|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 객체지향 시스템 설계 | | | | | |
| 학번 | 2018023390 | **이름** | 이서연 | **제출일** | 2021.06.20 |

1. **프로그램 실행 환경**
   1. OS : macOS BigSur (11.2.3)
   2. IDE : Eclipse (ver 4.18.0)
   3. java
      1. java version "15.0.1" 2020-10-20
      2. Java(TM) SE Runtime Environment (build 15.0.1+9-18)
      3. Java HotSpot(TM) 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
2. **Design pattern 설명**
   1. Singleton pattern
      1. mart object는 반드시 하나만 생성되어야 하기 때문에, 생성자를 private 로 할 수 있도록 했다.
      2. singleton 패턴 자체는Double checked looking 을 통해 구현하고자 했다. 실제로 Mart의 instance로 활용될 uniqueInstance 자체는 volatile static 으로 만들어지도록 했고, getInstance() 함수를 통해 null 체크 후, synchronization 후 한번 더 null check 를 해서 mart 가 만들어지도록 했다. 그렇게 만들어진 uniqueInstance를 리턴하도록 했다. mart 자체는 외부에서 만들어질 때도 getInstance()를 통해서만 만들어지도록 했다.

public static Mart getInstance() {

if(uniqueInstance == null) {

synchronized(Mart.class) {

if(uniqueInstance == null)

uniqueInstance = new Mart();

}

}

return uniqueInstance;

}

* 1. Observer pattern



* + 1. InventoryManager 는 abstract class로, observers, quantities, products를 Instance variable로 가지고, addObserver, deleteObserver, notifyObservers를 메소드로 갖는다.
       1. Instance variable
          1. **observers**는 현재 품절 혹은 재고 부족으로 대기 상태로 들어온 customer 들이 들어가게 된다.
          2. **quantities** 에는 해당 customer가 구매하려고 했던 물품의 수가 들어간다.
          3. **products**에는 해당 customer 가 구매하려고 했던 물품이 들어간다.
       2. Method
          1. **addObservers**(Observer observer, int quantity, Product product) :

observer(customer) , quantity (사려고 했던 양), product(사려고 했던 물품) 을 입력받아서 array List에 추가한다.

이 메소드는 기본적으로 MartSimulator에서 customer mode – shopping 실행시 호출되는 Mart 클래스의 buying method 에서 실행된다. 만약에 사려고 하는 물품의 quantity 가 고객이 원하는 quantity 보다 작으면 waitings 메소드(Mart class) 를 invoke 하면서 실행된다.

즉 고객이 사고자 하는 만큼의 양의 물건이 없을 때, observers 리스트에 추가하면서 일종의 대기상태로 만드는 것이다.

* + - * 1. **deleteObserver**(Observer observer, Product product)

observer와 product를 받아 일치하는 고객의 정보를 observer 에서 제거한다. observer와 product를 동시에 받도록 설계한 이유는, 어떤 구매자가, 재고가 없는 상품을 2회이상 선택해서 2회이상 대기상태로 들어갔을 때 observer정보만 가지고 정확한 index를 구할 수 없었기 때문이다. product의 정보와 비교해서 이후 상품이 늘어났을 때, 늘어난 상품이 존재하면 그 상품과 일치하는 인덱스에 해당하는 quantities, products를 제거하기 위해 product 정보도 받게 되었다.

observers array list에 대해 for loop을 돌면서 전달받은 observer와 동일한지 확인을 한 후, 만약에 동일하다면 그 인덱스에서 product가 전달받은 product와 동일한지 체크해 인덱스를 결정한다.

인덱스(index)의 초기값은 -1이며, index >= 0 이상이라면, quantities, products, observers에서 해당 인덱스의 값을 제거한다.

이 메소드는, Mart의 buying 메소드가 실행될 때 최종적으로 구매가 가능해진다면 실행된다. 그 customer 가 observers에 있는지와 상관없이 반드시 실행되고, 실행 되고나서 만약에 대기리스트에 있었다면 index 값이 0 이상일테니 대기리스트에 있을 때만 observers 및 다른 Instance variable이 반영되도록 했다.

