Decorator Pattern

The Decorator Pattern from GoF

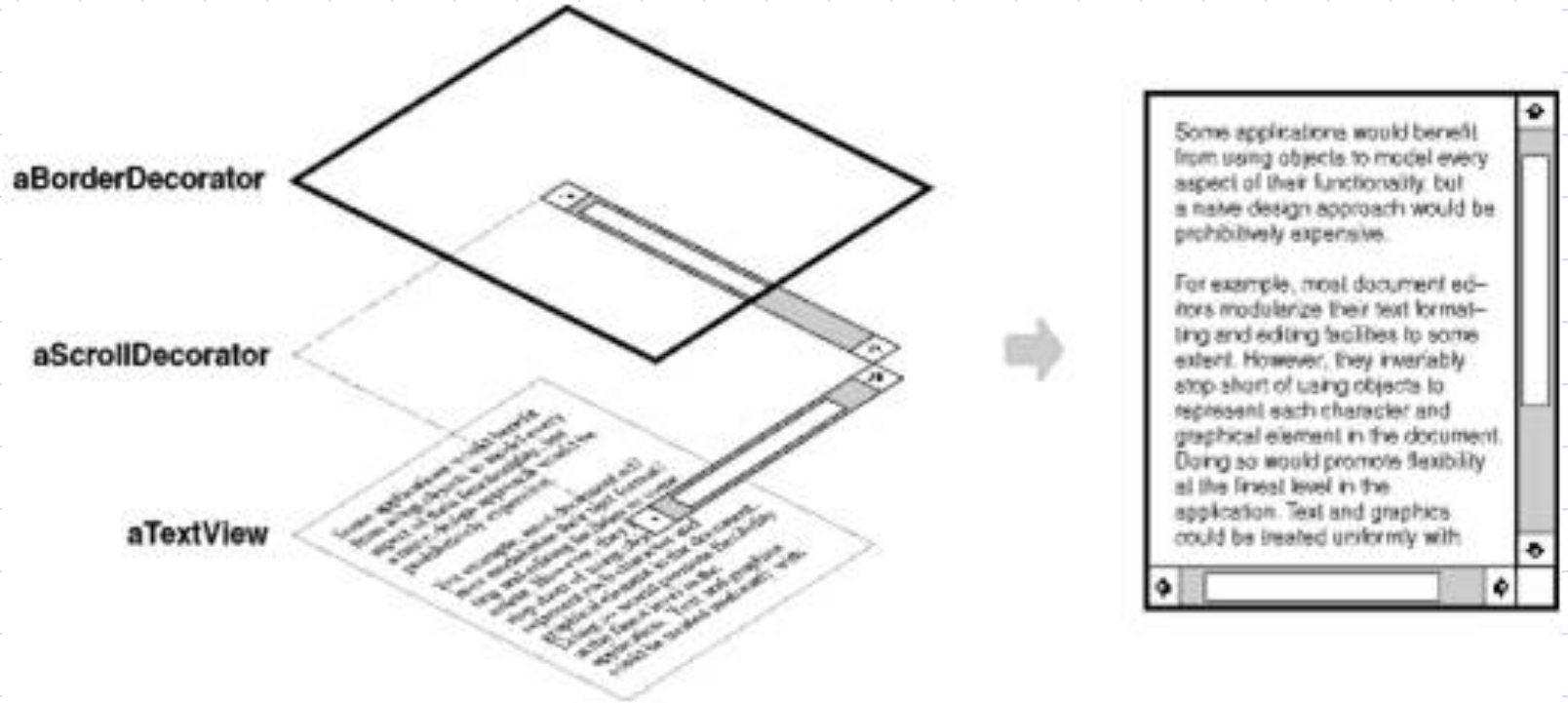
- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing to extend flexibility
- Motivation
  - Want to add properties to an existing object.
  - Examples
    - Add borders or scrollbars to a GUI component
    - Add headers and 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)

