프로젝트 보고서

팀명: 패턴 디자이너
팀원: 박범찬 (201904010), 김예은 (202004008)

1. 프로젝트 개요

이 프로젝트는 카페 시스템의 주문 및 결제, 메뉴 관리, 제조 완료 알림 등을 객체지향 프로그램으로 구현한 것이다. 각 기능마다 적절한 디자인 패턴을 적용하여 유연하고 효율적인 관리가 가능하다.

2. 객체 지향 설계

카페 시스템 프로젝트에서는 퍼사드 패턴, 팩토리 패턴, 템플릿 메서드 패턴, 옵저버 패턴을 사용하고 있다.

본 프로젝트는 카페 시스템을 소규모 단위로 작은 규모로 설계되었기 때문에 디자인 패턴 적용을 했을 때의 객체지향적으로 보이기 어려운 부분도 분명히 있을 것이다. 하지만 이는 데모 프로젝트이며, 규모가 커지면 커질 수록 해당 디자인 패턴 적용 방식을 그대로 참조하게 되면 객체지향적인 설계가 잘 이루어질 수 있을 것이다.

1. 퍼사드 패턴(Facade Pattern):

PaymentSystem에서 퍼사드 패턴이 사용되었다. 이 코드에서는 CardPayment 클래스가 복잡한 시스템을 대표하고, PaymentFacade 클래스가 퍼사드 역할을 한다.

CardPayment 클래스는 카드 결제 과정을 세 부분으로 나눠 insertCard(), waitAndPrintPaymentStatus(), ejectCard() 메소드를 제공한다. 이는 결제 시스템의 내부 동작을 나타내며, 일반적으로 클라이언트 코드는 이 모든 메소드를 올바른 순서로 호출해야 한다.

반면 PaymentFacade 클래스는 이 모든 과정을 한 번에 수행하는 processPayment() 메소드를 제공한다. 이 메소드 내부에서는 CardPayment의 모든 메소드를 올바른 순서로 호출한다. 따라서 클라이언트 코드는 processPayment()만 호출하면 된다.

이렇게 퍼사드 패턴을 사용하면 클라이언트 코드는 복잡한 시스템의 내부 동작 방식을 알 필요 없이 간단한 인터페이스를 통해 시스템을 사용할 수 있다. 또한 시스템의 내부 구현이 변경 되더라도 퍼사드 인터페이스는 변하지 않으므로, 클라이언트 코드는 변경에 영향을 받지 않다. 이는 결국 코드의 유지 관리성과 확장성을 향상시킨다.

2. 팩토리 메소드 패턴(Factory Method Pattern):

Menu파트에서 팩토리 메소드 패턴이 사용되었다. 팩토리 메소드 패턴에서는 객체를 생성하는 메소드가 추상화되어 상위 클래스에서 정의된다. 이 경우 CafeMenu가 그 상위 클래스에 해당하고, createOrder 메소드가 팩토리 메소드에 해당한다. 각 하위 클래스(AdeMenu, CoffeeMenu, LatteMenu)는 이 메소드를 재정의(오버라이딩)하여 각자의 메뉴에 맞는 음료 객체를 생성하고 패키지에 추가한다. 예를 들어, AdeMenu 클래스의 createOrder 메소드는

