2. Builder Design Pattern

The Builder Design Pattern is a creational design pattern that is used for constructing complex objects step by step. It is particularly useful when an object needs to be created with numerous possible configurations or when the construction process is intricate. The Builder pattern provides a way to construct a variety of objects using the same construction code.

Here are the key components of the Builder Design Pattern:

1. Director:

- The Director is responsible for orchestrating the construction of the object. It works with the Builder interface to build the object step by step.

- The Director does not know the specific details of how the product is built but uses the Builder interface to delegate the construction steps.

2. Builder:

- The Builder interface defines the steps to construct the product.

- It typically includes methods for creating and setting parts of the product.

- Concrete builders implement this interface, providing specific implementations for constructing the various parts of the product.

3. ConcreteBuilder:

- ConcreteBuilder classes implement the Builder interface.

- They provide specific implementations for building the different parts of the product.

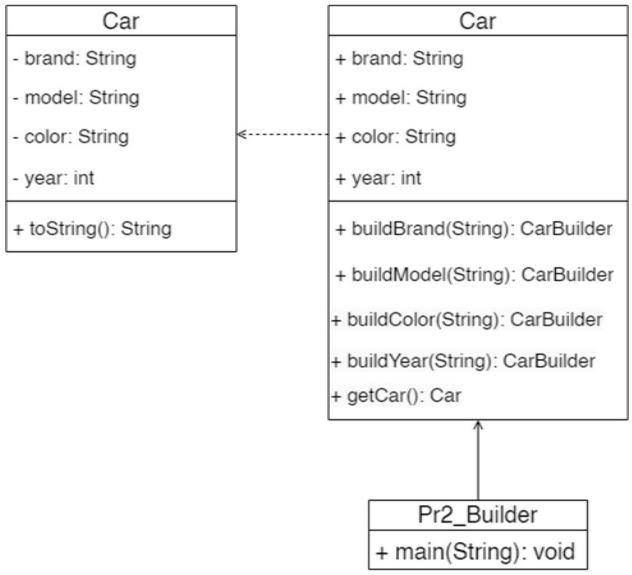
- A ConcreteBuilder is responsible for constructing a particular representation of the product.

4. Product:

- The Product is the complex object being constructed.

- It is the end result of the construction process.

UML Class Diagram:



Implementation:

```
/*
Builder pattern aims to “Separate the construction of a complex object from its
representation so that the same construction process can create different
representations.” It is used to construct a complex object step by step and the
final step will return the object. The process of constructing an object should
be generic so that it can be used to create different representations of the same
object.
*/

class Car {
    private String brand;
    private String model;
    private int year;
    private String color;

    // Constructor to initialize the Car with required attributes
    public Car(String brand, String model, int year, String color) {
        this.brand = brand;
        this.model = model;
        this.year = year;
```

```
        this.color = color;
    }

    // The toString() method returns the String representation of the object.
    // The toString() method returns the String representation of the object.
    // In this implementation, the Java compiler internally invokes the
toString()
    // method and prints the attributes of the car accordingly.

    public String toString() {
        return "\nCar{" +
                "brand='" + brand + '\'' +
                ", model='" + model + '\'' +
                ", year=" + year +
                ", color='" + color + '\'' +
                "}\n";
    }

}

// The builder class responsible for constructing a Car
class CarBuilder {

    // Temporary variables to store Car attributes during the building process
    private String brand;
    private String model;
    private int year;
    private String color;

    // Builder method for setting the brand attribute
    public CarBuilder buildBrand(String brand) {
        this.brand = brand;
        return this;
    }

    // Builder method for setting the model attribute
    public CarBuilder buildModel(String model) {
        this.model = model;
        return this;
    }

    // Builder method for setting the year attribute
    public CarBuilder buildYear(int year) {
        this.year = year;
        return this;
```

```
    }

    // Builder method for setting the color attribute
    public CarBuilder buildColor(String color) {
        this.color = color;
        return this;
    }

    // Final build method that constructs and returns the Car with specified
    // attributes
    public Car getCar() {
        return new Car(brand, model, year, color);
    }
}

// Main Driver Class
public class Pr2_Builder {

    // Main Method
    public static void main(String[] args) {

        Car porsche = new CarBuilder().buildBrand("Porsche").buildColor("Matte
Black").buildModel("911")
                .buildYear(1963).getCar();
        Car pagani = new
CarBuilder().buildBrand("Pagani").buildColor("White").buildModel("Huayra")
                .buildYear(2011).getCar();

        System.out.print(porsche);
        System.out.print(pagani);
        System.out.println();
    }
}
```

Output:

```
Car{brand='Porsche', model='911', year=1963, color='Matte Black'}

Car{brand='Pagani', model='Huayra', year=2011, color='White'}
```