**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 implementer is also an interface or abstract class.
* The abstraction contains a reference to the implementer. Children of the abstraction are referred to as refined abstractions, and children of the implementer are concrete 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.
* It increases the loose coupling between class abstraction and it’s implementation.

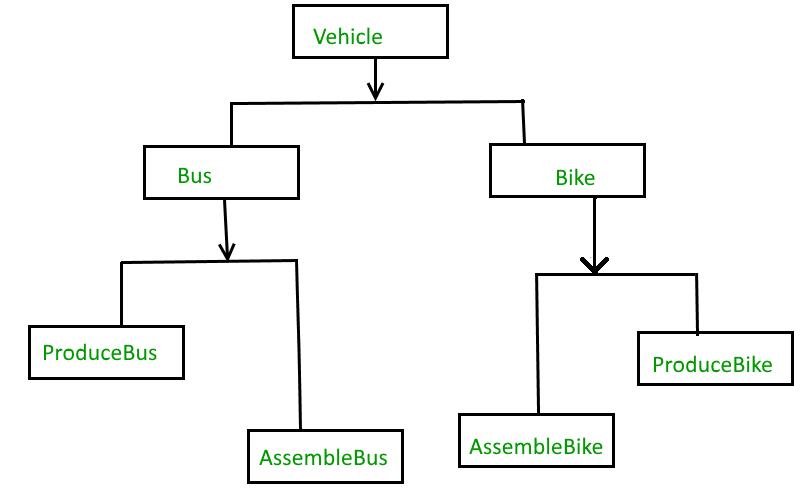
**Elements of Bridge Design Pattern**

* **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 implementers.
* **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 the Implementer interface.
* **Concrete Implementation** – Implements the above implementer by providing the concrete implementation.

**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 :

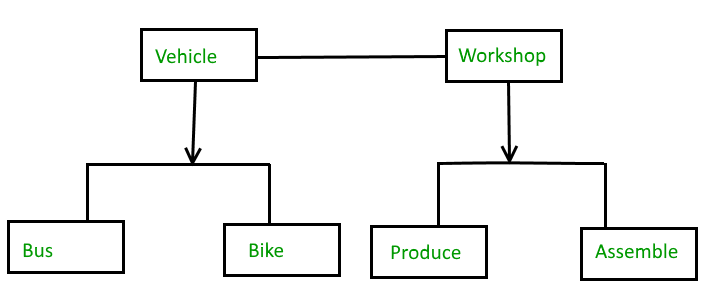
**Without Bridge Design Pattern**



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.



**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 features.
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.

