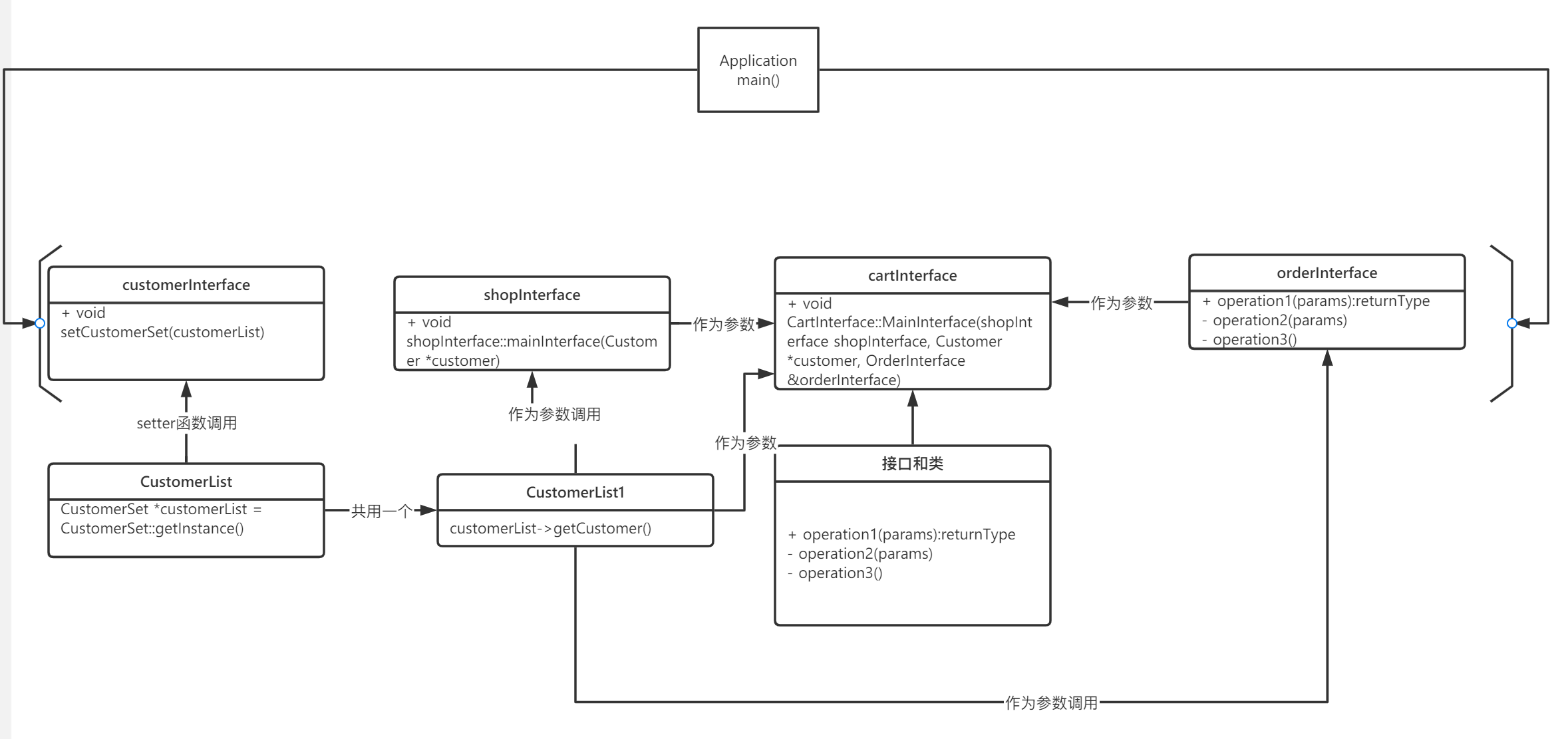


图 18依赖注入模式（Shop）

### 详述

依赖注入指当一个类的接口函数调用另一个类时，被调用类将自身注入调用类。使得在业务逻辑层面，调用类成为客户端client，被调用类成为服务器server，让这个局部调用过程和整体全局的调用过程拥有严格的业务逻辑。

依赖注入可以使用3种方式实现：通过参数传入、通过构造函数传入、setter函数。在全局Application的主过程内使用了通过参数传入的方式，然后进入具体类shopInterface调用后，即可使用通过构造函数传入、setter函数等完成。

### 核心代码

节选自Application.cpp

switch (choiceSub1) {  
 case 1: {  
 //依赖注入：将类作为参数传入  
 //依赖注入：通过setter函数传入  
 customerInterface.setCustomerSet(customerList);  
 customerInterface.CustomerMain();  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 2: {  
 //依赖注入：将类作为参数传入  
 shopInterface.mainInterface(customerList->getCustomer());  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 3: {  
 cout << "欢迎您，您正在查看本店货物信息一览大全！" << endl;  
 //依赖注入：将类作为参数传入  
 shopInterface.showCommodityList(shopInterface.getCommodityList());  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 4: {  
 // 浏览当前已有的广告和活动  
 facade.BrowseAd();  
 facade.BrowseAc();  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 5: {  
 //依赖注入：将类作为参数传入  
 cartInterface.MainInterface(shopInterface, customerList->getCustomer(), orderInterface);  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 6: {  
 //依赖注入：将类作为参数传入  
 orderInterface.MainInterface(customerList->getCustomer());  
 normal\_log->writeLog(2);  
 break;  
 }  
 case 0: {  
 break;  
 }  
 default: {  
 cout << "不存在的指令！请重新输入！" << endl;  
 normal\_log->writeLog(3);  
 break;  
 }  
 }

### 出处

|  |
| --- |
| *a. High-level modules should not depend on low-level modules. Both should depend on abstractions.*  *b. Abstractions should not depend on details. Details should depend on abstractions.*  We are already quite good at reusing low-level modules in the form of subroutine libraries. When high-level modules depend on low-level modules, it becomes very difficult to reuse those high-level modules in different contexts. However, when the high-level modules are independent of the low-level modules, then the high-level modules can be reused quite simply. This principle is at the very heart of framework design.[4] |

# 商品子系统

**Commodity Subsystem**

## 桥接模式

**Bridge**

模式场景：把抽象化与实现化解耦，使得二者可以独立变化

### 类图

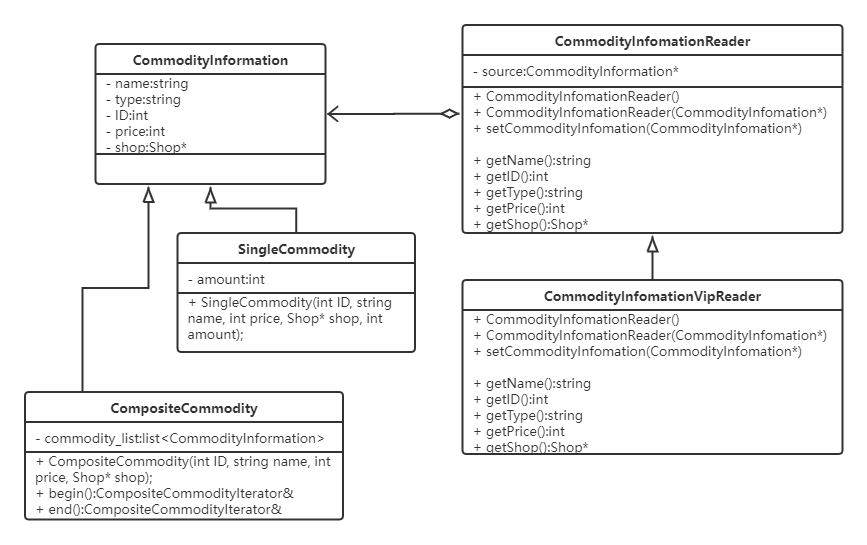


图 19商品子系统（Commodity）

### 详述

在商品子系统中，获得商品信息的功能使用了桥接模式，使得商品类的具体实现与其抽象功能分离，二者可以互不干扰地独立变化。

CommodityInformation是商品具体实现的基类，它只含有一些最基本的商品信息，比如商品名称，商品ID等。SingleCommodity是商品信息基类的一个实现，表示普通的单件商品，它有其特有的属性amount即库存。CompositeCommodity是商品信息基类的另一个实现，表示复合商品，他也有特有属性commodity\_list来存储其复合了哪些其他商品。

CommodityInformationReader是获取商品属性这一抽象功能的基类，它只有一些基本的方法函数，比如获取商品名称，获取商品价格等。CommodityInformationVipReader继承于这个基类，它针对于VIP用户，其getPrice()这个方法获取的商品价格是原价的八折。

商品实体基类可以派生出多种复杂的商品具体实现，而商品功能基类也可以派生出多样的方法，两边仅通过基类之间的友元关系“桥接”在一起，使得两种类型的类可被结构化改变而互不影响。

### 实现API

* CommodityInformationReader::getPrice():int，获得商品价格。
* CommodityInformationReader::getName():string，获得商品名称。
* CommodityInformationReader::getID():int，获得商品ID。
* CommodityInformationReader::getType():string，获得商品类型。
* CommodityInformationReader::getShop():Shop\*，获得商品的商铺。
* CommodityInformationVipReader::getPrice():int，VIP用户获得商品价格（打八折）。
* CommodityInformationVipReader::getName():string，VIP用户获得商品名称。
* CommodityInformationVipReader::getID():int，VIP用户获得商品ID。
* CommodityInformationVipReader::getType():string，VIP用户获得商品类型。
* CommodityInformationVipReader::getShop():Shop\*，VIP用户获得商品的商铺。

