객체지향개발론및실습

Laboratory 4. Factory Method와 Singleton 패턴

1. 목적

- 구체적인 객체의 생성을 하위 클래스로 위임하는 Factory 메소드 패턴을 실습하여 본다.
- 클래스가 오직 하나의 인스턴스만 가지도록 보장해주며, 이 인스턴스에 대한 광역적 접근 방법을 제공해주는 singleton 패턴을 실습해본다.
- Singleton 패턴을 확장하여 생성되는 인스턴스의 개수를 제한할 수 있으며, 이에 대해서도 실습해본다.

2. 개요

- 특정 회사 제품을 판매하지 않고 다양한 회사의 차량을 판매하는 업체가 있다고 가정하자. 이 업체는 사용자들이 각 사용자의 취향을 선택하면 그 취향에 맞는 차량을 추천하여 주는 시스템을 만들고자 한다.
- 차량은 크게 일반 승용과 OffRoad 차량으로 구분된다고 가정하자. 또 일반 승용은 Saloon, Compact, Sports로 세분화되고, OffRoad는 OriginalSUV와 CrossoverSUV로 세분화된다고 가정하자. 이와 같은 종류를 나타나기 위해 상속을 활용하였다고 가정하자.
- 이 실습에서는 복잡성을 줄이기 위해 각 차량은 설명, 차량색, 가격 세 가지 정보만 유지한다고 가정한다.
- 전체적인 클래스 관계도는 그림 2.1과 같다.

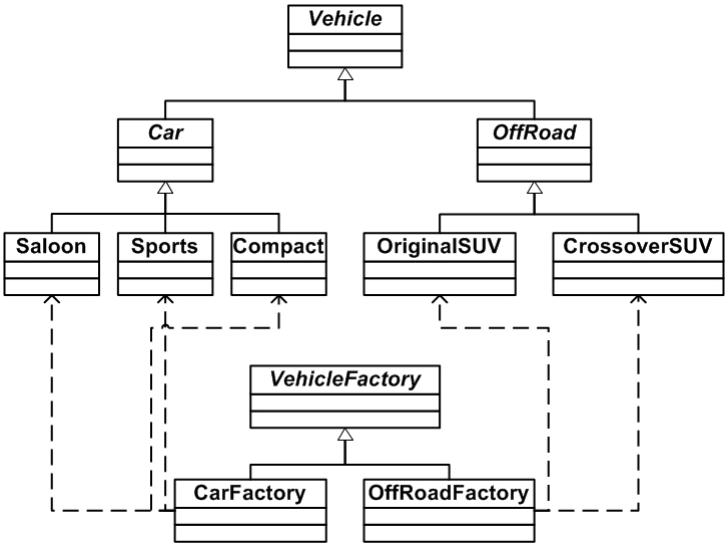

그림 2.1: 실습 4의 클래스 관계도

3. 실습

- 클래스 관계도에 제시된 일부 클래스들은 el에 제공되어 있음. 해당 파일을 다운받아 완성하시오. 완성해야 하는 클래스는 `VehicleFactory`, `TestProgram`이다.

3.1 Vehicle 클래스

- 추상 클래스임
- 열거형 `Color`를 다음과 같이 정의함

```
public enum Color {UNPAINTED, BLUE, BLACK, PERLWHITE, WHITE, SILVER, GRAY, RED};
```

- 구성요소
  - 차량 색: `Vehicle.Color`
  - 차량 설명: `String`
- 연산
  - 생성자: `public Vehicle()`
  - 생성자: `public Vehicle(String description)`
  - `public void paint(Vehicle.Color color)`
    - 사후조건: 주어진 색으로 차량 색을 도색함. 실제로는 멤버변수를 주어진 값으로 설정함
  - `public abstract int cost()`
  - `public String toString()`
    - 사후조건: 다음과 같은 형태의 문자열을 반환함
      차량정보, 차량색, 가격

3.2 Car 클래스

- `Vehicle` 클래스를 상속받아 정의되며, 이 클래스도 추상 클래스임. 일반 승용 차량을 나타내는 클래스임
- 연산
  - 생성자: `public Car(String description)`

3.3 OffRoad 클래스

- `Vehicle` 클래스를 상속받아 정의되며, 이 클래스도 추상 클래스임. OffRoad 타입의 차량을 나타내는 클래스임
- 연산
  - 생성자: `public OffRoad(String description)`

