Introduction

The implementation to extend is taken from an GeeksForGeeks (https://www.geeksforgeeks.org/bridge-design-pattern/) article. It is used as an example to explain how the Bridge Design Pattern works.

Structure of the example

The example Bridge is between a Vehicle superclass and a Workshop interface.

Vehicle (abstract class)

- Constructor takes in workshops, contains an abstract method manufacture()
- Subclass Car
- Subclass Bike

Workshop (interface)

- Contains only the abstract method work()
- Subclass Produce
- Subclass Assemble

BridgePattern (class)

- Demonstrates how the whole system works

New Functionality

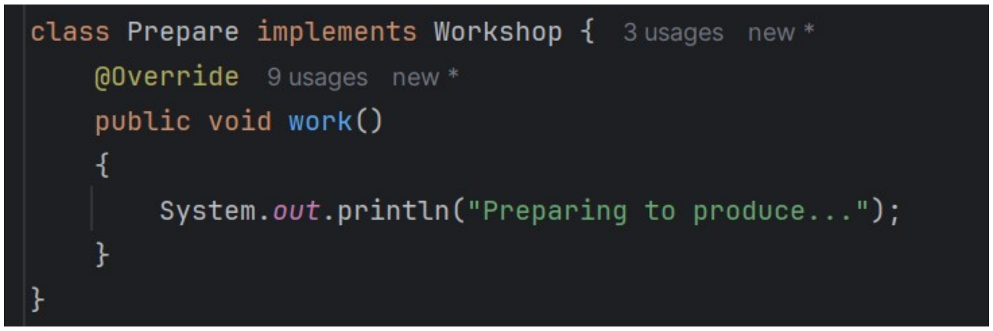
To cover both sides of the Bridge Design Pattern, there will be implementations of both the Vehicle class and the Workshop interface.

The Vehicle class will get a new subclass Motorcycle, that will work largely in the same way as the other Vehicle subclasses.

The new Workshop implementation will be Prepare, in which the the system prepares to manufacture the Vehicle. In practice it will work with a console log, like the other Workshop subclasses.

Implementation

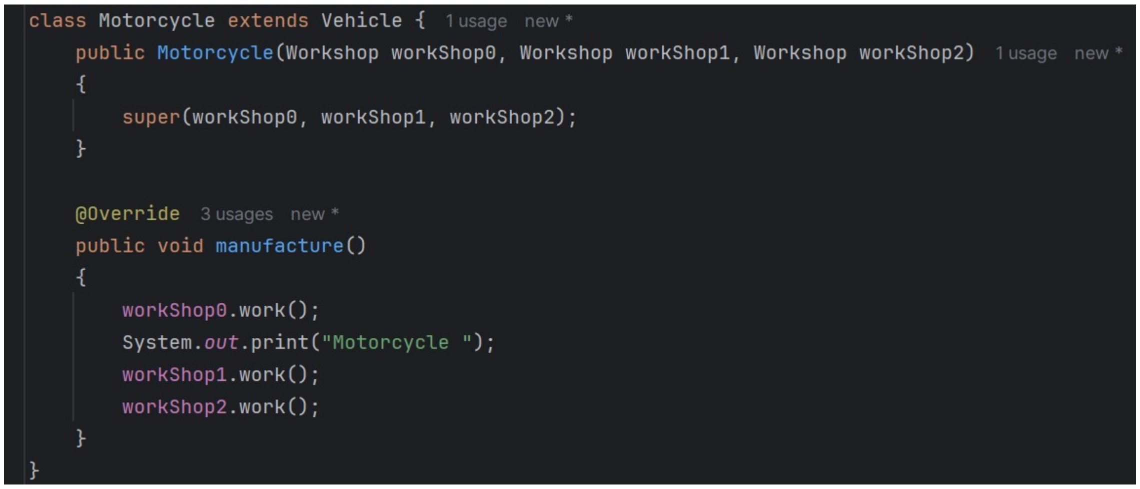
The new functionality is mainly covered in two classes named Prepare and Motorcycle. The vehicle class had to be modified a bit to accept this new Workshop implementation, and the manufacture() method of each subclass was changed to include it.



```java
class Prepare implements Workshop {   3 usages   new *
    @Override   9 usages   new *
    public void work()
    {
        System.out.println("Preparing to produce...");
    }
}
```

