Search articles...

Bridge Design Pattern

Topic Tags:

System Design LLD

Problem Statement: Separating the Unseparable ⚖️

Imagine you're designing a drawing application that supports different shapes (Circle, Rectangle, etc.) and rendering methods (Vector and Raster). Your goal is to create a system that can render any shape using any rendering method.

Each shape has specific attributes and behaviors, and each rendering method has its own way of drawing the shapes. For instance:
• Circle has attributes like radius and a method to draw itself.
• Rectangle has attributes like width and height and a method to draw itself.
• Vector Rendering uses mathematical equations to draw shapes.
• Raster Rendering uses pixels to draw shapes.

The Problem: Shapes and rendering methods are tightly coupled. Adding a new shape or a new rendering method requires modifying existing code, making the system fragile and hard to maintain.

The Challenge: How can you design a clean and scalable solution that allows adding new shapes or rendering methods independently without affecting existing code?

Solving It the Traditional Way: A Messy Solution 🛠️

Let's look at a naive approach to solving this problem:

Shape.java

Java

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

Circle.java

Java

```java
public class Circle extends Shape {
    @Override
    public void rasterDraw() {
        System.out.println("Drawing Circle using Raster Rendering");
    }
}
```

Rectangle.java

Java

```java
public class Rectangle extends Shape {
    @Override
    public void rasterDraw() {
        System.out.println("Drawing Rectangle using Raster Rendering");
    }
}

public class DrawingApp {
    public static void main(String[] args) {
        Shape circle = new Circle();
```

```
        circle.rasterDraw();
        Shape rectangle = new Rectangle();
        rectangle.rasterDraw();
    }
}
```

The Problems with This Approach:

1. No Flexibility for Rendering Methods: If we want to add Vector Rendering, we need to modify every shape class.
2. Code Duplication: Each shape must implement rendering logic, leading to repeated code.
3. Poor Scalability: Adding new shapes or rendering methods requires significant changes, increasing maintenance overhead.

Interviewer's Follow-up Questions: Can We Improve the Code? 🤔

An interviewer might ask:
1. What if we need to add new shapes? For example, adding a Triangle or Polygon.
2. What if we need to support new rendering methods? For instance, rendering shapes using a 3D API.
3. How can we ensure minimal changes to existing code when adding new features?
4. Can we decouple shapes from rendering methods to make the system more modular?
These questions highlight the need for a better design that separates shape-specific logic from rendering-specific logic.

Ugly Code: When We Realize the Code Needs Restructuring 🤦

Let's say the logic for rendering becomes more complex:

Java

```
import java.util.Scanner;
public class DrawingApp {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    // Display available shapes
    System.out.println("Available shapes:");
    System.out.println("1. Circle");
    System.out.println("2. Rectangle");
    System.out.println("3. Triangle");
```

```
10      System.out.println("4. Square");
11      System.out.println("5. Hexagon");
12      System.out.print("Enter the number for desired shape: ");
13      int shapeChoice = scanner.nextInt();
14      String shapeType = getShapeType(shapeChoice);
15      // Display available rendering methods
16      System.out.println("nAvailable rendering methods:");
17      System.out.println("1. Raster");
18      System.out.println("2. Vector");
19      System.out.println("3. 3D");
20      System.out.println("4. Wireframe");
21      System.out.print("Enter the number for desired rendering method: ");
22      int renderChoice = scanner.nextInt();
23      String renderType = getRenderType(renderChoice);
24
25
26      // Process the choices
27      if (shapeType != null && renderType != null) {
28        drawShape(shapeType, renderType);
29      } else {
30        System.out.println("Invalid selection. Please try again.");
31      }
32      scanner.close();
33    }
34
35    private static String getShapeType(int choice) {
36      switch (choice) {
37        case 1:
38          return "Circle";
39        case 2:
40          return "Rectangle";
41        case 3:
42          return "Triangle";
43        case 4:
44          return "Square";
45        case 5:
46          return "Hexagon";
47        default:
48          return null;
49      }
50    }
```

```
51
52    private static String getRenderType(int choice) {
53      switch (choice) {
54        case 1:
55          return "Raster";
56        case 2:
57          return "Vector";
58        case 3:
59          return "3D";
60        case 4:
61          return "Wireframe";
62        default:
63          return null;
64      }
65    }
66
67    private static void drawShape(String shapeType, String renderType) {
68      // Circle combinations
69      if (shapeType.equals("Circle") && renderType.equals("Raster")) {
70        System.out.println("Drawing Circle using Raster Rendering");
71      } else if (shapeType.equals("Circle") && renderType.equals("Vector")
72        System.out.println("Drawing Circle using Vector Rendering");
73      } else if (shapeType.equals("Circle") && renderType.equals("3D")) {
74        System.out.println("Drawing Circle using 3D Rendering");
75      } else if (shapeType.equals("Circle") && renderType.equals("Wirefram
76        System.out.println("Drawing Circle using Wireframe Rendering");
77      }
78      // Rectangle combinations
79      else if (shapeType.equals("Rectangle") && renderType.equals("Raster"
80        System.out.println("Drawing Rectangle using Raster Rendering");
81      } else if (shapeType.equals("Rectangle") && renderType.equals("Vecto
82        System.out.println("Drawing Rectangle using Vector Rendering");
83      } else if (shapeType.equals("Rectangle") && renderType.equals("3D"))
84        System.out.println("Drawing Rectangle using 3D Rendering");
85      } else if (shapeType.equals("Rectangle")
86          && renderType.equals("Wireframe")) {
87        System.out.println("Drawing Rectangle using Wireframe Rendering");
88      }
89      // Triangle combinations
90      else if (shapeType.equals("Triangle") && renderType.equals("Raster")
91        System.out.println("Drawing Triangle using Raster Rendering");
```

```
92      } else if (shapeType.equals("Triangle") && renderType.equals("Vector
93        System.out.println("Drawing Triangle using Vector Rendering");
94      } else if (shapeType.equals("Triangle") && renderType.equals("3D"))
95        System.out.println("Drawing Triangle using 3D Rendering");
96      } else if (shapeType.equals("Triangle") && renderType.equals("Wirefr
97        System.out.println("Drawing Triangle using Wireframe Rendering");
98      }
99      // Square combinations
100     else if (shapeType.equals("Square") && renderType.equals("Raster"))
101       System.out.println("Drawing Square using Raster Rendering");
102     } else if (shapeType.equals("Square") && renderType.equals("Vector")
103       System.out.println("Drawing Square using Vector Rendering");
104     } else if (shapeType.equals("Square") && renderType.equals("3D")) {
105       System.out.println("Drawing Square using 3D Rendering");
106     } else if (shapeType.equals("Square") && renderType.equals("Wirefram
107       System.out.println("Drawing Square using Wireframe Rendering");
108     }
109     else {
110       System.out.println("Unsupported combination of shape and rendering
111     }
112   }
113 }
```

