15 Structural: Decorator Pattern for Dynamic UI Enhancements

**Scenario** Your team is building a design-tool prototype. Each UI element (for now, just a TextView) must be able to receive optional visual effects at **runtime** such as borders, shadows, and scroll indicators.  
 The core widget code must stay closed for modification while new effects can be introduced independently and combined in any order.

#### **Tasks**

1. **Analysis** (analysis/decorator\_choice.md)  
   * Explain why inheritance fails when every combination of effects is required ( TextViewWithBorderAndShadowAndScroll… ).
   * List two benefits of choosing Decorator over subclassing for this scenario.
2. **Implementation** (src/main/java/clean/)  
   * UIComponent – target interface with render().
   * Concrete component TextView.
   * Abstract ComponentDecorator wrapping a UIComponent.
   * Concrete decorators:  
     1. BorderDecorator
     2. ShadowDecorator
     3. ScrollDecorator (adds vertical scroll bars).
   * Demo.java – build three variants:  
     1. plain TextView,
     2. bordered view,
     3. bordered + shadow + scroll. Print output.
3. **Unit tests** (src/test/java/)  
   * BorderAddsBehaviourTest – capture System.out; assert extra line “Adding border…” appears.
   * DecoratorStackOrderTest – wrap in two orders (Border→Shadow vs Shadow→Border) and assert output order differs accordingly.
   * RuntimeCompositionTest – use a List<ComponentDecorator> to attach random effects at runtime, assert no exception.
4. **Reflection** (reflection.md)  
   * Discuss how Decorator honours the Open/Closed Principle here.
   * Identify a potential performance cost of deep decorator chains.
   * Suggest a strategy to keep chains readable (builder, factory, or UI DSL).

#### **Deliverables**

analysis/decorator\_choice.md

src/main/java/clean/UIComponent.java

src/main/java/clean/TextView.java

src/main/java/clean/ComponentDecorator.java

src/main/java/clean/BorderDecorator.java

src/main/java/clean/ShadowDecorator.java

src/main/java/clean/ScrollDecorator.java

src/main/java/clean/Demo.java

src/test/java/BorderAddsBehaviourTest.java

src/test/java/DecoratorStackOrderTest.java

src/test/java/RuntimeCompositionTest.java

reflection.md

README.md

## **Detailed Solution**

### **1 Core Interfaces & Component**

/\* UIComponent.java \*/

package clean;

public interface UIComponent {

void render();

}

/\* TextView.java \*/

package clean;

public class TextView implements UIComponent {

@Override public void render() { System.out.println("Rendering TextView."); }

}

### **2 Decorator Hierarchy**

/\* ComponentDecorator.java \*/

package clean;

public abstract class ComponentDecorator implements UIComponent {

protected final UIComponent component;

protected ComponentDecorator(UIComponent c){ this.component = c; }

@Override public void render(){ component.render(); }

}

/\* BorderDecorator.java \*/

package clean;

public class BorderDecorator extends ComponentDecorator {

public BorderDecorator(UIComponent c){ super(c); }

@Override public void render(){

super.render();

System.out.println("Adding border to the component.");

}

}

/\* ShadowDecorator.java \*/

package clean;

public class ShadowDecorator extends ComponentDecorator {

public ShadowDecorator(UIComponent c){ super(c); }

@Override public void render(){

super.render();

System.out.println("Adding shadow to the component.");

}

}

/\* ScrollDecorator.java \*/

package clean;

public class ScrollDecorator extends ComponentDecorator {

public ScrollDecorator(UIComponent c){ super(c); }

@Override public void render(){

super.render();

System.out.println("Adding vertical scrollbar.");

}

}

### **3 Demo**

/\* Demo.java \*/

package clean;

public class Demo {

public static void main(String[] args){

UIComponent plain = new TextView();

UIComponent bordered = new BorderDecorator(new TextView());

UIComponent fancy = new ScrollDecorator(

new ShadowDecorator(

new BorderDecorator(

new TextView())));

System.out.println("== Plain ==");

plain.render();

System.out.println("\n== Bordered ==");

bordered.render();

System.out.println("\n== Bordered + Shadow + Scroll ==");

fancy.render();

}

}

**Console**

== Plain ==

Rendering TextView.

== Bordered ==

Rendering TextView.

Adding border to the component.

== Bordered + Shadow + Scroll ==

Rendering TextView.

Adding border to the component.

Adding shadow to the component.

Adding vertical scrollbar.

### **4 Test Outline (JUnit 5)**

/\* BorderAddsBehaviourTest \*/

ByteArrayOutputStream out = new ByteArrayOutputStream();

System.setOut(new PrintStream(out));

new BorderDecorator(new TextView()).render();

assertTrue(out.toString().contains("Adding border"));

/\* DecoratorStackOrderTest \*/

String a = capture(() -> new BorderDecorator(new ShadowDecorator(new TextView())).render());

String b = capture(() -> new ShadowDecorator(new BorderDecorator(new TextView())).render());

assertNotEquals(a, b); // order matters

/\* RuntimeCompositionTest \*/

UIComponent comp = new TextView();

for(ComponentDecorator d : List.of(new BorderDecorator(comp),

new ShadowDecorator(comp),

new ScrollDecorator(comp))){

d.render();

}

*capture(Runnable r) is a helper that redirects System.out and returns the string.*

### **Reflection (high-level)**

* **Open/Closed** – adding GlowDecorator requires zero edits to TextView or existing decorators.
* **Chain readability** – could introduce an EffectBuilder that stores decorators in a list and applies them in one call.
* **Performance** – each decorator adds an extra object and stack frame; keep chains shallow or cache common compositions.

The Decorator pattern here lets UI designers layer any combination of visual effects in runtime configuration files or a GUI without recompiling the core widget library.