3.4 Saloon 클래스

- `Car` 클래스를 상속받아 정의됨. 대형 승용 차량을 나타내는 클래스임
- 연산
  - 생성자: `public Saloon(String description)`
  - `public int cost()`
    - 사후조건: 이 종류의 차량 가격을 반환함. 만원 단위이며, 각자 알아서 값을 반환하면 됨. 수업시간에 다룬 커피 예제와 동일한 방식임

3.5 Compact 클래스

- `Car` 클래스를 상속받아 정의됨. 중형 승용 차량을 나타내는 클래스임
- 연산
  - 생성자: `public Compact(String description)`
  - `public int cost()`

3.6 Sports 클래스

- `Car` 클래스를 상속받아 정의됨. 스포츠 타입의 차량을 나타내는 클래스임
- 연산
  - 생성자: `public Sports(String description)`
  - `public int cost()`

3.7 OriginalSUV 클래스

- `OffRoad` 클래스를 상속받아 정의됨. 전형적인 SUV 차량을 나타내는 클래스임
- 연산
  - 생성자: `public OriginalSUV(String description)`
  - `public int cost()`

3.8 CrossoverSUV 클래스

- `OffRoad` 클래스를 상속받아 정의됨. Crossover SUV 차량을 나타내는 클래스임
- 연산
  - 생성자: `public CrossoverSUV(String description)`
  - `public int cost()`

3.9 VehicleFactory 클래스

- 추상 클래스로 고객에게 추천하고 싶은 차량을 만들어주는 클래스임
- 열거형 `DrivingStyle`를 다음과 같이 정의함

```
public enum DrivingStyle {ECONOMICAL, MIDRANGE, LUXURY, POWERFUL};
```

- 연산
  - `public final Vehicle build(DrivingStyle style, Vehicle.Color color)`
    - 사후조건: 주어진 차량 스타일에 맞는 차량을 `selectVehicle` 메소드를 이용하여 생성하고, 그 차량을 도색하여 반환함
  - `protected abstract Vehicle selectVehicle(DrivingStyle style)`
    - 사후조건: 주어진 차량 스타일에 맞는 차량을 생성함. Factory 메소드임

3.10 CarFactory 클래스

- `VehicleFactory`를 상속받아 정의됨. 일반 승용 차량을 생성할 때 사용됨
- 연산
  - `protected Vehicle selectVehicle(VehicleFactory.DrivingStyle style)`
    - 사후조건: 주어진 차량 스타일에 맞는 승용 차량을 생성함. Factory 메소드임

3.11 OffRoadFactory 클래스

- `VehicleFactory`를 상속받아 정의됨. OffRoad 종류의 차량을 생성할 때 사용됨
- 연산
  - `protected Vehicle selectVehicle(VehicleFactory.DrivingStyle style)`
    - 사후조건: 주어진 차량 스타일에 맞는 OffRoad 차량을 생성함. Factory 메소드임

3.12 테스트 프로그램

- 테스트 프로그램은 el.koretech.ac.kr에서 다운받을 수 있으며, `TestProgram` 클래스의 내부 클래스인 `ButtonListener`의 `actionPerformed` 메소드를 완성해야 함. 차량의 종류에 따라 `CarFactory` 또는 `OffRoadFactory`를 이용하여 차량을 생성하여야 함
- 두 가지 고민
  - `selectVehicle` 메소드가 왜 **protected**로 선언되었을까?
  - 차량의 색깔에 따라 추가비용이 발생할 경우 어떻게 수정해야 할까?

4. 숙제

- 테스트 프로그램 소개에 제시된 두 가지 고민 중 차량의 색깔에 따라 추가비용을 계산하도록 프로그램을 수정하시오.
- `PERLWHITE`만 10만원 추가비용이 발생하도록 프로그램을 수정하시오.