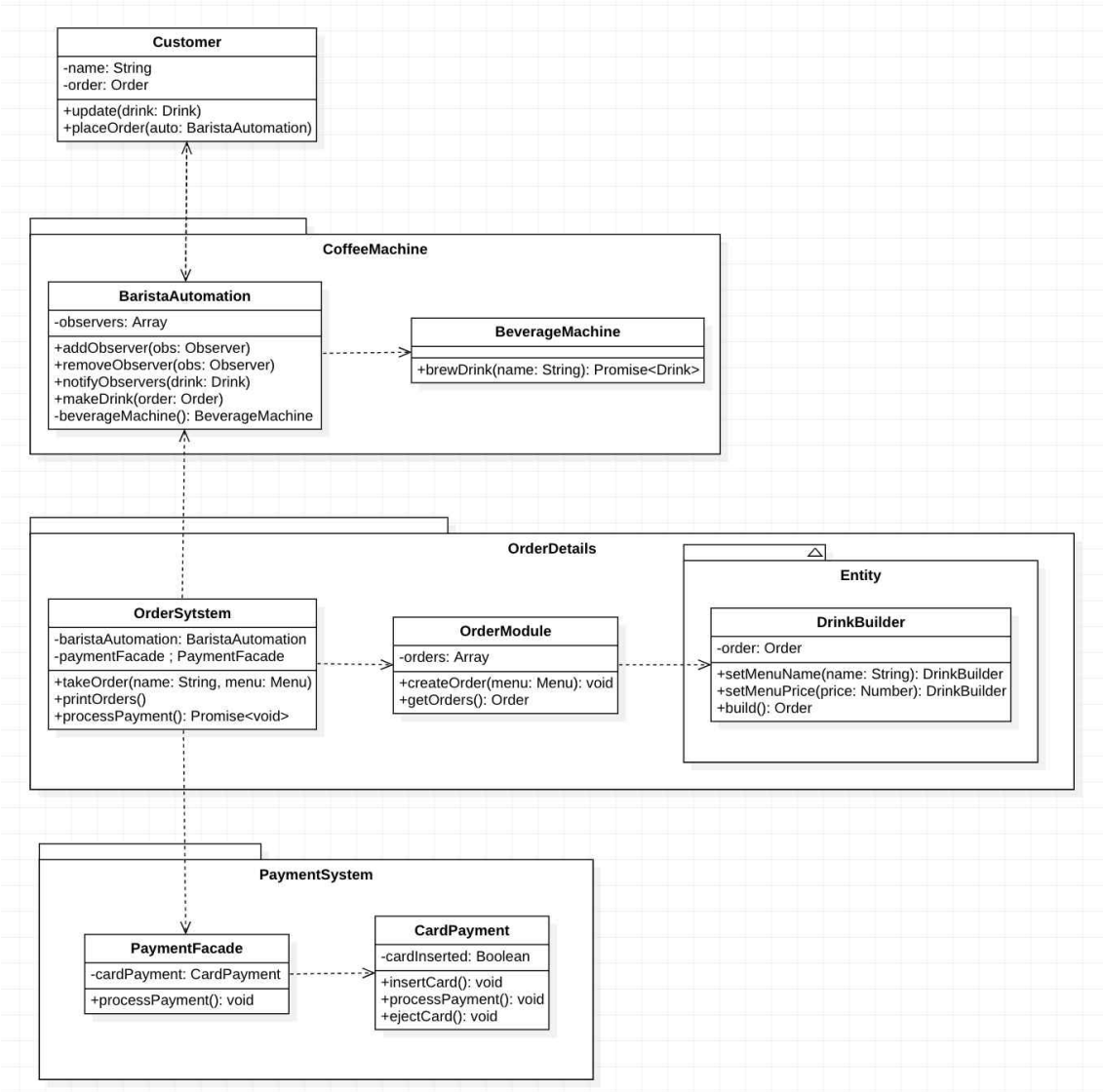
LemonAde, GreenGrapeAde, GrapefruitAde를 생성하고 패키지에 추가한다. 이는 팩토리 메서드 패턴의 핵심 원리인 "객체 생성을 서브 클래스로 위임"을 잘 보여주는 예시이다.
이렇게 하면 객체 생성 로직이 클라이언트 코드로부터 분리되고, 어떤 메뉴 클래스를 사용하더라도 동일한 인터페이스(createOrder)를 통해 음료 객체를 생성할 수 있다. 이를 통해 코드의 유연성과 확장성이 향상된다.
더 추가하면, 새로운 메뉴 클래스가 필요하다면 CafeMenu를 상속받아 createOrder 메소드를 재정의하기만 하면 되므로, 새로운 타입의 메뉴를 쉽게 추가할 수 있다. 이런 방식으로 팩토리 메소드 패턴은 코드의 유지보수성과 확장성을 향상시킨다.

3. 템플릿 메서드 패턴(Template Method Pattern):
Menu파트에서는 템플릿 메서드 패턴 또한 사용되었다. CafeMenu 클래스가 템플릿 메서드 패턴의 상위 클래스 역할을 한다. CafeMenu 클래스는 createOrder() 메서드를 제공하는데, 이 메서드가 바로 알고리즘의 골격을 나타낸다. 그러나 CafeMenu에서 createOrder()는 구체적인 구현 없이 메서드만 정의해두었다.
그리고 AdeMenu, CoffeeMenu, LatteMenu 클래스들은 CafeMenu 클래스를 상속받아 createOrder() 메서드를 각각 다르게 구현하였다. 이 클래스들은 createOrder() 메서드 안에서 특정 음료를 주문 패키지에 추가하는 방식으로 알고리즘의 골격에 맞춰 구체적인 내용을 구현했다.
이런 식으로 템플릿 메서드 패턴을 사용하면, 상위 클래스에서 알고리즘의 골격을 정의하고 하위 클래스에서 구체적인 내용을 구현함으로써 코드의 중복을 최소화하고, 재사용성을 높이며, 하위 클래스 간의 일관성을 유지할 수 있다. 또한 이 패턴을 통해 캡슐화를 강화하고, 확장에는 열려있고 변경에는 닫혀있는 구조를 만들 수 있다.