* + - * 1. **notifyObservers**()

모든 observers의 구성요소들에 대해서 update 메소드를 실행시킨다. update 는 아래(ii)에서 설명될 것이다.

이 메소드는, Mart의 setNumber 메소드가 실행될 때 quantity가 0 이상이라면 실행된다. setNumber 메소드는 salesList의 product의 quantity 가 변화할 때 실행되는데, addInventory가 실행될 때, 또는 물건을 구매할 때 실행된다. 물건을 구매할 때는 quantitiy가 0이하로 들어오기 때문에 0이상일 때만 notifyObservers 가 실행될 수 있도록 했다.

product 를 넣어줄 때, Mart의 salesList 상의 인덱스로 들어가도록 했다.

* + 1. Observer Interface 및 이를 implement 하는 Customer 의 class 경우 generator (Inventory Manager) , quantity, product를 입력으로 받는 update 메소드를 가지며, 디자인 패턴의 구현에 있어서는 update 함수만 관련이 있는 것 같아 자세한 다른 설명은 아래 3.메소드 설명 부분에서 하고, 여기서는 update method만 설명하겠다.
       1. **update**(InventoryManager manager, int quantity, int product)
          1. InventoryManager (Mart) , quantity (사려고 했던 양), product(사려고 한 물건의 인덱스) 를 받으며, Mart 의 buying 메소드를 실행한다. notifyObservers 에서 product를 받을 때 인덱스값으로 받기 때문에 product는 1 더해서 넣어준다.
          2. buying 이 ExpiredException을 일으키도록 되어있어서 exception handling 을 해줘야한다.
          3. 이 메소드는, notifyObservers가 실행될 때, 즉 상품이 추가되었을 때 실행되는데, 고객이 대기상태에 들어갈 때 당시 quantity 만큼, 당시 선택했던 product 에 대해 buying 을 진행한다. buying 메소드에서 다른 조건 (유통기한, Quantity 등) 을 통과하면 성공적으로 구매하고, deleteObservers 가 실행되면서 observers 에서 해당customer 가 제거되게 된다.

1. **메소드 설명**
   1. 목록

[i. interface Payable 3](#_Toc75120132)

[1. class Cash 3](#_Toc75120133)

[2. class Credit 4](#_Toc75120134)

[ii. class Product 4](#_Toc75120135)

[1. class Food 4](#_Toc75120136)

[2. class Manufactured 5](#_Toc75120137)

[iii. abstract class InventoryManager 5](#_Toc75120138)

[1. class Mart 5](#_Toc75120139)

[iv. interface Observer 5](#_Toc75120140)

[1. class Customer 6](#_Toc75120141)

[v. class MartSimulator (main) 6](#_Toc75120142)

* 1. 설명
     1. **interface Payable**

public void pay(int amount) throws NotEnoughBalanceException 을 헤딩으로 가지는 Interface 이다. NotEnoughBalance Exception을 throw 하는 이유는, 메소드에서 throw를 하고 나중에 이걸 invoke 할 때 예외처리를 하고자 이렇게 구현했다.

* + - 1. class **Cash**
         1. Instance variable : private String **currency** (통화량) , private int **amount** (잔액)
         2. Method

public Cash (String currency, int amount) : 생성자, currency, amount 를 입력받아 초기화 한다.

public String toString () : “[currency] , [amount] won” 형태의 string 을 반환함.

**public void pay(int amount) throws NotEnoughBalanceException**

입력받은 amount 만큼을 amount 에서 빼는 메소드로, 프로그램상에서는 paying 단계에서 Cash로 결재하면 실행된다. Invoke 자체는 Mart Simulator 에서 paying 메소드 invoke 🡪 Customer 클래스 에서 paying 메소드 Invoke 🡪 (만약 결재수단이 cash 일 경우) 이 메소드가 실행된다.

만약에 입력받은 금액이 이 object 가 가진 총액보다 크면 NotEnoughBalanceException이 발생한다. 예외처리 자체(catch)는 Customer class의 paying 메소드에서 진행된다.

