Bridge Design Pattern

Sean Kelley, Joey King, Aidan O'Brien

What is the Bridge Design Pattern?

Introduction

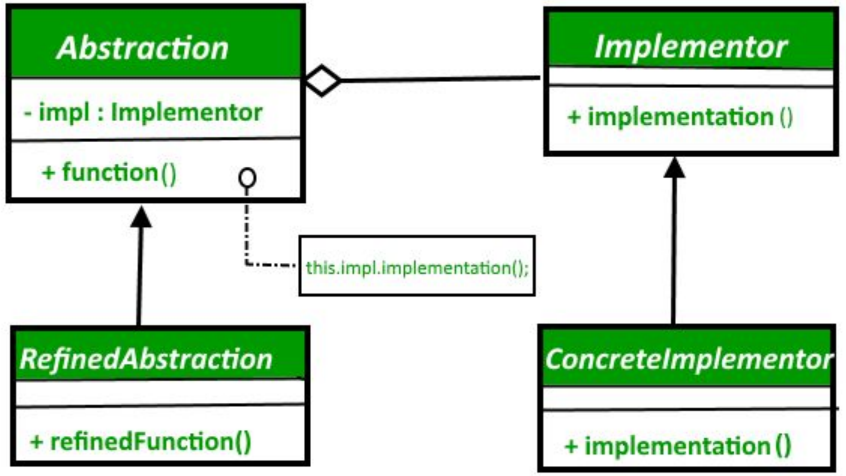
- The Bridge Design Pattern is a structural design pattern that allows for the abstraction and implementation to be developed separately from each other
- The client code can access the abstraction without the implementation
- The abstraction and implementer are both interfaces (abstract classes)
- The abstraction references the implementer
- Children of the abstraction are called “refined abstractions” and children of the implementer are called “concrete implementers”

Definitions

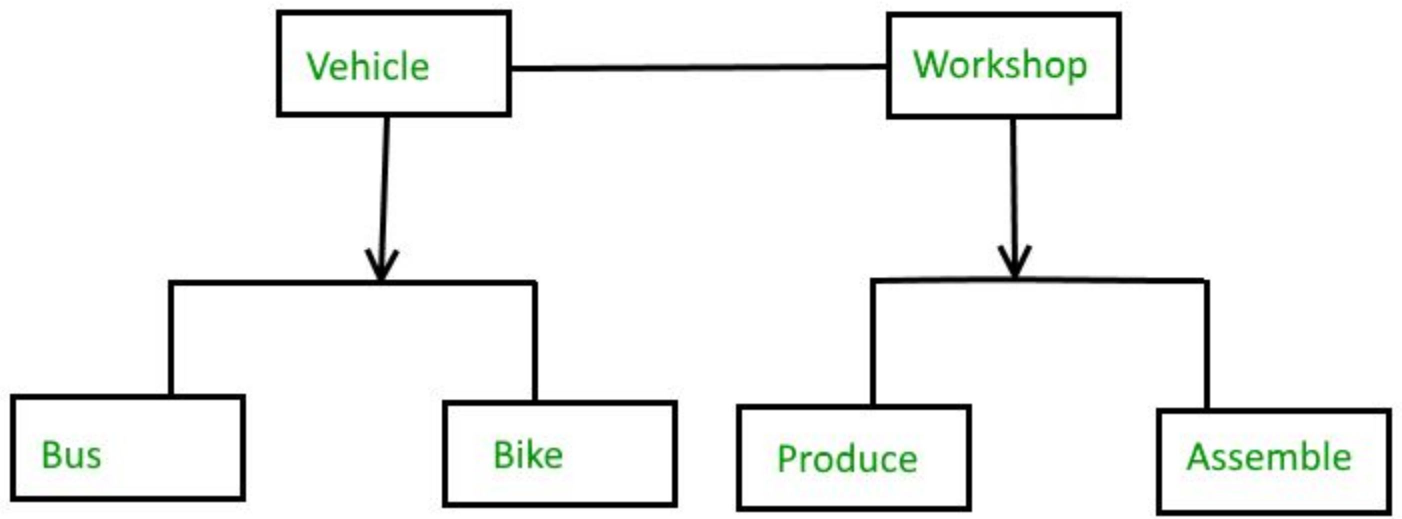
- Abstraction
  - The process of hiding certain details and showing only essential information to the user
- Refined Abstraction
  - Takes the finer details one level below and hides the finer elements from implementers
- Implementer
  - Defines the interface for implementation classes and does not need to correspond exactly with the abstraction interface.
- Concrete Implementation
  - Provides implementation for all of the methods in the implementer and leaves no room for unimplemented methods

