객체지향개발론및실습

Laboratory 3. Decorator 패턴: StarBuzz 커피숍 응용

1. 목적

- 객체에 동적으로 새로운 책임(기능, 상태)을 추가할 수 있도록 해주는 **장식 패턴**에 대해 숙제를 포함하여 다음을 실습해 본다.
  - 장식 패턴에서 장식된 것을 제거하는 방법을 구현하여 본다.
  - 장식 패턴의 객체 생성을 추상화해주는 생성 메소드를 구현하여 본다.
  - 장식을 제한하는 방법을 구현하여 본다.
  - 장식 패턴을 이용하여 장식한 객체를 서로 비교하는 방법을 구현하여 본다.

2. 개요

- 커피를 판매할 때 고객들은 다양한 첨가물(스팀우유, 두유, 모카, 크림 등)을 추가할 수 있다. 이에 대한 가격을 유연하게 계산하기 위해 이들을 장식 패턴을 통해 모델링하여 구현할 수 있다.
- 이미 StarBuzz 커피숍 응용은 이론 시간에 설명 및 실습하였다.

3. 실습 1

- 실습 1에서는 추가된 책임을 제거하는 방법을 실습해본다.
- 수업 시간에 완성한 첫 번째 StarBuzz 커피숍 응용을 복사하여 새 프로젝트를 만들고 수정하시오.
- 맨 마직막으로 추가한 첨가물을 제거하는 방법을 생각하여 보자.
- 테스트 프로그램을 통해 알 수 있듯이 커피 음료는 항상 **Beverage** 타입으로 유지된다.
- 이 때문에 **removeCondiment**라는 메소드는 **CondimentDecorator** 타입에서만 필요한 것이지만 **Beverage** 타입에 추가해야 사용이 용이하다.
- 예)

```
Beverage beverage = new DarkRoast();
beverage = new Mocha(beverage);
beverage = new Mocha(beverage);
beverage = new Whip(beverage);
System.out.printf("%s: %,d원%n",
    beverage.getDescription(), beverage.cost());
beverage = beverage.removeCondiment();
System.out.printf("%s: %,d원%n",
    beverage.getDescription(), beverage.cost());
```

- 이 실습에서는 **Beverage** 클래스와 각 장식자 클래스에 다음 메소드를 추가한다.
  - **public Beverage removeCondiment()**

    - 사후조건: 현재 클래스의 타입이 **Beverage**이면 **this**를 반환하고, **CondimentDecorator** 타입이면 장식한 **beverage**를 반환한다.

- 장식자 클래스를 만들면서 그것이 장식할 클래스를 수정해야 하는 것은 바람직하지 않다. 실제로 **Beverage** 클래스에 **removeCondiment** 메소드를 추가하지 않고 **CondimentDecorator** 타입에만 추가하여 사용할 수 있다. 이 경우 테스트 프로그램은 다음과 같이 변경이 필요하다.

```
Beverage beverage = new DarkRoast();
beverage = new Mocha(beverage);
beverage = new Mocha(beverage);
beverage = new Whip(beverage);
System.out.printf("%s: %,d원%n",
    beverage.getDescription(), beverage.cost());
if(beverage instanceof CondimentDecorator condiment){
    beverage = condiment.removeCondiment();
}
System.out.printf("%s: %,d원%n",
    beverage.getDescription(), beverage.cost());
```

- 이처럼 구현하기 위해서는 각 장식자에서 장식 대상을 유지하지 않고 **CondimentDecorator**에서 유지하는 것이 효과적이다. 이 이후부터는 항상 **CondimentDecorator**에서 장식 대상을 유지하는 형태로 구현한다.
- C++에서는 다형성이 지원되는 상속 관계에서 포인터나 참조 타입이 가리키는 객체의 타입이 **CondimentDecorator** 타입인지 검사하는 방법은 크게 두 가지 방법이 있다.
    - 첫째, **dynamic_cast**를 이용하는 방법이다. **dynamic_cast**를 해당 타입으로 실행 시간에 변환하였을 때 성공하면 주어진 객체는 해당 타입의 객체(해당 타입 또는 후손)가 된다.
    - 둘째, **typeid**를 이용하는 방법이다. 하지만 정확하게 일치하는 경우만 검사할 수 있으므로 조상 타입을 검사하는 목적으로 사용하기 어렵다.
- 파이썬은 **isinstance**를 이용하면 된다.