## 组合模式

**Composite**

模式场景：用于把一组相似的对象当作一个单一的对象

### 类图

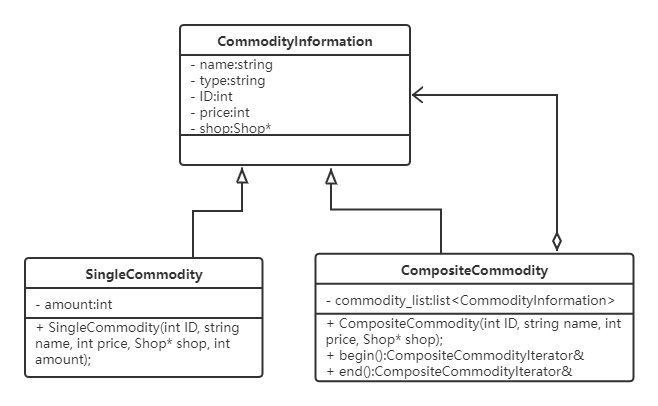


图 20组合模式（Commodity）

### 详述

组合模式用于把一组相似的对象当作一个单一的对象，依据树形结构来组合对象，用来表示部分以及整体层次。

CommodityInformation是商品基类，派生出来SingleCommodity单一商品类以及CompositeCommodity组合商品类。单一商品类特有属性amount表示库存，复合商品类特有属性commodity\_list存储指向商品基类的指针。商品基类既作为所有商品的统一接口，也作为聚合的对象存在于复合商品类的属性中，从而实现组合模式。

### 实现API

* CommodityInfomationSetter::setName(string):void，设置商品名称。
* CommodityInfomationSetter::setID(int):void，设置商品ID。
* CommodityInfomationSetter::setType(string):void，设置商品类型。
* CommodityInfomationSetter::setPrice(int):void，设置商品价格。
* CommodityInfomationSetter::addCommodity(CommodityInformation\*):bool，向组合商品中添加商品，如果调用者为单一商品类，则返回false。
* CommodityInfomationSetter::removeCommodity(CommodityInformation\*):bool，删除组合商品中的某件商品，如果调用者为单一商品类，则返回false。

## 迭代器模式

**Iterator**

模式场景：用于顺序访问集合对象的元素，且不需要知道集合对象的底层表示

### 类图

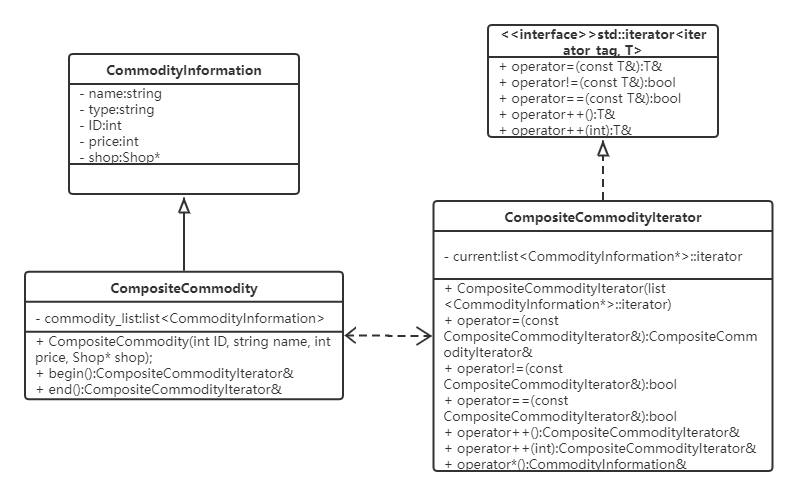


图 21迭代器模式（Commodity）

### 详述

CompositeCommodityIterator复合商品迭代器继承于STL中的iterator<iterator\_tag, T>虚基类，重载了赋值、自加、等于等操作符，使得编写代码时可以轻易使用迭代器遍历复合商品中的所有商品。

### 实现API

* CompositeCommodityIterator::opteraor=(CompositeCommodityIterator&)，迭代器赋值。
* CompositeCommodityIterator::opteraor==(CompositeCommodityIterator&)，迭代器等于判断。
* CompositeCommodityIterator::opteraor!=(CompositeCommodityIterator&)，迭代器不等于判断。
* CompositeCommodityIterator::operator++()，迭代器自加。
* CompositeCommodityIterator::operator\*()，迭代器取值。
* CompositeCommodity::begin()，获取复合商品的第一个元素的迭代器。
* CompositeCommodity::end()，获取复合商品的最后一个元素的迭代器。

## 享元模式

**Flyweight**

模式场景：用于减少创建对象的数量，以减少内存占用和提高性能

### 类图

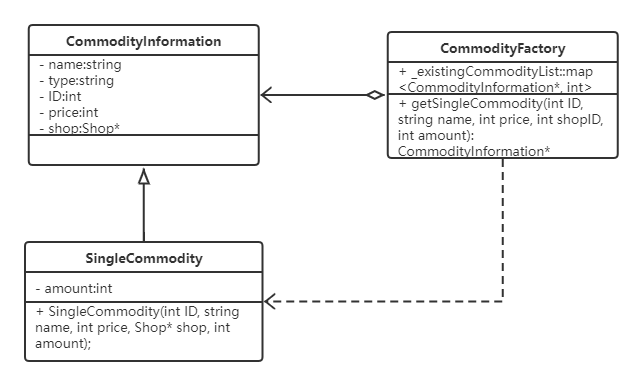


图 22享元模式（Commodity）

### 关键代码

/\*  
 \* Flyweight设计模式  
 \* 工厂在生成对象的时候优先检查哈希表里有没有，有的话直接调用该对象，没有的话再重新生成。  
 \* 可以大大减少内存的使用，提高运行效率。  
 \*/  
CommodityInformation \* CommodityFactory::getSingleCommodity(int ID, string name, int price, int shopID, int amount) {  
 for (auto iter:\_existingCommodityList) {  
 reader.setCommodityInformation(iter.first);  
 if (reader.getID() == ID)  
 return iter.first;  
 else  
 continue;  
 }  
 SingleCommodity \*singleCommodity = new SingleCommodity(ID, name, price, shopID, amount);  
 \_existingCommodityList[singleCommodity]++;  
 return singleCommodity;  
}

### 详述

在CommodityFactory商品工厂类中添加属性\_existingCommodityList，它保存了所有商品的指针以及对应的商品ID，数据结构使用了std::map，它的查找时间复杂度为，提高运行效率。在每次构造商品之前，首先遍历\_existingCommodityList，如果需要构造的商品ID已经存在，则直接返回这个已经存在的商品，不存在的情况下才调用构造函数生成SingleCommodity单件商品。从而减少应用的空间使用。

### 实现API

getSingleCommodity(int ID, string name, int price, int shopID, int amount):CommodityInformation\*，构造新商品。

## 命令模式

Command

模式场景：请求以命令的形式包裹在对象中，并传给调用对象

### 类图

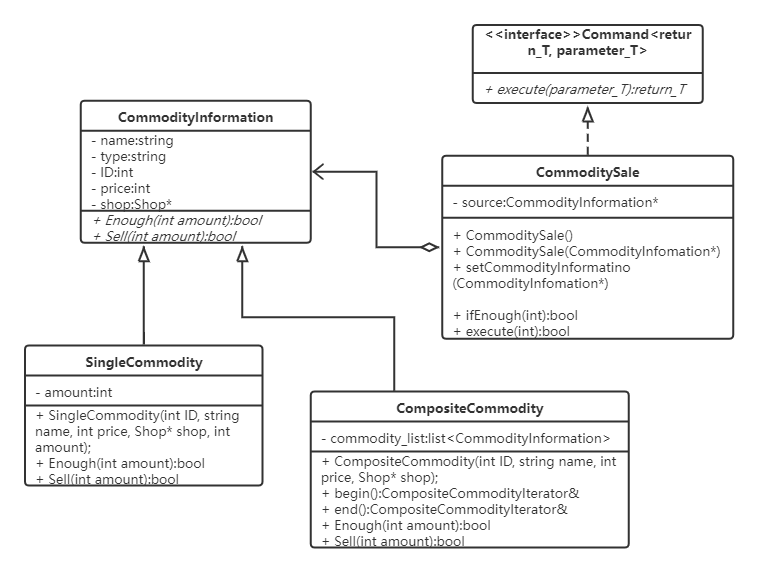


图 23命令模式（Commodity）

### 详述

Command为命令的虚基类，成员为一个execute()纯虚函数。CommoditySale类具体功能是卖出一定数量的某件商品。它承并实现了Command类，并且聚合了CommodityInformation类，表示这个命令类控制的哪件商品。

