Tutorial 8

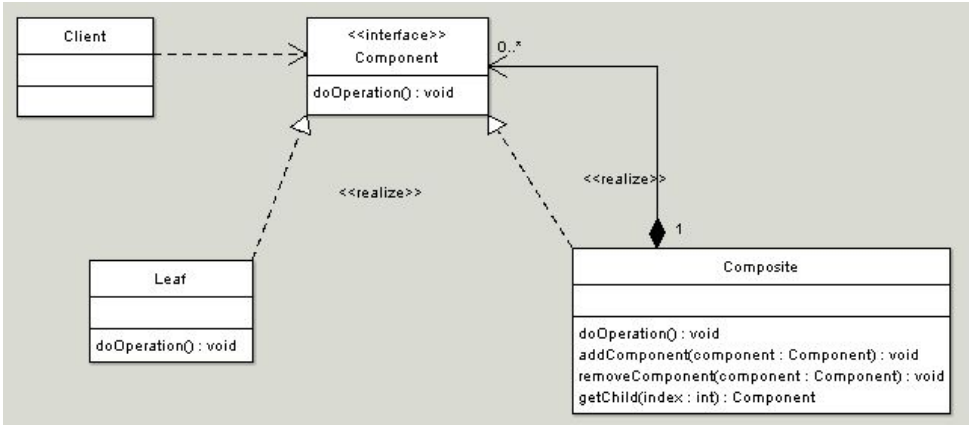
The Composite Pattern

Motivation

- Motivation - we want to treat a single object the in the same (or a similar) way as a collection of those objects
- The single object is called the **leaf**, the collection the **composite**
- There may be common operations both the leaf and the composite want to perform.
- The composite may have additional operations, or different implementations of the common operations

Solution

- Solution: Designate a component interface which has the common operations of both the leaf and the composite
- Leaves just implement this interface
- Components implement, and extend with new operations



UML for the Composite Pattern

Question 2

Which design principle does this solution violate and why?

Suggest a suitable design pattern to improve the design of the above solution.

Question 3

In a composite pattern, discuss the implication of placing the add(), remove() and getChild() methods in the Component interface over the Composite class?

Decorator Pattern

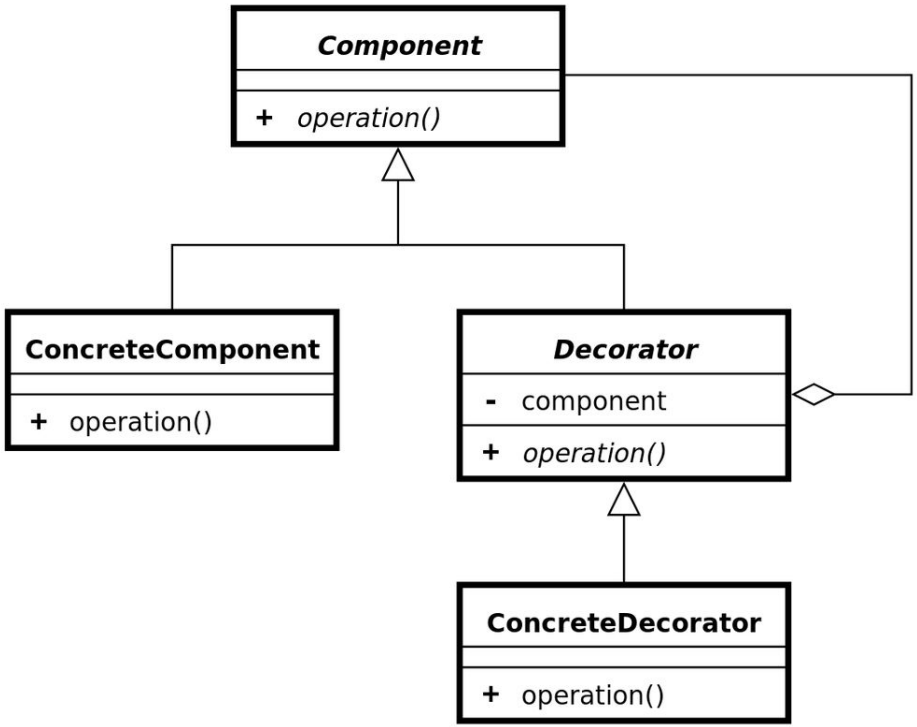
Motivation

- Motivation: we want to extend an objects functionality dynamically at runtime
- For example, the user decides they want an additional feature / turn on a setting