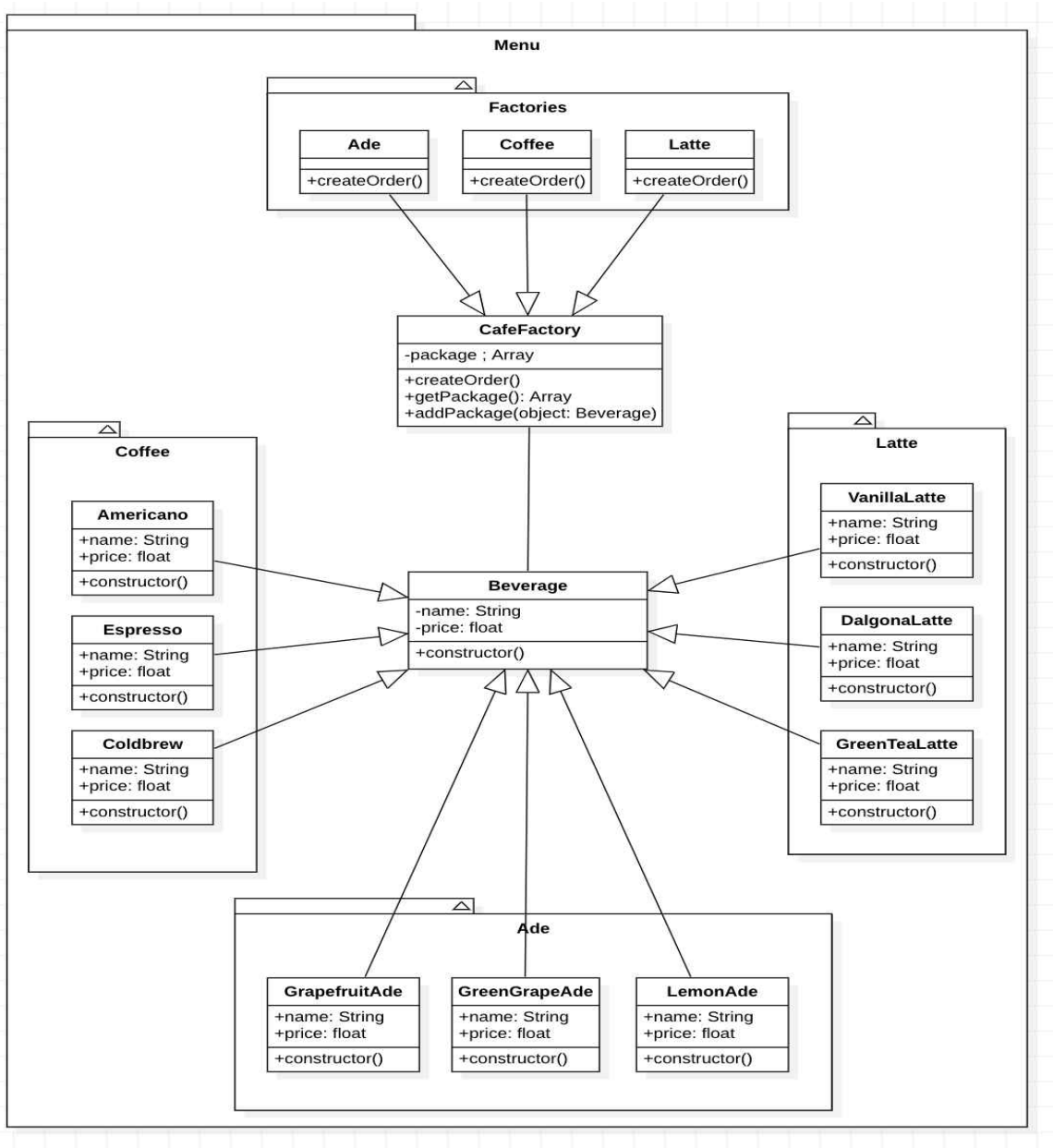
4. 옵저버 패턴(Observer Pattern):
코드 내에서 옵저버 패턴은 BaristaAutomation과 Customer 클래스를 통해 적용되었다. 옵저버 패턴은 객체의 상태 변화를 관찰하는 객체들이 있는 경우, 상태 변화가 일어나면 이를 관찰하고 있는 객체들에게 변화를 알려주는 패턴이다.
1. BaristaAutomation 클래스: 이 클래스는 'Subject' 또는 'Observable' 역할을 한다. 이 객체의 상태가 변하면 이를 'Observer'에게 알려준다. 이 클래스는 Observer 목록(this.observers)을 유지하고, Observer를 추가(addObserver)하거나 제거(removeObserver)하는 메서드를 제공한다. 또한 notifyObservers 메서드를 통해 모든 Observer에게 상태 변경을 알린다.
2. Customer 클래스: 이 클래스는 'Observer' 역할을 합니다. update 메서드를 통해 BaristaAutomation 클래스로부터 상태 변화를 통보받는다. 여기서 상태 변화는 음료 제조 완료 상태를 의미한다.
따라서, Customer 객체가 주문을 하면(placeOrder), 해당 Customer는 BaristaAutomation의 Observer로 등록된다. 그런 다음 BaristaAutomation에서 음료를 만들기 시작하고(makeDrink), 음료 제조가 완료되면 모든 Observer에게 이를 알린다(notifyObservers). 이때 Customer의 update 메서드가 호출되어 주문한 음료가 제조 완료되었음을 알게 된다.
이와 같이 옵저버 패턴은 한 객체의 상태 변화를 다른 객체들이 자동으로 업데이트 받을 수 있도록 해주므로, 객체 간의 의존도를 최소화하고 코드 유지 보수를 용이하게 합니다.