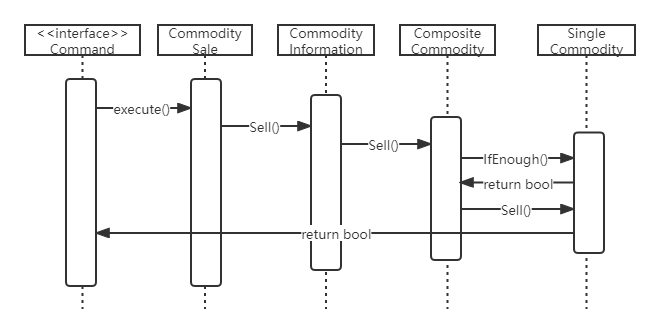


图 24商品售卖流程图

复合商品售卖流程：CommoditySale命令类调用CommodityInformation商品基类的虚函数Sell()，这个函数在复合商品类CompositeCommodity复合商品类中实现，它使用迭代器依次调用SingleCommodity单件商品类的IfEnough()方法，如果所有单件商品的库存都充足，则调用Sale()方法依次将商品卖出，并返回true，否则返回false。

### 实现API

* CommoditySale::execute()，商品售卖命令。
* CommodityDisplay::execute()，商品展示命令（复合商品会依次展示其中包含的单件商品）。

# 活动与广告子系统

**Advertisement & Activity Subsystem**

## 功能与实现

**Functions & Implementation**

### 广告子系统

* 主要功能：
  1. 添加广告
  2. 浏览广告
* 实现方案：
  1. 结构体AdNode代表的变量类型用于存储一个广告的具体内容，包括广告ID以及广告内容。
  + struct AdNode//广告   
    {   
     int AdID;   
     string Content;   
       
     AdNode(int ID, string Cont)   
     {   
     AdID = ID;   
     Content = Cont;   
     }   
    };
  1. AdvertisingSystem类用于对所有当前存在的广告进行管理，是广告系统的主体，通过开放Accept函数使用Visitor模式进行功能实现与扩展。为了保持广告系统的唯一性，使用了单例模式。
  + class AdvertisingSystem//广告子系统（单例）   
    {   
    public:   
     ~AdvertisingSystem();   
       
     static AdvertisingSystem\* GetAdSystemInstance();   
       
     void Accept(AdVisitor\* Visitor);   
       
     void PrintAd(int Index);//打印广告   
       
    private:   
       
     AdvertisingSystem();   
       
     static AdvertisingSystem\* AdSystemInstance;   
       
     vector<AdNode\*> AdvertisingList;   
    };
  1. AdVisitor类作为AdvertisingSystem的访问者，用于实现与扩展广告系统的功能，并保持AdvertisingSystem类的简洁性。AdVisitor\_Browse类与AdVisitor\_Add类均继承于AdVisitor类，分别实现了广告系统的浏览功能和添加功能。
  + class AdVisitor//广告visitor父类   
    {   
    public:   
       
     virtual ~AdVisitor() = default;   
       
     virtual void Visit(AdvertisingSystem\* AdSystem) = 0;   
    };   
    class AdVisitor\_Browse : public AdVisitor//浏览广告visitor子类   
    class AdVisitor\_Add : public AdVisitor//添加广告visitor子类

### 活动子系统

* 主要功能：
  1. 添加活动
  2. 浏览活动
  3. 计算活动价格并选择最优方案
  4. 根据商品推荐相关活动
* 实现方案：
  1. AcNode类是单个活动的基类，用于存储所有活动都拥有的内容，包括活动ID以及活动内容。
  + class AcNode//活动父类   
    {   
    public:   
       
     AcNode(int ID, string Cont);//两类活动id不可相同   
       
     int AcID;   
     string Content;   
    };
  + 从AcNode类派生出两个子类：AcNode\_Discount、AcNode\_FullReduction，分别代表打折类活动和满减类活动。
  + AcNode\_Discount中存储了打折商品列表以及打折力度这两个关键属性，AcNode\_FullReduction中存储了满减的阈值以及力度这两个关键属性。
  + 通过这些关键属性便可以提供：1.是否能使用该活动；2.使用该活动的价格为多少，这两个函数，进一步便可以提供完整的“ ExecuteActivity”这个函数用于获取通过该活动的价格。
  + class AcNode\_Discount : public AcNode//打折类活动子类   
    {   
    public:   
       
     AcNode\_Discount(int ID, string Cont, vector<int>& CommodityInformationIDList, float DisRate);   
       
     float ExecuteActivity(CommodityInformation\* BuyCommodityInformation, int Amount); // 若商品打折，则返回商品打完折的价格，否则返回原价   
       
     bool IsSatisfy(CommodityInformation\* BuyCommodityInformation);//判断商品是否打折   
       
    private:   
       
     float CalPrice(CommodityInformation\* BuyCommodityInformation, int Amount);//返回商品打完折后的价格   
       
     vector<int> SatisfyCommodityInformationIDList;   
       
     float DiscountRate;   
    };   
       
    class AcNode\_FullReduction : public AcNode//满减类活动子类   
    {   
    public:   
       
     AcNode\_FullReduction(int ID, string Cont, float Thd, float RedAmount);   
       
     float ExecuteActivity(map<CommodityInformation\*, int>& BuyCommodityInformation); //若商品达到满减金额，则返回商品满减后的价格，否则返回原价   
       
    private:   
       
     bool IsSatisfy(map<CommodityInformation\*, int>& BuyCommodityInformation);//判断商品是否满减   
       
     float CalPrice(map<CommodityInformation\*, int>& BuyCommodityInformation);//返回商品满减后的价格   
       
     float Threshold;   
     float ReductionAmount;   
    };
  1. ActivitySystem类用于管理当前存在的活动，通过开放Accept函数使用Visitor模式进行功能实现与扩展。为了保持活动系统的唯一性，使用了单例模式。因为存在两种活动，为了处理起来更加高效，使用了两个vector分别对其进行存储。
  + class ActivitySystem//活动子系统   
    {   
    public:   
     ~ActivitySystem();   
       
     static ActivitySystem\* GetAcSystemInstance();   
       
     void Accept(AcVisitor\* Visitor);   
       
     void PrintAc\_D(int Index);//打印打折类活动   
     void PrintAc\_F(int Index);//打印满减类活动   
       
    private:   
       
     ActivitySystem();   
       
     static ActivitySystem\* AcSystemInstance;   
       
     vector<AcNode\_Discount\*> DiscountActivityList;   
     vector<AcNode\_FullReduction\*> FullReductionActivityList;   
    };
  1. AcVisitor类作为ActivitySystem的访问者类，用于实现与扩展活动系统的功能，并保持ActivitySystem类的简洁性。
  + class AcVisitor//活动Visitor父类   
    {   
    public:   
     virtual ~AcVisitor() = default;   
     virtual void Visit(ActivitySystem\* AcSystem) = 0;   
    };
  + 为了实现添加活动功能，派生出了AcVisitor\_Add类：
  + class AcVisitor\_Add : public AcVisitor//添加活动Visitor子类
  + 为了实现浏览活动功能，派生出了AcVisitor\_Browse类：
  + class AcVisitor\_Browse : public AcVisitor//浏览活动Visitor子类
  + 为了实现推荐活动功能，派生出了AcVisitor\_Recommend类：
  + class AcVisitor\_Recommend : public AcVisitor//推荐活动Visitor子类
  + 为了实现计算最优价格功能，派生出了AcVisitor\_CalPrice类，因为存在两种活动方式（打折、满减），且两种活动计算价格的方法以及输入都不一样，故需要分开处理，所以再从AcVisitor\_CalPrice类派生出AcVisitor\_CP\_Discount类和AcVisitor\_CP\_FullRedu类分别用于计算打折类活动价格和满减类价格：
  + class AcVisitor\_CalPrice : public AcVisitor//计算最优活动Visitor子类   
    class AcVisitor\_CP\_Discount : public AcVisitor\_CalPrice//计算最优打折活动Visitor子类   
    class AcVisitor\_CP\_FullRedu : public AcVisitor\_CalPrice//计算最优满减活动Visitor子类
  1. Interpreter类用于解释活动方案并打印输出，在计算最优价格以及推荐活动这两个函数中都会涉及到对一些活动进行组织并给予显示，通过一个解释器使得这种组织更加方便和准确：
  + class Interpreter   
    {   
    public:   
       
     Interpreter();   
       
     void MakeInterpretation(string Code);//解释Code含义   
       
    private:   
       
     ActivitySystem\* AcSystemInstance;   
    };

### 子系统API

* 主要功能：
  1. 管理广告和活动两个子系统，包括创建子系统和使用对应功能。
  2. 对外部提供简洁的接口，隐藏后面的细节。
* 实现方案：
  1. 因为要管理多个子系统，故使用了外观模式。
  2. 存储了广告子系统和活动子系统，以及一个解释器用于解释输出。
  3. 提供的接口包括：广告方面（浏览广告、添加广告）、活动方面（浏览活动、添加活动、计算最优价格、推荐活动）。
  + class Facade//外观模式   
    {   
    public:   
       
     Facade();//构造广告子系统、活动子系统、解释器   
       
     // Ad   
     void BrowseAd();//浏览全部广告   
     void BrowseAd(int BrowseAdID);//浏览单个广告   
     void AddAd(int ID, string Cont);//添加单个广告   
       
