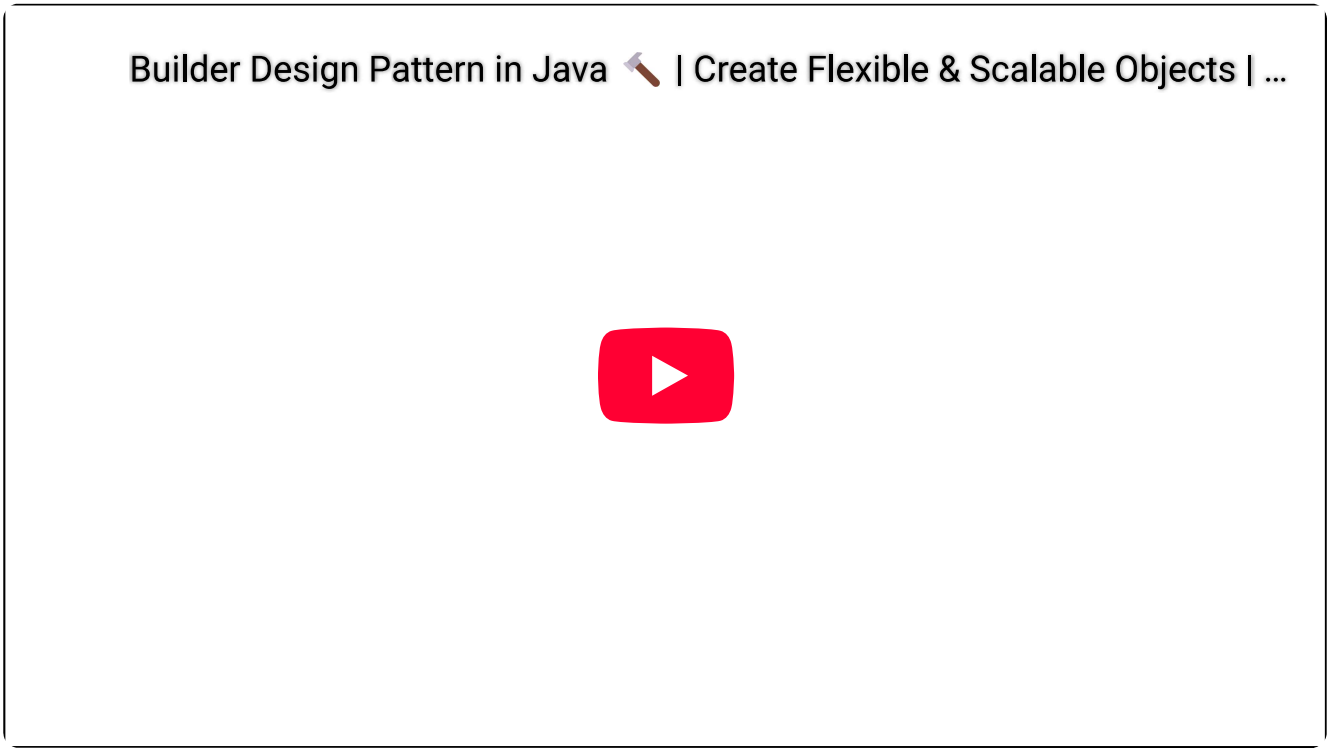
Search articles...

Builder Design Pattern



Topic Tags:

LLD System Design

1. What is the Builder Design Pattern? 🛠️

> *The Builder Design Pattern is a creational design pattern that helps in constructing complex objects step-by-step. This pattern is especially useful when an object has many attributes or optional fields, allowing you to create different configurations of the object easily and clearly.*

Instead of using a constructor with many parameters (which can be cumbersome and error-prone), the Builder pattern allows you to separate the construction process from the final object. It allows you to build an object piece by piece, and the final product is assembled only when you call the build() method.
In short, it's a way to construct an object in a flexible, readable, and maintainable manner.

2. The Traditional Way: The Problems We Face with Constructors ⚙️

Now, let's revisit the traditional approach where constructors are used to create objects. Consider we have a Car class with many attributes, some of which are optional.

Why Constructors?

Constructors were introduced to ensure that objects are created in a valid state right when they are instantiated. The constructor allows you to initialize an object with necessary values and guarantees that all required properties are set up right away.

Java

```java
public class Car {
  private String engine;
  private int wheels;
  private int seats;
  private String color;
  private boolean sunroof;
  private boolean navigationSystem;
  public Car(String engine, int wheels, int seats, String color,
      boolean sunroof, boolean navigationSystem) {
    this.engine = engine;
    this.wheels = wheels;
    this.seats = seats;
    this.color = color;
    this.sunroof = sunroof;
    this.navigationSystem = navigationSystem;
  }
}
```

While this approach works, as we can see, there are several drawbacks:

Problem #1: Passing Unnecessary Values 🛑

When you need to set optional attributes, such as sunroof or navigation system, you have to pass values for all parameters, even if they aren't necessary.

```
Car car = new Car("V8", 4, 5, "Red", false, false); // The client needs to pass `fa
```

Problem #2: Constructor Overloading and Huge Combinations ⚒️

If a car has many optional attributes, you end up with multiple constructors, each for different combinations of parameters. This results in code duplication and leads to messy and unmanageable code.

