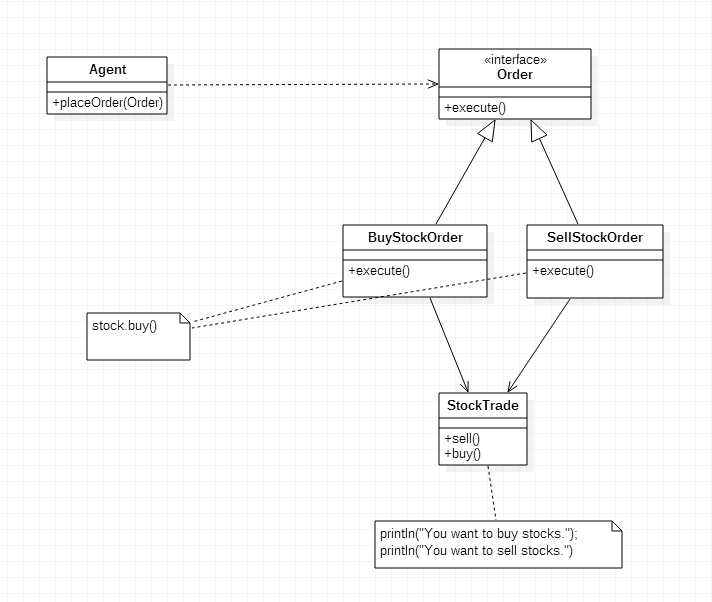
*객체지향 설계와 패턴*  
**LAB ASSIGNMENT #11**

## Weight 10% 마감 6월 4일(금)

**목적: 커맨드 패턴, 데코레이터, 반복자**

문제 #1: 커맨드 패턴

다음은 주식을 거래하기 위한 프로그램의 클래스 설계이다.



(1) 위 설계를 만족하는 각 클래스를 Java 언어로 코딩하라.

public class Agent {  
 void placeOrder(Order order) {  
 order.execute();  
 }  
}

public interface Order {  
 void execute();  
}

public class BuyStockOrder implements Order {  
 private StockTrade buy;  
  
 public BuyStockOrder(StockTrade buy) {  
 this.buy = buy;  
 }  
  
 @Override  
 public void execute() {  
 buy.buy();  
 }  
}

public class SellStockOrder implements Order {  
 private StockTrade sell;  
  
 public SellStockOrder(StockTrade sell) {  
 this.sell = sell;  
 }  
  
 @Override  
 public void execute() {  
 sell.sell();  
 }  
}

public class StockTrade {  
 public void sell() {  
 System.*out*.println("You want to sell stocks.");  
  
 }  
  
 public void buy() {  
 System.*out*.println("You want to buy stocks.");  
 }  
}

커맨드 패턴이 적용된 곳은 BuyStockOrder와 SellStockOrder 클래스로 각각 매도 행위와 매수 행위를 객체로 캡슐화한 커맨드객체이다. 여기서 실제 행위를 수행하는 Receiver클래스는 StockTrade 클래스이고, 커맨드 객체들이 해당 인스턴스로 구체적인 행동을 수행한다.

(2) 다음 클라이언트 프로그램으로 실행시켜 보고 결과를 캡처하라.

// Client

public class Client {

public static void main(String[] args) {

StockTrade stock = new StockTrade();

BuyStockOrder buyStockOrder = new BuyStockOrder(stock);

SellStockOrder sellStockOrder = new SellStockOrder(stock);

Agent agent = new Agent();

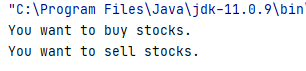
agent.placeOrder(buyStockOrder ); // Buy Shares

agent.placeOrder(sellStockOrder ); // Sell Shares

}

}

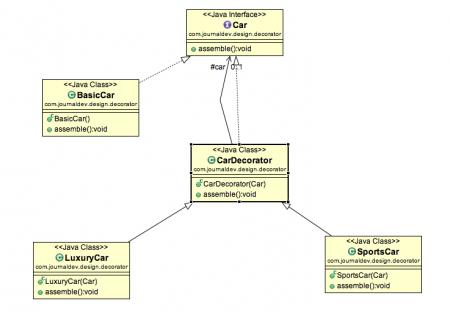
<실제결과>



문제 2: 데코레이터 패턴

여러 종류의 자동차를 구현하고 싶다. Basic car에 장식으로 Sport car 또는 Luxury car를 확장할 수 있게 다음과 같이 데코레이터 패턴으로 설계하였다.