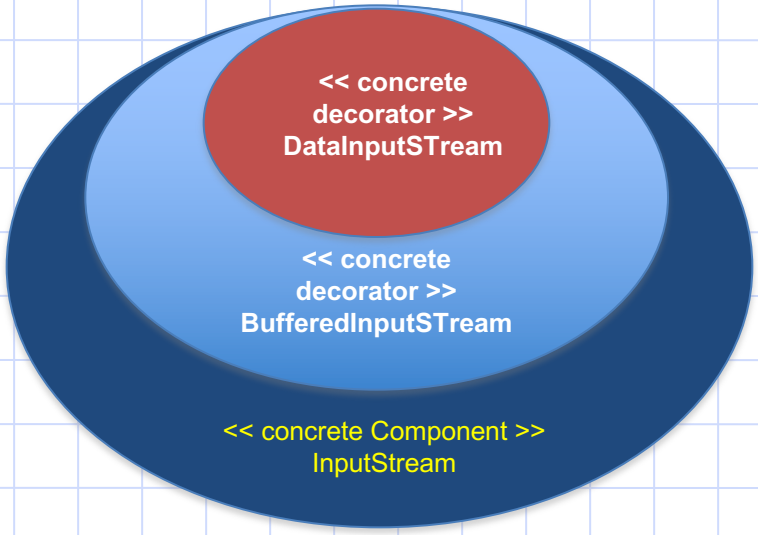
A Very Common Application

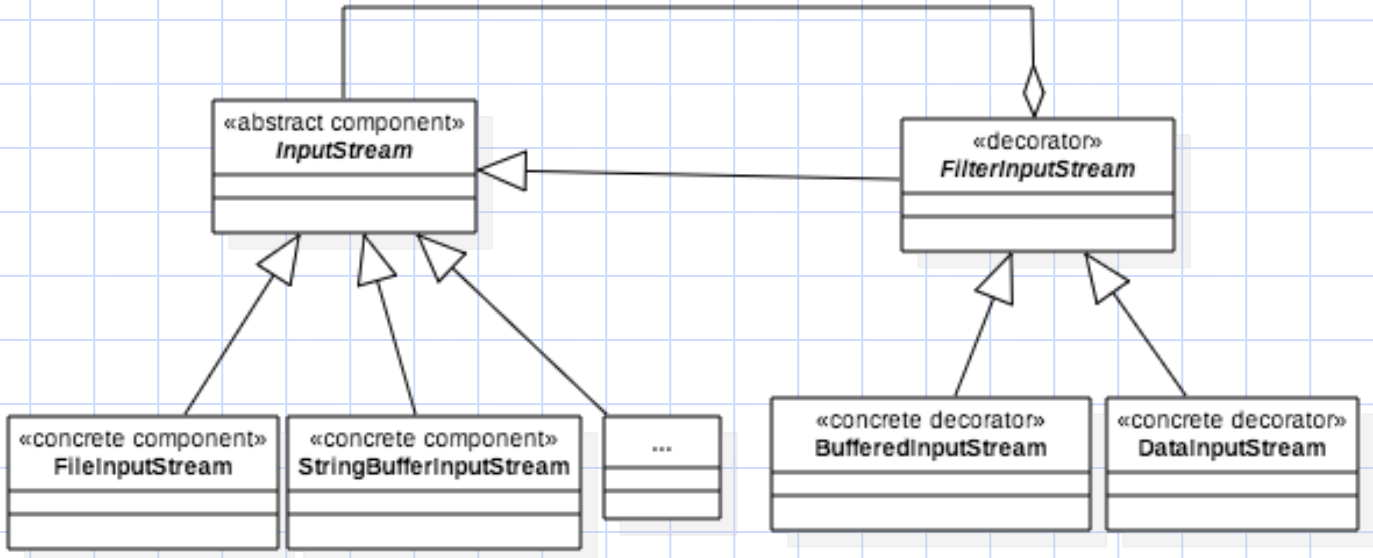
- A TextView GUI component that we want to add different kinds of borders and/or scrollbars to it:



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

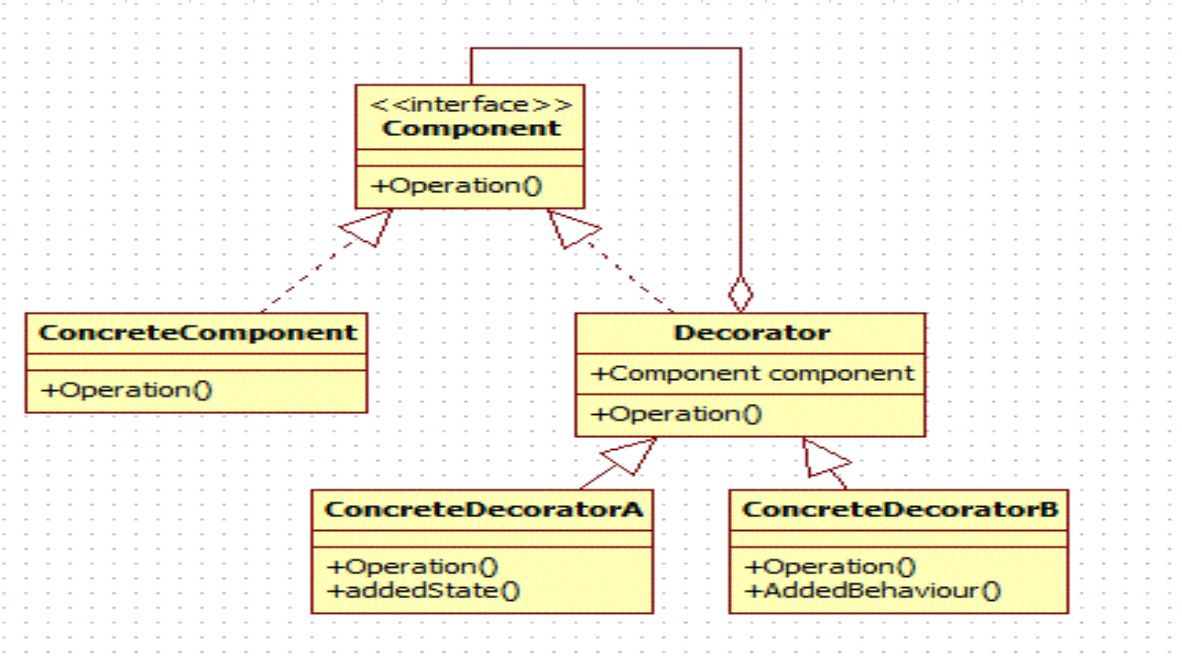
Another Example of decorator pattern used for Java I/O Stream