Examples

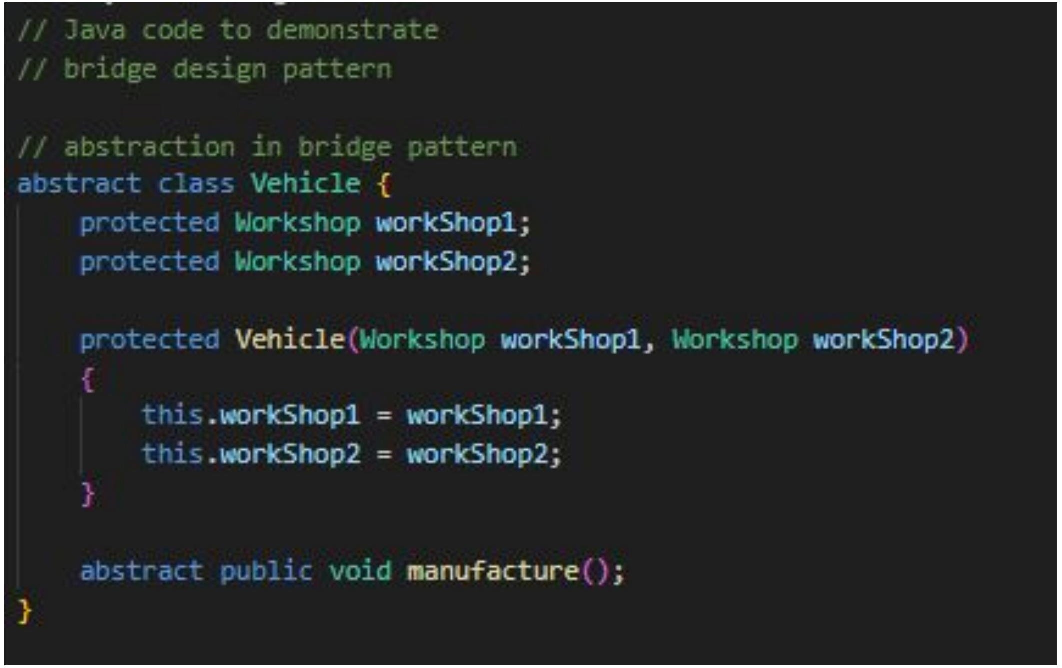
Example 1: UML Diagram



Example 2: Basic Implementation

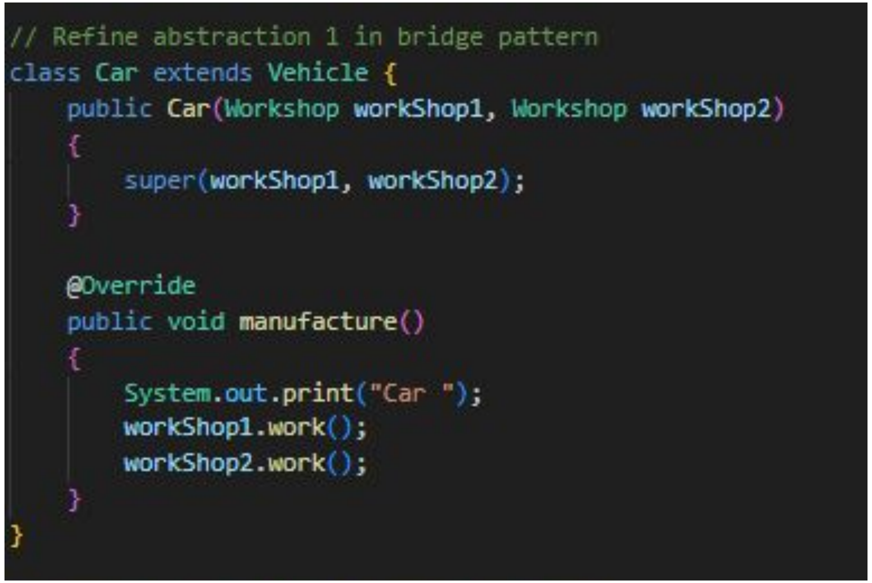


The Code

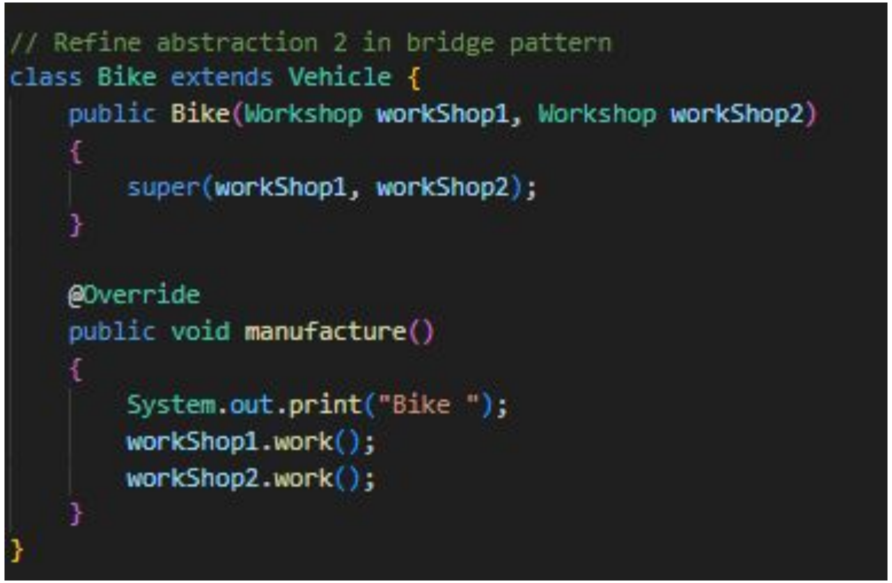


```java
// Java code to demonstrate
// bridge design pattern

// 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();
}
```

The Code (Cont.)