Car에 CarDecorator로 Luxury나 Sports등을 첨가한 모습이다. 첨가될때마다 assemble()함수로 인해 내용을 첨가할 수 있다.



위 설계를 코딩하고 다음과 같은 프로그램으로 테스팅하라. 제시된 출력 결과가 나와야 한다.

package com.journaldev.design.test;

import com.journaldev.design.decorator.BasicCar;

import com.journaldev.design.decorator.Car;

import com.journaldev.design.decorator.LuxuryCar;

import com.journaldev.design.decorator.SportsCar;

public class DecoratorPatternTest {

public static void main(String[] args) {

Car sportsCar = new SportsCar(new BasicCar());

sportsCar.assemble();

System.out.println("\n\*\*\*\*\*");

Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar())); sportsLuxuryCar.assemble();

}

}

public interface Car {  
 void assemble();  
}

public class BasicCar implements Car {  
 public BasicCar() {  
 *//this.assemble();* }  
  
 @Override  
 public void assemble() {  
 System.*out*.print("Basic Car. ");  
 }  
}

public abstract class CarDecorator implements Car {  
 CarDecorator(Car car) {  
 car.assemble();  
 }  
  
 public abstract void assemble();  
  
}

public class LuxuryCar extends CarDecorator {  
 LuxuryCar(Car car) {  
 super(car);  
 }  
  
 @Override  
 public void assemble() {  
 System.*out*.print("Adding features of Luxury Car. ");  
 }  
}

public class SportsCar extends CarDecorator {  
 public SportsCar(Car car) {  
 super(car);  
 }  
  
 @Override  
 public void assemble() {  
 System.*out*.print("Adding features of Sports Car. ");  
 }  
}

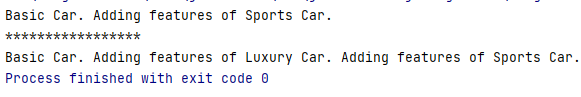
출력 결과:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.

<실제 결과>



문제 3: 반복자 패턴

다음은 Topic 자료에 대하여 반복자 패턴을 적용하기 위한 코드의 일부이다.

public class Topic

{

    private String name;

    public Topic(String name) {

        super();

        this.name = name;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

}

public interface Iterator<E>

{

void reset(); // reset to the first element

E next(); // To get the next element

E currentItem(); // To retrieve the current element

boolean hasNext(); // To check whether there is any next element or not.

}

public class Main

{

    public static void main(String[] args)

    {

        Topic[] topics = new Topic[5];

        topics[0] = new Topic("topic1");

        topics[1] = new Topic("topic2");

        topics[2] = new Topic("topic3");

        topics[3] = new Topic("topic4");

        topics[4] = new Topic("topic5");

        List<Topic> list = new TopicList(topics);

        Iterator<Topic> iterator = list.iterator();

        while(iterator.hasNext()) {

            Topic currentTopic = iterator.next();

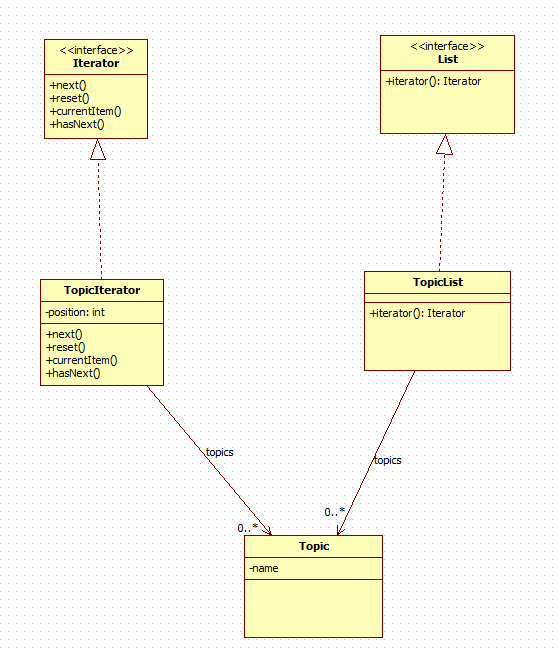
            System.out.println(currentTopic.getName());

        }

    }

}

(1) 위의 메인 프로그램이 수행되기 위하여 필요한 클래스를 추가하여 설계하고 UML 클래스 다이어그램으로 나타내라.



