Eduardo Pons

Plane Builder Pattern Implementation

Overview

This document outlines the implementation and benefits of the Builder pattern applied to the `Plane` class in the `Semana3.Builder` package. The Builder pattern is a design pattern used to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Problem Statement

In many scenarios, constructing complex objects with multiple parameters can become cumbersome, especially when there are optional parameters. For the `Plane` class, managing these parameters directly through a constructor can lead to a constructor with a large number of parameters, reducing code readability and maintainability.

Solution

The Builder pattern provides a flexible solution for constructing complex objects. The `Plane` class utilizes a nested static `PlaneBuilder` class to handle object construction. This approach allows for incremental and readable object creation, especially when dealing with optional parameters.

Code Implementation

Plane Class

The `Plane` class represents an aircraft with various attributes such as engine type, wingspan, seating capacity, commercial status, and autopilot capability.

java
Copiar código

```
public class Plane {
    private String engineType;
    private Double wingspan;
    private Integer capacity;
    private boolean isComercial;
    private boolean hasAutoPilot;

    private Plane(PlaneBuilder builder){
        this.engineType = builder.engineType;
        this.wingspan = builder.wingspan;
```

```
        this.capacity = builder.capacity;
        this.isComercial = builder.isComercial;
        this.hasAutoPilot = builder.hasAutoPilot;
    }

    @Override
    public String toString(){
        return "Plane [engineType=" + engineType + ", wingspan=" +
wingspan + ", seatingCapacity=" + capacity +
        ", is a comercial plane=" + isComercial + ", hasAutopilot="
+ hasAutoPilot + "]";
    }

    public static class PlaneBuilder{
        private String engineType;
        private Double wingspan;
        private Integer capacity;
        private boolean isComercial;
        private boolean hasAutoPilot;

        public PlaneBuilder setEngineType(String engineType) {
            this.engineType = engineType;
            return this;
        }

        public PlaneBuilder setWingspan(double wingspan) {
            this.wingspan = wingspan;
            return this;
        }

        public PlaneBuilder setCapacity(int capacity) {
            this.capacity = capacity;
            return this;
        }

        public PlaneBuilder setIsComercial(boolean isComercial){
            this.isComercial = isComercial;
            return this;
        }

        public PlaneBuilder setHasAutoPilot(boolean hasAutoPilot){
            this.hasAutoPilot = hasAutoPilot;
            return this;
        }

        public Plane build(){
            return new Plane(this);
        }
```

```
    }
}
```

Main Class

The `Main` class demonstrates the use of the `PlaneBuilder` to create instances of `Plane`.

java
Copiar código

```
public class Main {
    public static void main(String[] args) {
        Plane Boeing777 = new Plane.PlaneBuilder()
            .setEngineType("GE90")
            .setWingspan(64.8)
            .setCapacity(301)
            .setIsComercial(true)
            .setHasAutoPilot(true)
            .build();

        System.out.println(Boeing777);

        Plane privateJet = new Plane.PlaneBuilder()
        .setEngineType("PW800")
        .setWingspan(10)
        .setCapacity(20)
        .setHasAutoPilot(true)
        .build();

        System.out.println(privateJet);
    }
}
```

Benefits of Using the Builder Pattern

1. **Encapsulation of Construction Logic**: The Builder pattern encapsulates the construction logic of the `Plane` class, making it easier to manage and understand.
2. **Flexible Object Creation**: It allows for the creation of different types of `Plane` objects with varying configurations without requiring multiple constructors.
3. **Readability and Maintainability**: The builder approach improves code readability by clearly defining which parameters are being set. It also simplifies maintenance and future changes to the object's construction process.
4. **Immutability**: The `Plane` object is immutable after construction, ensuring that its state cannot be altered once created.
5. **Default Values**: The Builder pattern allows for setting default values for optional parameters, which simplifies object creation when certain attributes are not specified.

Eduardo Pons

Conclusion

The Builder pattern is an effective solution for managing the construction of complex objects like `Plane`. It enhances code clarity, flexibility, and maintainability while ensuring that the `Plane` objects are created in a consistent and controlled manner.