Java

```java
public class Car {
  public Car(String engine, int wheels, int seats, String color,
      boolean sunroof, boolean navigationSystem) {
    ...
  }
  public Car(String engine, int wheels, int seats, String color) {
    ...
  }
  public Car(String engine, int wheels, int seats) {
    ...
  }
}
```

The more attributes you add, the more combinations you have, leading to constructor bloat. Every time you add a new feature (like sports seats or a premium sound system), you need to add more constructors.

Problem #3: Lack of Readability 📉

The client code becomes hard to read because of unlabeled parameters in constructors. Here's a constructor that takes in multiple parameters, but it's hard to tell what each parameter represents:

Java

```java
Car car = new Car("V8", 4, 5, "Red", true, false);
```

This is difficult to understand at a glance. What do the true and false values represent? The client would have to refer to documentation to know which value represents the sunroof or navigation system, which isn't ideal.

3. Solving the Problem with Constructors: Follow-up Question by the Interviewer 🎤

An interviewer might ask:
• What if you need to add more optional attributes?
• What if the object creation needs to be more flexible, especially when dealing with large objects or more parameters?
The client realizes that as the number of attributes grows, constructors become harder to maintain. They quickly realize that constructor overloading doesn't scale well.

4. Shifting to the Builder Design Pattern 🔧

Why is it Named the "Builder" Pattern? 🏗️

The Builder Design Pattern is so named because it allows you to build an object step-by-step. The builder is responsible for assembling an object, and you control the process by setting attributes one by one. Instead of passing all parameters in a constructor, you pass only the ones you care about, and the builder takes care of the rest.

How Does It Work?

Let's see how we can implement the Builder Pattern to create a Car with flexibility and clarity.

Java

```java
public class Car {
  private String engine;
  private int wheels;
  private int seats;
  private String color;
  private boolean sunroof;
  private boolean navigationSystem;
  // Car constructor should be private, ensuring it's only created through
  // builder
  private Car(CarBuilder builder) {
    this.engine = builder.engine;
    this.wheels = builder.wheels;
    this.seats = builder.seats;
    this.color = builder.color;
    this.sunroof = builder.sunroof;
    this.navigationSystem = builder.navigationSystem;
  }
  // Getter methods for the fields
  public String getEngine() {
```

```
    return engine;
  }
  public int getWheels() {
    return wheels;
  }
  public int getSeats() {
    return seats;
  }
  public String getColor() {
    return color;
  }
  public boolean hasSunroof() {
    return sunroof;
  }
  public boolean hasNavigationSystem() {
    return navigationSystem;
  }
  @Override
  public String toString() {
    return "Car [engine=" + engine + ", wheels=" + wheels + ", seats=" + s
        + ", color=" + color + ", sunroof=" + sunroof
        + ", navigationSystem=" + navigationSystem + "]";
  }
  // CarBuilder nested class
  public static class CarBuilder {
    private String engine;
    private int wheels = 4; // Default value
    private int seats = 5; // Default value
    private String color = "Black"; // Default value
    private boolean sunroof = false; // Default value
    private boolean navigationSystem = false; // Default value
    // Builder methods to set attributes
    public CarBuilder setEngine(String engine) {
      this.engine = engine;
      return this;
    }
    public CarBuilder setWheels(int wheels) {
      this.wheels = wheels;
      return this;
    }
    public CarBuilder setSeats(int seats) {
```

```
    this.seats = seats;
    return this;
  }
  public CarBuilder setColor(String color) {
    this.color = color;
    return this;
  }
  public CarBuilder setSunroof(boolean sunroof) {
    this.sunroof = sunroof;
    return this;
  }
  public CarBuilder setNavigationSystem(boolean navigationSystem) {
    this.navigationSystem = navigationSystem;
    return this;
  }
  // Build method to create a Car object
  public Car build() {
    return new Car(
        this); // Return a new Car created using the builder's values
  }
 }
}
```

Client Code

Here's how the client would use the CarBuilder to create Car objects:

Java

```
public class Main {
  public static void main(String[] args) {
    // Creating a car using the Builder pattern
    Car.CarBuilder builder = new Car.CarBuilder();
    Car car1 = builder.setEngine("V8")
                  .setColor("Red")
                  .setSeats(5)
                  .setSunroof(true)
                  .build(); // The build method returns the final product
    System.out.println(car1);
    // Creating another car with different specifications
    Car car2 = builder.setEngine("V6")
```

```
13                      .setColor("Blue")
14                      .setSeats(4)
15                      .build(); // Sunroof and Navigation are default
16      System.out.println(car2);
17    }
18  }
```

Why is the CarBuilder Nested in the Car Class?

1. Encapsulation:
The CarBuilder is tightly related to the Car class, so it's grouped inside it. This makes it clear that the builder is for creating Car objects.

2. Access to Private Fields:

The CarBuilder can directly access private fields of Car (like engine, wheels) without needing getters/setters.