TopicList의 iterator()메소드를 통해 컬렉션 안의 모든 원소들에 대해 접근할 수 있다. iterator()메소드는 TopicIterator 객체를 반환하는데, TopicIterator 클래스는 Topic 배열에 대한 접근을 책임지고 있는 클래스이다. Topic 배열에 대한 접근에 관한 attribute로는 Topic배열 그자체인 topics, 그리고 topics안의 위치를 의미하는 position이 있다. 메소드로는 다음 Topic객체를 리턴하는 next(), 현재 Topic객체를 리턴하는 currentItem, 위치를 처음으로 돌리는 reset(), 다음 객체가 존재하는지 확인하는 hasNext()가 있다.

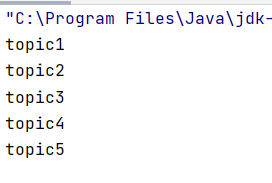
(2) 코드가 제시되지 않은 클래스에 대하여 구현하고 실행하라.

public interface List<E>  
{  
 Iterator<E> iterator();  
}

public class TopicList implements List<Topic>  
{  
 private Topic[] topics;  
  
 public TopicList(Topic[] topics)  
 {  
 this.topics = topics;  
 }  
  
 @Override  
 public Iterator<Topic> iterator() {  
 return new TopicIterator(topics);  
 }  
}

public class TopicIterator implements Iterator<Topic> {  
  
 private Topic[] topics;  
 private int position;  
  
 public TopicIterator(Topic[] topics)  
 {  
 this.topics = topics;  
 position = 0;  
 }  
  
 @Override  
 public void reset() {  
 position = 0;  
 }  
  
 @Override  
 public Topic next() {  
 return topics[position++];  
 }  
  
 @Override  
 public Topic currentItem() {  
 return topics[position];  
 }  
  
 @Override  
 public boolean hasNext() {  
 if(position >= topics.length)  
 return false;  
 return true;  
 }  
}

(3) 실행 결과를 캡처하라.

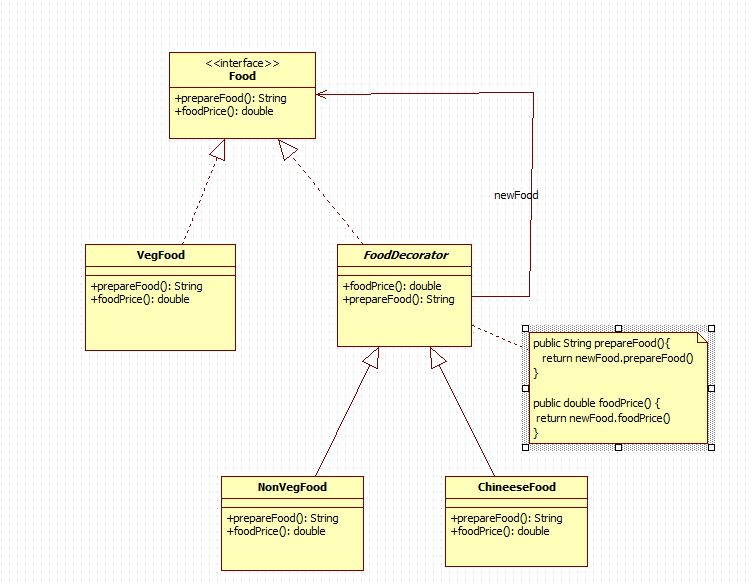


**문제 4: 데코레이터 패턴**

어느 채식 전문 식당에서 메뉴를 주문하는 프로그램을 개발하려고 한다. 기본적으로 채식류(Vegetarian Food)을 주문할 수 있고 Non-Vegetarian 메뉴를 주문하면 채식에 구운 치킨과 치킨 커리(Roasted Chicken and Chicken Curry)가 장식된다. 또한 중국 음식을 주문하면 채식 음식에 볶음밥(Fried Rice)과 야채튀김(Manchurian)이 나온다.

(1) DecoratorPatternCustomer 라는 메인 프로그램이 메뉴를 준비한다고 할 때 Food, VegeFood와 데코레이션을 위한 Food의 클래스를 설계하여 클래스 다이어그램으로 그려라.

VegFood와 Food에 첨가를 담당하는 데코레이터 클래스 FoodDecorator는 동일한 인터페이스를 가진다. 동시에 FoodDecorator클래스는 첨가의 대상이 될 Food 객체(newFood)를 attribute로 가지고 있어야한다. 예를 들어, newFood에 ChineeseFood를 첨가했을 때 가격을 계산해보면, (newFood의 가격) + (ChineeseFood첨가물의 가격)이 될테고 이 계산은 ChineeseFood 클래스의 foodPrice()메소드에서 행해진다. 즉, super.foodPrice()가 newFood의 가격에 해당하고, 65.0이 첨가된 가격에 해당하게 되는 것이다.



