BRIDGE DESIGN PATTERN

- The **Bridge Design Pattern** is a **structural pattern** that decouples an abstraction (what is to be done) from its implementation (how it is done).
- You usually use it when you have **two orthogonal hierarchies** (e.g., shape vs. color, message type vs. message channel, etc.).
- You want to avoid a deep inheritance hierarchy that multiplies combinations of features.
- You need to combine multiple variations of behavior or implementation at runtime.
- These two hierarchies are "bridged" via composition — not inheritance — allowing you to mix and match independently.

REAL-TIME USE CASES OF BRIDGE PATTERN

1. Message Sending System (Email, SMS, Push Notification)
   Abstraction: Message (e.g., TextMessage, AlertMessage, NotificationMessage)
   Implementation: MessageSender (e.g., EmailSender, SMSSender, PushSender)
2. Payment Gateway Integration
   Abstraction: Payment (e.g., OrderPayment, SubscriptionPayment)
   Implementation: PaymentGateway (e.g., PayPal, Stripe, Razorpay)
   Real-time: An **e-commerce app** allows different payments (UPI, Card, Wallet) via different providers.
3. Database Drivers (JDBC Example)
   Abstraction: JDBC API (Connection, Statement, ResultSet)
   Implementation: Specific Database Driver (MySQL, PostgreSQL, Oracle)
   Real-time: JDBC defines an **API abstraction**, and actual DB vendors provide **bridge implementations**.

THE PROBLEM: DRAWING SHAPES

- Imagine you're building a cross-platform graphics library. It supports rendering shapes like circles and rectangles using different rendering approaches:
  - Vector rendering – for scalable, resolution-independent output
  - Raster rendering – for pixel-based output

Now, you need to support:

- Drawing different shapes (e.g., Circle, Rectangle)
- Using different renderers (e.g., VectorRenderer, RasterRenderer)

NAIVE IMPLEMENTATION: SUBCLASS FOR EVERY COMBINATION

- You might start by creating a class hierarchy that looks like this:

```
abstract class Shape {
    public abstract void draw();
}
```

```
    class VectorCircle extends Shape {
    public void draw() {
        System.out.println("Drawing Circle as VECTORS");
    }
}
```

```
class RasterCircle extends Shape {
    public void draw() {
        System.out.println("Drawing Circle as PIXELS");
    }
}
```

```
class VectorRectangle extends Shape {
    public void draw() {
        System.out.println("Drawing Rectangle as VECTORS");
    }
}
```

```
class RasterRectangle extends Shape {
    public void draw() {
        System.out.println("Drawing Rectangle as PIXELS");
    }
}
```

```
public class App {
    public static void main(String[] args) {
        Shape s1 = new VectorCircle();
        Shape s2 = new RasterRectangle();

        s1.draw(); // Drawing Circle as VECTORS
        s2.draw(); // Drawing Rectangle as PIXELS
    }
}
```