3. Logical Grouping:

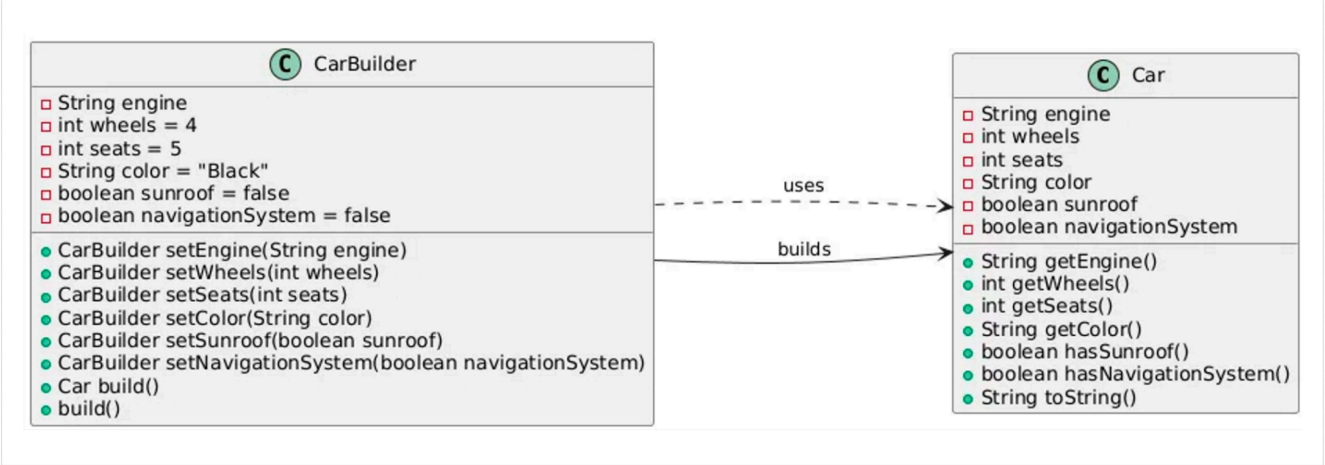
By nesting, we keep the CarBuilder and Car together, making the code cleaner and easier to understand.

Why is the CarBuilder Static?

1. No Need for Car Instance:
The CarBuilder doesn't need an instance of Car to create a new one, so it's made static. You can use the builder without creating a Car object first.

2. Efficiency:
It avoids unnecessary object creation. You don't need to instantiate Car just to use the builder.

3.Simpler Usage:
The static builder allows clients to create a Car object directly with Car.CarBuilder() without needing a separate builder instance.



Explanation of the Diagram:

1. Car Class:
○ Contains the attributes (engine, wheels, seats, color, sunroof, navigationSystem) and methods to retrieve them.
○ The Car constructor is private, ensuring it is created only through the CarBuilder.

2. **CarBuilder Class:**
○ Has the same attributes as Car, but they are mutable, and it allows setting these attributes via builder methods.
○ The build() method is used to create a Car object by passing the builder as a parameter to the Car constructor.

3. **Relationships:**

○ The CarBuilder is used by Car to construct a Car object, and the CarBuilder class returns a Car instance using the build() method.
○ The CarBuilder class is nested inside the Car class.

5. Solving the Interviewer's Follow-Up Questions with the Builder 🛠️

• What if we only want to set some attributes?
With the Builder pattern, you can only set the attributes you care about, and the rest of the attributes will take default values.
For example, if the client doesn't care about the sunroof or navigation system, they can skip those methods, and the car will be created with default values for those fields.

• What if I want to add new attributes in the future?
The Builder pattern makes this easy. You can simply add a new setter method to the builder class. No need to change the client code or the rest of the builder methods. For instance, you could add a "sportsSeats" feature later by adding one line in the CarBuilder class, and the client doesn't have to modify their existing code.

6. Real-life Use Cases of the Builder Pattern 🏗️

• **Building Complex Meals** 🍔:
Imagine creating a custom meal order (e.g., selecting burger size, toppings, drinks). The Builder Pattern lets you choose only the options you care about, making the process cleaner.

• **Creating Documents** 📑:
When creating complex documents (reports, articles), where sections might vary (like titles, images, or tables), the Builder Pattern helps assemble them step-by-step.

• **User Profile Creation** 👤: \
When building user profiles in apps, where there are multiple options (name, email, preferences), the Builder Pattern allows customization without cluttering the code.

🎉 Conclusion: Building Complex Objects the Smart Way 🔨

The Builder Design Pattern is an excellent solution for creating complex objects in a flexible, clear, and maintainable way. Unlike constructors, which can become messy and unmanageable with many parameters, the Builder pattern allows you to create objects step-by-step, setting only the attributes you care about. It provides default values, allows easy extensibility, and keeps the client code clean and understandable.

This pattern is ideal when you need to handle complex object creation without sacrificing flexibility or readability. Whether you're building cars, meals, or user profiles, the Builder Pattern ensures that objects are constructed in an organized, step-by-step manner.