5. SerialNumberGenerator 실습

- 일련번호 생성기: 보통 생성되는 한 클래스의 인스턴스의 수를 유지하기 위해서는 **static** 멤버변수를 정의하여 생성자가 생성될 때마다 카운트를 증가하도록 하는 방법을 사용할 수 있다.

```
public class BankAccount{
    private static int count = 0;
    private int accountNumber;
    public BankAccount(){
        ++count;
        accountNumber = count;
    }
}
```

- 위 예처럼 사용하지 않고 일련번호를 생성하는 별도의 클래스를 정의하자. 하지만 이 클래스의 인스턴스를 자유롭게 생성할 수 있다면 각 객체에 고유한 일련번호를 부여하는 것이 어렵다. 따라서 이 클래스는 singleton으로 모델링되어야 한다.

```
public class BankAccount{
    private int accountNumber;
    public BankAccount(){
        accountNumber =  SerialNumberGenerator.getInstance().getNext();
    }
    public String toString(){
        return String.format("계좌번호: %d", accountNumber);
    }
}
```

- `SerialNumberGenerator` 클래스를 완성하시오. 이 때 내부 클래스를 이용한 구현방법을 사용하여 구현하시오.
- 테스트 프로그램

```
public class SerialNumberGeneratorTest {
    public static void main(String[] args) {
        System.out.println(SerialNumberGenerator.getInstance().getNext());
        System.out.println(SerialNumberGenerator.getInstance().getNext());
    }
}
```

6. Doubleton 실습

- Singleton은 보통 하나의 인스턴스만 생성하도록 제한한다. 이 실습에서는 두 개의 인스턴스만 생성되도록 제한하며, 각 인스턴스를 요청마다 번갈아 제공하는 클래스를 만들어본다.
- 이 실습에서 열거형을 이용하여 Doubleton을 만들어 보자.
- `Doubleton` 클래스를 완성하시오.

```
public enum Doubleton{
    FIRST, SECOND;
    public static Doubleton getInstance(){

    }
}
```

- 테스트 프로그램

```
public class DoubletonTest {
    public static void main(String[] args) {
        Doubleton first = Doubleton.getInstance();
        Doubleton second = Doubleton.getInstance();
        Doubleton third = Doubleton.getInstance();
        System.out.println(first);
        System.out.println(second);
        System.out.println(third);
    }
}
```

- 이와 같이 열거형을 이용하여 `Doubleton`을 구현하였을 때 문제점은?

7. Multiton 실습

- `Doubleton`에서는 두 개의 인스턴스만 생성하도록 제한하였다. 이 실습에서는 특정 개수의 인스턴스만 생성되도록 제한하며, 각 인스턴스마다 문자열로 된 키가 있어 사용자는 해당 키를 이용하여 인스턴스를 요청할 수 있도록 한다. 만약 제한된 개수 이상의 객체 생성을 요청하면 IllegalArgumentException 예외를 발생하도록 구현하시오.
- 이를 위해 자바 라이브러리 java.util.Hashmap 클래스를 사용한다.
- `Multiton` 클래스를 완성하시오.

```
public class Multiton{
    private static final HashMap<String,Multiton> registry
        = new HashMap<>();
    private static final int NUMBEROFINSTANCE = 3;
    public static Multiton getInstance(String key){

    }
}
```

- 테스트 프로그램

```
public class MultitonTest {
    public static void main(String[] args) {
        Multiton instance1 = Multiton.getInstance("first");
        Multiton instance2 = Multiton.getInstance("second");
        Multiton instance3 = Multiton.getInstance("third");
        Multiton instance4 = Multiton.getInstance("first");
        if(instance1==instance4) System.out.println("올바르게 동작중");
        Multiton instance5 = Multiton.getInstance("fourth");
    }
}
```

8. Multiton 실습 - 두 번째

- 이전 예제에서는 키를 이용하고 있음. 키를 이용하지 않고 정해진 개수에 객체만 생성하도록 제한하고, 각 생성된 객체를 공평하게 사용하도록 Multiton을 구현하시오.

```
public class Multiton{
    private static final int NUMBEROFINSTANCE = 3;
    private static final Multiton[] objectPool
        = new Multiton[NUMBEROFINSTANCE];
    public static Multiton getInstance(){

    }
}
```

- 테스트 프로그램

```
public class MultitonTest {
    public static void main(String[] args) {
        Multiton instance1 = Multiton.getInstance();
        Multiton instance2 = Multiton.getInstance();
        Multiton instance3 = Multiton.getInstance();
        Multiton instance4 = Multiton.getInstance();
        if(instance1==instance4) System.out.println("올바르게 동작중");
    }
}
```