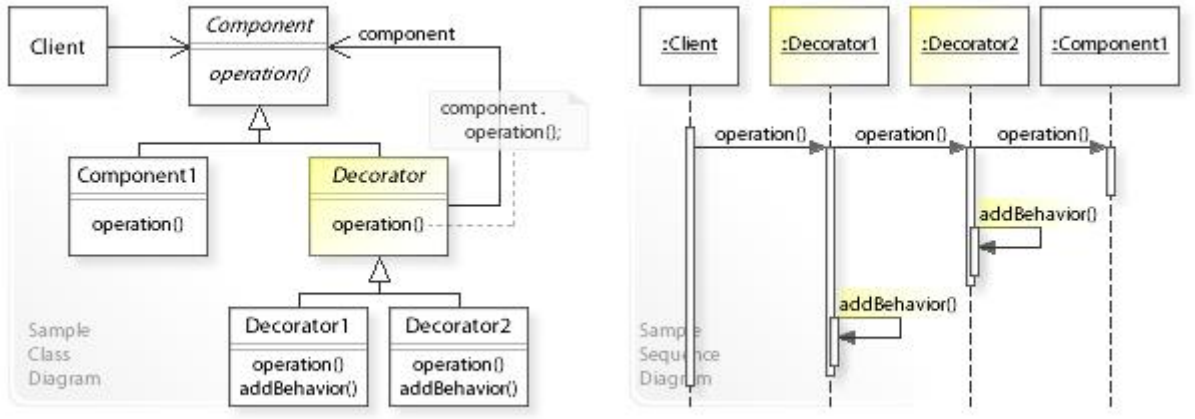
DESIGN PATTERNS: DECORATOR

Structural Patterns

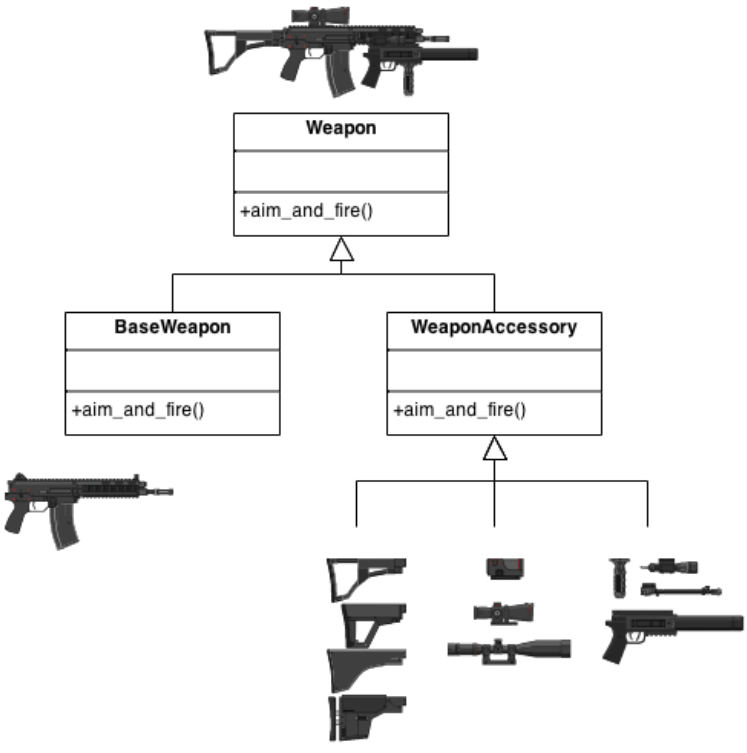
1. Design Pattern Description

Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box. → CASCADING or CHAINING features