3.1 검토사항

- 실습 1에서 문제 제기한 장식 패턴에서 장식된 것을 제거하는 기능을 제공하는 것이 논리적이지 않고 필요 없는 기능이라고 주장되는 경우도 있다. 동적으로 추가된 장식이 제거된 어떤 객체가 필요하면 기존 객체를 이용하지 않고 새로 만들 수 있다.
- 특히, 제거 기능을 제공하기 위해 사용된 기법은 기술적으로는 가능한 기법이지만 기본 타입에 원래 필요 없는 메소드가 추가되어야 하기 때문에 논리적이라고 보기도 어렵다.

4. 실습 2

- 자바의 reflection 클래스들을 이용하여 장식자를 이용한 객체 생성 부분을 추상화할 수 있다. 우리가 사용하고자 하는 형태는 다음과 같다.

```
Beverage beverage
    = BeverageFactory.createCoffee("HouseBlend", "Mocha", "Whip","Mocha");
```

- 위 **static** 메소드를 구현하시오. 이를 위해 **Class** 클래스의 **forName** 메소드와 **Constructor** 클래스의 **getConstructor** 메소드, **newInstance** 메소드의 사용이 필요하다.
- 예) **DarkRoast** 클래스 정보를 가진 **Class** 객체 얻기

```
Class<?> coffeeClass = Class.forName("DarkRoast");
```

이때 해당 클래스가 **Beverage**의 하위 클래스인지 검사하고 싶으면 다음을 추가한다.

```
Class<? extends Beverage> coffeeClass
    = Class.forName("DarkRoast").asSubclass(Beverage.class);
```

추가로 할 수 있는 예외 검사는?

- 예) `Class` 객체로부터 `public` 기본 생성자 얻기

```
Constructor<? extends Beverage> constructor = coffeeClass.getConstructor();
```

`public`이 아니면 `getDeclaredConstructor`를 대신 이용함. 첨가물의 경우에는 기본 생성자가 아님. 기본 생성자가 아닌 경우에는 생성자 요청 메소드에 인자로 생성자의 매개변수 타입을 전달한다.

  - 예) `getConstructor(Beverage.class)`는 `Beverage` 타입을 하나 받는 생성자를 요청한다.
  - 참고. 생성자의 매개변수 타입이 원시타입이면 어떻게? `getConstructor(int.class)`

- 예) `Constructor` 객체를 이용하여 생성자 호출 (객체 생성)

```
Beverage beverage = (Beverage)constructor.newInstance();
```

- 이 개념이 어려우면 다음 클래스가 있을 때 reflection 라이브러리를 이용하여 각 생성자를 사용하여 객체를 생성해 보자.

```
class A{
    private int x = 0;
    private String s = "";
    public A(){}
    public A(int x){ this.x = x; }
    public A(int x, String s){
        this(x);
        this.s = s;
    }
}
```

- 파이썬은 전역 식별자 해시 테이블을 반환하여 주는 `globals`를 이용함
  - 클래스 이름은 전역 식별자 해시 테이블에 등록되어 있으며, 그것의 타입 object가 value 값으로 저장되어 있음
  - 타입 객체의 `__bases__`과 같은 것을 이용하여 클래스의 부모 클래스 정보를 얻을 수 있음
- C++는 동적 언어가 아니므로 실행 시간에 타입 정보를 얻을 수 없다. 이에 열거형으로 커피와 첨가물을 정의하고 다음과 같은 형태의 생성 메소드를 만든다.

```
enum class CoffeeType {DARKROAST, HOUSEBLEND};
enum class CondimentType {MILK, MOCHA, WHIP};
Beverage* createCoffee(CoffeeType coffee,
    std::initializer_list<CondimentType> condiments);

Beverage* beverage
    = createCoffee(CoffeType::HOUSEBLEND,
        {CondimentType::MOCHA, CondimentType::WHIP, CondimentType::MOCHA});
```

5. 실습 3

- 장식 패턴은 보통 동일 장식자를 이용하여 여러 번 장식할 수 있고, 장식하는 순서가 중요하지 않다.
- 하지만 응용에 따라 여러 번 장식하는 것을 제한하고 싶을 수 있고, 장식하는 순서를 제한하고 싶을 수 있다.
- 이 실습에서는 여러 번 장식하는 것을 제한하는 것을 구현해 본다.
- 우선 `Milk` 장식자는 한 번만 장식할 수 있도록 구현하여 보자. 장식할 때 기존에 `Milk`로 장식되어 있는지 어떻게 알 수 있을까?
- 이것을 일반화할 수 있을까? 규칙을 정하고 규칙대로 동작하도록 구현할 수 있을까? 규칙의 예는 다음과 같음

  - double mocha는 OK, triple mocha 이상은 NO!
  - milk는 double milk 이상 NO!
  - expresso는 whip으로 장식 불가
- 이와 같은 제한을 장식 대상에 구현하면 장식 대상과 장식자와 더 밀접한 관계가 형성되는 문제가 있다.

