Bridge Design Pattern

The Bridge design pattern allows you to separate the abstraction from the implementation.It is a structural design pattern.

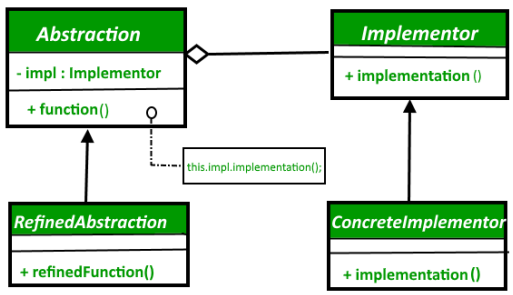
There are 2 parts in Bridge design pattern :

1. Abstraction
2. Implementation

This is a design mechanism that encapsulates an implementation class inside of an interface class.

- The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
- The abstraction is an interface or abstract class and the implementor is also an interface or abstract class.
- The abstraction contains a reference to the implementor. Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors. Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction's implementor at run-time. Changes to the implementor do not affect client code.
- It increases the loose coupling between class abstraction and it's implementation.

UML Diagram of Bridge Design Pattern



Elements of Bridge Design Pattern

- **Abstraction** – core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
- **Refined Abstraction** – Extends the abstraction takes the finer detail one level below. Hides the finer elements from implemetors.
- **Implementer** – It defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementer interface.
- **Concrete Implementation** – Implements the above implementer by providing concrete implementation.

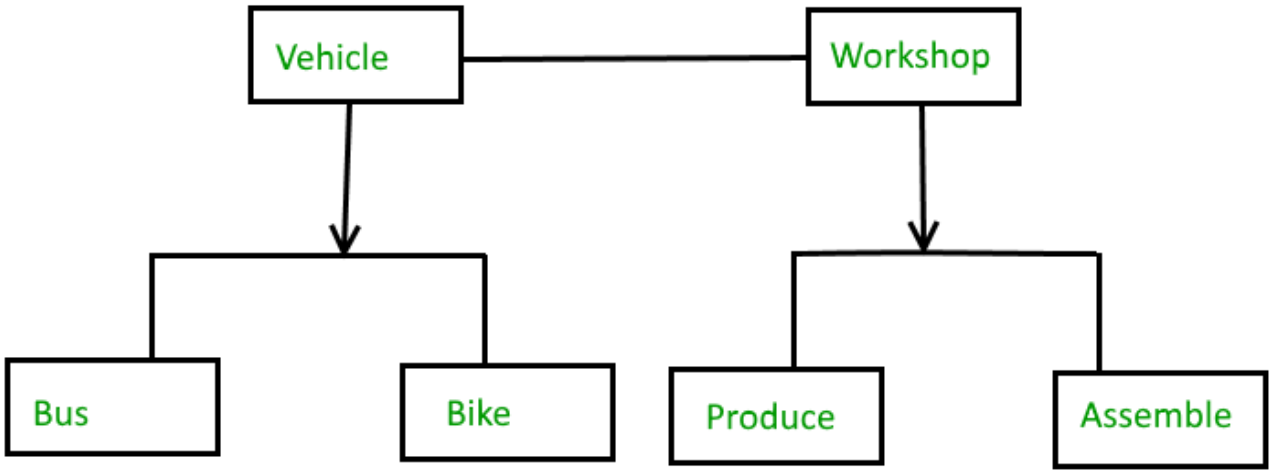
Lets see an Example of Bridge Design Pattern :

```
// Java code to demonstrate
// bridge design pattern

// abstraction in bridge pattern
abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;

    protected Vehicle(Workshop workShop1, Workshop workShop2)
    {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }

    abstract public void manufacture();
}

// Refine abstraction 1 in bridge pattern
class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture()
    {
```

```
        workShop2.work();
    }
}

// Refine abstraction 2 in bridge pattern
class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture()
    {
        System.out.print("Bike ");
        workShop1.work();
        workShop2.work();
    }
}

// Implementor for bridge pattern
interface Workshop
{
    abstract public void work();
}

// Concrete implementation 1 for bridge pattern
class Produce implements Workshop {
    @Override
    public void work()
    {
        System.out.print("Produced");
    }
}

// Concrete implementation 2 for bridge pattern
class Assemble implements Workshop {
    @Override
    public void work()
    {
        System.out.print(" And");
        System.out.println(" Assembled.");
    }
}

// Demonstration of bridge design pattern
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();
    }
}
```

Output :

```
Car Produced And Assembled.
```



Advantages

1. Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
2. It is used mainly for implementing platform independence feature.
3. It adds one more method level redirection to achieve the objective.
4. Publish abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
5. Use bridge pattern to run-time binding of the implementation.
6. Use bridge pattern to map orthogonal class hierarchies
7. Bridge is designed up-front to let the abstraction and the implementation vary independently.

This article is contributed by **Saket Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

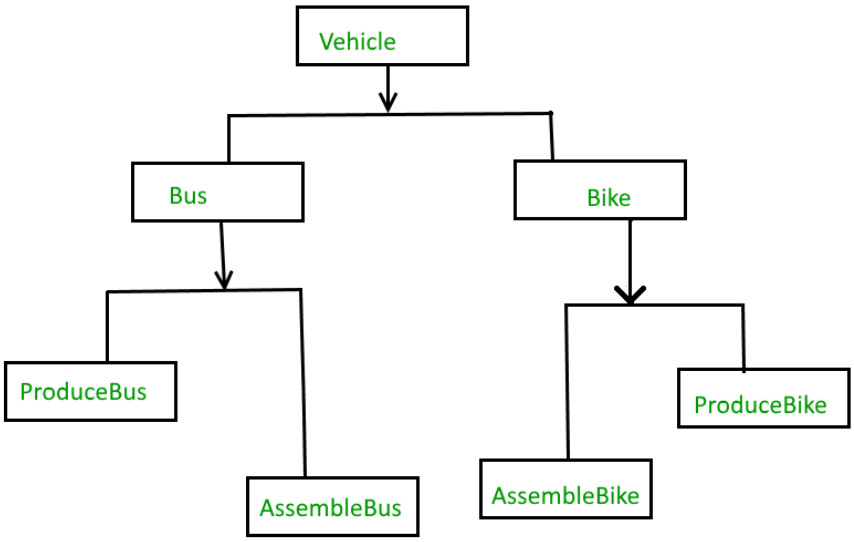
Here we're producing and assembling the two different vehicles using Bridge design pattern.

When we need bridge design pattern

The Bridge pattern is an application of the old advice, "prefer composition over inheritance". It becomes handy when you must subclass different times in ways that are orthogonal with one another.

For Example, the above example can also be done something like this :

Without Bridge Design Pattern



But the above solution has a problem. If you want to change the Bus class, then you may end up changing ProduceBus and AssembleBus as well and if the change is workshop specific then you may need to change the Bike class as well.

With Bridge Design Pattern

You can solve the above problem by decoupling the Vehicle and Workshop interfaces in the below manner.