- Call the extendable objects **Components**, which implement some **Component** interface (similar to Composite pattern)
- Simple components (**undecorated**) just implement this interface

Solution

- Define a special **Decorator** abstract class, which implements Component
- Decorator aggregates a **Component** (the interface)
- Special feature extending decorators can then extend the Decorator class to
    - override existing operations
    - implement new operations
    - add state



UML for the Decorator Pattern (with one ConcreteDecorator)

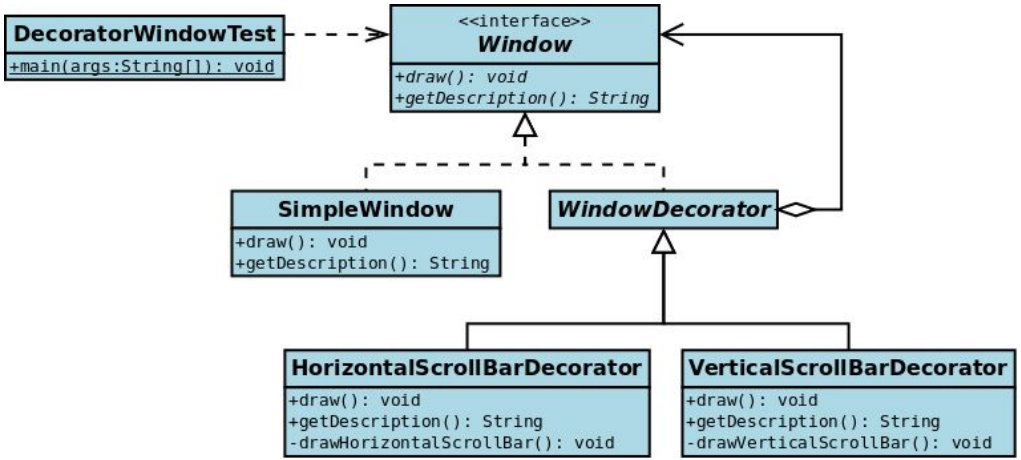
Example: Dynamically Navigable Windows

- Motivation: Windows in a windowing system must be scrollable to allow extra horizontal or vertical content
- However, scrollbars should only be displayed when it is necessary - if everything fits in the screen, they shouldn’t appear

Example: Dynamically Navigable Windows

- Component is **Window** with a draw() method, and **SimpleWindow** is the simple implementation with no scrollbars

- Define a ScrollingWindowDecorator abstract class, and create two implementations, **HorizontalScrollBarDecarator**, and **VerticalScrollBarDecorator**, with drawHorizontalScrollBar(), and drawVerticalScrollBar()



UML for the Window Example:
https://en.wikipedia.org/wiki/Decorator_pattern#First_example_(window/scrolling_scenario)

Why not Inheritance/Subclassing?

- Alternative Solution: extend SimpleWindow with VerticalScrollBarWindow and HorizontalScrollBarWindow
- How to create window with both Vertical and Horizontal Scrolling?
  - Create new class, HorizontalVerticalScrollBarWindow, extend from SimpleWindow
  - Reuse of code, but otherwise not too bad

Why not Inheritance/Subclassing?

- How to add resizing feature to windows?
- ResizableWindow, extend from SimpleWindow
- How about a resizable window with a vertical scroll bar?
- VerticalScrollBarResizableWindow?
- HorizontalVerticalScrollBarResizableWindow?

Advantages of Decorator Pattern

- Simplest approach to dynamic feature extension when there are many potential features
- Behavior is added at runtime (not compile time, like using inheritance)

Disadvantages of Decorator Pattern

- For small cases, may be overkill
- Instantiating an object with many decorations is a bit painful
    - new ResizableDecorator(new VerticalScrollBarDecorator(new HorizontalScrollBarDecorator(new SimpleWindow())))

Question 4

Write a decorator that converts all uppercase characters to lowercase in the input stream.

```
public class LowerCaseInputStream extends FilterInputStream {

      public LowerCaseInputStream(InputStream is) {
            super(is);
      }

      public int read() throws IOException {
            int c = super.read();
            return (c == -1 ? c: Character.toLowerCase(c));
      }

      public int read (byte[] b, int offset, int len) throws IOException {
            int result = super.read(b, offset, len);
            for (int i = offset; i < offset + result; i ++) {
                  b[i] = (byte)Character.toLowerCase((char)b[i]);
            }
            return result;
      }
}
```