6. 실습 4

- 장식 패턴에서는 다양한 장식 객체를 이용하여 장식 대상을 장식할 수 있다. 이렇게 장식된 객체에서 가장 아래에 있는 장식 대상 객체를 알고 싶을 수 있다. 예를 들어 장식을 제한해야 할 때에도 가장 아래에 있는 장식 대상 객체가 어떤 타입인지 알아야 할 수 있다.
- 새로운 메소드를 추가하지 않고도 가장 아래에 있는 장식 대상 객체의 타입을 알 수 있다. 하지만 다음과 같은 메소드를 **Beverage**에 추가하는 것이 필요하다. 이 메소드가 동작하도록 필요한 클래스를 수정하시오.

  ```
  Class<? extends Beverage> getCoffee();
  ```

  반환 타입을 **Beverage**로 할 수 있지만 이 메소드는 해당 객체의 타입을 알고 싶은 것이기 때문에 이 실습에서는 **getClass**를 호출하여 반환하도록 구현한다.
- 파이썬의 경우에는 **getClass** 대신에 **__class__**를 반환하도록 구현한다.
- C++는 동적 언어가 아니므로 클래스 정보 대신에 **Beverage***를 반환하도록 구현한다.

7. 과제 1

- 장식된 객체의 등가 여부를 비교하는 기능을 추가하여 보자. 자바에서는 **equals** 메소드를 통해 두 객체가 같은지 여부를 확인하며, 파이썬은 **__eq__** 메소드를 이용하고, C++는 **operator==**와 **operator!=**를 다중정의한다.
- **equals** 메소드를 **Beverage** 클래스와 **CondimentDecorator** 클래스에만 구현하면 원하는 기능을 얻을 수 있다.
- 보통 **equals** 메소드는 모든 멤버 변수를 비교하여 같은지 여부를 결정하지만 커피 예제에서는 멤버 변수를 이용한 비교는 필요 없다.
- 주의사항. 장식된 순서가 다르더라도 첨가물이 같고, 커피가 같은 종류이면 **true**를 얻을 수 있어야 한다.

8. 과제 2

- 실습 2에서 만든 생성 메소드에 장식 제한 기능을 추가하여 보자.
- 각 장식마다 그것의 장식 규칙을 나타내는 **Restriction** 클래스를 다음과 같이 정의하여 사용하여 보자.

```
1 class Restriction{
2     public int maxAddition = 0;
3     public Set<String> exclusionList  = new HashSet<>();
4 }
```

  여기서 **maxAddition**이 0이면 제한이 없는 것이고, **maxAddition**이 k이면 k번 추가할 수 있는 것이다. **exclusionList**는 해당 첨가물을 추가할 수 없는 커피 목록이다.
- **BeverageFactory** 클래스에 **Map<String, Restriction> restrictionTable**를 **static** 멤버로 추가한 다음, 존재하는 첨가물을 이 맵에 기본값으로 추가한다.
- 파이썬은 **restrictionTable**이 **dict** 타입이 된다.
- 다음과 같이 제한 규칙을 지정할 수 있다.

```
BeverageFactory.addAdditionRestriction("Whip", 1);
BeverageFactory.addAdditionRestriction("Milk", 1);
BeverageFactory.addAdditionRestriction("Mocha", 2);
BeverageFactory.addCoffeeRestriction("Mocha", "DarkRoast");
```

이를 위해 **addAdditionRestriction**과 **addCoffeeRestriction** **static** 메소드를 추가하여야 한다. 참고로 위와 같이 제한을 두면 크림과 우유는 오직 한번만 첨가할 수 있고, 모카는 최대 두번까지만 첨가할 수 있다. 또 DarkRoast 커피에는 모카를 첨가할 수 없다.

- C++는 다음과 같은 구조체를 사용하며,

```
struct Restriction{
    int maxAddition{0};
    std::unordered_set<CoffeeType> exclusionList;
};
```

제한 테이블은 다음을 사용한다.

```
class BeverageFactory{
    static inline std::unordered_map<CondimentType, Restriction>
        restrictionTable{};
public:
    static void addAdditionRestriction(
        CondimentType condiment, int maxAddition);
    static void addCoffeeRestriction(
        CondimentType condiment, CoffeeType coffee);
    static Beverage* createCoffee(
        CoffeeType coffee, std::initializer_list<CondimentType> condiments);
};
```