Definition of Decorator Pattern

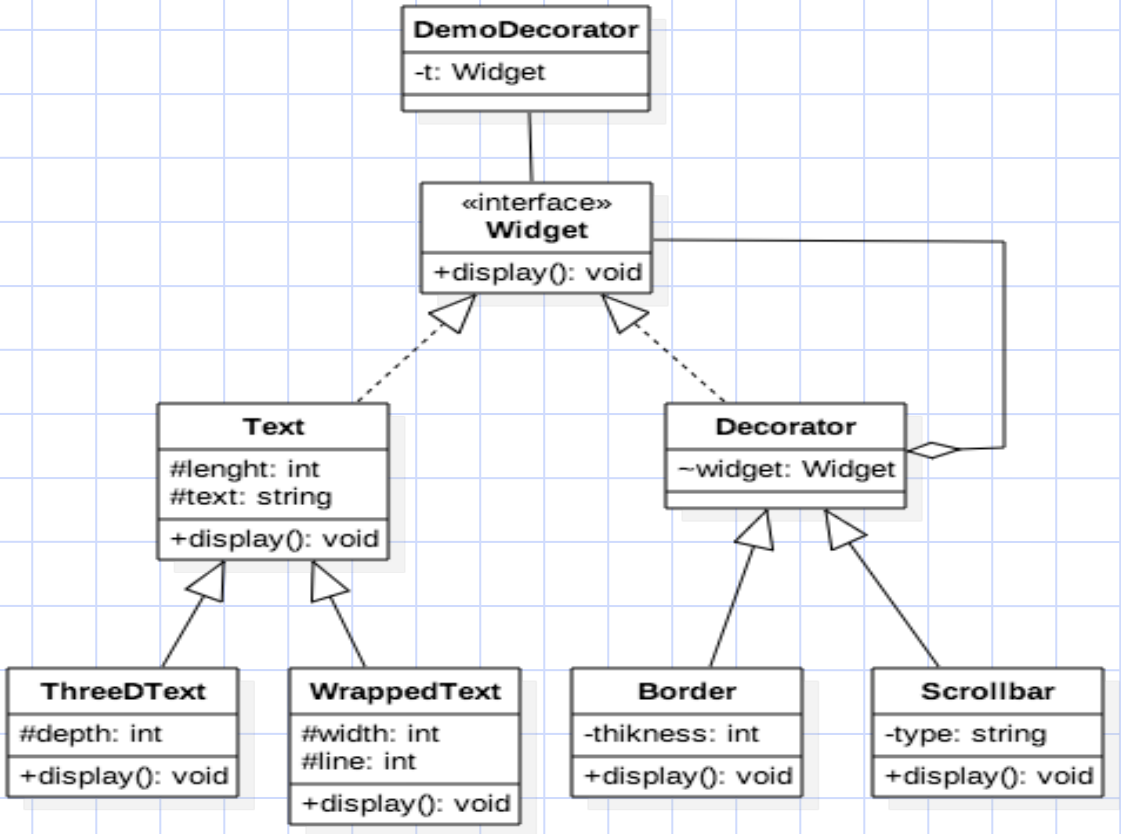
- The Decorator is a **structural** pattern.
- It's used to form structurally complex formed object from many different objects.
- Let's take a look at the following diagram that models the concept of Decorator Pattern:



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

Decorator Pattern 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.



What About Using Inheritance

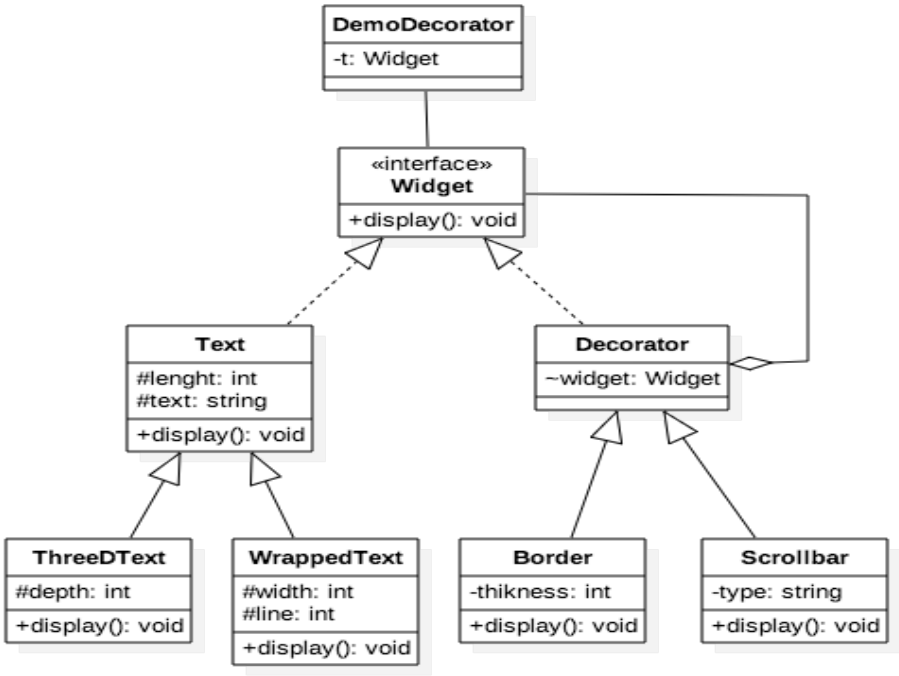
- One possible solution is to use inheritance only to create all possible options of windows such as:
  - Window
  - Window with border
  - Window with vertical scrollbar
  - Window with horizontal scrollbar
  - Window with vertical and horizontal scrollbar
  - Window with vertical and horizontal scrollbar and board
  - Many more …

