The Bridge Pattern is a structural design pattern that separates an object's interface from its implementation.

It decouples the abstraction (high-level logic) from the implementation details (low-level logic) by providing a bridge structure that connects them.

This pattern is particularly useful when you have multiple dimensions of variation and want to avoid an explosion of classes or interfaces.

Using the Bridge Pattern, you can achieve the following benefits:

* **Decoupling**: The abstraction (shape type) and implementation details (rendering mode) are decoupled, allowing them to vary independently.
* **Flexibility**: You can easily add new shape types or rendering modes without creating a large number of classes.
* **Simplified** Code: The number of classes is reduced, making the code more maintainable and easier to understand.
* Extensibility: You can extend the system by adding new shape types or rendering modes without impacting existing code.

*interface Shape* {
 *void* draw();
}

*abstract class* ShapeType *implements Shape* {
 *protected RenderingMode* renderingMode;

 *public* ShapeType(*RenderingMode* renderingMode) {
 *this*.renderingMode = renderingMode;
 }

*// public abstract void drawShape();*}

*class* Circle *extends* ShapeType {

 *public* Circle(*RenderingMode* renderingMode) {
 *super*(renderingMode);
 }

 @Override
 *public void* draw() {
 System.*out*.print("Drawing a ");
 renderingMode.render();
 }
}

*class* Rectangle *extends* ShapeType {
 *public* Rectangle(*RenderingMode* renderingMode) {
 *super*(renderingMode);
 }

 @Override
 *public void* draw() {
 System.*out*.print("Drawing a ");
 renderingMode.render();
 }
}

*// Other shape types...

interface RenderingMode* {
 *void* render();
}

*class* TwoDRendering *implements RenderingMode* {
 *public void* render() {
 System.*out*.println("2D shape");
 }
}

*class* ThreeDRendering *implements RenderingMode* {
 *public void* render() {
 System.*out*.println("3D shape");
 }
}

*public class* Bridge {
 *public static void* main(String[] args) {
 *RenderingMode* twoD = *new* TwoDRendering();
 *RenderingMode* threeD = *new* ThreeDRendering();

 *Shape* circle = *new* Circle(twoD);
 *Shape* rectangle = *new* Rectangle(threeD);

 circle.draw();
 rectangle.draw();
 }
}

*/\*\*

Abstraction = Shape
Implementation = Color

You decouple abstraction (Shape) from implementation (Color).

You can add new shapes or colors independently without modifying
existing code (Open/Closed Principle).

 \*/

interface Color* {
 *void* applyColor();
}
*class* RedColor *implements Color* {
 *public void* applyColor() {
 System.*out*.println("Applying red color");
 }
}

*class* GreenColor *implements Color* {
 *public void* applyColor() {
 System.*out*.println("Applying green color");
 }
}
*abstract class* Shape {
 *protected Color* color;

 *// Bridge: injecting implementation
 public* Shape(*Color* color) {
 *this*.color = color;
 }

 *public abstract void* draw();
}

*class* Circle *extends* Shape {
 *public* Circle(*Color* color) {
 *super*(color);
 }

 *public void* draw() {
 System.*out*.print("Drawing Circle with ");
 color.applyColor();
 }
}

*class* Square *extends* Shape {
 *public* Square(*Color* color) {
 *super*(color);
 }

 *public void* draw() {
 System.*out*.print("Drawing Square with ");
 color.applyColor();
 }
}

*public class* Bridge2 {
 *public static void* main(String[] args) {
 Shape redCircle = *new* Circle(*new* RedColor());
 Shape greenSquare = *new* Square(*new* GreenColor());

 redCircle.draw(); *// Output: Drawing Circle with Applying red color* greenSquare.draw(); *// Output: Drawing Square with Applying green color* }
}