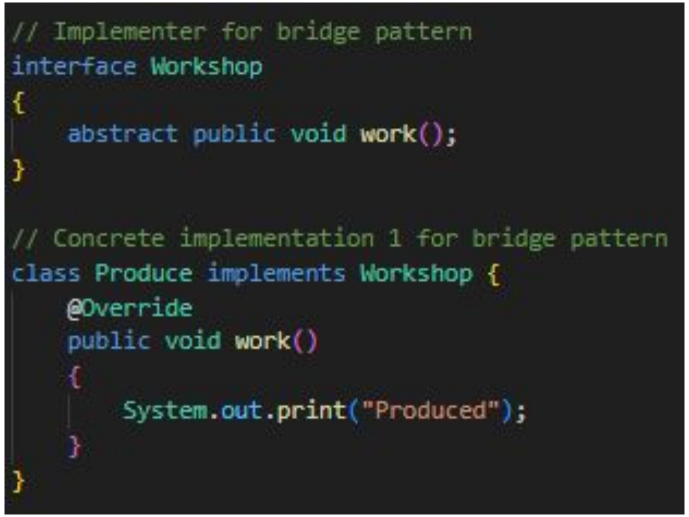


```
// Refine abstraction 1 in bridge pattern
class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

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

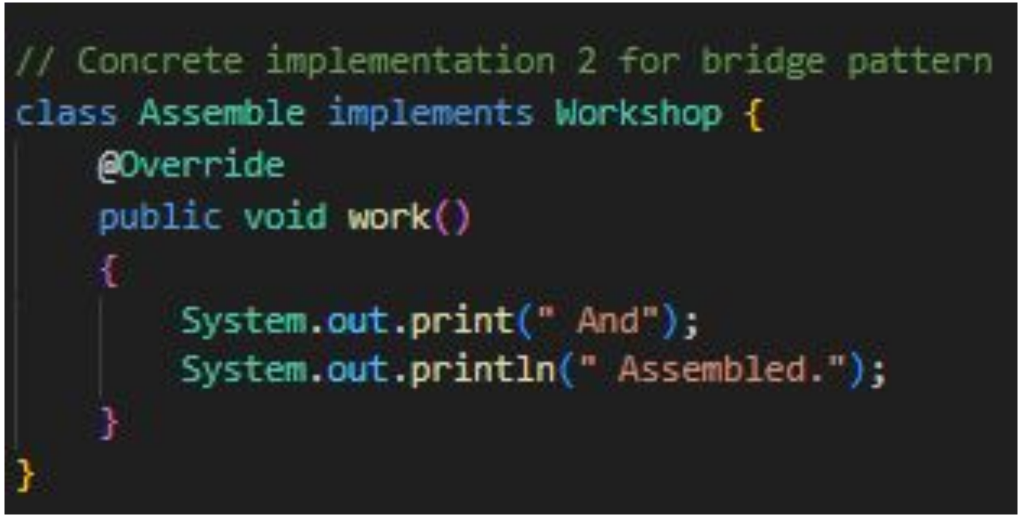


```
// Refine abstraction 2 in bridge pattern
class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2)
    {
