**TYPES OF PATTERNS:**

**1. Creational Patterns**

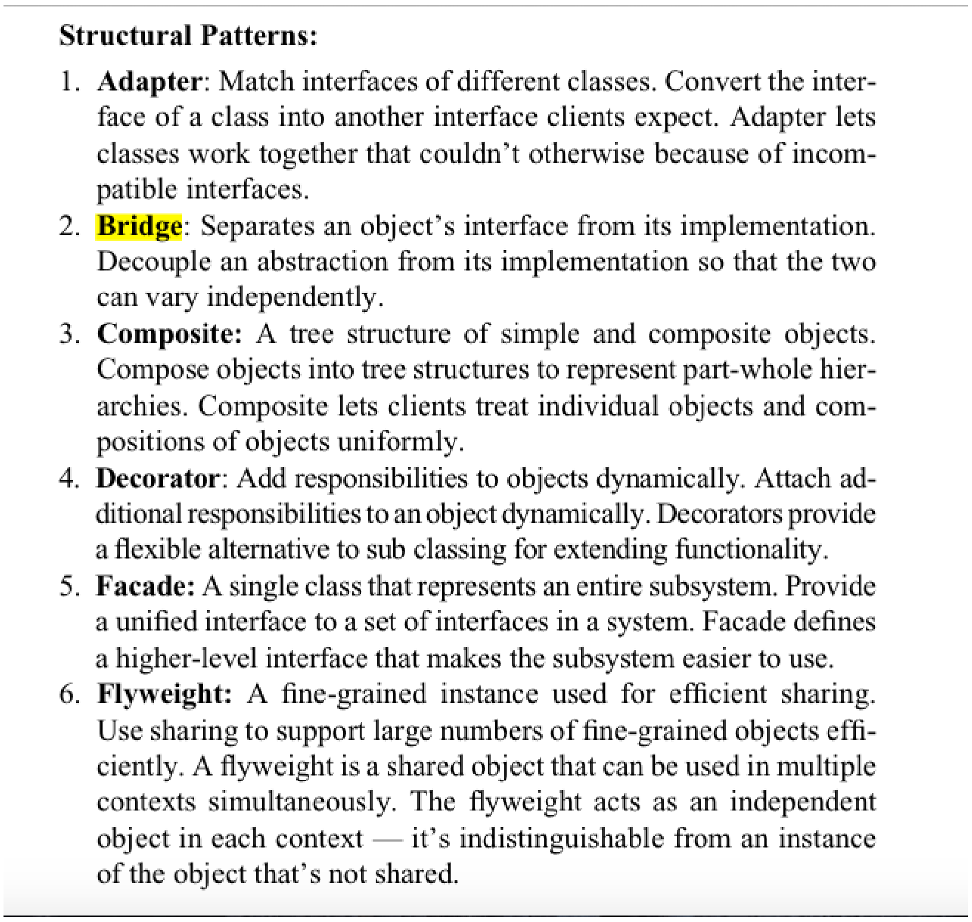
These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. (Factory, singleton,etc.)

**2 Structural Patterns**

These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. (Adapter, Bridge, Facade, Proxy,etc.)

**3 Behavioral Patterns**

These design patterns are specifically concerned with communication between objects. (Command, interpreter, observer, etc.)



