Lesson 9 Bridge and Decorator

The Bridge Pattern

1. Intent

   Decouple an abstraction from its implementation so that the two can vary independently.

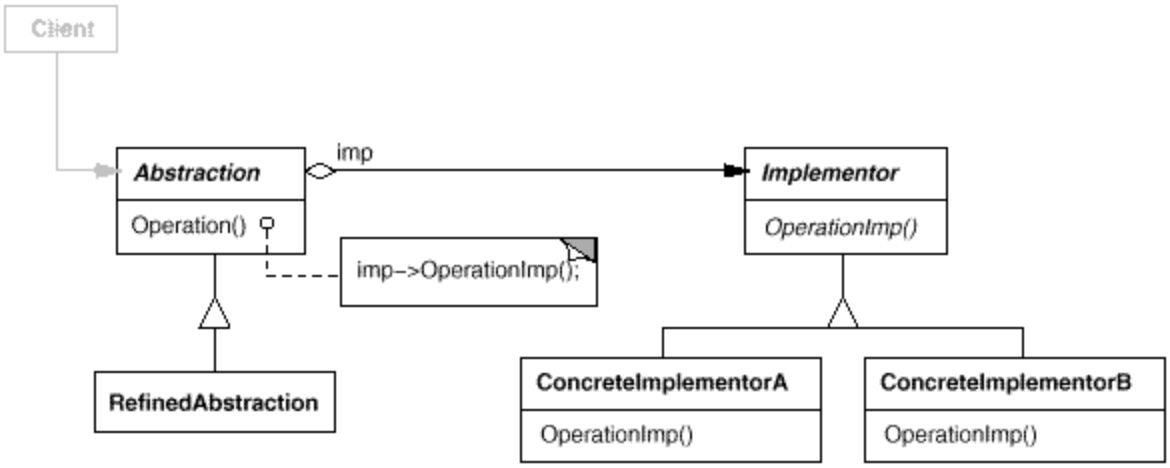
2. Motivation

   a. Avoid permanent binding between abstraction and implementation by favoring composition over inheritance.
   b. Changes in implementation should have no impact to the client, as long as the interface stays the same.
   c. There are existing implementations of a certain API you want to use in your business logic (methods).

3. Applicability

   Use the Bridge pattern when
   a. You want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
   b. Changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
   c. You want to share an implementation among multiple objects, and this fact should be hidden from the client.

4. Structure



5. Participants

a. Abstraction
   - defines the abstraction's interface.
   - maintains a reference to an object of type Implementor.

b. RefinedAbstraction
   - Concrete/sub-type of Abstraction.

c. Implementor
   - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.

d. ConcreteImplementor
   - implements the Implementor interface and defines its concrete implementation.

6. Consequences

The Bridge pattern has the following consequences:

a. Decoupling interface and implementation.
b. Improved extensibility. You can extend the Abstraction and Implementation hierarchies independently.
c. Hiding implementation details from clients.
d. Allowing us to build new services/APIs on top of existing implementations (of different granularities, for example).

7. How to implement the Bridge pattern?

a. Abstraction

```
public abstract class Shape {
	protected DrawingAPI drawAPI;

	protected Shape(DrawingAPI drawAPI) {
		this.drawAPI = drawAPI;
	}

	public abstract void draw();
}
```

b. Refined abstraction

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

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

	public void draw() {
		drawAPI.drawQuadrant(x-radius, y, x, y+radius);
		drawAPI.drawQuadrant(x, y+radius, x+radius, y);
		drawAPI.drawQuadrant(x+radius, y, x, y-radius);
		drawAPI.drawQuadrant(x, y-radius x-radius, y);
	}
}
```