     //Ac   
     void BrowseAc();   
     void BrowseAc(int BrowseAcID);   
     void AddAcD(int ID, string Cont, vector<int>& CommodityInformationReaderList, float DisRate);//添加打折类活动   
     void AddAcF(int ID, string Cont, float Thd, float RedAmount);//添加满减类活动   
       
     float CalOptimalDecision(map<CommodityInformation\*, int>& BuyCommodityInformationReaderMap);//寻求最优惠价格，并打印此时活动列表   
     void RecommendActivity(vector<CommodityInformation\*>& RelatedCommodityInformationReaderList);//打印与商品相关的所有活动   
     void RecommendActivity(map<CommodityInformation\*, int>& RelatedCommodityInformationReaderMap);//打印与商品相关的所有活动   
       
    private:   
     AdvertisingSystem\* AdSystemInstance;   
     ActivitySystem\* AcSystemInstance;   
     Interpreter\* ActivityInterpreter;   
    };

## 解释器模式

**Interpreter**

模式场景：提供了评估语言的语法或表达式的方式

### 类图

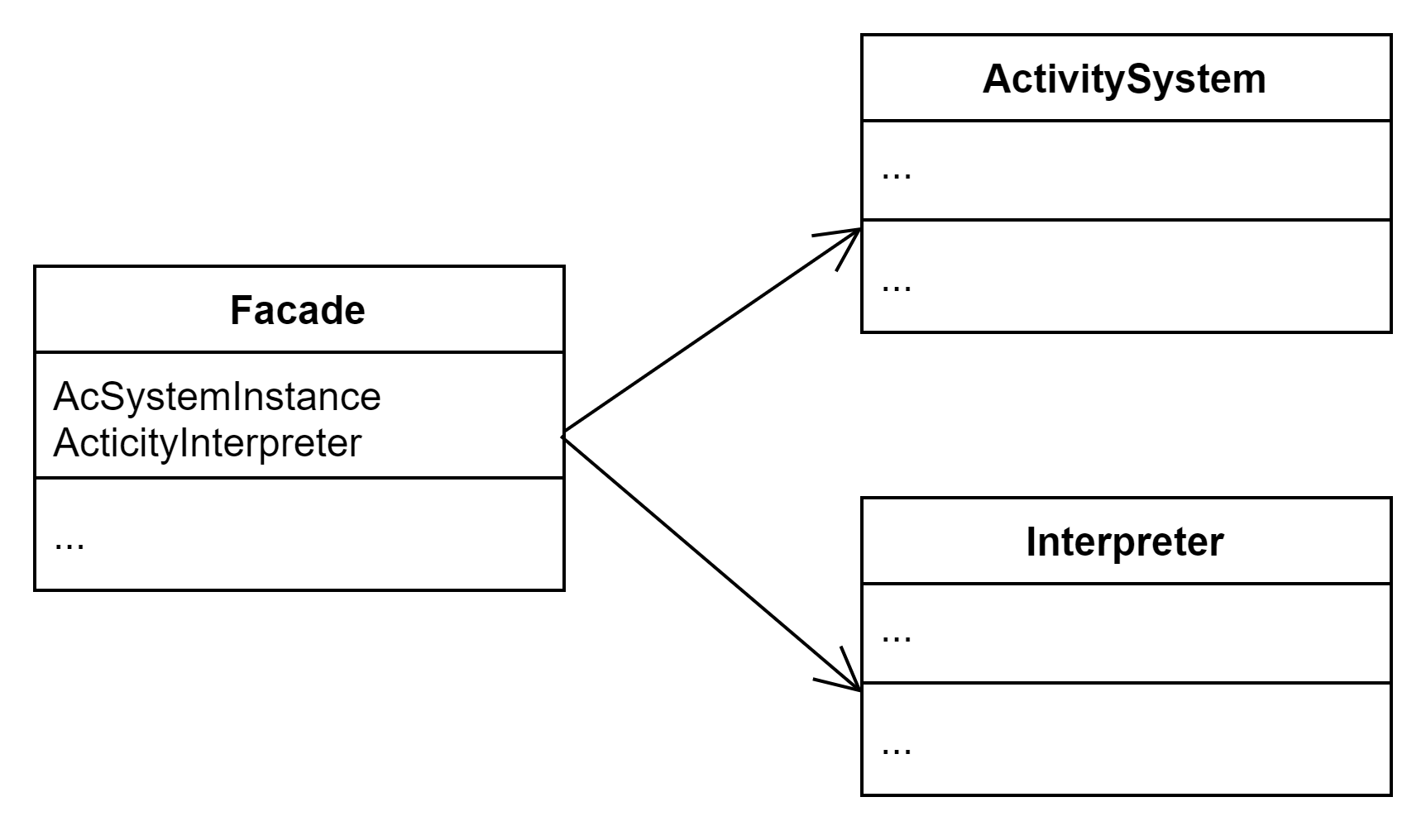


图 25解释器模式（AD）

### 详述

在计算最优价格时没有直接打印出最优价格方案，而是构造了一个表达最优活动方案的字符串DecisionCode，需要输出时再交给解释器进行解释输出。

void AcVisitor\_CP\_Discount::Visit(ActivitySystem\* AcSystem)   
{   
 string DecisionCode;   
 DecisionCode.push\_back('1'); // 1代表需要解释的Code为最优活动列表   
 DecisionCode.push\_back('1'); // 1代表所选方案为打折活动类   
 ……   
 for (auto BuyCommodityInformation : SaveBuyCommodityInformationMap)   
 {   
 DecisionCode.push\_back(120); // 后续位均代表对应的活动的下标，120即为不使用活动   
 ……   
 for (int i = 0; i < AcSystem->DiscountActivityList.size(); i++)   
 {   
 float CalPrice = ……   
 if (CalPrice < MinPrice)   
 {   
 MinPrice = CalPrice;   
 DecisionCode[DecisionCode.length() - 1] =static\_cast<char>(i); // 对应位置存储对应活动下标   
 }   
 }   
 TotalMinPrice += MinPrice;   
 }   
   
 OptimalPrice = TotalMinPrice;   
 OptimalDecisionCode = DecisionCode;   
}   
void AcVisitor\_CP\_FullRedu::Visit(ActivitySystem\* AcSystem)   
{   
 string DecisionCode;   
 DecisionCode.push\_back('1'); // 1代表需要解释的Code为最优活动列表   
 DecisionCode.push\_back('2'); // 2代表所选方案为满减活动类   
 ……   
}

同样，在推荐活动是也没有直接打印所有推荐的活动，而是构建了一个表达推荐活动方案的字符串RecommendCode，需要输出时再交给解释器进行解释输出。

void AcVisitor\_Recommend::Visit(ActivitySystem\* AcSystem)   
{   
 string RecommendCode;   
 RecommendCode.push\_back('2'); // 2代表需要解释的Code为推荐活动列表   
 RecommendCode.push\_back('-1'); // 与最优活动编码对齐   
   
 set<int> RecommendAcSet;   
   
 ……   
   
 for (auto Elem : RecommendAcSet)   
 {   
 RecommendCode.push\_back(static\_cast<char>(Elem)); // 每一位均存储一个相关的活动的下标   
 }   
   
 RecommendActivityCode = RecommendCode;   
}

解释器接受一个字符串输入进行解释，解释规则与编码规则一一对应：

void Interpreter::MakeInterpretation(string Code)   
{   
 if (Code.length() < 2) // 长度小于2即是一个错误编码，正常编码位数均在两位以上   
 return;   
   
 if (Code[0] == '1') // 第一位为1代表需要解释的Code为最优活动方案   
 {   
 if (Code[1] == '1') // 第二位为1代表选择的活动类别为打折类活动   
 {   
 cout << "最优活动类别为打折类，具体配置如下：" << endl;   
 for (int i = 2; i < Code.length(); i++)   
 {   
 if (Code[i] == 120) // 编码为120代表不使用活动或没有活动满足   
 {   
 cout << "###第" << i-1 << "件商品不活动" << endl;   
 continue;   
 }   
 cout << "###第" << i-1 << "件商品使用活动为：" << endl;   
 AcSystemInstance->PrintAc\_D(Code[i]); // 寻找对应的活动进行展示   
 }   
 }   
 else if (Code[1] == '2') // 第二位为2代表选择的活动类别为满减类活动   
 {   
 ……   
 }   
   
 }   
 else if (Code[0] == '2') // 第一位为2代表需要解释的Code为推荐活动方案   
 {   
 cout << "推荐活动为：" << endl;   
   
 for (int i = 2; i < Code.length(); i++)   
 {   
 AcSystemInstance->PrintAc\_D(Code[i]); // 对Code里的每一位代表的活动进行显示   
 }   
 }   
}

## 迭代器模式

**Iterator**

模式场景：用于顺序访问集合对象的元素，且不需要知道集合对象的底层表示

### 详述

迭代器是一种不暴露内部结构而遍历集合的设计模式，我们两种方式使用了vector的迭代器：

遍历集合元素的位置：

for (vector<AdNode\*>::size\_type ix = 0; ix != AdSystem->AdvertisingList.size(); ix++){}

遍历集合元素：

for (vector<AdNode\*>::iterator iter = AdSystem->AdvertisingList.begin();   
 iter != AdSystem->AdvertisingList.end(); ++iter){}

我们使用迭代器来访问 **广告子系统AdSystem** 中的 **广告列表AdvertisingList** 以及 **活动子系统AcSystem** 中的 **活动列表DiscountActivityList** 。

## 单例模式

**Singleton**

模式场景：多用户登录管理每个应用程序只有一个，可以应用单例模式

### 类图



图 26迭代器模式（AD）

### 详述

单例模式是指，一个类只可以创建一个实例，在我们的项目中 **广告子系统AdSystem** 和 **活动子系统AcSystem** 只能有一个，它们分别存放着所有广告与活动，并且对它们进行操作。

