Decorator Pattern

The Decorator Pattern from GoF

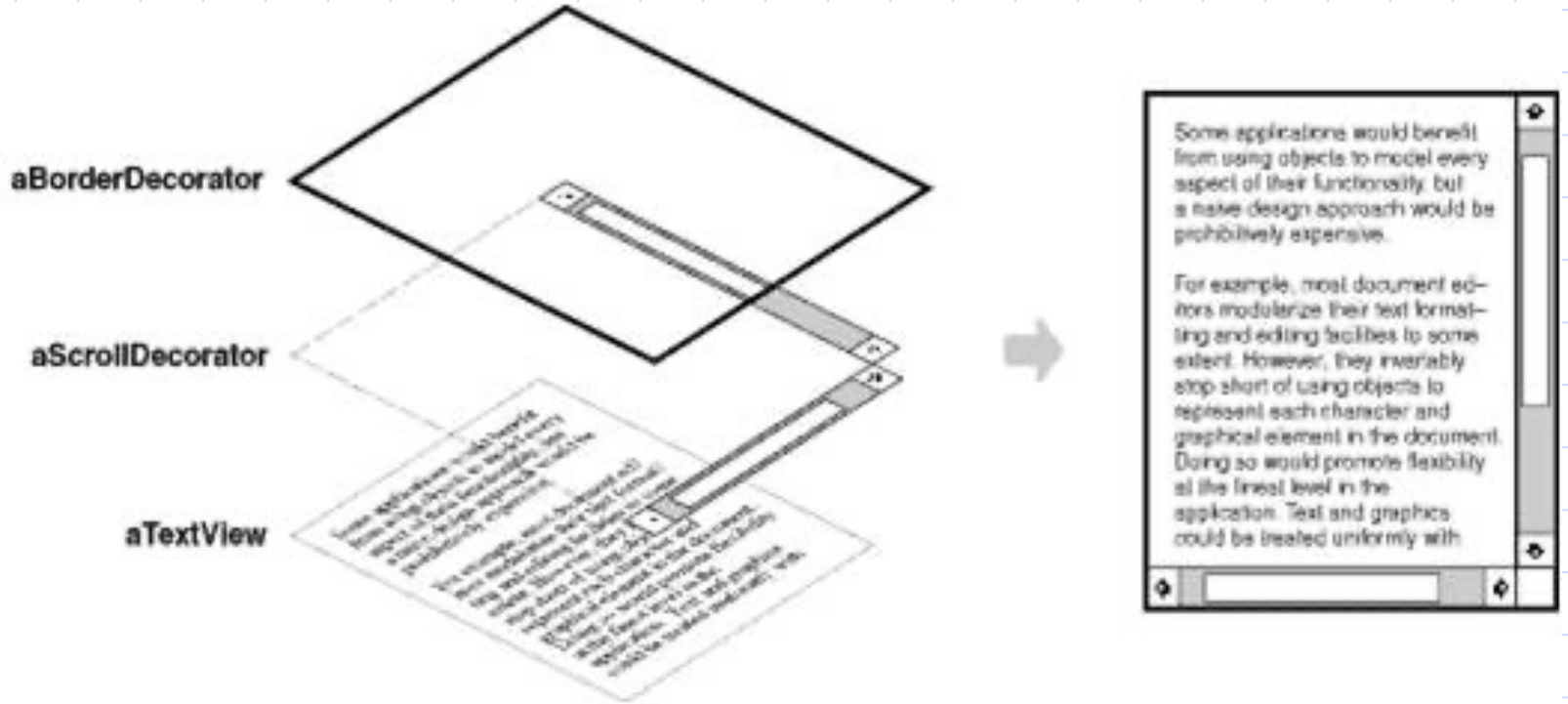
- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing to extend flexibility
- Motivation
  - Want to add properties to an existing object.
  - Examples
    - Add borders or scrollbars to a GUI component
    - Add headers and footers to an advertisement
    - Add stream functionality such as reading a line of input or compressing a file before sending it over the wire

When and Where Can be Used?

- Use Decorator
  - To add responsibilities to individual objects dynamically without affecting other objects
  - When extending classes is impractical
    - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination (this inheritance approach is on the next few slides)