**Structural Patterns:**

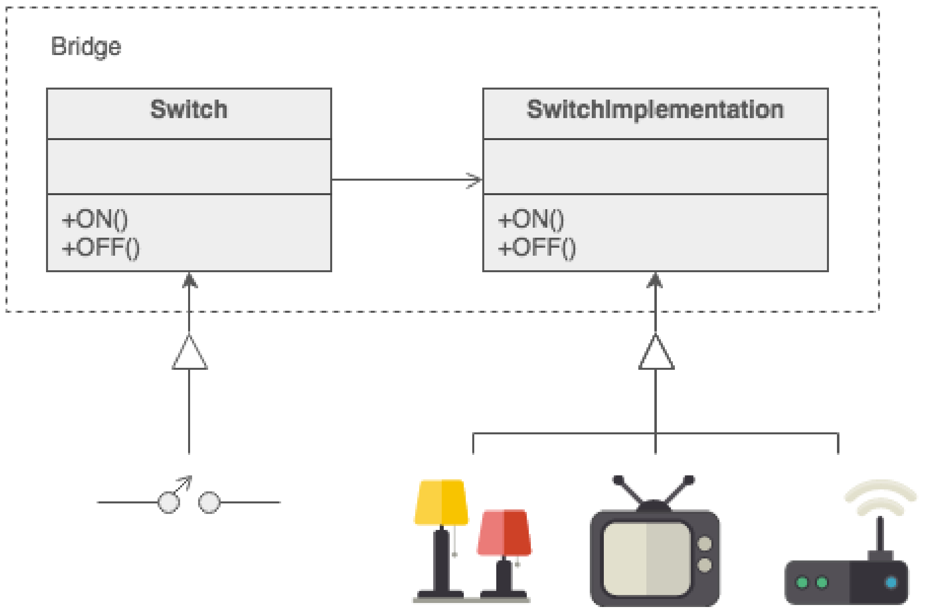
1. **Adapter**: Match interfaces of different classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
2. **Bridge**: Separates an object's interface from its implementation. Decouple an abstraction from its implementation so that the two can vary independently.
3. **Composite:** A tree structure of simple and composite objects. Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
4. **Decorator**: Add responsibilities to objects dynamically. Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.
5. **Facade:** A single class that represents an entire subsystem. Provide a unified interface to a set of interfaces in a system. Facade defines a higher-level interface that makes the subsystem easier to use.
6. **Flyweight:** A fine-grained instance used for efficient sharing. Use sharing to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context — it's indistinguishable from an instance of the object that's not shared.

**Bridge Pattern**

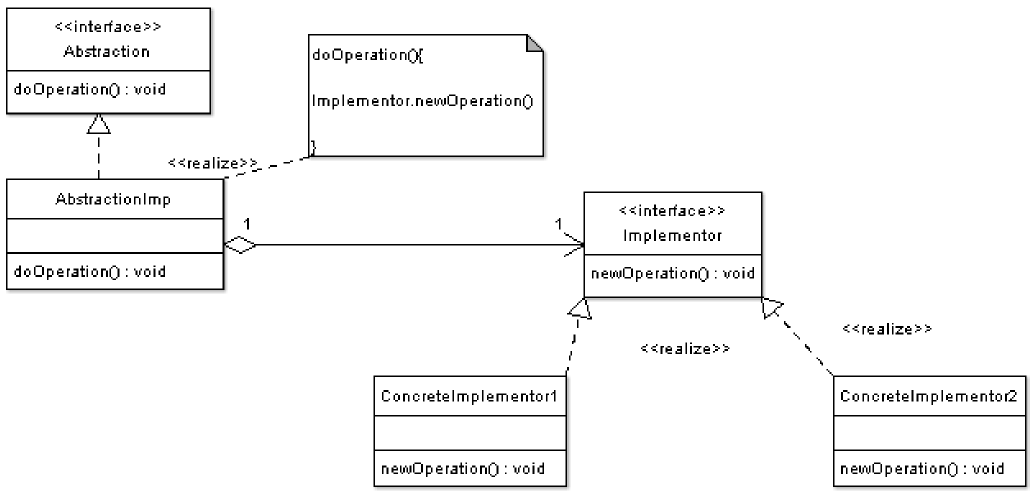
Bridge is used where we need to **decouple an abstraction from its implementation so that the two can vary independently.** This type of design pattern comes under **structural pattern** as this pattern **decouples implementation class and abstract class by providing a bridge structure between them.**

## Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



**STRUCTURE**



This pattern involves an interface which acts as a bridge which makes the f**unctionality of concrete classes independent from interface implementer classes.** Both types of classes can be altered structurally without affecting each other.

The participants classes in the bridge pattern are:

1.Abstraction - Abstraction defines abstraction interface.