WHY THIS QUICKLY BREAKS DOWN

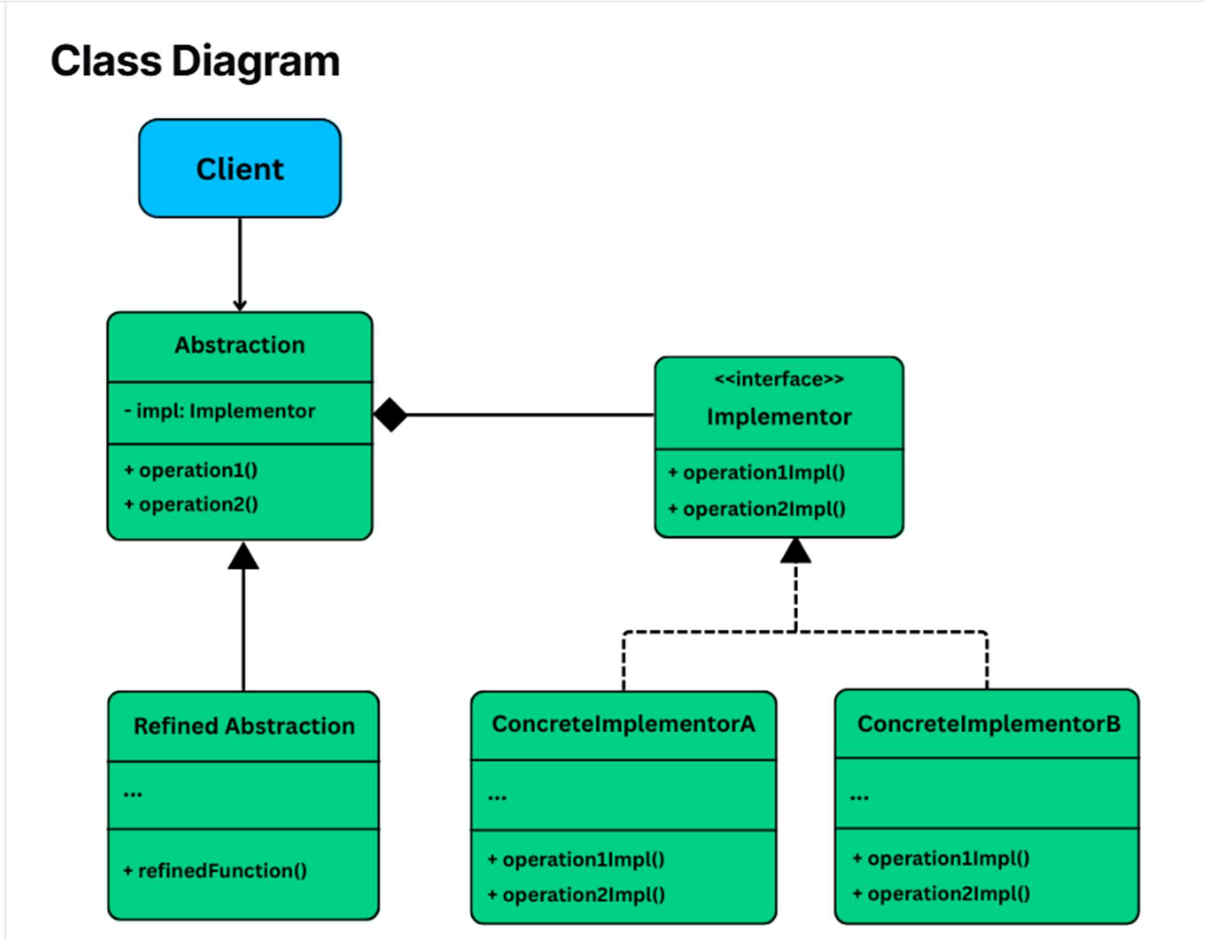
- **Class Explosion** - Every new combination of shape and rendering method requires a new subclass
- **Tight Coupling** - Each class ties together shape logic and rendering logic. You can’t reuse rendering behavior independently of the shape
- **Violates Open/Closed Principle -** If you want to support a new rendering engine, you must modify or recreate every shape for that renderer.

WHAT WE REALLY NEED

- Separates the abstraction (Shape) from its implementation (Renderer)
- Allows new renderers to be added without touching shape classes
- Enables new shapes to be added without modifying or duplicating renderer logic

WHAT IS THE BRIDGE PATTERN

- The Bridge Design Pattern lets you split a class into two separate hierarchies — one for the **abstraction** and another for the **implementation** — so that they can evolve independently.
- In the Bridge Pattern, **"abstraction has-a implementation**" — the abstraction delegates work to an implementor object.



- **Abstraction (e.g., Shape)** - The high-level interface that defines the abstraction's core behavior. It maintains a reference to an Implementor and delegates work to it.
- **RefinedAbstraction (e.g., Circle, Rectangle)** - A concrete subclass of Abstraction that adds additional behaviors or logic. It still relies on the implementor for actual execution.
- **Implementor (e.g., Renderer) -** An interface that declares the operations to be implemented by concrete implementors. These are the low-level operations**.**
- **ConcreteImplementors (e.g., VectorRenderer,RasterRenderer) -** Platform- or strategy-specific classes that implement the Implementor interface. They contain the actual logic for performing the delegated operations.

IMPLEMENTING BRIDGE

- Let's now implement the Bridge Pattern to decouple our **Shape abstraction** (e.g., Circle, Rectangle) from the **Renderer implementation** (e.g., VectorRenderer, RasterRenderer).

DEFINE THE IMPLEMENTOR INTERFACE (RENDERER)

```
interface Renderer {
   void renderCircle(float radius);
   void renderRectangle(float width, float height);
}
```

CREATE CONCRETE IMPLEMENTATIONS OF THE RENDERER

```
class VectorRenderer implements Renderer {
    @Override
    public void renderCircle(float radius) {
        System.out.println("Drawing a circle of radius " + radius + " using VECTOR rendering.");
    }

    @Override
    public void renderRectangle(float width, float height) {
        System.out.println("Drawing a rectangle " + width + "x" + height + " using VECTOR rendering.");
    }
}
```

- RasterRenderer

```
class RasterRenderer implements Renderer {
    @Override
    public void renderCircle(float radius) {
        System.out.println("Drawing pixels for a circle of radius " + radius + " (RASTER).");
    }

    @Override
    public void renderRectangle(float width, float height) {
        System.out.println("Drawing pixels for a rectangle " + width + "x" + height + " (RASTER).");
    }
}
```