1) 주문 과정 다이어그램

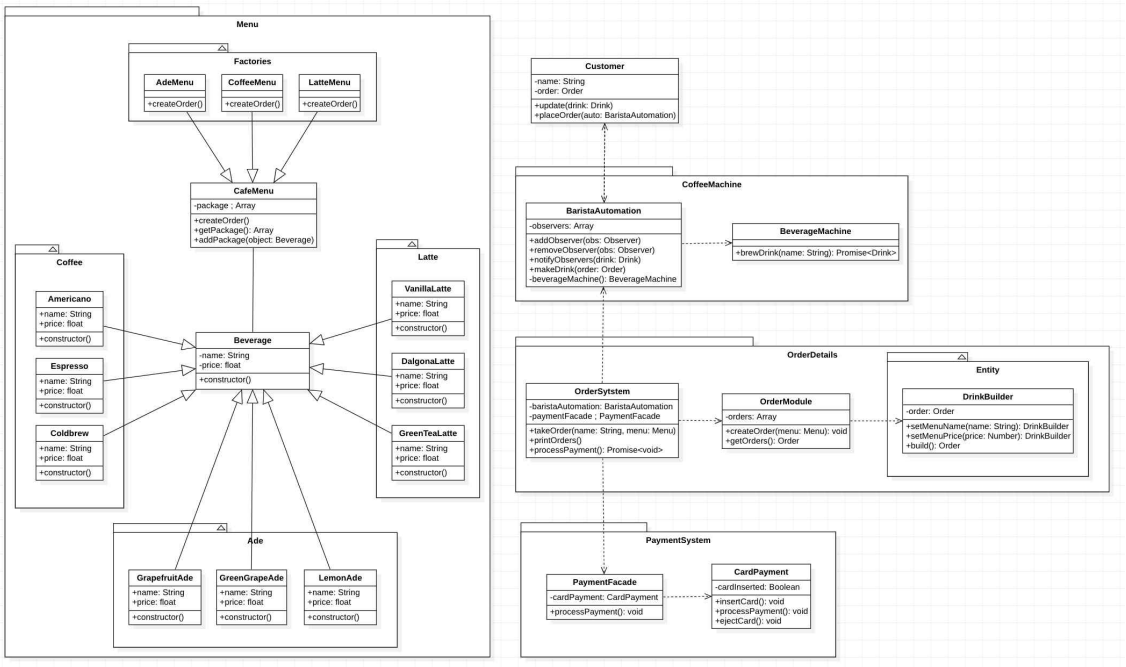
(Facade Pattern, Template Method Pattern, Observer Pattern)