- But the problem with this solution is that:

  it does not allow the client to select any option that he desires. For example you can not have Window with vertical scrollbar and border. You have to develop another class for this purpose

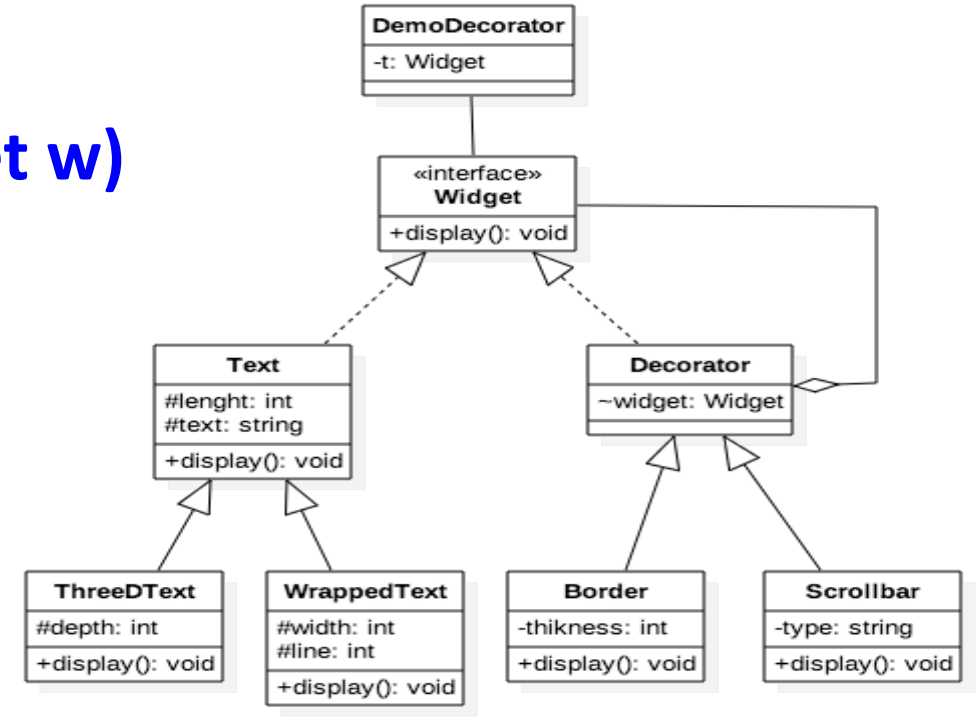
Step 1: Definition of a Component

```
public interface Widget
{
   void display();
}
```