以 **广告子系统AdSystem** 为例，我们单例的实现如下：

1、首先在类的定义时，需要有一个私有、静态的、指向当前类的指针，用于存放单例；  
并且将构造函数设为私有，避免外部创建该类的实例，取而代之的是静态公有成员函数 **GetAdSystemInstance（）**；

class AdvertisingSystem   
{   
public:   
 static AdvertisingSystem\* GetAdSystemInstance();   
 ……   
private:   
 AdvertisingSystem();   
 static AdvertisingSystem\* AdSystemInstance;   
 ……   
};

2、然后要将指向单例的指针初始化为 **nullptr** ；

AdvertisingSystem \*AdvertisingSystem::AdSystemInstance = nullptr;

最后**GetAdSystemInstance（）**的实现如下：  
若指向单例的指针为 **nullptr** ，则说明还未生成单例，需要生成一个当前类的实例，并让指针指向这个实例；  
此时指针一定指向单例的实例，所以只需返回该指针即可；

AdvertisingSystem \*AdvertisingSystem::GetAdSystemInstance() {   
 if (AdSystemInstance == nullptr)   
 AdSystemInstance = new AdvertisingSystem();   
 return AdSystemInstance;   
}

## 访问者模式

**Visitor**

模式场景：使用了一个访问者类，改变元素类的执行算法

### 类图

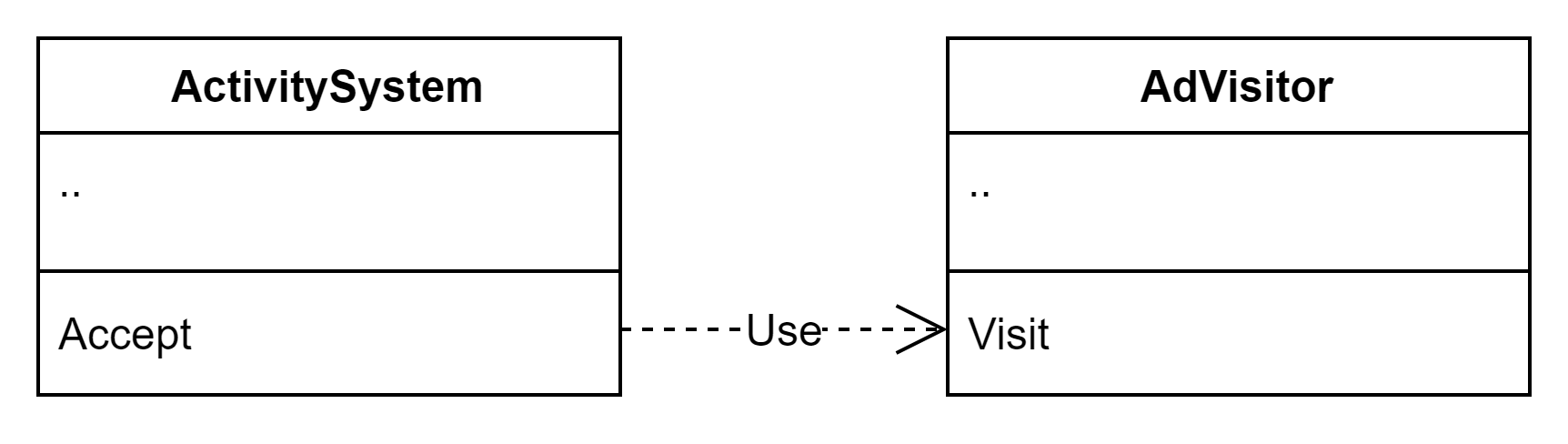


图 27访问者模式（AD）

### 详述

访问者的定义（源于GoF《Design Pattern》）是：表示一个作用于某对象结构中的各元素的操作。它使你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。  
我们在设计 **广告与活动子系统** 时，不确定后续需要进行哪些操作，又不想在后续再修改原来类的代码，所以使用了 **访问者** 设计模式。  
以 **广告子系统AdSystem** 为例，我们将说明 **访问者** 设计模式的实现：

1、在 **广告子系统AdSystem** 中写入 **Accept** 函数，用于获得 **访问者**：

class AdvertisingSystem   
{   
public:   
 void Accept(AdVisitor\* Visitor);   
 ……   
private:   
 ……   
};

**Accept** 函数的实现方法是：调用传入的 **访问者** 的 **Visit** 函数，将当前类实例传给 **访问者** 。

void AdvertisingSystem::Accept(AdVisitor \*Visitor) {   
 Visitor->Visit(this);   
}

2、创建 **访问者** 的父类： **Visit** 函数表示要进行的具体操作，这里是纯虚函数，具体操作要在子类实现。

class AdVisitor//广告visitor父类   
{   
public:   
 virtual ~AdVisitor() = default;   
 virtual void Visit(AdvertisingSystem\* AdSystem) = 0;   
};

3、为每一个操作创建一个 **访问者** 的子类，以 **添加广告** 为例：  
相较于 **访问者** 的父类，子类可以添加私有成员变量，如这里的 **广告编号AdID** 、**广告内容Content**；

class AdVisitor\_Add : public AdVisitor   
{   
public:   
 AdVisitor\_Add(int ID, string Cont);   
 void Visit(AdvertisingSystem\* AdSystem) override;   
private:   
 int AdID;   
 string Content;   
};

子类需要自己的构造函数：如这里 **添加广告** 访问者子类的构造函数需要将参数传入：

AdVisitor\_Add::AdVisitor\_Add(int ID, string Cont)   
{   
 AdID = ID;   
 Content = Cont;   
}

**访问者子类** 的代表的具体操作将在 **Visit** 函数里实现：例如这里 **添加广告** 访问者子类的 **Visit** 函数实现如下：

void AdVisitor\_Add::Visit(AdvertisingSystem\* AdSystem)   
{   
 if (AdID < 0) // 验证广告编号合法性   
 {   
 cout << "ID < 0 !!!" << endl;   
 return;   
 }   
 AdNode\* ad = new AdNode(AdID, Content); // 创建广告实例   
 AdSystem->AdvertisingList.push\_back(ad); // 将广告加到广告集合   
}