```
public class Rectangle extends Shape {
	private int x1, y1, x2, y2;

	public Rectangle(int x1, int y1, int x2, int y2, DrawingAPI
drawAPI) {
		super(drawAPI);
		this.x1 = x1;
		this.y1 = y1;
		this.x2 = x2;
		this.y2 = y2;
	}

	public void draw() {
		drawAPI.drawLine(x1, y1, x2, y1);
		drawAPI.drawLine(x2, y1, x2, y2);
		drawAPI.drawLine(x2, y2, x1, y2);
		drawAPI.drawLine(x1, y2, x1, y1);

	}
}
```

c. Implementor

```
public interface DrawingAPI {
	public void drawLine(int x1, int y1, int x2, int y2);
	public void drawQuadrant(int x1, int y1, int x2, int y2);
}
```

d. Concrete Implementor

```
public class DrawingAPIImpl1 implements DrawingAPI {

	@Override
	public void drawLine(int x1, int y1, int x2, int y2){
		ExistingDrawingTool1.drawLine(x1, y1, x2, y2);
	}

	@Override
	public void drawQuadrant(int x1, int y1, int x2, int y2) {
		ExistingDrawingTool1.drawQuadrant(x1, y1, x2, y2);
	}

}
```

e. Existing implementations you build a bridge for.

```
public class ExistingDrawingTool1 {
	public static void drawLine(int x1, int y1, int x2, int y2) {
		System.out
				.println("This method draws a line from
point1(x1, y1) to point2(x2, y2");
	}

	public static void drawQuadrant(int x1, int y1, int x2, int y2) {
		System.out
```

```
                        .println("This method draws a quarter circle
from point1(x1, y1) to point2(x2, y2");
      }

}
```

Pencil and Paper 9-1

The Bridge pattern appears to have much in common with the Strategy pattern. Note the similarity between their GOF intent statements:
Bridge Intent: “Decouple an abstraction from its implementation so that the two can vary independently.”
Strategy Intent: “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”
Also note the similarity between their UML diagrams.

a. Specify the deciding factors that determine which of these two patterns should be applied in a particular situation, emphasizing the reasons why the Bridge pattern is considered to be structural while the Strategy pattern is considered to be behavioral.
b. Supply examples to demonstrate this distinction.

Submit a Word document with your answers.

The Decorator Pattern

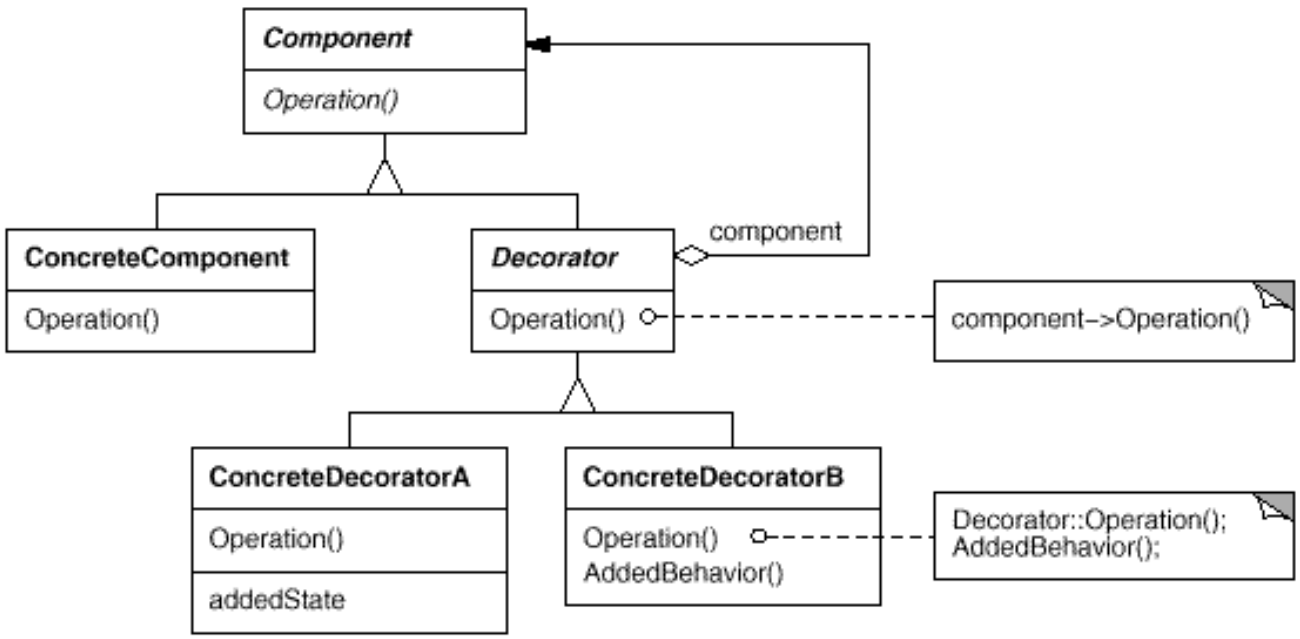
1. Intent

   Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

2. Motivation

   a. Sometimes we want to add responsibilities to individual objects, not to an entire class; or modify the responsibilities of an object at runtime.
   b. Subclassing does not support the above normally. But even if it does, it would create an exploding number of subclasses that are hard to manage.

3. Structure



4. Participants

a. Component
   - defines the interface for objects that can have responsibilities added to them dynamically.

b. ConcreteComponent
   - defines an object to which additional responsibilities can be attached.

c. Decorator
   - maintains a reference to a Component object and defines an interface that conforms to Component's interface.

d. ConcreteDecorator
   - adds responsibilities to the component.

5. Applicability

Use the Decorator Pattern

a. to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

b. when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

6. How to implement the Decorator Pattern

a. Component interface or abstract class that can be decorated dynamically.