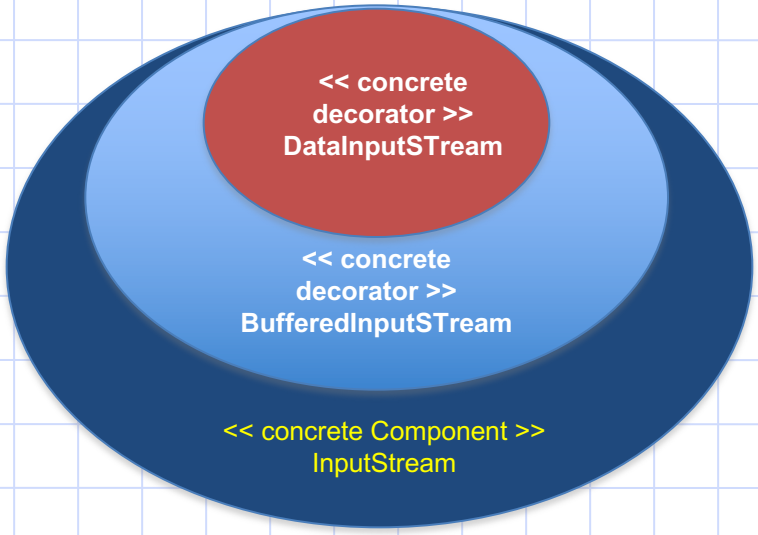
A Very Common Application

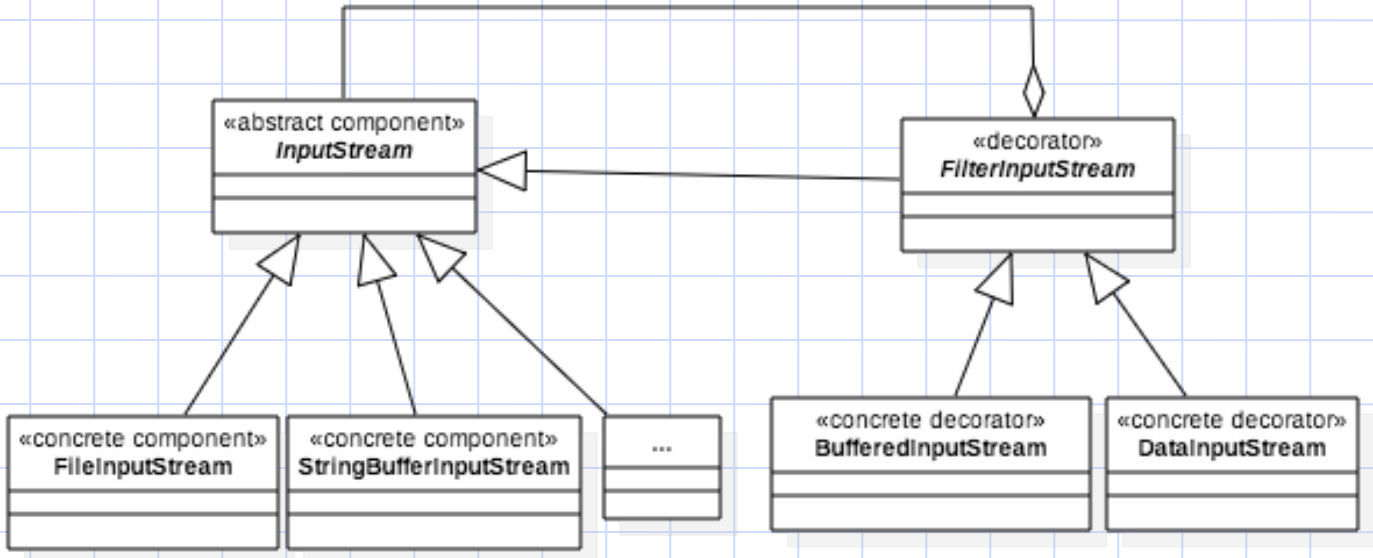
- A TextView GUI component that we want to add different kinds of borders and/or scrollbars to it:



From: https://shishirkumarblog.files.wordpress.com/2011/07/decorator_class_explosion1.jpg

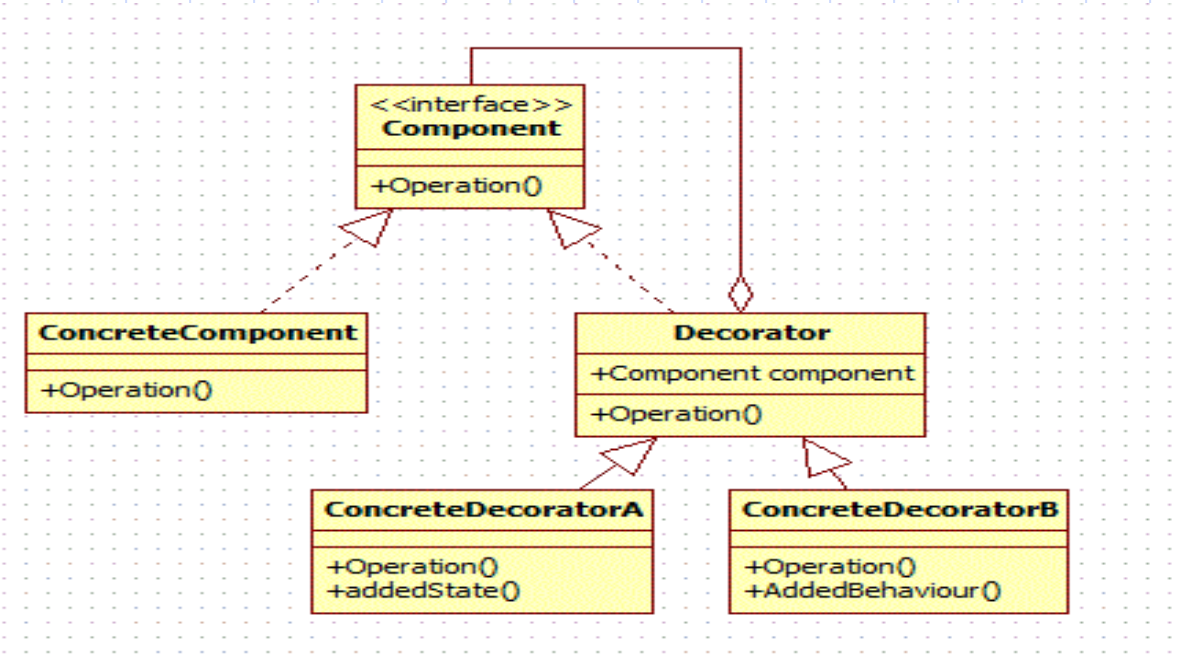
Another Example of decorator pattern used for Java I/O Stream