## 策略模式

**Strategy**

模式场景：一个类的行为或其算法可以在运行时更改

### 类图

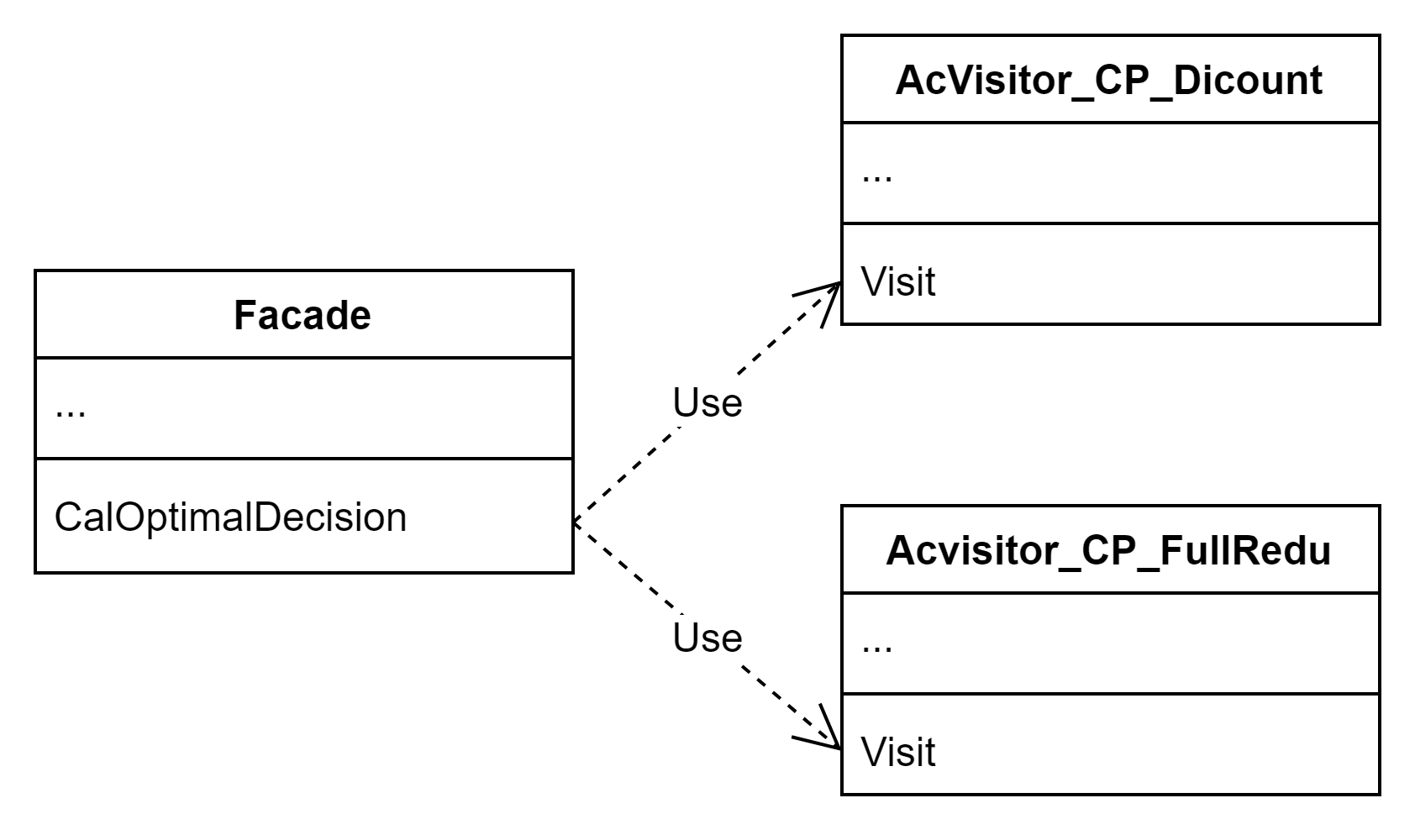


图 28策略模式（AD）

### 详述

策略模式用于实现一个行为有多种选择的情况，并且将每种选择需要做的操作封装起来，方便调用。  
我们这里在 **活动子系统** 的 **访问者AcVisitor** 中使用了策略模式，是为了更方便的区别使用 **打折类活动** 与 **满减类活动** 。  
具体来说，我们需要找出购买一系列商品时，通过各种活动优惠后，最优的价格，在这里，我们有两种策略：一是参与 **打折类活动** ，二是参与 **满减类活动** ，然后对比两者谁更优惠。  
在使用 **策略模式** 时，我们首先创建了指向所有策略的父类 **AcVisitor\_CalPrice** 的指针，这个指针既可以指向 **打折策略类AcVisitor\_CP\_Discount** ，又可以指向 **满减策略类AcVisitor\_CP\_FullRedu** 。

float Facade::CalOptimalDecision(map<CommodityInformation\*, int>& BuyCommodityInformationMap)   
{   
 AcVisitor\_CalPrice\* CalPriceVisitor;// 指向所有策略的父类   
   
 CalPriceVisitor = new AcVisitor\_CP\_Discount(BuyCommodityInformationMap);// 策略1，使用打折活动   
 AcSystemInstance->Accept(CalPriceVisitor);   
 ……   
 delete CalPriceVisitor;   
   
 CalPriceVisitor = new AcVisitor\_CP\_FullRedu(BuyCommodityInformationMap);// 策略2，使用满减活动   
 AcSystemInstance->Accept(CalPriceVisitor);   
 ……   
}

## 外观模式

**Facade**

模式场景：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

### 类图

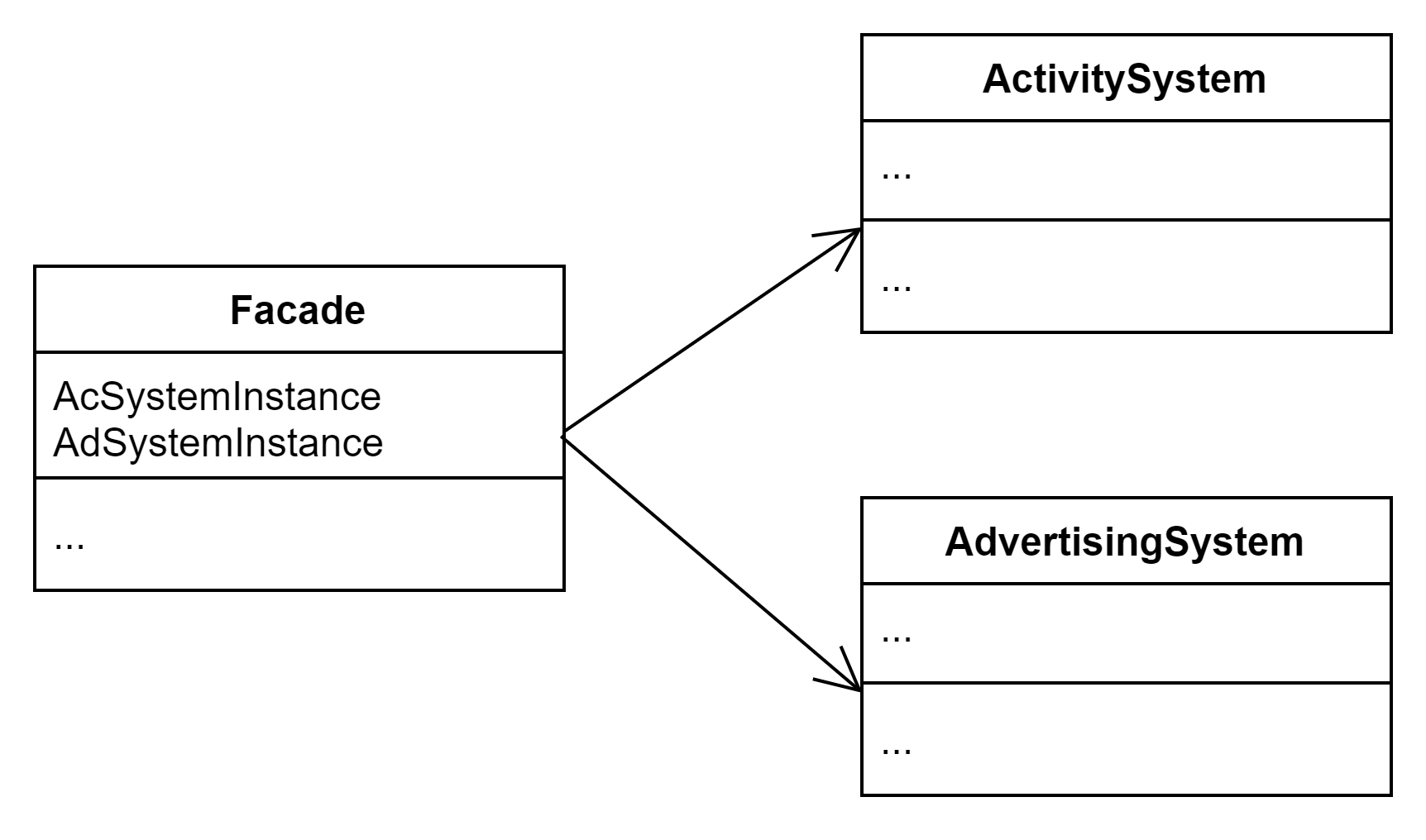


图 29外观模式（AD）

### 详述

外观模式的意义在于，为几个子系统的对外接口提供一个统一的界面，使子系统更容易使用。  
这里我们给予 **广告子系统AdvertisingSystem** 与 **活动子系统ActivitySystem** 一个外观：

class Facade//外观模式   
{   
public:   
   
 Facade();//构造广告子系统、活动子系统、解释器   
   
 // 广告操作：   
   
 void BrowseAd();//浏览全部广告   
 void BrowseAd(int BrowseAdID);//浏览单个广告   
 void AddAd(int ID, string Cont);//添加单个广告   
   
 // 活动操作：   
   
 void BrowseAc();//浏览全部活动   
 void BrowseAc(int BrowseAcID);//浏览单个活动   
 void AddAcD(int ID, string Cont, vector<int>& CommodityInformationReaderList, float DisRate);   
 //添加打折类活动   
 void AddAcF(int ID, string Cont, float Thd, float RedAmount);   
 //添加满减类活动   
 float CalOptimalDecision(map<CommodityInformation\*, int>& BuyCommodityInformationReaderMap);   
 //寻求最优惠价格，并打印此时活动列表   
 void RecommendActivity(vector<CommodityInformation\*>& RelatedCommodityInformationReaderList);   
 //打印与商品相关的所有活动   
 void RecommendActivity(map<CommodityInformation\*, int>& RelatedCommodityInformationReaderMap);   
 //打印与商品相关的所有活动   
   
private:   
 AdvertisingSystem\* AdSystemInstance;   
 ActivitySystem\* AcSystemInstance;   
 Interpreter\* ActivityInterpreter;   
};

在外观构造函数中，我们生成了两个子系统与解释器的实例，并给它们添加了一系列初始化广告、活动内容；

Facade::Facade()   
{   
 AdSystemInstance = AdvertisingSystem::GetAdSystemInstance();// 广告子系统   
 AcSystemInstance = ActivitySystem::GetAcSystemInstance();// 活动子系统   
 ActivityInterpreter = new Interpreter();// 解释器   
   
 AddAd(1, "双11开门红不熬夜！每日干果限量低价放送！");//添加广告   
 ……   
}

另外还定义了一系列其他子系统需要调用的接口，例如，**浏览所有广告** 操作，原本需要生成观察者 、Accept观察者、释放观察者三步操作，现在封装成一个函数 **BrowseAd** ，更加方便：

void Facade::BrowseAd()   
{   
 AdVisitor\_Browse\* AdBrowse = new AdVisitor\_Browse();   
 AdSystemInstance->Accept(AdBrowse);   
 delete AdBrowse;   
}

# 购物车子系统

**Shopping List Subsystem**

## 功能与实现

**Functions & Implementation**

**主要功能**

1. 添加/删除商品
2. 浏览商品
3. 生成订单

**实现方案**

构造Cart类包含对购物车进行的所有操作。

class Cart {  
public:  
 // 按数量增加某ID的商品  
 void add(int id, int amount);  
 // 移除购物车内某ID的所有商品  
 void remove(int id);  
 // 按数量移除购物车内某ID的商品  
 void remove(int id, int amount);  
 // 展示购物车内的所有商品信息  
 void display();  
 // 调用活动类的接口：计算最优惠价格  
 float calculateOptionalPrice(map<CommodityInformation \*, int> commodities);  
 // 结算购物车内所有商品并生成订单  
 void pay();  
 // 结算购物车内某ID的商品并生成订单  
 void pay(int id);  
 // 按数量结算购物车内某ID的商品并生成订单  
 void pay(int id, int amount);  
};