2) 메뉴판 클래스 다이어그램
(Factory Pattern)



3) 최종 다이어그램


Menu
Factories
AdeMenu
+createOrder()
CoffeeMenu
+createOrder()
LatteMenu
+createOrder()
CafeMenu
-package ; Array
+createOrder()
+getPackage(): Array
+addPackage(object: Beverage)
Coffee
Americano
+name: String
+price: float
+constructor()
Espresso
+name: String
+price: float
+constructor()
Coldbrew
+name: String
+price: float
+constructor()
Beverage
-name: String
-price: float
+constructor()
Latte
VanillaLatte
+name: String
+price: float
+constructor()
DalgonaLatte
+name: String
+price: float
+constructor()
GreenTeaLatte
+name: String
+price: float
+constructor()
Ade
GrapefruitAde
+name: String
+price: float
+constructor()
GreenGrapeAde
+name: String
+price: float
+constructor()
LemonAde
+name: String
+price: float
+constructor()
Customer
-name: String
-order: Order
+update(drink: Drink)
+placeOrder(auto: BaristaAutomation)
CoffeeMachine
BaristaAutomation
-observers: Array
+addObserver(obs: Observer)
+removeObserver(obs: Observer)
+notifyObservers(drink: Drink)
+makeDrink(order: Order)
-beverageMachine(): BeverageMachine
BeverageMachine
+brewDrink(name: String): Promise<Drink>
OrderDetails
OrderSytstem
-baristaAutomation: BaristaAutomation
-paymentFacade ; PaymentFacade
+takeOrder(name: String, menu: Menu)
+printOrders()
+processPayment(): Promise<void>
OrderModule
-orders: Array
+createOrder(menu: Menu): void
+getOrders(): Order
Entity
DrinkBuilder
-order: Order
+setMenuName(name: String): DrinkBuilder
+setMenuPrice(price: Number): DrinkBuilder
+build(): Order
PaymentSystem
PaymentFacade
-cardPayment: CardPayment
+processPayment(): void
CardPayment
-cardInserted: Boolean
+insertCard(): void
+processPayment(): void
+ejectCard(): void

3. 구현 코드

1-1) 주문 시스템 관리 및 리스트 출력

```
// 주문 시스템을 관리하는 핵심 클래스로 현재 주문 리스트를 출력하는 기능을 제공한다
// 주문을 받아 처리하고, 결제를 진행한 뒤, 바리스타 자동화 시스템에게 음료 제조를 요청한다.


import BaristaAutomation from '../CoffeeMachine/BaristaAutomation.js';
import Customer from '../Customer.js';
import PaymentFacade from '../PaymentSystem/CardPaymentFacade.js';
import { createOrder, getOrders } from './OrderModule.js';

class OrderSystem {
    constructor() {
        this.baristaAutomation = new BaristaAutomation();
        this.paymentFacade = new PaymentFacade();
    }

    takeOrder(customerName, menu) {
        createOrder(menu);
        const order = getOrders()[getOrders().length - 1];
        const customer = new Customer(customerName, order);

        this.processPayment().then(() => {
            customer.placeOrder(this.baristaAutomation);
        });
    }

    printOrders() {
        const orders = getOrders();
        console.log("\n현재 주문 리스트: \n", orders);
    }

    processPayment() {
        // PaymentFacade의 processPayment() 메서드가 Promise를 반환하도록 변경
        return new Promise((resolve) => {
            this.paymentFacade.processPayment();
            setTimeout(resolve, 3000);
        });
    }
}

export default OrderSystem;
```

1-2) 주문 생성 및 리스트 관리

```
// 주문을 생성하고, 주문 리스트를 관리하는 모듈로 주문 리스트를 가져오는 기능을 포함함
// DrinkBuilder 클래스를 사용하여 주문을 생성하고, 이를 주문 리스트에 추가한다

import DrinkBuilder from './Entity/DrinkBuilder.js';

let orders = [];

function createOrder(menu) {
    let drinkBuilder = new DrinkBuilder();
    drinkBuilder.setMenuName(menu.name);
    drinkBuilder.setMenuPrice(menu.price);
    let order = drinkBuilder.build();
    orders.push(order);
}

function getOrders() {
    return orders;
}

export { createOrder, getOrders };
```