        super(workShop1, workShop2);
    }

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

The Code (Cont.)

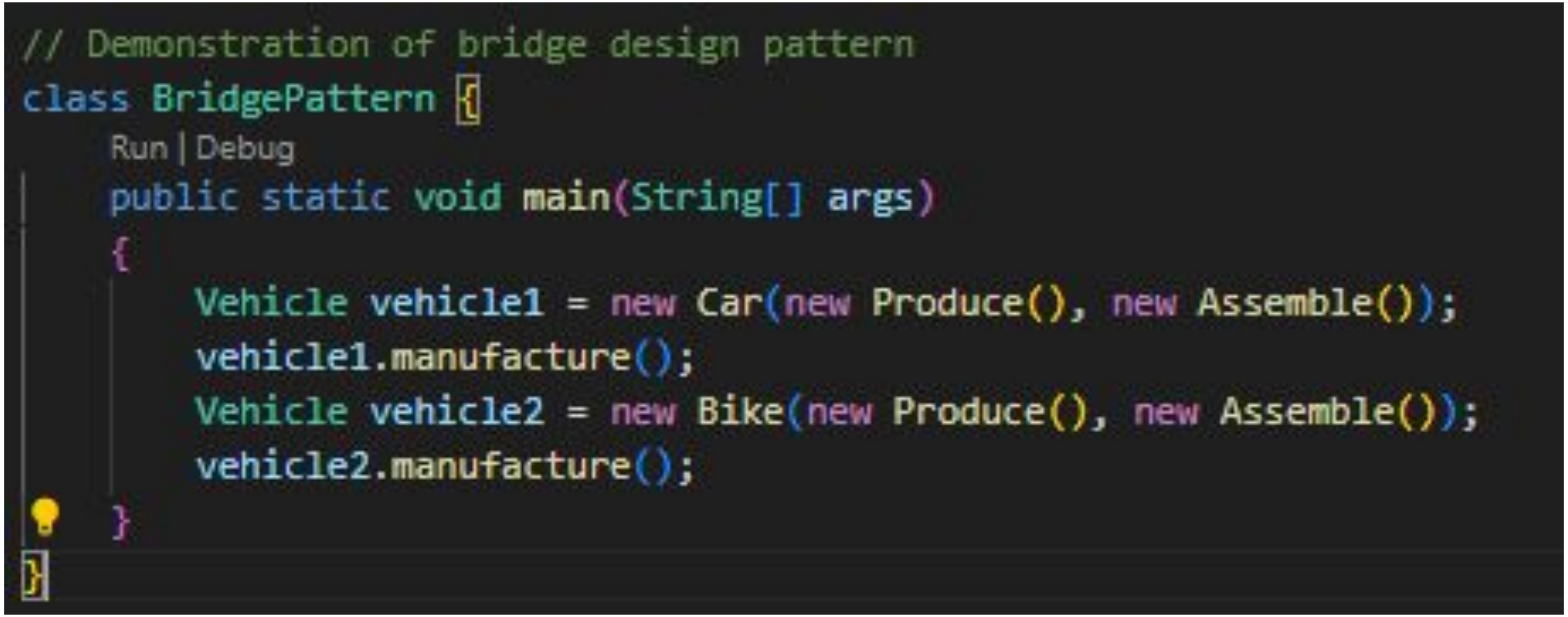


```java
// Implementer for bridge pattern
interface Workshop
{
    abstract public void work();
}

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



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

The Code(Cont.)



