Assignment 16: Bridge

Runzhou Zhu

1. Introduction

A bridge design pattern is a structural design pattern. It decouples abstraction from implementation so that they can change independently. The abstraction and the implementer could both be interface or abstract class.

Children of the abstraction can have different implementers. Since we can change the reference to the implementer in the abstraction, we are able to change the abstraction's implementer at run-time. Changes to the implementer do not affect client code.

I chose the Geeksforgeeks as the example implementation.
The line is: https://www.geeksforgeeks.org/bridge-design-pattern/

In the example, the abstraction is Vehicle and implementer is Workshop. The refined abstraction - Bike or Car can have different (or the same) concrete implementations - Assemble and Produce.

When we change the methods in Assemble or Produce, the Bike and Car class can still work normally.

2. New Functionality

I added a new vehicle - the EV, and 2 new workshops - the AutomatedProduceLine and InstantChargeStation.

I added this new functionality because the original example on the website did include the case that different abstraction could use different implementers. So, I added the AutomatedProduceLine to put the Assemble and Produce together and added a new implementer - the InstantChargeStation.

3. Implementation

```
// AutomatedProduceLine.java


public class AutomatedProduceLine implements Workshop {
    Workshop produce = new Produce();
    Workshop assemble = new Assemble();
```

```
    @Override
    public void work() {
        produce.work();
        assemble.work();
    }
}
```

```
// InstantChargeStation.java

public class InstantChargeStation implements Workshop {
    @Override
    public void work() {
        System.out.println("Charged");
    }
}
```

```
// EV.java

public class EV extends Vehicle {
    public EV(Workshop workshop1, Workshop workshop2) {
        super(workshop1, workshop2);
    }

    @Override
    public void manufacture() {
        System.out.println("EV");
        workshop1.work();
        workshop2.work();
    }
}
```

```
// BridgePattern.java

public class BridgePattern {
    public static void main(String[] args) {
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
```

```
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();

        // new functionality
        Vehicle vehicle3 = new EV(new AutomatedProduceLine(), new
InstantChargeStation());
        vehicle3.manufacture();
    }
}
```

4. Verification

```
C:\Users\Calorie\.jdks\openjdk-23.0.2\bin\java.exe
Car
Produced And Assemble.
Bike
Produced And Assemble.
EV
Produced And Assemble.
Charged

Process finished with exit code 0
```

5. Conclusion

In conclusion, the customization of the example is completed and works fine.

I decided to merge Assemble and Produce together and they work normally. I could also create a new class instead of using the old two.