1-3) 주문 객체 생성

```
// Builder 패턴을 이용하여 주문 객체를 생성하는 클래스
// 메뉴 이름과 가격을 설정한 뒤, 주문 객체를 생성하여 반환한다
// 이 클래스를 이용하면 주문 객체를 유연하게 생성하고 관리할 수 있다

class DrinkBuilder {
    constructor() {
        this.order = {};
    }

    setMenuName(menuName) {
        this.order.menuName = menuName;
        return this;
    }

    setMenuPrice(menuPrice) {
        this.order.menuPrice = menuPrice;
        return this;
    }

    build() {
        return this.order;
    }
};

export default DrinkBuilder;
```

2-1) 결제

```
// 카드 결제 과정을 구현한 클래스
// 카드를 삽입하고, 결제 상태를 출력하며, 카드를 제거하는 기능을 제공한다


class CardPayment {
    insertCard() {
        console.log('카드를 삽입하세요.');
    }

    waitAndPrintPaymentStatus() {
        console.log('결제 진행중...');
        setTimeout(() => {
            console.log('결제 완료!');
        }, 3000);  // 3초 후 결제 완료
    }

    ejectCard() {
        console.log('카드를 제거하세요.');
    }
}

export default CardPayment;
```

2-2) 결제 내부

```
// CardPayment 클래스를 감싸고 있는 퍼사드 패턴을 구현한 클래스
// 카드 결제 과정을 한 번에 수행하는 processPayment 메서드를 제공한다


import CardPayment from './CardPayment.js';

class PaymentFacade {
    constructor() {
        this.cardPayment = new CardPayment();
    }

    processPayment() {
        this.cardPayment.insertCard();
        this.cardPayment.waitAndPrintPaymentStatus();
        this.cardPayment.ejectCard();
    }
}

export default PaymentFacade;
```

3-1) 카페 메뉴 생성 및 관리

```
// 메뉴 패키지를 생성하고 관리하는 클래스
// 음료 객체를 포함하는 배열로, 새로운 음료 객체를 추가하거나 패키지 전체를 가져올 수 있다

class CafeFactory {
    constructor() {
        this.package = new Array();
    }
    createOrder() {}
    getPackage() {
        return this.package;
    }
    addPackage(object) {
        this.package.push(object);
    }
}

export default CafeFactory;
```

3-2) 카페 메뉴 분류

```
// CafeMenu를 상속 받아 에이드 메뉴를 생성하는 클래스
// createOrder 메서드에서 자몽에이드, 청포도에이드, 레몬에이드를 주문 패키지에 추가한다

import GrapefruitAde from '../Beverages/Ade/GrapefruitAde.js';
import GreenGrapeAde from '../Beverages/Ade/GreenGrapeAde.js';
import LemonAde from '../Beverages/Ade/LemonAde.js';
import CafeMenu from '../CafeMenu.js';

class AdeMenu extends CafeMenu {
    createOrder() {
        this.addPackage(new LemonAde());
        this.addPackage(new GreenGrapeAde());
        this.addPackage(new GrapefruitAde());
    }
}

export default AdeMenu;
```

3-3) 카페 메뉴 세부 분류

```
// Beverage를 상속 받아 자몽에이드를 정의하는 클래스
// 자몽에이드의 이름과 가격이 설정되어 있다
import Beverage from '../Beverage.js';

class GrapefruitAde extends Beverage {
    constructor() {
        super();
        this.name = "자몽에이드";
        this.price = 4.5;
    }
}

export default GrapefruitAde;
```

4-1) 음료 자동 제조 머신

```
// 음료를 제조하는 로직을 포함하는 클래스
// 각 음료는 3초의 제조 시간을 가정하며, 제조 완료 시 Promise가 resolve되어 제조 완료가 알려진다

class BeverageMachine {
    brewDrink(drinkName) {
        return new Promise((resolve) => {
            console.log(`음료기계: ${drinkName} 제조 시작...`);
            setTimeout(() => {
                console.log(`음료기계: ${drinkName} 완성!`);
                resolve({ name: drinkName });
            }, 3000);
        });
    }
};

export default BeverageMachine;
```