Why is This Code Problematic ? :

1. Hardcoding: Adding new shapes or rendering methods requires modifying multiple conditional branches.
2. Tightly Coupled Logic: Shapes and rendering methods are entangled, making the code hard to extend or debug.
3. Fragile Design: A single change can break the entire system.

The Savior: Bridge Design Pattern 👼

The Bridge Design Pattern is ideal for this problem. It separates abstraction (Shape) from its implementation (Rendering Method) so that the two can evolve independently.
In essence, the Bridge Pattern decouples what is being done (shapes) from how it is being done (rendering methods).

How the Bridge Design Pattern Works 🔧

1. Abstraction: Define the high-level interface for shapes.
2. Implementor: Define the interface for rendering methods.
3. Concrete Implementor: Provide specific implementations of the rendering methods (e.g., Raster, Vector).
4. Refined Abstraction: Implement shapes by using rendering methods through composition.

Solving the Problem with Bridge Design Pattern 🌐

Step 1: Define the Renderer Interface

This interface provides methods to render shapes:

Renderer.java

Java

```java
// Renderer.java - Interface for rendering methods
public interface Renderer {
    void renderCircle(double radius);
    void renderRectangle(double width, double height);
}
```

Step 2: Create Concrete Implementations for Rendering Methods

In this step,we will Create concrete classes that implement the Renderer interface. These classes will provide specific implementations for the rendering methods defined in the interface.

RasterRenderer.java

Java

```java
// RasterRenderer.java - Concrete implementation for Raster Rendering
public class RasterRenderer implements Renderer {
    @Override
    public void renderCircle(double radius) {
        System.out.println("Raster Rendering: Drawing Circle with radius " +
    }
    @Override
    public void renderRectangle(double width, double height) {
        System.out.println("Raster Rendering: Drawing Rectangle with width "
```

```
    }
}
```

VectorRenderer.java

Java

```java
// VectorRenderer.java - Concrete implementation for Vector Rendering
public class VectorRenderer implements Renderer {
    @Override
    public void renderCircle(double radius) {
        System.out.println("Vector Rendering: Drawing Circle with radius " +
    }
    @Override
    public void renderRectangle(double width, double height) {
        System.out.println("Vector Rendering: Drawing Rectangle with width "
    }
}
```

Step 3: Define the Abstract Shape Class :

In this step, we will create an abstract class that represents a generic shape. This abstract class will define common properties and methods that all shapes must have, such as dimensions and rendering capabilities

Shape.java

Java

```java
// Shape.java - Abstract class for shapes
public abstract class Shape {
    protected Renderer renderer;
    // Constructor to accept a renderer
    public Shape(Renderer renderer) {
        this.renderer = renderer;
    }
    public abstract void draw();
}
```

Step 4: Create Refined Shape Classes :

In this step, we will create concrete classes that extend the abstract Shape class. These refined shape classes will provide specific implementations for the abstract methods defined in the Shape class.