**子系统接口**

在CartCommand类及其子类中实现。

构造CartCommand基类代表对购物车的Command基本操作，以此根据购物车的不同功能分别构造不同的由基类派生的派生类（每个操作对应一个派生类）。同时构造一个ExecuteCommands的宏命令类为若干操作提供统一执行的接口。

class CartCommand {  
public:  
 // 购物车命令的执行  
 virtual void operation() = 0;  
};  
  
/// 对CartCommand类进行存储的宏命令类，调用execute()类进行执行  
class ExecuteCommands {  
 list<CartCommand\*> commands;  
public:  
 void addCommand(CartCommand \*command);  
 void removeCommand(CartCommand \*command);  
 // 宏命令的执行操作方法，将执行其中所有命令的operate()方法  
 void execute();  
};  
  
/// 增加商品的Command类  
class AddCommodity : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 移除某一ID的商品的Command类  
class RemoveCommodity : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 按数量移除某一ID的商品的Command类  
class RemoveSomeCommodity : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 展示购物车所有商品信息的Command类  
class DisplayCart : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 展示购物车内所有商品/购物车内某一ID的所有商品/某一数量的某一ID的商品的最优惠价格的Command类  
class CalculateOptionalPrice : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 结算购物车内所有商品的Command类  
class PayAll : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 结算购物车内某一ID的所有商品的Command类  
class PaySingleCommodity : public CartCommand {  
public:  
 void operation() override;  
}  
  
/// 按数量结算购物车内某一ID的商品的Command类  
class PaySomeCommodity : public CartCommand {  
public:  
 void operation() override;  
}

执行方法(example)

ExecuteCommands \*commands = new ExecuteCommands();   
commands->addCommand(new AddCommodity(params[]));   
commands->addCommand(new RemoveCommodity(params[]));   
commands->addCommand(new DisplayCart());   
commands->addCommand(new PayAll());   
commands->execute();

## 命令模式

**Command**

模式场景：请求以命令的形式包裹在对象中，并传给调用对象

### 类图

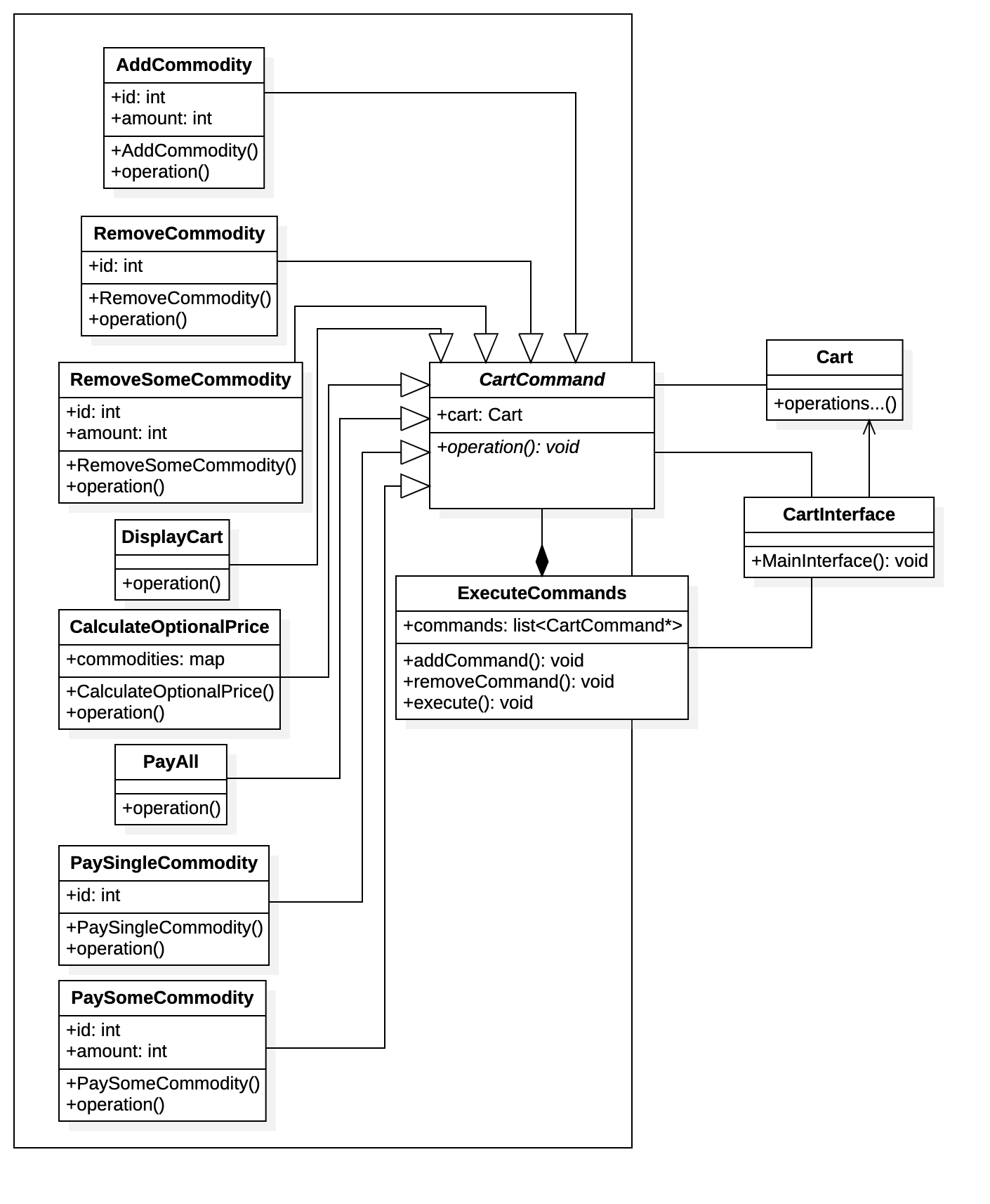


图 30命令模式（ShoppingList）

### 实现过程

Command模式应用于将对购物车的每种操作封装成一个独立的类的CartCommand类及其派生类中。

将Cart类主要供顾客调用的成员方法分别封装为独立的类：

表 2 Cart类成员函数

|  |  |
| --- | --- |
| 成员方法 | Command |
| add(int id, int amount) | AddCommodity |
| remove(int id) | RemoveCommodity |
| remove(int id, int amount) | RemoveSomeCommodity |
| display() | DisplayCart |
| calculateOptionalPrice(map<CommodityInformation\*, int>) | CalculateOptionalPrice |
| pay() | PayAll |
| pay(int id) | PaySingleCommodity |
| pay(int id, int amount) | PaySomeCommodity |

在每个Command派生类中定义一个成员Cart类型的成员属性cart，当调用Command派生类的operation()成员方法时，即对其私有属性cart提供的对购物车操作的方法进行操作。

### 实现API

AddCommodity::AddCommodity(int id, int amount);

**接受参数：**商品ID(int类型)，商品数量(int类型)

**执行功能：**对购物车进行增加商品操作

**返回值：**无

RemoveCommodity::RemoveCommodity(int id);

**接受参数：**商品ID(int类型)

**执行功能：**对购物车进行移除商品操作

**返回值：**无

RemoveCommodity::RemoveCommodity(int id, int amount);

**接受参数：**商品ID(int类型)，商品数量(int类型)

**执行功能：**对购物车进行按数量移除商品操作

**返回值：**无

DisplayCart::DisplayCart();

**接受参数：**无

**执行功能：**对购物车进行展示所有商品操作

**返回值：**无

CalculateOptionalPrice::CalculateOptionalPrice();

**接受参数：**无

**执行功能：**对购物车进行计算所有商品优惠价格并显示操作

**返回值：**无

CalculateOptionalPrice::CalculateOptionalPrice(int id);

**接受参数：**商品ID(int类型)

**执行功能：**对购物车进行计算某件商品优惠价格并显示操作

**返回值：**无

CalculateOptionalPrice::CalculateOptionalPrice(int id, int amount);

**接受参数：**商品ID(int类型)，商品数量(int类型)

**执行功能：**对购物车进行计算某一数量的某件商品优惠价格并显示操作

**返回值：**无

PayAll::PayAll();

**接受参数：**无

**执行功能：**对购物车进行结算所有商品操作

**返回值：**无

PaySingleCommodity::PaySingleCommodity(int id);

**接受参数：**商品ID(int类型)

**执行功能：**对购物车进行结算某一商品操作

**返回值：**无

PaySomeCommodity::PaySomeCommodity(int id, int amount);

**接受参数：**商品ID(int类型)，商品数量(int类型)

**执行功能：**对购物车进行按数量结算某一商品操作

**返回值：**无

void CartCommand::operation();

**接受参数：**无

**执行功能：**执行CartCommand派生类实例的相应的对Cart(购物车)进行的操作

**返回值：**无

void ExecuteCommands::execute();