4-2) 음료 완료 고객 알림

```
// 음료를 만드는 자동화 시스템을 제어하며, observer 패턴을 통해 음료 제조 완료 시 알림을 주는 클래스
// 인스턴스를 사용하여 음료를 제조하고, 제조 완료 시 등록된 observer에게 알림을 준다

import BeverageMachine from './BeverageMachine.js';

class BaristaAutomation {
    constructor() {
        this.observers = [];
        this.beverageMachine = new BeverageMachine();
    }

    addObserver(observer) {
        this.observers.push(observer);
    }

    removeObserver(observer) {
        this.observers = this.observers.filter(obs => obs !== observer);
    }

    notifyObservers(drink) {
        this.observers.forEach(obs => obs.update(drink));
    }

    makeDrink(order) {
        console.log(`자동화 시스템: 주문하신 ${order.menuName} 제조를 시작합니다.`);
        this.beverageMachine.brewDrink(order.menuName).then(drink => {
            this.notifyObservers(drink); // 음료 제조가 완료된 후에 observer에게 알림
        });
    }
};

export default BaristaAutomation;
```

4-3) 고객 주문서 및 음료 제조 완료 알림

```
// 고객을 표현하는 클래스로 고객 이름과 주문을 가지고 있다
// 진동벨을 가지고 있어, 주문 음료가 제조 완료되었을 때 진동벨이 울린다
// 이 클래스는 Observer 패턴을 이용하여 음료 제조 완료를 통보받는다
// 여러 모듈과 상호작용하여 카페 주문과 음료 제조 과정을 구현한다

class Customer {
    constructor(name, order) {
        this.name = name;
        this.order = order;
    }

    update(drink) {
        if (drink.name === this.order.menuName) {
            console.log(`${this.name}: ${drink.name} 완성! 진동벨 울림.`);
        }
    }

    placeOrder(baristaAutomation) {
        baristaAutomation.addObserver(this);
        baristaAutomation.makeDrink(this.order);
    }
}

export default Customer;
```

4. 테스트 코드

1) 주문

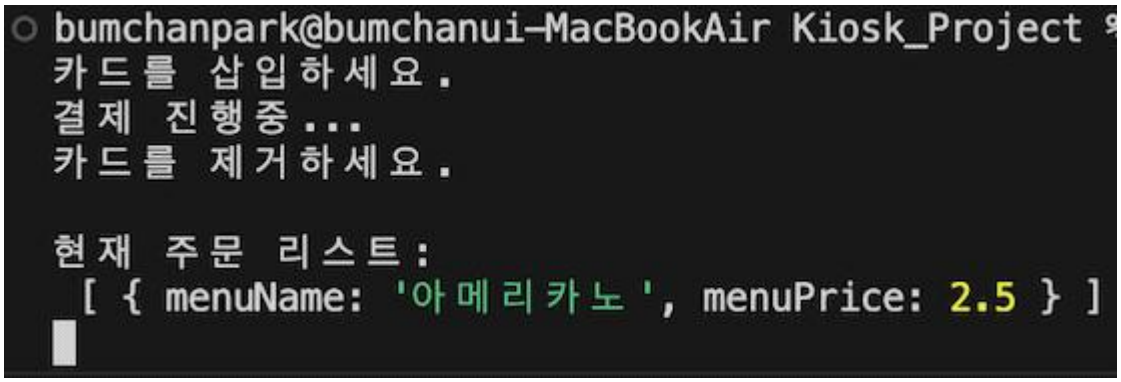
```
// 전체 카페 시스템의 흐름을 나타내는 테스트 코드
// 현재 코드상에선, 아메리카노를 주문한다고 가정한다.
// 주문을 하면 리스트에 적용이 되고, 카드로 결제를 하면 커피 제조가 된 뒤, 진동벨이 울리는 과정을 거친다.

import Americano from './Menu/Beverages/Coffee/Americano.js';
import OrderSystem from './OrderDetails/OrderSystem.js';

const customerName = '박범찬(고객)';

let orderSystem = new OrderSystem();
let americano = new Americano();

orderSystem.takeOrder(customerName, americano);

orderSystem.printOrders();
```