(2) 아래와 같은 실행 결과가 나올 수 있게 위의 설계를 코딩하라.

========= Food Menu ============

1. Vegetarian Food.

2. Non-Vegetarian Food.

3. Chinese Food.

4. Exit

Enter your choice: 1

Veg Food

50.0

========= Food Menu ============

1. Vegetarian Food.

2. Non-Vegetarian Food.

3. Chinese Food.

4. Exit

Enter your choice: 2

Veg Food With Roasted Chicken and Chicken Curry

200.0

========= Food Menu ============

1. Vegetarian Food.

2. Non-Vegetarian Food.

3. Chinese Food.

4. Exit

Enter your choice: 3

Veg Food With Fried Rice and Manchurian

public interface Food {  
 public String prepareFood();  
  
 public double foodPrice();  
}

public class VegFood implements Food {  
 @Override  
 public String prepareFood() {  
 return "Veg Food";  
 }  
  
 @Override  
 public double foodPrice() {  
 return 50.0;  
 }  
}

public abstract class FoodDecorator implements Food {  
 private Food newFood;  
  
 public FoodDecorator(Food newFood) {  
 this.newFood = newFood;  
 }  
  
 @Override  
 public String prepareFood() {  
 return newFood.prepareFood();  
 }  
  
 @Override  
 public double foodPrice() {  
 return newFood.foodPrice();  
 }  
}

public class NonVegFood extends FoodDecorator {  
 public NonVegFood(Food newFood) {  
 super(newFood);  
 }  
  
 @Override  
 public String prepareFood() {  
 return super.prepareFood() + " With Roasted Chicken and Chicken Curry ";  
 }  
  
 @Override  
 public double foodPrice() {  
 return super.foodPrice() + 150.0;  
 }  
}

public class ChineeseFood extends FoodDecorator {  
 public ChineeseFood(Food newFood) {  
 super(newFood);  
 }  
  
 @Override  
 public String prepareFood() {  
 return super.prepareFood() + " With Fried Rice and Manchurian ";  
 }  
  
 @Override  
 public double foodPrice() {  
 return super.foodPrice() + 65.0;  
 }  
}

import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
  
public class DecoratorPatternCustomer {  
 private static int *choice*;  
  
 public static void main(String[] args) throws NumberFormatException, IOException{  
 do {  
 System.*out*.print("========= Food Menu ============ \n");  
 System.*out*.print("1. Vegetarian Food. \n");  
 System.*out*.print("2. Non-Vegetarian Food.\n");  
 System.*out*.print("3. Chineese Food. \n");  
 System.*out*.print("4. Exit \n");  
 System.*out*.print("Enter your choice: ");  
  
 final BufferedReader br = new BufferedReader(new InputStreamReader(System.*in*));  
  
 *choice* = Integer.*parseInt*(br.readLine());  
  
 switch (*choice*) {  
 case 1:  
 {  
 VegFood vf = new VegFood();  
 System.*out*.println(vf.prepareFood());  
 System.*out*.println(vf.foodPrice());  
 }  
 break;  
  
 case 2:  
 {  
 Food f1 = new NonVegFood((Food) new VegFood());  
 System.*out*.println(f1.prepareFood());  
 System.*out*.println(f1.foodPrice());  
 }  
 break;  
 case 3:  
 {  
 Food f2 = new ChineeseFood((Food) new VegFood());  
 System.*out*.println(f2.prepareFood());  
 System.*out*.println(f2.foodPrice());  
 }  
 break;  
 default: {  
 System.*out*.println("Other than these no food available");  
 }  
 return;  
 }  
  
 } while (*choice* != 4);  
  
 }  
}

**제출물**

제출하여야 할 것:

1. UML로 표현한 설계와 파악한 패턴과 어디에 적용되었는지 표시하고 어떤 효과가 있었는지 기술하라.

2. 구현한 결과:

\* 원시코드

\* 실행 한 후의 결과 화면

**평가**

과제 #11다음과 같은 기준으로 평가할 것임:

1. 패턴 마이닝의 정확성

2. 구현의 정확성

3. 설계, 구현 결과물에서 볼 수 있는 설계에 대한 이해 수준