* + - 1. class **Credit**
         1. Instance variable : private String bank , private int limit, private int amountUsed
         2. Method

public Credit(String bank, int limit, int amountUsed) : 생성자, bank, limit, amountUsed 를 입력받아 초기화 한다.

public String toString() : “[bank] , Amount Used : [amount used] won (Limit : [limit])” 형태의 string 을 반환함.

public String getBank() : bank를 반환하는 getter

public boolean payCheck(int amount) : 입력받은 금액이 limit – amountUsed 보다 적은지 큰지 Boolean 값으로 반환한다.

**public void pay(int price) throws NotEnoughBalanceException**

Cash에서와 마찬가지로, 입력받은 price만큼 사용한 것으로 차감해야하는데, price와는 반대로 amountUsed 에 더해야한다.

최대 사용한도는 limig – amountUsed 이므로, 처음에 price 가 이보다 큰지 작은지 확인한다. 만약에 price 가 더 크면 NotenoughBalanceException을 발생시키고, Cash에서와 마찬가지로 customer class의 paying 메소드에서 예외처리가 진행된다.

* + 1. class Product

- Instance variable : private String name , private int price, private int quantity

- Method

1. public Product(String name, int price, int quantity) / public Product(Product obj, int quantity) : 생성자, 첫번째 것은 name, price, quantity 를 입력받아 생성하고, 두번째 것은 product와 quantity 를 입력받아 생성한다. product에서 name, price 를 초기화 하고, quantitys는 입력받은 것으로 초기화 한다. 첫번째 것은 맨 처음에 main 실행될 때 리스트를 만들 때 필요하고, 두번째 것은 customer의 shoppingCart 에 넣어줄 때 필요하다.
2. getName() / getPrice() / getQuantity() : name, price, quantity를 리턴하는 getter
3. public String toString() : “(Quantity : [quantity]), [name], [price]won” 형태의 string 을 반환한다.
4. public boolean equals(Product product) : 전달받은 product의 이름과 price 가 동일하면 true를 리턴하고, 아니면 false를 리턴한다.
5. public void setQuantity(int quantity) : quantity를 변경하는 setter
   * + 1. class **Food**
          1. Instance variable : private LocalDateTime expirationDateTime (유통기한)
          2. Method

public Food(String name, int price, int quantity, LocalDateTime time) : 생성자, super()를 이용해 name, quantity, price는 초기화 하고, time을 expirationDateTime으로 초기화 한다.

public boolean **isExpired(LocalDateTime present)** : 입력받은 present보다 expirationtime이 이전이면 true를 반환하고 아니면 false 를 반환한다. (즉 유통기한이 지났으면 true이다)

public String toString() : super.toString() 에 “, Best before : [expirationDateTime] “ 형태의 string 을 더한 형태의 string을 반환한다.

public void setExpiration(int elongated) : Expiration을 변경하는 setter. LocalDateTime 의 메소드인 plusDays() 를 이용해 elongated 만큼 유통기한을 늘려준다.

* + - 1. class **Manufactured**
         1. Instance variable : public String brand
         2. Method

public Manufactured(String name, int price, int quantity, String brand) : 생성자, super()를 이용해 name, quantity, price는 초기화 하고 this.brand를 전달받은 brand 로 초기화 한다.

public String toString() : super.toString() 에 “, Brand : [brand]” 를 더한 형태의 string 을 출력할 수 있도록 한다.

* + 1. abstract class InventoryManager

2 – B Observer패턴 구현 설명에서 해당 클래스에 대한 설명을 기술 했으므로, 여기서는 생략하도록 하겠다. (해당 부분에서 설명된 메소드 외에 사용된 메소드는 없다.)