IntelliJ IDEA screenshot of the implementation of the Prepare class

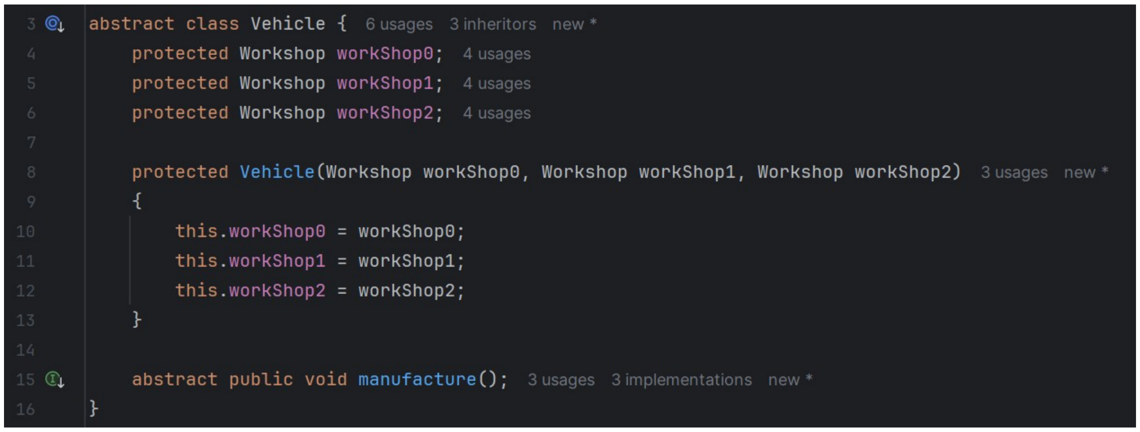
The Prepare class contains the work-method from the interface and simply console logs “Preparing to produce…” when called.



```java
class Motorcycle extends Vehicle {   1 usage   new *
    public Motorcycle(Workshop workShop0, Workshop workShop1, Workshop workShop2)   1 usage   new *
    {
        super(workShop0, workShop1, workShop2);
    }

    @Override   3 usages   new *
    public void manufacture()
    {
        workShop0.work();
        System.out.print("Motorcycle ");
        workShop1.work();
        workShop2.work();
    }
}
```

IntelliJ IDEA screenshot of the implementation of the Motorcycle class

From the implementation of the Motorcycle class, it can be seen how the rest of the Vehicle subclasses were modified too, seeing as they’re all the exact same. The line, workShop0.work() was added to the manufacture() method so the new Workshop gets to be used.



```java
abstract class Vehicle {   6 usages   3 inheritors   new *
    protected Workshop workShop0;   4 usages
    protected Workshop workShop1;   4 usages
    protected Workshop workShop2;   4 usages

    protected Vehicle(Workshop workShop0, Workshop workShop1, Workshop workShop2)   3 usages   new *
    {
        this.workShop0 = workShop0;
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();   3 usages   3 implementations   new *
}
```

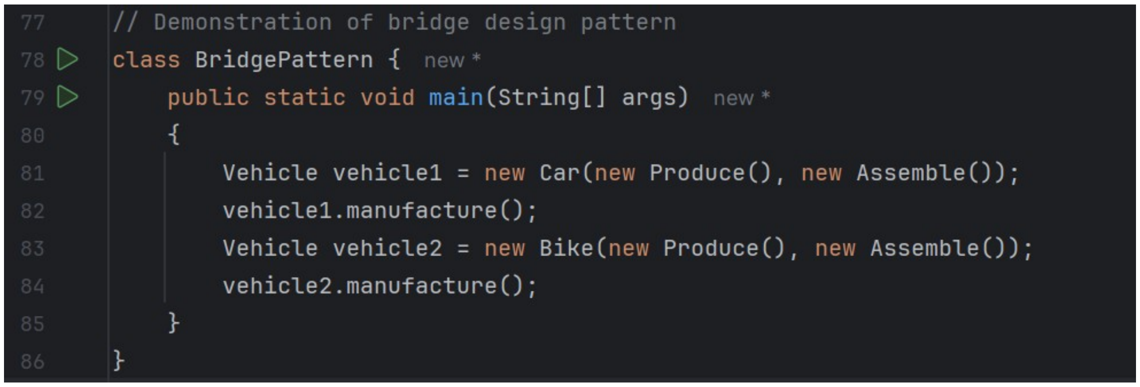
IntelliJ IDEA screenshot of the modified Vehicle class

The Vehicle class was modified to take in a third Workshop from the constructor. It is named as workShop0 to fit the other existing variable names, but in the example it is the Prepare class.

All of the subclass constructors that extend Vehicle had to be modified to reflect this.