**接受参数：**无

**执行功能：**执行宏命令ExecuteCommands内所有CartCommand实例对应的操作

**返回值：**无

### 关键代码

以AddCommodity类为例：

AddCommodity类将Cart的add()方法封装为一个Command类。AddCommodity类构造实例时即将add()方法的参数存为私有成员属性，在调用operation()方法时对成员属性Cart类实例cart进行操作。

class Cart {  
 ...  
public:  
 void add(int id, int amount);  
}  
  
class CartCommand {  
public:  
 // 需要AddCommodity类实现的纯虚方法  
 virtual void operation() = 0;  
}  
  
class AddCommodity : public CartCommand {  
private:  
 int id, amount;  
 // 存储的Cart类实例  
 Cart cart;  
public:  
 AddCommotity(int id, int amount) : id(id), amount(amount) {}  
 void operation() override {  
 // 间接对Cart实例进行操作  
 cart.add(id, amount);  
 }  
}

## 观察者模式

**Observer**

模式场景：当一个对象被修改时，自动通知依赖它的其他对象

### 类图

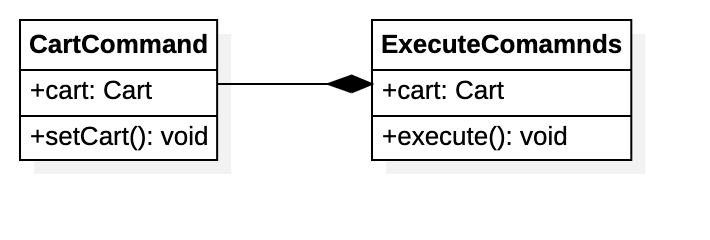


图 31观察者模式（ShoppingList）

### 实现过程

Observer模式与Command模式搭配使用，在购物车子系统的Command模式中，每个对购物车进行操作的CartCommand类都拥有私有成员属性：Cart类实例cart，在调用operation()成员方法对Cart实例进行操作时需要预先对其进行设置。

在Command的宏命令类ExecuteCommands的成员方法execute()中，在依次调用其私有成员属性：CartCommand类的list中存储的Command的operation()方法前，先对其进行Cart的设置。

同时对于不同顾客的不同的Cart实例，在其改变时通过预先设置ExecuteCommands实例的cart，再通过execute()方法对其内部的CartCommand列表的广播机制，达到灵活改变和更新cart的功能。

### 实现API

void ExecuteCommands::setCart(Cart cart);

**接受参数：**Cart类实例cart

**执行功能：**设置宏命令的Cart(购物车)实例

**返回值：**无

void ExecuteCommands::execute()

**接受参数：**无

**执行功能：**广播宏命令存储的Cart实例给其内部的CartCommand

**返回值：**无

### 代码样例

class CartCommand {   
private:   
 Cart cart;   
public:   
 // 被ExecuteCommands实例调用的响应函数：通过被广播修改cart   
 void setCart(Cart cart) { this->cart = cart; }   
 void operation();   
}   
   
class ExecuteCommands {   
private:   
 std::list<CartCommand\*> commands;   
 Cart cart;   
public:   
 void setCart(Cart cart) { this->cart = cart; }   
 void execute() {   
 for (auto command : commands) {   
 // 通过广播修改其内部list的所有CartCommand实例的cart   
 command->setCart(cart);   
 command->operation();   
 }   
 }   
}

# 订单子系统

**Order Subsystem**

## 状态模式

**State**

模式场景：类的行为是基于它的状态改变的

### 类图

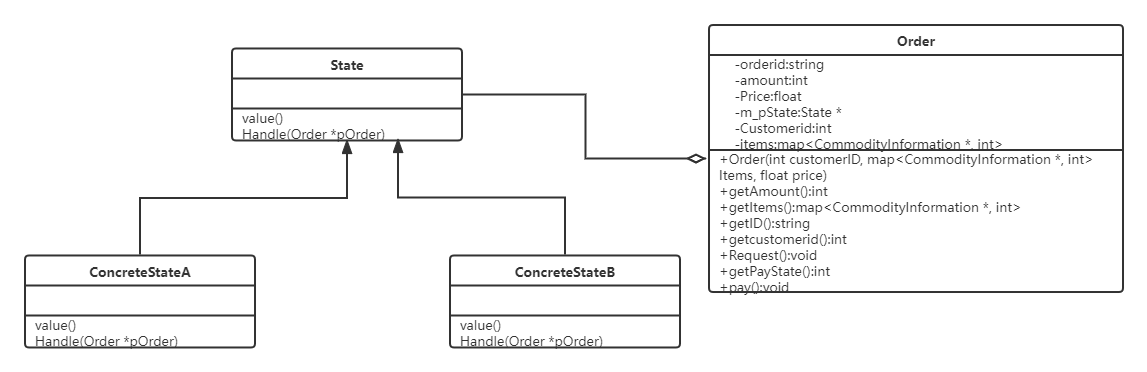


图 32状态模式（Order）

### 详述

状态（State）模式是指对有状态的对象，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。状态模式把受环境改变的对象行为包装在不同的状态对象里，其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。

环境类（Order）角色：也称为上下文，它定义了客户端需要的接口，内部维护一个当前状态，并负责具体状态的切换。作为state的载体。

抽象状态（State）角色：定义一个接口，用以封装环境对象中的特定状态所对应的行为。记录订单的状态

具体状态（ConcreteStateA/B）角色：实现抽象状态所对应的行为，并且在需要的情况下进行状态切换。分别记录已支付和未支付两个状态。

### 实现API

State::value() :int，状态标志位。

State::Handle(Order \*pOrder):void，打印状态函数。

Order::Request():void，打印订单状态。

Order::getPayState():int，获得状态标志位。

Order::pay():void，支付（改变订单状态） 。

# 小组成员贡献

**Contribution of Members**

表 3小组成员贡献表

|  |  |  |
| --- | --- | --- |
| 成员姓名 | 工作内容 | 贡献比例 |
| 李睿捷 | 组织协调组员、店铺子系统、耦合所有子系统 | 20% |
| 陆天宇 | 顾客子系统、系统整体debug | 17% |
| 张峰 | 订单子系统、协助完成店铺子系统 | 10% |
| 莫海洋 | 购物车子系统、协助完成店铺子系统 | 10% |
| 张福泰 | 广告与活动子系统 | 9% |
| 余林旭 | 广告与活动子系统 | 9% |
| 闫书玮 | 广告与活动子系统 | 9% |
| 李林洲 | 商品子系统，文档整合与排版 | 13% |
| 席宇琛 | 完善系统整体、PPT制作 | 3% |

# 图表索引

**Chart Index**

[表 1设计模式列表 7](#_Toc86868553)

[表 2 Cart类成员函数 58](#_Toc86868554)

[表 3小组成员贡献表 65](#_Toc86868555)

[图 1模块模式（UI） 9](#_Toc86942252)

[图 2建造者模式（Customer） 13](#_Toc86942253)

[图 3观察者模式（Customer） 15](#_Toc86942254)

[图 4中介者模式（Customer） 18](#_Toc86942255)

[图 5备忘录模式（Customer） 19](#_Toc86942256)

[图 6模板方法（Customer） 21](#_Toc86942257)

[图 7单例模式（Customer） 22](#_Toc86942258)

[图 8抽象工厂模式（Shop） 24](#_Toc86942259)

[图 9工厂模式（Shop） 25](#_Toc86942260)

[图 10空对象模式（Shop） 27](#_Toc86942261)

[图 11观察者模式（Shop） 28](#_Toc86942262)

[图 12过滤器模式（Shop） 30](#_Toc86942263)

[图 13代理模式（Shop） 32](#_Toc86942264)

[图 14适配器模式（Shop） 34](#_Toc86942265)

[图 15装饰器模式（Shop） 35](#_Toc86942266)

[图 16责任链模式（Shop） 37](#_Toc86942267)

[图 17原型模式（Shop） 39](#_Toc86942268)

[图 18依赖注入模式（Shop） 41](#_Toc86942269)

[图 19商品子系统（Commodity） 44](#_Toc86942270)

[图 20组合模式（Commodity） 46](#_Toc86942271)

[图 21迭代器模式（Commodity） 47](#_Toc86942272)

[图 22享元模式（Commodity） 48](#_Toc86942273)

[图 23命令模式（Commodity） 50](#_Toc86942274)

[图 24商品售卖流程图 51](#_Toc86942275)

[图 25解释器模式（AD） 58](#_Toc86942276)

[图 26迭代器模式（AD） 61](#_Toc86942277)

[图 27访问者模式（AD） 63](#_Toc86942278)

[图 28策略模式（AD） 65](#_Toc86942279)

[图 29外观模式（AD） 66](#_Toc86942280)

[图 30命令模式（ShoppingList） 71](#_Toc86942281)

[图 31观察者模式（ShoppingList） 75](#_Toc86942282)

[图 32状态模式（Order） 77](#_Toc86942283)

# 参考文献

**References**

1. Crockford, D. *JavaScript: The Good Parts.* Sebastopol CA: O'Reilly Media, 2008. 40~42.
2. Martin, R. C. *Agile Software Development, Principles, Patterns, and Practices.* Edinburgh Gate Essex: Pearson, 2014. 189~192.
3. Kayal, D. *Pro Java™ EE Spring Patterns: Best Practices and Design Strategies Implementing Java™ EE Patterns with the Spring Framework.* US: Apress, 2008. 98~106.
4. Martin, R. C. *Agile Software Development, Principles, Patterns, and Practices.* Edinburgh Gate Essex: Pearson, 2014. 127~139.