* + - 1. class **Mart**
         1. instance variable : public static int transactionNum= 1000; private ArrayList<Product> salesList
         2. Method

private Mart() : 생성자, singleton 패턴을 사용해야해서 private로 선언하고, arrayList를 만들어 초기화 함.

public static Mart getInstance() : Mart instance 를 만드는 실질적인 생성자의 역할을 하는 method 인데, 2-A singleton 패턴 구현 설명에서 자세히 설명했으니 여기서는 생략하겠다.

public void setSaleList(Product product) : salesList에 새로운 Product를 추가하는 함수.

public void printSaleList() : for-each loop을 돌면서 “[순서]. product.toString()” 형태의 string을 출력한다.

public void printExpired(LocalDateTime present) : salesList에 있는 상품들 중에 food인 상품들에 대해서, present에 대해서 Food클래스의 isExpired리턴값이 true이면 (유통기한이 지났으면) printSalesList와 같은 형태로 출력한다.

public void setExpiration(int elongated, LocalDateTime present) : expiration을 늘릴 때 사용되는 메소드, salesList를 돌면서 유통기한이 지났는지 체크하고 유통기한이 지난 상품들은 elongated만큼 늘려준다. 늘리는건 Food Class의 setExpiration으로 한다.

public void waitings(Customer customer, int quantity, Product product) : addObserver 를 실행하는 함수, Observer 패턴 구현에 사용됨. 해당 customer 를 대기상태에 넣겠다는 경고메시지 를 출력함.( "We are currently out of stock, so we'll put you on a waiting list.")

public int getIndex(Product product) : 특정 product의 salesList에서 index를 반환함.

public void **buying(**int product, int quantity, LocalDateTime present, Customer customer, int recheck) throws ExpiredException

product (salesList상에서의 Index + 1), quantity, present, customer 를 입력받는다.

만약 전달받은 product가 Food의 인스턴스 이면서, 유통기한이 지났을 경우, ExpiredException을 throw 한다. 예외 처리 (catch)는 MartSimulator에서 이루어진다.

만약 전달받은 quantity가 salesList에 존재하는 quantity보다 많을 경우 waitings 메소드를 실행해 customer를 일종의 대기상태로 만든다. 자세한 설명은 Observer 패턴에 설명된다. 혹시 재차 실행이 되었을 때 한번 더 customer가 들어가는 것을 방지하기 위해 recehck가 1이 아닐 때만 waitings가 실행되도록 했다. customer 의 update function을 통해 실행될 때는 recheck=1로 들어온다.

앞선 조건 2개를 통과하면 salesList의 해당 product에 대해 입력받은 quantity만큼을 빼준다. (Product클래스의 setQuantity method 활용) 그리고 Customer class의 addShoppingCart 메소드를 실행한다. 대기중에 있던 customer 를 observers에서 제거하기 위해 deleteObserver() 메소드도 실행한다. 자세한 내용은 2-B Observer 패턴 구현 설명에 기술하였으므로 생략한다.

public void **setNumber**(int product, int quantity)

salesList에서 product의 quantity를 변경하는 메소드, 주로 addInventory 에서 쓰인다. quantity가 0 이상이면 notifyObservers를 호출한다.

* + 1. interface Observer

**public abstract void update(InventoryManager generator, int quantity, int product)** 의 method heading 하나만 가지며, inventoryManager generator 외에 quantity와 product를 추가적으로 전달받는다. Mart class에서 update 구현을 할 때 buying Method를 invoke 하기 위해 그렇게 했다.

* + - 1. class **Customer**
         1. Instance variable : private String name, private ArrayList<Payable> wallet, private ArrayList<Product> shoppingCart // 각각 이름, payable 목록, 구매할 product 목록이다.
         2. Method

public Customer(String name) : 생성자, 이름만을 입력받고, payable과 shopping list는 빈 arrayLIst를 만들어 초기화 한다.

public boolean equals(Customer customer) : 이름과 wallet이 동일하면 true를 리턴한다.

public void addWallet(Payable wallet)

Payable을 전달받아 wallet에 추가하는 함수. customerLIst MartSimulator 초반 부분에서 초기화 할 때 실행되는데, 생성자와 분리해서 구현했다.

public String getName() / public ArrayList<Payable> getWallet() : 이름과 wallet을 반환하는 getter, ArrayList의 경우 동일한 arrayList를 복사해서 리턴하도록 했다.

public void addShoppingCart(Product product) : shoppingCart 에 product를 추가한다.

public void printShoppingCart()

shoppingCart 의 정보를 출력하는 메소드로 깔끔하게 보이게 하기 위해 문자열 포맷팅을 했다. 아래와 같이 출력된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

public void **paying(**PrintWriter receipt, int pay)

우선 try block 안에서wallet.get(pay-1)에서 pay 메소드를 실행한다. 만약에 잔액이 부족한 것 등의 문제가 생기면, NotEnoughBalanceException exception이 발생하고, 아래 catch block 으로 넘어간다. while loop를 돌도록 했는데, 만약에 NotEnougBalance Exception이 발생하면, 다시 pay 입력을 받도록 한다.

