Assignment 3: Prototype Pattern

What is Prototype Design Pattern?

Prototype is a creational design pattern that lets you copy existing objects (prototypes) without making your code dependent on their classes.

Intend

- ✓ Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- ✓ Co-opt one instance of a class for use as a breeder of all future instances.
- ✓ The new operator considered harmful.

Context: *Car*

In the context of car manufacturing, the prototype design pattern can be used to create new car models by copying and modifying existing car prototypes. This allows for a more efficient and cost-effective way of creating new cars, as the process of creating a new car from scratch can be time-consuming and costly.

For example, a car manufacturer may have a prototype of a sports car that they have been using to test different design features. They can then use this prototype as a starting point to create a new car model by making changes to the design and components of the prototype. This allows the manufacturer to quickly create new car models without having to go through the entire process of designing and building a car from scratch.

Problem

Say you have an object, and you want to create an exact copy of it. How would you do it? First, you have to **create a new object** of the same class. Then you have to **go through all the fields** of the original object and **copy their values** over to the new object.

Nice! But there's a catch. Not all objects can be copied that way because some of the **object's fields may be private** and **not visible from outside** of the object itself.

There's one more problem with the direct approach. Since you have to know the object's class to create a duplicate, your code becomes **dependent** on that class. If the extra dependency doesn't scare you, there's another catch. Sometimes you only know the interface that the object follows, but not its concrete class, when, for example, a parameter in a method accepts any objects that follow some interface.

Solution

The Prototype pattern delegates the **cloning** process to the actual objects that are being cloned. The pattern declares a **common interface** for all objects that support cloning. This interface lets you clone an object **without coupling** your code to the class of that object. The implementation of the **clone** method is very similar in all classes. The method creates an object of the current class and carries over all of the field values of the old object into the new one.

You can **even copy private fields** because most programming languages let objects access private fields of other objects that belong to the same class. An object that supports cloning is called a ***prototype***.

Implementation (Code)

Car.java

```
public class Car {
    private String brand;
    private String engine;
    private String color;
    private String model;

    public String getBrand(){
        return brand;
    }
    public void setBrand(String brand){
        this.brand = brand;
    }
    public String getEngine(){
        return engine;
    }
    public void setEngine(String engine){
        this.engine = engine;
    }
    public String getColor(){
        return color;
    }
    public void setColor(String color){
        this.color = color;
    }
    public String getModel(){
        return model;
    }
    public void setModel(String model){
        this.model = model;
    }
    public String toString(){
        return "Car [Model = " + model + ", Engine = " + engine + ", Color = " + color + ", Brand =
" + brand + "]\n";
    }
}
```

BuildCar.java

```
import java.util.*;
public class CarShop implements Cloneable {
   private String shopName;
   List<Car> cars = new ArrayList<>();
   public String getshopName(){
      return shopName;
   }
   public void setshopName(String shopName){
      this.shopName = shopName;
   }

   public List<Car> getCars(){
      return cars;
   }
   public void setCars(List<Car> getCars){
      this.cars = cars;
   }
   public void loadData(){
      for (int i=65 ; i<=68 ; i++){
         Car c = new Car();
         c.setModel(String.valueOf(i));
         c.setBrand("Brand"+i);
         int flag = 0;
         c.setEngine("Petrol");
         c.setColor("#FF" + i + flag + "D");
         if (i%2==0){
            flag = 1;
            c.setEngine("Diesel");
            c.setColor("#FF" + i + flag + "D");
         }
         getCars().add(c);
      }
   }
   public String toString(){
      return "BookShop [ ShopName = " + shopName + ", Cars = " + cars+ " ]";
   }
   protected CarShop clone() throws CloneNotSupportedException {
      CarShop shop =  new CarShop();
      for ( Car c : this.getCars() ){
         shop.getCars().add(c);
      }
      return shop;
   }
}
```

Showroom.java

```
public class Main {
   public static void main(String[] args) throws CloneNotSupportedException {
      CarShop cs1 = new CarShop();
      cs1.setshopName("Shop 1");
      cs1.loadData();

      CarShop cs2 = cs1.clone();
      cs1.getCars().remove(2);
      cs2.setshopName("Shop 2");

      System.out.println(cs1);
      System.out.println(cs2);
   }
}
```

Output

```
BookShop [ ShopName = Shop 1, Cars = [Car [Model = 65, Engine =
 Petrol, Color = #FF650D, Brand = Brand65]
, Car [Model = 66, Engine = Diesel, Color = #FF661D, Brand = Brand66]
, Car [Model = 68, Engine = Diesel, Color = #FF681D, Brand = Brand68]
] ]
BookShop [ ShopName = Shop 2, Cars = [Car [Model = 65, Engine =
 Petrol, Color = #FF650D, Brand = Brand65]
, Car [Model = 66, Engine = Diesel, Color = #FF661D, Brand = Brand66]
, Car [Model = 67, Engine = Petrol, Color = #FF670D, Brand = Brand67]
, Car [Model = 68, Engine = Diesel, Color = #FF681D, Brand = Brand68]
] ]
```

Applicability

1. Use the Prototype pattern when your code shouldn't depend on the concrete classes of objects that you need to copy.
2. Use the pattern when you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.