Definition of Decorator Pattern

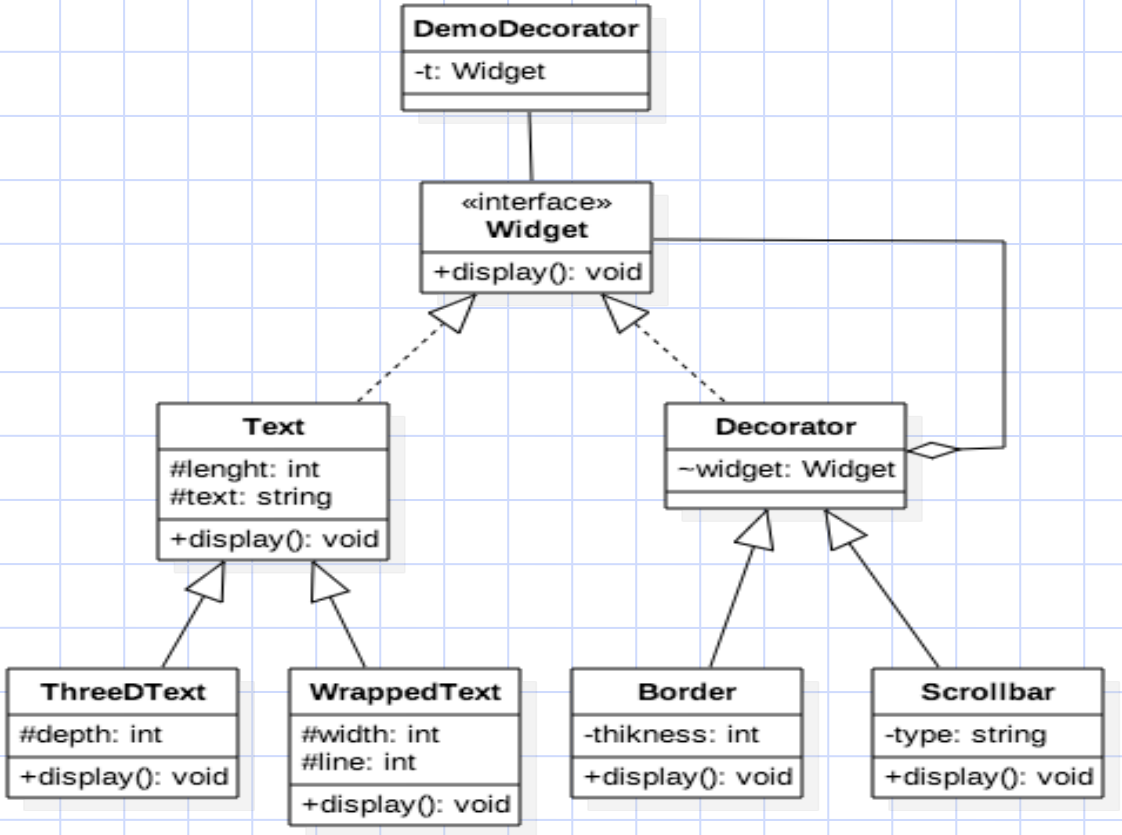
- The Decorator is a **structural** pattern.
- It's used to form structurally complex formed object from many different objects.
- Let's take a look at the following diagram that models the concept of Decorator Pattern:



- Please notice that component interface can be replaced with an abstract class

Decorator Pattern
Learning by Example

- Let's define a set of interface and classes that can be used to develop an application that uses components such as text that can be furnished with additional features (decorators) such as border, scrollbar, or more as needed.



The Java implementation of a simple version of this model will be given and further details will be discussed during lecture

Benefits of Decorator Pattern

- Provides a flexible alternative to sub-classing for extending functionality
- Allows behaviour modification at runtime rather than compilation time
- Difficulty of wide variety of permutation can be solved, and you can wrap a component with any number of decorators.
- Supports the most important principle of reusability and maintainability, which is:
  - classes should be open for extension but closed for modification