Assignment 16: Bridge design pattern

Niko Mehiläinen

Selected implementation:
https://jojozhuang.github.io/programming/design-pattern-bridge/

My GitHub project:
https://github.com/mehiis/DesignPatterns/tree/master/src/main/java/Structural/Bridge

Introduction

The Bridge design pattern is a structural design pattern that separates an abstraction from its implementation. This pattern helps in organizing code by separating different functionalities into individual class hierarchies, making the code more maintainable and flexible.

This design pattern is good for creating customizable class elements, where different classes can be combined without impacting the overall structure. The abstraction holds a reference to the implementation.

As an example I chose a project where a bridge design pattern was utilized. In this example project **Vehicle** works in the role of the *abstraction* and **Workshops** are the *implementation*. The idea is that we want to manufacture different kinds of vehicles in the workshops, but the manufacturing steps can vary. In this example the design pattern allows to mix and match different vehicles and workshops without changing their code.

New Functionality

I introduced a **Paint** workshop implementation, extending an already working implementation with more functionality. As an extra the **Paint** workshop class takes a color enum as a parameter, which allows the client to decide the color of the vehicle. I also modified the already existing **Vehicle** class to include a constructor that takes three **Workshop** parameters.

This lets us create different kinds of vehicles with varying amounts of workshops, while still allowing the **Vehicle** class to be used as intended. This demonstrates how simple and straightforward the extension is without touching the already existing implementation.

Implementation

My new painting functionality was made with **Workshop** implementation.

```java
public class Paint implements Workshop {  2 usages  new *
    private final PaintColors color;  2 usages

    public Paint(PaintColors color) {  2 usages  new *
        this.color = color;
    }

    @Override  6 usages  new *
    public void work() {
        System.out.print("painted in " + color + " color.\n");
    }
}
```

The client code with the originally created **Vehicle** classes and my new implementation.

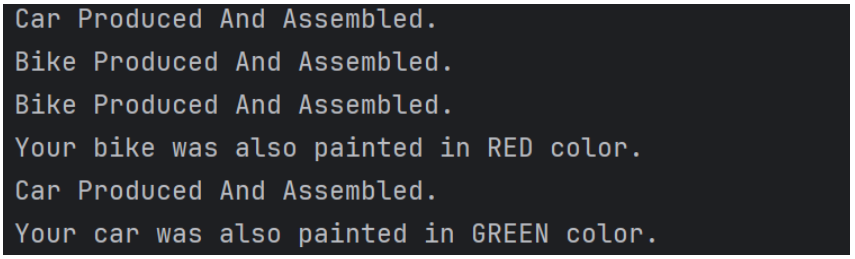
```java
public class Client {  new *
    public static void main(String[] args) {  new *
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();

        //MY CODE
        Vehicle vehicle3 = new Bike(new Produce(), new Assemble(), new Paint(PaintColors.RED));
        vehicle3.manufacture();
        Vehicle vehicle4 = new Car(new Produce(), new Assemble(), new Paint(PaintColors.GREEN));
        vehicle4.manufacture();
        //MY CODE ENDS
    }
}
```

The modified **Vehicle** class with an additional constructor that can take up to three workshops.

```java
public abstract class Vehicle {  6 usages  2 inheritors  new *
    protected Workshop workShop1;  4 usages
    protected Workshop workShop2;  4 usages
    protected Workshop workShop3; //MY CODE  5 usages

    protected Vehicle(Workshop workShop1, Workshop workShop2)  2 usages  new *
    {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    //MY CODE
    protected Vehicle(Workshop workShop1, Workshop workShop2, Workshop workShop3) {  2 usages  new *
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
        this.workShop3 = workShop3;
    }
    //MY CODE END


    abstract public void manufacture();  4 usages  2 implementations  new *
}
```

Verification

To ensure that the program works as intended and remains in its original form, I preserved original client code, while adding new vehicles to test the newly added functionality. In the following image we validate the functionality with a simple print statement to the IDE's console.



Conclusion

With the **bridge design pattern** we could easily and more importantly flexibly add a new workshop functionality. Without the design pattern, we would be forced to manually add the new paint functionality to each vehicle, but we would also face a problem if we didn't want to paint the vehicle. For example if the vehicles come with a default color that we do not want to change.

The bridge design pattern lets us easily extend our classes and create various objects with different attributes with minimal effort. This promotes loose coupling and avoids so called **combinatorial explosion**. A situation where the number of potential combinations of features grows exponentially.