```
// Demonstration of bridge design pattern
class BridgePattern {
    Run | Debug
    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();
    }
}
```

Pros and Cons

Pros:

- Promotes Encapsulation
  - The Bridge pattern separates abstraction from its implementation, allowing changes in one without affecting the other, promoting flexibility and scalability within the codebase.
- Easy Maintenance
  - Maintenance becomes easy, since changes in one place often won’t affect other major areas.
- Reusability
  - Components can be reused across different implementations, leading to code that is more reusable and modular.

Cons:

- Requires Some Design Skills
    - Requires prior knowledge of design skills, especially a good understanding of abstraction and implementation hierarchies.
- Over Complications
    - In some circumstances, using the idea of Bridge Design can result in the overengineering of a problem, and cause unintended consequences in the final product.
- Increased Number of Classes
    - The pattern often requires the developer to create more classes, resulting in extra work.

Real World Applications

Real World Applications

- Remote Controlled Devices
  - A lot of devices have a remote (abstraction) that controls devices (implementation) like TVs, stereos, DVD players, etc.
  - The Bridge pattern is used to separate the remote from the devices they control, allowing you to add new types of remotes or devices without modifying existing code.
- Payment Processing
  - When dealing with online shopping websites such as Amazon or eBay, you might need to process payments through different payment gateways such as PayPal, Stripe, or Square
  - Bridge pattern can be used to separate the abstraction of payment processing from the implementation of specific payment gateway
  - Enables you to integrate new payment gateways or switch between existing ones without changing the core payment processing code

Real World Applications (Cont.)

- Audio Playback
  - In a music player application, you may need to support different audio file formats like MP3, WAV, or FLAC, as well as different audio output devices like speakers or headphones
  - You can separate the abstraction of audio playback from the implementation of audio file decoding and output Using the Bridge pattern, allowing you to add support for new formats or devices without rewriting any of the core code
- Video Streaming Platforms
  - Streaming platforms like Netflix or Hulu need to support various devices such as smart TVs, phones, computers, etc.
  - Bridge pattern can be applied to separate the abstraction of video playback from the implementation of specific devices, allows the platform to deliver a consistent viewing experience across different devices

Resources

Bridge Design Pattern - GeeksforGeeks

Concrete class in Java (prepbytes.com)

Java Abstraction (w3schools.com)