```
public interface GUIContainer {
    public void paint();
    public String getDescription();
}
```

b. Concrete component class that can have responsibilities added (decorated) dynamically.

```
public class Window implements GUIContainer {

      @Override
      public void paint() {
            // implementation
      }

      @Override
    public String getDescription() {
        return "A Window Container";
    }
}
```

c. Decorator interface or abstract class.

```
public abstract class WindowDecorator implements GUIContainer {
    protected GUIContainer windowToBeDecorated;

    public WindowDecorator (GUIContainer windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }

    @Override
    public void paint() {
        windowToBeDecorated.paint();
    }

    @Override
    public String getDescription() {
        return windowToBeDecorated.getDescription();
    }
}
```

d. Concrete decorator classes.

```
public class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (GUIContainer
windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void paint() {
        super.paint();
        paintVerticalScrollBar();
    }

    private void paintVerticalScrollBar() {
        // implementation
    }
```

```
    @Override
    public String getDescription() {
        return super.getDescription() + ", adding vertical scrollbar";
    }
}
```

```
public class HorizontalScrollBarDecorator extends WindowDecorator {

    public HorizontalScrollBarDecorator (GUIContainer
windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void paint() {
        super.paint();
        paintHorizontalScrollBar();
    }

    private void paintHorizontalScrollBar() {
        // implementation
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", adding horizontal
scrollbar";
    }
}
```

e. Client

```
public class Client{
    public static void main(String[] args) {

       //decorate the Window with both vertical and horizontal scroll bars
        GUIContainer decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator (new Window()));

        System.out.println(decoratedWindow.getDescription());
    }
}
```

Lab 9-2

In real world projects, the Decorator Pattern is used to implement GUI components as mentioned in the GoF book. In the Java platform, the pattern is used to develop the InputStream and OutputStream subclasses. In addition, people also use it to develop debugging/profiling tools.

Let's write 2 List profiler classes with the Decorator Pattern – LinkedListProfiler and ArrayListProfiler. Then compare how much it takes to run some key methods and print the result that looks like the following –

	ArrayList	LinkedList
boolean add(E e)	…milliseconds	…milliseconds
boolean remove(Object o)	…milliseconds	…milliseconds
boolean contains(Object o)	…milliseconds	…milliseconds
int size()	…milliseconds	…milliseconds