만약에 에러없이 끝났다면, 결제가 정상적으로 된 것일 것이다. receipt에 printShoppingCart()에서 출력했던 것 과 완전히 동일하게 출력을 하고, 아래에 결제정보의 class에 따라 결제 정보를 출력한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

public void update(InventoryManager manager, int quantity, int product) : 2-B에서 설명했으므로 생략한다.

만약에 입력받은 pay가 적절치 못해서 NotEnoughBalanceException이 발생했다면 이 때 reselect 변수로 다시 pay의 입력을 받고 paying 메소드를 재귀적으로 호출하는데, 이 때 0을 입력받으면 결재를 취소하는 것으로 하고 shopping cart 에 있었던 것들이 다시 salesList의 quantity를 올려준다. 이 customer의 shopping Cart 초기화 한다. 0, 그리고 wallet 의 크기에서 벗어나는 다른 값을 입력했을 경우 exception 처리하고, 위에서와 마찬가지로 처리한다. 제대로 입력하면 paying을 재귀적으로 다시 Invoke 한다.

* + 1. class **MartSimulator** (main)
       1. public static LocalDateTime present : LocalDateTime.of(2021, 5, 27, 15, 00)로 초기화 한다.
       2. main
          1. “Mart.txt”로 mart의 sales 초기화

getInstance 함수로 Mart 인스턴스를 생성한 이후, Mart.txt 에서 맨 첫줄에 주어지는 product의 갯수 (n\_product)만큼 forloop을 돌며 반복해서 라인단위로 string 을 읽은 후 “,” 단위로 Split 해 맨 첫번째 단어에 따라 Food, Manufactured 객체를 생성해 Mart 의 salesList로 넣는다. (Mart class의 setSalesList 활용)

input 을 처음 읽을 때 에러가 발생하면 (FileNotFoundException) 프로그램을 종료한다.

* + - * 1. “CustomerWallets.txt”로 customerList 초기화

만약에 “CustomerWallets.txt”가 존재하지 않으면 (FileNotFoundException) 프로그램을 종료한다.

맨 첫줄에 주어지는 n\_customer만큼 반복해서 라인단위로 string 을 읽은 후 “,” 단위로 splig 해 맨 첫번째 단어 즉 이름으로 Customer 객체를 생성한다. 이름 뒤에 붙어서 나오는 payable 의 수만큼 반복해 라인을 읽고, “,” 단위로 끊어 Credit 또는 Cash 객체를 생성해 customer의 wallet 에 추가한다. (Customer class의 addWallet 메소드) Credit 또는 Cash는 첫번째 단어 정보를 보고 만들 수 있다.

* + - * 1. 프로그램 시작 지점.

Manager mode, Customer mode, End of program 3가지 선택지 중 하나 고를 수 있으며, 1~3 사이의 integer 값을 입력으로 받지 않으면 Exception을 발생시킨다. integer 가 아닌 값이 들어왔을 때는 run time exception인 InputMismatchException이 발생되어 이를 처리하는 catch block 이 있고, 만약에 1-3 사이의 값이 들어오지 않으면 User defined Exception인 InvalidAccessException 이 발생하게 된다. 처음부터 끝까지 while(true) 안에 있어 계속 반복된다.

1을 선택했을 경우: ManagerMode 메소드를 실행한다.

2를 선택했을 경우

customerList의 목력을 출력한 뒤, customer 를 입력받는다. 이 때 0을 입력하면 continue가 실행되면서 다시 while loop 시작지점 (manager mode, customer mode, end of program 중 고르는 부분) 으로 돌아간다. 역시 예외처리가 들어가며, 0-3 사이 정수값이 들어가지 않으면 C – i 에서와 마찬가지로 예외처리가 진행된다. 정상적으로 선택이 된다면 customerMode 메소드를 실행한다.