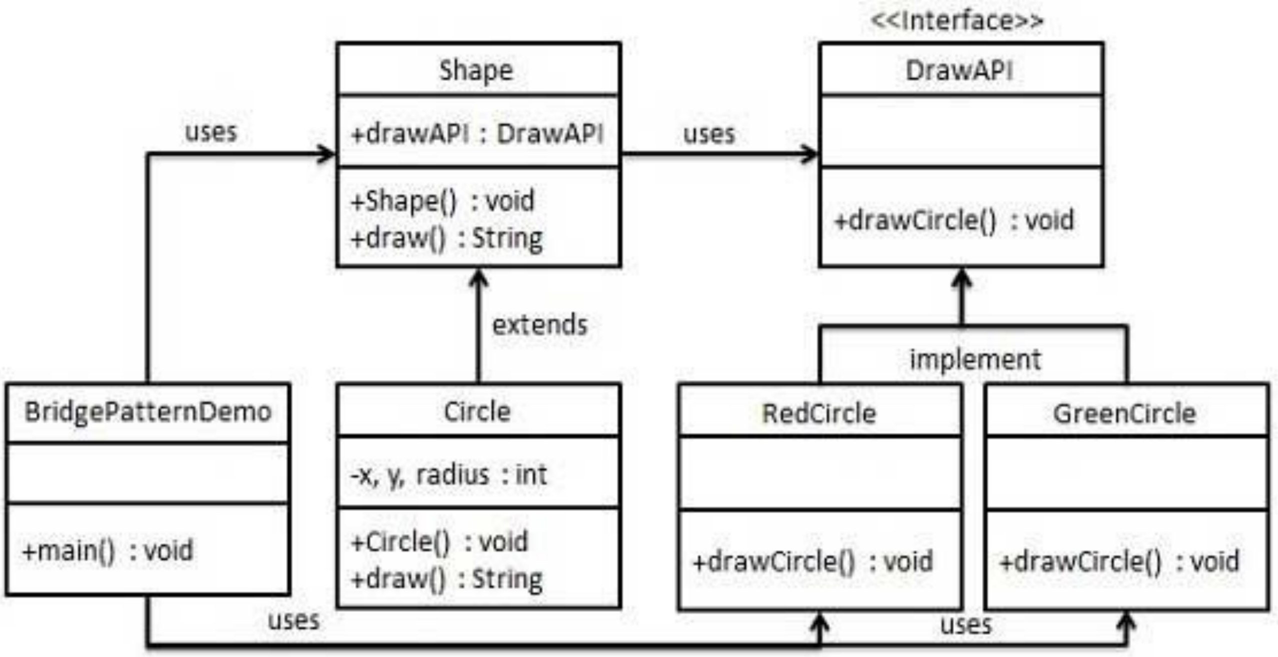
2.AbstractionImpl - Implements the abstraction interface using a reference to an object of type Implementor.

3.Implementor - Implementor defines the interface for implementation classes. This interface does not need to correspond directly to abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by Implementor interface.

4.ConcreteImplementor1, ConcreteImplementor2 - Implements the Implementor interface.
Description

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.
**Implementation**

We've an interface DrawAPI interface which is acting as a **bridge implementer** and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.



Bridge Pattern UML Diagram

Step 1

Create bridge implementer interface.

**DrawAPI.java**

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

Step 2

Create concrete bridge implementer classes implementing the DrawAPI interface.

**RedCircle.java**

```
public class RedCircle implements DrawAPI {

   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}
```

**GreenCircle.java**

```
public class GreenCircle implements DrawAPI {

   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: "
         + radius +", x: " +x+", "+ y +"]");
   }
}
```

Step 3

Create an abstract class Shape using the DrawAPI interface.

**Shape.java**

```
public abstract class Shape {
   protected DrawAPI drawAPI;
   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

Step 4

Create concrete class implementing the Shape interface.

**Circle.java**

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }

   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Step 5

Use the Shape and DrawAPI classes to draw different colored circles.

**BridgePatternDemo.java**

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

Step 6

Verify the output.

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[  color: green, radius: 10, x: 100, 100]
```

**Adapter vs. Bridge**

1. Adapter makes things work after they're designed; Bridge makes them work before they are.
2. Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.