Verification

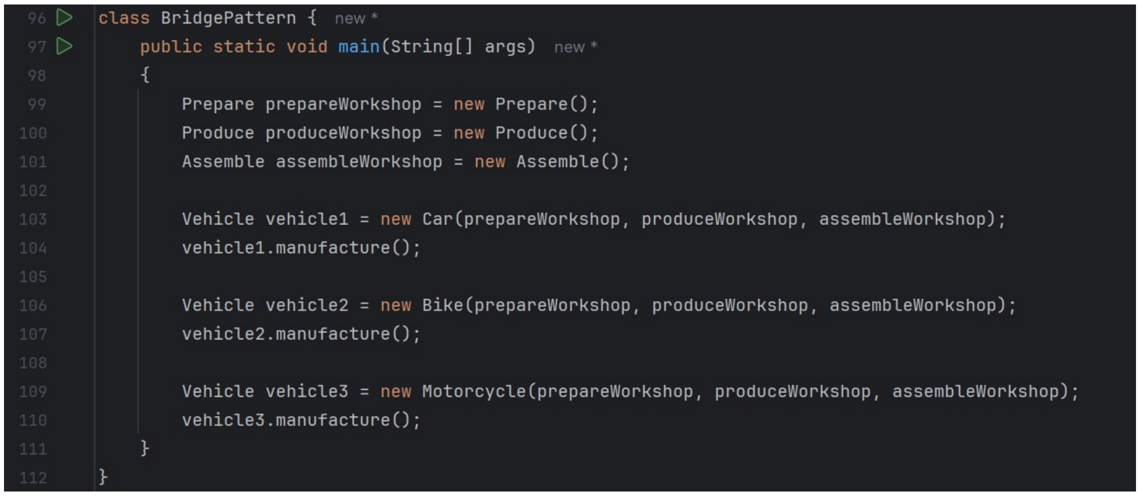
The demonstration is kept very similar to the original, so it is easy to tell what has changed.



```java
// Demonstration of bridge design pattern
class BridgePattern {  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();
    }
}
```

IntelliJ IDEA screenshot of the original demonstration

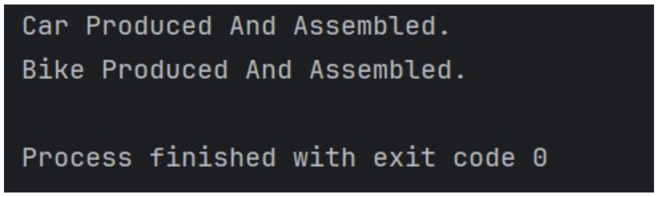
The original demonstration class simply creates two new Vehicles, Car and Bike and then calls the manufacture method on them.



```java
class BridgePattern {  new *
    public static void main(String[] args)  new *
    {
        Prepare prepareWorkshop = new Prepare();
        Produce produceWorkshop = new Produce();
        Assemble assembleWorkshop = new Assemble();

        Vehicle vehicle1 = new Car(prepareWorkshop, produceWorkshop, assembleWorkshop);
        vehicle1.manufacture();

        Vehicle vehicle2 = new Bike(prepareWorkshop, produceWorkshop, assembleWorkshop);
        vehicle2.manufacture();

        Vehicle vehicle3 = new Motorcycle(prepareWorkshop, produceWorkshop, assembleWorkshop);
        vehicle3.manufacture();
    }
}
```

IntelliJ IDEA screenshot of the slightly modified demonstration

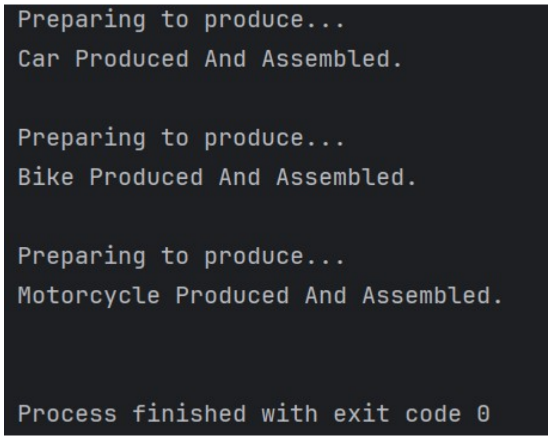
In the modified version, a new Vehicle object is added to demonstrate the Motorcycle class. Note how every object takes in three Workshop objects instead of two. To avoid creating multiple objects unnecessarily, the Workshops are refactored to be constructed only once.



```
Car Produced And Assembled.
Bike Produced And Assembled.

Process finished with exit code 0
```

Screenshot of the original demonstration output

In the original implementation, the vehicle type part (for ex. “Car”) is printed from the manufacture() method, “Produced” is printed from Produce Workshop, and “And Assembled” is printed from the Assemble Workshop.



```
Preparing to produce...
Car Produced And Assembled.

Preparing to produce...
Bike Produced And Assembled.

Preparing to produce...
Motorcycle Produced And Assembled.


Process finished with exit code 0
```

Screenshot of modified demonstration output

In the new demonstration, two things can be noticed: first, every call to manufacture -method prints an additional “Preparing to produce…” line from the Prepare Workshop. Also, a new Motorcycle type Vehicle is manufactured.

Conclusion

Two new classes were added: A new Vehicle subclass Motorcycle, and a new Workshop subclass Prepare. Adding a Vehicle class was easy and didn’t require modifying other classes.

Adding a new Workshop subclass (Prepare) needed slightly more effort, and required modifying the Vehicle class as well as all of its subclasses.

This could’ve been avoided, if for example instead of adding a new variable to the constructor, the Prepare class was called in the place of some other Workshop class. The work() method called by the manufacture() method would’ve worked anyway, since Prepare still implements the Workshop interface. In hindsight, perhaps replacing a Workshop method like this would’ve been more faithful to how the Bridge Design Pattern could usually be used.