3을 선택했을 경우

프로그램을 종료한다. (System.exit(0))

* + - * 1. **public static void ManagerMode(Scanner keyboard, Mart mart)**

C에서와 마찬가지로 while(true) 안에서 실행되며, 0을 입력시 메소드 자체를return 해버려 while loop에서 탈출한다. 0-3 사이 integer 값이 들어오지 않으면 C에서와 마찬가지로 예외처리를 한다.

1이 입력되었을 경우 addInventory 메소드를 실행하고, 2가 입력되었을 경우 replaceExpired가 실행된다.

* + - * 1. public static void **addInventory**(Scanner keyboard, Mart mart)

처음 실행 되자마자 saleList를 프린트 한다.

그 이후로는 while(true) loop 안에서 구현되고, product, quantity를 입력받는다. 만약에 정상적으로 product와 quantity를 입력했으면 (혹은 product에 0을 입력했으면) return 되어 빠져나온다. 어떤 종류든 Exception이 발생하면 while loop에서 탈출하지 못하기 때문에 다시 입력을 받는다.

정상적으로 입력을 했을 경우, Mart 클래스의 setNumber(product,quantity)가 실행된다. 특정 salesList의 저 product 에 대해 quantity가 전달된 값만큼(quantity) 만큼 증가한다.

* + - * 1. public static void **replaceExpired(**Scanner keyboard, Mart mart)

늘리고싶은 유통기한을 integer 로 입력받아(expandedDate) 그만큼 늘린다. integer 값으로 입력하지 않을 때를 대비해 try block 안에서 실행된다. 제대로 된 정수를 입력하지 않으면 while loop을 탈출하지 못하므로 계속 실행된다.

* + - * 1. public static void **customerMode**(Scanner keyboard, Customer customer, Mart mart)

4가지 선택지 중에 하나 입력을 받고, 이 역시 앞에서처럼 while loop 안에서 실행된다. 0-4 사이의 integer 값이 들어오지 않으면 exception이 발생하고 catch block 에서 예외처리가 이루어진다. 제대로 실행된다면 try block 안에 return 이 되기 때문에 while loop을 탈출한다.

1을 선택했을 경우 shopping 메소드, 2를 선택했을 경우 printShopppingCart 메소드, 3을 선택했을 경우 paying 메소드, 4를 선택했을 경우 printWallet 메소드가 실행된다.

* + - * 1. public static void **shopping**(Scanner keyboard, Mart mart, Customer customer)

while(true)안에서 실행되고, 초반에 Mart 클래스의 printSaleList() 메소드를 통해 salesList의 상황을 출력한다. 그 후 어떤 product를 고를지 입력을 받는데, 0을 입력받으면 리턴되면서 메소드가 종료되어 전단계로 돌아간다. 입력받은 product가 0이하, salesList의 size 이상일 경우 exception을 일으킨다. quantity는 0 이하로 들어오면 exception을 일으킨다.

만약에 성공적으로 입력되었으면, Mart 클래스의 buying 함수를 통해 구매를 진행한다.

* + - * 1. public static void printShoppingCart(Customer customer)

Customer 클래스의 printShoppingCart() 메소드를 실행한다.

* + - * 1. public static void **paying**(Customer customer, Scanner keyboard)

output 이 될 receipt를 만든다. (PrintWriter) 시작할 때 Customer 클래스의 shopping cart와 printWallet 메소드를 실행한다. 이후 어떤 것으로 지불할지 선택한다. 만약에 pay가 제대로 들어왔으면, receipt 가 제대로 만들어졌으면 Customer 클래스의 pay 메소드가 실행된다.

paying 메소드가 실행될 때마다 Mart 클래스의 transactionNum이 1씩 올라간다.

고로 해당 메소드를 3번 (성공적으로) 실행하면 receipt 가 아래와 같이3번 만들어진다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * 1. public static void **printWallet**(Customer customer)

customer 의 wallet 을 받아(getWallet) 출력한다.