Circle.java :

Java

```java
// Circle.java - Refined abstraction for Circle
public class Circle extends Shape {
    private double radius;
    public Circle(Renderer renderer, double radius) {
        super(renderer);
        this.radius = radius;
    }
    @Override
    public void draw() {
        renderer.renderCircle(radius);
    }
}
```

Rectangle.java

Java

```java
// Rectangle.java - Refined abstraction for Rectangle
public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(Renderer renderer, double width, double height) {
        super(renderer);
        this.width = width;
        this.height = height;
    }
    @Override
    public void draw() {
        renderer.renderRectangle(width, height);
```

```
    }
}
```

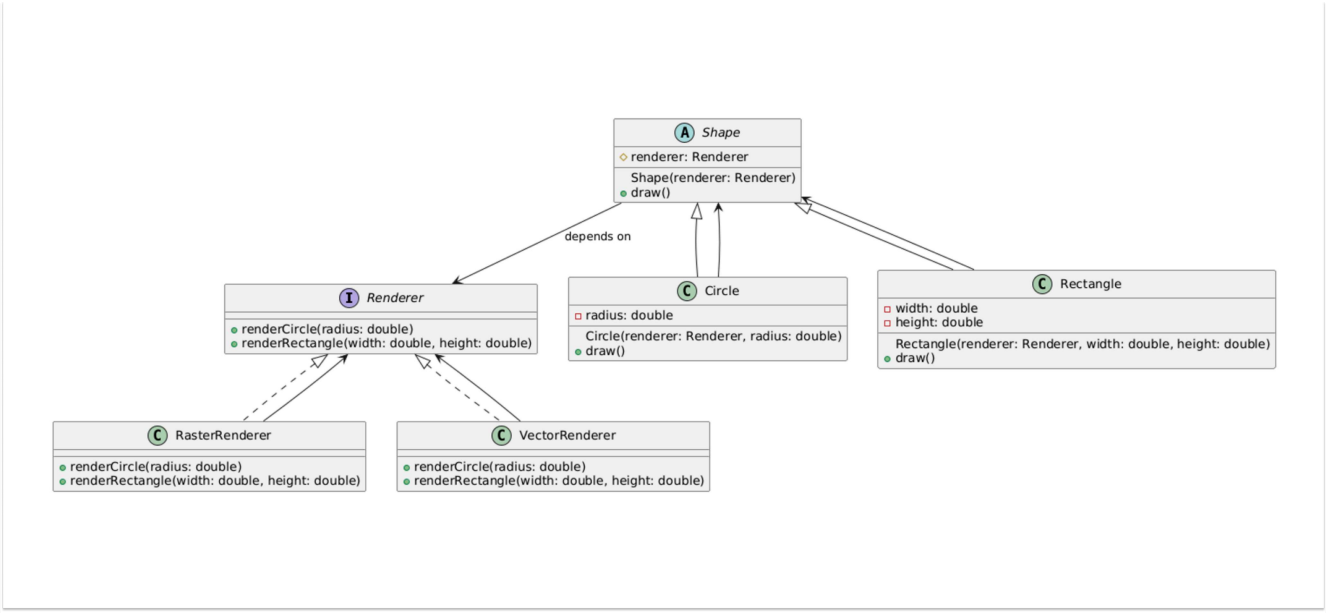
Step 5: Use the Bridge Pattern in the Application :

In this step, we will integrate the Bridge Pattern into our application to decouple the abstraction (shape) from its implementation (renderer). This allows us to vary both independently, promoting flexibility and scalability in our design.

DrawingApp.java :

Java

```java
// DrawingApp.java
public class DrawingApp {
    public static void main(String[] args) {
        Renderer rasterRenderer = new RasterRenderer();
        Renderer vectorRenderer = new VectorRenderer();
        Shape rasterCircle = new Circle(rasterRenderer, 5);
        Shape vectorCircle = new Circle(vectorRenderer, 5);
        Shape rasterRectangle = new Rectangle(rasterRenderer, 10, 5);
        Shape vectorRectangle = new Rectangle(vectorRenderer, 10, 5);
        rasterCircle.draw();
        vectorCircle.draw();
        rasterRectangle.draw();
        vectorRectangle.draw();
    }
}
```



Advantages of Using the Bridge Design Pattern 🏆

1. Decoupling:
Shapes and rendering methods are decoupled, allowing them to vary independently.

2. Scalability:
Adding a new shape or rendering method requires minimal changes.

3. Reusability:
Rendering logic can be reused across different shapes.

4. Maintainability:
Code is cleaner and easier to maintain, as each class has a single responsibility.

Real-life Use Cases of the Bridge Pattern 🌎

1. Graphics Libraries:
Libraries like OpenGL use a bridge-like pattern to separate shapes from rendering logic.

2. UI Frameworks:
UI frameworks often decouple widgets from their look-and-feel using the Bridge Pattern.

3. Persistence Mechanisms:
ORM libraries use bridges to decouple object manipulation from database-specific queries.

Conclusion 🎯

The Bridge Design Pattern is a powerful tool for decoupling abstraction from implementation. In our drawing application example, it allows shapes and rendering methods to evolve independently, making the system flexible, scalable, and maintainable. Whether you're designing a graphics library, a UI framework, or any system with varying abstractions and implementations, the Bridge Pattern is an excellent choice for clean architecture and long-term growth.