2) 메뉴판

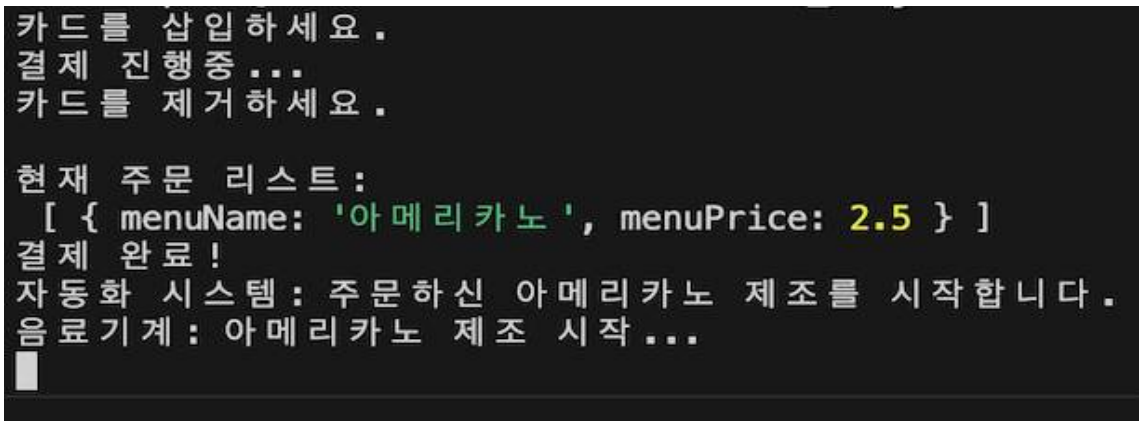
```
// 메뉴 폴더들 안의 파일을 불러와 정상적으로 잘 작동하는지 테스트하는 파일

import AdeMenu from './Factories/AdeMenu.js';
import CoffeeMenu from './Factories/CoffeeMenu.js';
import LatteMenu from './Factories/LatteMenu.js';

const customer1 = new CoffeeMenu();
customer1.createOrder();
console.log(customer1.getPackage());

const customer2 = new AdeMenu();
customer2.createOrder();
console.log(customer2.getPackage());

const customer3 = new LatteMenu();
customer3.createOrder();
console.log(customer3.getPackage());
```

5. 테스트 결과 화면

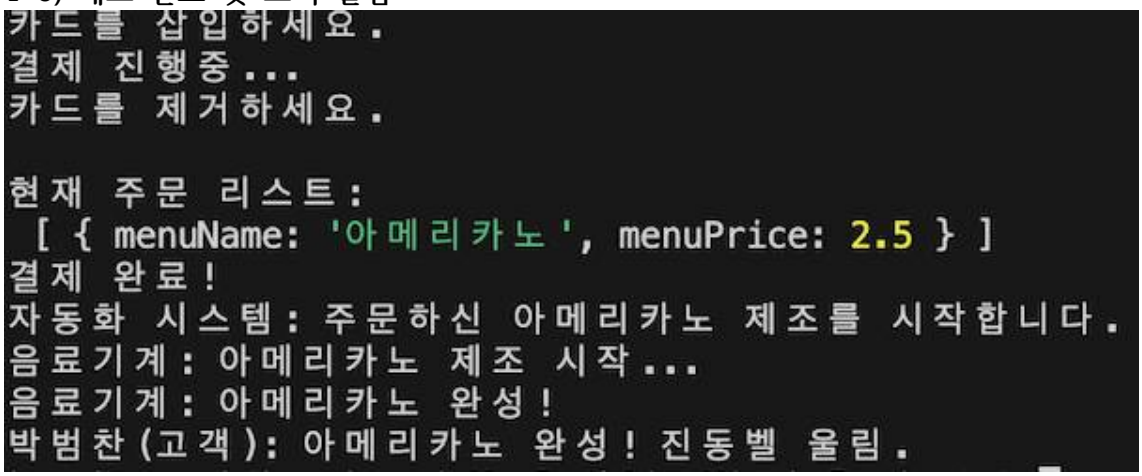
1) 주문



1-2) 음료 제조



1-3) 제조 완료 및 고객 알림



2) 메뉴판

```
[
  Americano { name: '아메리카노', price: 2.5 },
  Espresso { name: '에스프레소', price: 2.5 },
  Coldbrew { name: '콜드브루', price: 3.5 }
]
[
  LemonAde { name: '레몬에이드', price: 4.5 },
  GreenGrapeAde { name: '청포도에이드', price: 4.5 },
  GrapefruitAde { name: '자몽에이드', price: 4.5 }
]
[
  DalgonaLatte { name: '달고나라떼', price: 5.5 },
  GreenTeaLatte { name: '녹차라떼', price: 5.5 },
  VanillaLatte { name: '바닐라라떼', price: 5.5 }
]
```

부록 1. 참여율

박범찬: 50%
김예은: 50%

부록 2. 프로젝트 수행 인증 사진