**Builder Design Pattern**

The builder pattern is **a design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming**. The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

**When should we use the Builder Design Pattern?**

***Problem 1: If you have an object that takes too many parameters***

If you have an object that takes too many parameters, you started passing many "null" parameters in its constructor while creating this object.

// constructor like that

public Pizza(int size, boolean cheese, boolean olive, boolean pepperoni, boolean bacon, boolean mushroom, boolean mozarella) {...}

// creating instance

Pizza myFavoritePizza = new Pizza(20, true, true, null, null,null,null) ;

***Problem 2: If you have multiple constructors with different parameter combinations.***

If your object has multiple constructors and it is not clear which one to use, if you are confused about which minimum fields you need to fill in to create this object.

public class Pizza {

public Pizza(int size) { … }

public Pizza(int size, boolean cheese) { … }

public Pizza(int size, boolean cheese, boolean pepperoni) { … }

public Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { … }

public Pizza(int size, boolean cheese, boolean olive, boolean pepperoni, boolean bacon, boolean mushroom, boolean mozarella) { … }

}

***Problem 3: If you want the object to be immutable after creation***

In the following lines, the pizza object is created and then the properties of the object are changed. If the 2nd and 3rd lines are not written at this stage, a pizza object with no size and content is created, which is actually against the nature of the pizza object we designed.

Pizza myFavoritePizza = new Pizza();

myFavoritePizza.setSize(20);

myFavoritePizza.setCheese(true);

cook();

myFavoritePizza.setSize(10);

Moreover, the size can be changed even after the cook() method in line 4 is running. Worst of all, you do not know that the object has been changed.

If you want to create an object with your minimum parameter set and keep these fields as they were created, you can use the **Builder** design pattern.

In the Pizza example above, assuming that the pizza name and size are mandatory fields and we want to ensure that it is not changed later, we can implement it as follows.

public class Pizza {

private final int size;

private final String name;

private boolean cheese;

private boolean olive;

private boolean pepperoni;

private boolean bacon;

private boolean mushroom;

private boolean mozarella;

private Pizza(PizzaBuilder pizzaBuilder) {

this.size = pizzaBuilder.size;

this.name = pizzaBuilder.name;

this.cheese = pizzaBuilder.cheese;

this.olive = pizzaBuilder.olive;

this.pepperoni = pizzaBuilder.pepperoni;

this.bacon = pizzaBuilder.bacon;

this.mushroom = pizzaBuilder.mushroom;

this.mozarella = pizzaBuilder.mozarella;

}

public static class PizzaBuilder {

private final int size;

private final String name;

private boolean cheese;

private boolean olive;

private boolean pepperoni;

private boolean bacon;

private boolean mushroom;

private boolean mozarella;

public PizzaBuilder(int size, String name) {

this.size = size;

this.name = name;

}

public PizzaBuilder withOptionalCheese(boolean cheese) {

this.cheese = cheese;

return this;

}

public PizzaBuilder withOptionalOlive(boolean olive) {

this.olive = olive;

return this;

}

public PizzaBuilder withOptionalPepperoni(boolean pepperoni) {

this.pepperoni = pepperoni;

return this;

}

public PizzaBuilder withOptionalMushroom(boolean mushroom) {

this.mushroom = mushroom;

return this;

}

public PizzaBuilder withOptionalBacon(boolean bacon) {

this.bacon = bacon;

return this;

}

public PizzaBuilder withOptionalMozarella(boolean mozarella) {

this.mozarella = mozarella;

return this;

}

public Pizza buildPizza() {

return new Pizza(this);

}

}

}

**What Did We Do While Using the Builder Design Pattern?**

* By making the constructor of our pizza class **private**, we ensured that this object **is not called** from outside.
* We created a builder class (PizzaBuilder) and took the required fields (defined as final) "size" and "name" as parameters in the constructor of this class. Thus, a new object cannot be created without these two parameters.
* We wrote methods like withOptionalCheese() that fill optional parameters one by one for optional fields.
* Finally, we prepared the buildPizza() method and made it return the Pizza object out of the builder class.

After making the above arrangements, we can create the Pizza object as follows.

* While creating the builder object in the 4th line, the mandatory fields must be set and created.
* Optional parameters are created in the 5th and 6th lines.
* In the 7th line, a new Pizza object is generated with the prepared builder object.

public class PizzaMain {

public static void main(String[] args) {

PizzaBuilder myPizzaBuilder = new Pizza.PizzaBuilder(15, "Italiano");

myPizzaBuilder.withOptionalOlive(true);

myPizzaBuilder.withOptionalMozarella(true);

Pizza myFavoritePizza = myPizzaBuilder.buildPizza();

}

}