DEFINE THE ABSTRACTION (SHAPE)

- This class holds a reference to the renderer and defines a general draw() method.

```
abstract class Shape {
    protected Renderer renderer;

    public Shape(Renderer renderer) {
        this.renderer = renderer;
    }

    public abstract void draw();
}
```

CREATE CONCRETE SHAPES

```
class Circle extends Shape {
    private final float radius;

    public Circle(Renderer renderer, float radius) {
        super(renderer);
        this.radius = radius;
    }

    @Override
    public void draw() {
        renderer.renderCircle(radius);
    }
}
```

```
class Rectangle extends Shape {
    private final float width;
    private final float height;

    public Rectangle(Renderer renderer, float width, float height) {
        super(renderer);
        this.width = width;
        this.height = height;
    }

    @Override
    public void draw() {
        renderer.renderRectangle(width, height);
    }
}
```

CLEINT CODE

```
public class BridgeDemo {
    public static void main(String[] args) {
        Renderer vector = new VectorRenderer();
        Renderer raster = new RasterRenderer();

        Shape circle1 = new Circle(vector, 5);
        Shape circle2 = new Circle(raster, 5);

        Shape rectangle1 = new Rectangle(vector, 10, 4);
        Shape rectangle2 = new Rectangle(raster, 10, 4);

        circle1.draw();    // Vector
        circle2.draw();    // Raster
        rectangle1.draw();  // Vector
        rectangle2.draw();  // Raster
    }
```

}

WHAT WE ACHIEVED

- **Decoupled abstractions from implementations**: Shapes and renderers evolve independently
- **Open/Closed compliance**: You can add new renderers or shapes without modifying existing ones
- **No class explosion**: Avoided the need for every shape-renderer subclass
- **Runtime flexibility**: Dynamically switch renderers based on user/device context
- **Clean, extensible design**: Each class has a single responsibility and can be composed as needed

// Implementor

```
interface MessageSender {
    void sendMessage(String message);
}

// Concrete Implementors
class EmailSender implements MessageSender {
    public void sendMessage(String message) {
        System.out.println("Sending Email: " + message);
    }
}

class SMSSender implements MessageSender {
    public void sendMessage(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

// Abstraction
abstract class Message {
    protected MessageSender sender;
    public Message(MessageSender sender) {
        this.sender = sender;
    }
    abstract void send(String text);
}

// Refined Abstraction
class TextMessage extends Message {
    public TextMessage(MessageSender sender) { super(sender); }
    public void send(String text) {
        sender.sendMessage("Text Message: " + text);
    }
}

class AlertMessage extends Message {
    public AlertMessage(MessageSender sender) { super(sender); }
    public void send(String text) {
        sender.sendMessage("Alert: " + text);
    }
}

// Usage
public class BridgeDemo {
    public static void main(String[] args) {
        Message msg1 = new TextMessage(new EmailSender());
        msg1.send("Hello User!");

        Message msg2 = new AlertMessage(new SMSSender());
        msg2.send("System Down!");
    }
}
```

PAYMENT GATEWAY BRIDGE PATTERN

```
//implementor
interface PaymentGateway {
    void processPayment(double amount);
}

// Concrete Implementors
class PayPalGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " via PayPal.");
    }
}

class StripeGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment of $" + amount + " via Stripe.");
    }
}

class RazorpayGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment of ₹" + amount + " via Razorpay.");
    }
}

// Abstraction
abstract class Payment {
    protected PaymentGateway gateway;

    public Payment(PaymentGateway gateway) {
        this.gateway = gateway;
    }

    public abstract void pay(double amount);
}

// Refined Abstractions
class OrderPayment extends Payment {
    public OrderPayment(PaymentGateway gateway) {
        super(gateway);
    }

    @Override
    public void pay(double amount) {
        System.out.println("Order Payment initiated...");
        gateway.processPayment(amount);
    }
}

class SubscriptionPayment extends Payment {
    public SubscriptionPayment(PaymentGateway gateway) {
        super(gateway);
    }

    @Override
    public void pay(double amount) {
        System.out.println("Subscription Payment initiated...");
        gateway.processPayment(amount);
    }
```

```
}

// Client
public class BridgePaymentDemo {
    public static void main(String[] args) {
        // Use Case 1: Order payment using PayPal
        Payment order = new OrderPayment(new PayPalGateway());
        order.pay(250.75);

        // Use Case 2: Subscription payment using Stripe
        Payment subscription = new SubscriptionPayment(new StripeGateway());
        subscription.pay(99.99);

        // Use Case 3: Order payment using Razorpay
        Payment orderInIndia = new OrderPayment(new RazorpayGateway());
        orderInIndia.pay(1500.00);
    }
}
```