Extending a Bridge pattern implementation

```
+----------------+                   +----------------+
|   Vehicle      |<----------------->|   Workshop     |
| <<abstract>>   |                   | <<interface>>  |
+----------------+                   +----------------+
        |                                     |
        |                                     |
   +----+-----+                        +------+-----+
   |          |                        |            |
+------+   +------+              +----------+   +----------+
|  Car |   | Bike |              |  Produce |   | Assemble |
+------+   +------+              +----------+   +----------+
```

1. Introduction

This implementation demonstrates the manufacturing of different types of Vehicles, without having to create multiple methods to each Object such as:

- **ProduceCar()** & **AssembleCar()**
- **ProduceCar()** & **AssembleBike()**

This way we can create Vehicle objects with different combinations of workshops, enabling flexibility and scalability without modifying existing code or creating redundant methods for each vehicle-workshop pair.

```java
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();
}
```

Link to the original implementation: https://www.geeksforgeeks.org/system-design/bridge-design-pattern/

2. New Functionality

I added a new Truck object in the motivation to just add Trucks. Also added the functionality to give the Vehicle objects a paint job.

```java
public static void main(String[] args) {
    // Original 2-workshop combinations
    Vehicle vehicle1 = new Car(new Produce(), new Assemble());
    vehicle1.manufacture();

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

```
        vehicle2.manufacture();

        // new vehicle Truck
        Vehicle vehicle3 = new Truck(new Produce(), new Assemble());
        vehicle3.manufacture();

        // only produce and paint a truck
        Vehicle truck2 = new Truck(new Produce(), new PaintJob());
        truck2.manufacture();

        // create a new painted car with all 3 workshops
        Vehicle paintedCar = new Car(new Produce(), new Assemble(), new
PaintJob());
        paintedCar.manufacture();

    }
```

3. Implementation

The new Structure

```
    +----------------+                   +----------------+
    |   Vehicle      |<----------------->|   Workshop     |
    | <<abstract>>   |                   | <<interface>>  |
    +----------------+                   +----------------+
            |                                     |
            |                                     |
       +----+-----+----------+             +------+---------+--------------+
       |          |          |             |                |              |
    +-------+  +-------+ +--------+     +-----------+  +-----------+  +-----------+
    |  Car  |  | Bike  | | Truck  |     |  Produce  |  | Assemble  |  | PaintJob  |
    +-------+  +-------+ +--------+     +-----------+  +-----------+  +-----------+
```

Added classes and new methods:

```
public class Truck extends Vehicle {

    public Truck(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    // New Constructor implemented to take in 3 workshops
    public Truck(Workshop workShop1, Workshop workShop2, Workshop workShop3) {
        super(workShop1, workShop2, workShop3);
    }

    @Override
    public void manufacture() {
        System.out.print("Truck ");
        workShop1.work();
        workShop2.work();
        if (workShop3 != null) {
            workShop3.work();
        }
        System.out.println();
    }
}
```

```
public class PaintJob implements Workshop {

    @Override
    public void work() {
        System.out.print(" And Painted");
    }
}
```

```
public abstract class Vehicle {

    protected final Workshop workShop1;
    protected final Workshop workShop2;
    protected final Workshop workShop3; // Added Third Workshop

    // Original constructor
    public Vehicle(Workshop workShop1, Workshop workShop2) {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
        this.workShop3 = null; // added this to set the third workshop as null if
its not initialized
    }

    // HERE is the new Added secondary Constructor to use all three workshops
    public Vehicle(Workshop workShop1, Workshop workShop2, Workshop workShop3) {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
        this.workShop3 = workShop3;
    }

    abstract public void manufacture();
}
```

4. Verification

Verification of the newly added features:

```
// Original functionality - Car with Produce and Assemble
Vehicle vehicle1 = new Car(new Produce(), new Assemble());
```

```
vehicle1.manufacture();
// Output: Car Produced And Assembled


// Added Functionality - Truck with Produce and Assemble
Vehicle vehicle2 = new Truck(new Produce(), new Assemble());
vehicle2.manufacture();
// Output: Truck Produced And Assembled

// With Both new Functionalities - Bike with Produce, Assemble and PaintJob
Vehicle vehicle3 = new Bike(new Produce(), new Assemble(), new PaintJob());
vehicle3.manufacture();
// Output: Bike Produced And Assembled And Painted
```

5. Conclusion

While this implementation may not be fully optimized, it effectively demonstrates the flexibility of the Bridge pattern. The pattern allows the system to grow and adapt by decoupling abstraction from implementation, making it easy to extend functionality and combine different objects without modifying existing code.

Link to the code in my Github: https://github.com/onnikiv/suunnittelumallit/tree/main/src/main/java/bridge