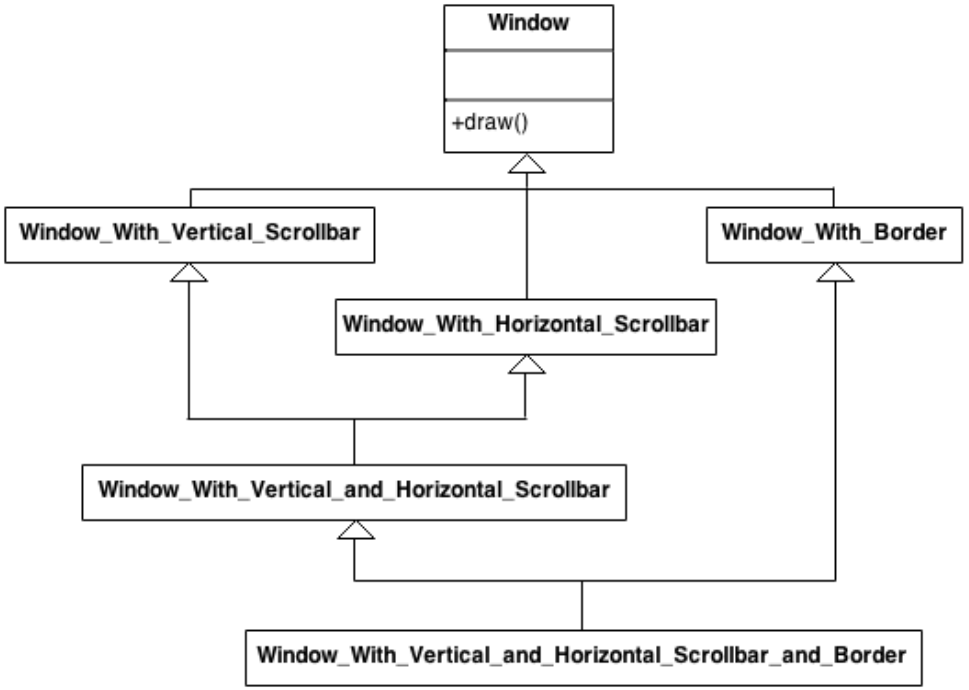


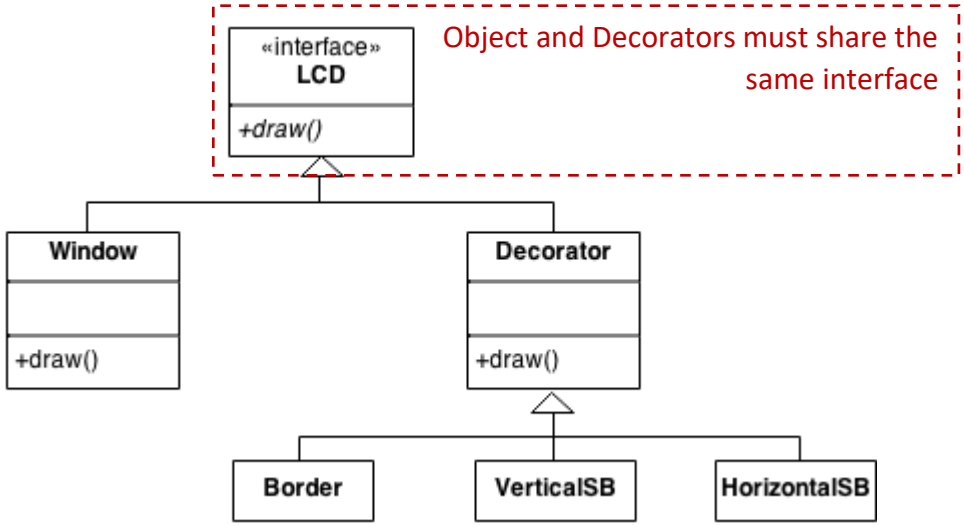

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```



- A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.
- Composite and Decorator are usually used in concert: object aggregation (allow aggregated objects access parent properties) + override some of these properties on parts of the composition.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.

2. Design Pattern Example





```
// The Window interface class
public interface Window {
    void draw(); // Draws the Window
    String getDescription(); // Returns a description of the Window
}

// Implementation of a simple Window without any scrollbars
class SimpleWindow implements Window {
    @Override
    public void draw() {
        // Draw window
    }
    @Override
    public String getDescription() {
        return "simple window";
    }
}
```

The following classes contain the decorators for all `Window` classes, including the decorator classes themselves.

```
// abstract decorator class - note that it implements Window
abstract class WindowDecorator implements Window {
    protected Window windowToBeDecorated; // the Window being decorated

    public WindowDecorator (Window windowToBeDecorated) {
        this.windowToBeDecorated = windowToBeDecorated;
    }
    @Override
    public void draw() {
        windowToBeDecorated.draw(); //Delegation
    }
    @Override
    public String getDescription() {
        return windowToBeDecorated.getDescription(); //Delegation
    }
}

// The first concrete decorator which adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }
```

```
    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", including vertical scrollbars";
    }
}

// The second concrete decorator which adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    @Override
    public void draw() {
        super.draw();
        drawHorizontalScrollBar();
    }

    private void drawHorizontalScrollBar() {
        // Draw the horizontal scrollbar
    }

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

3. Existing Example in FEM-MAT-OO

No.

4. Design Proposal in FEM-MAT-OO

Monitoring?