Step 2: Defining an Abstract Decorator

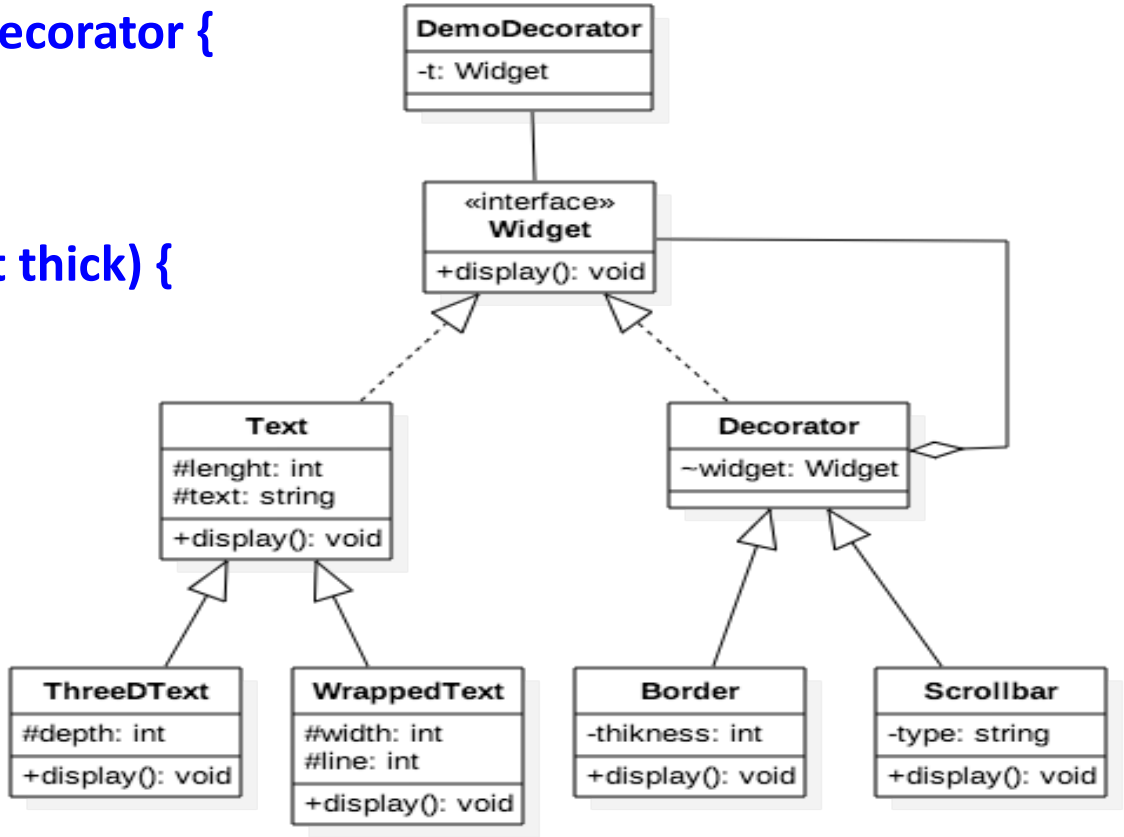
```
abstract class Decorator implements Widget
{
    Widget widget;
    public Decorator(Widget w)
    {
      widget = w;
    }
}
```



Step 3: Defining a Concrete Decorator

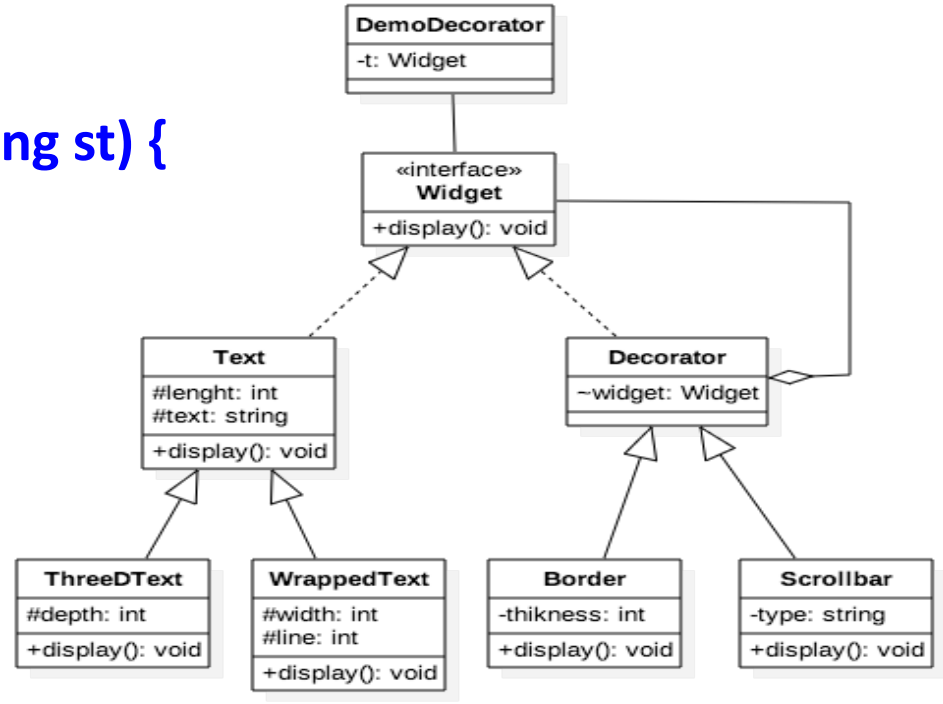
- Now Let's build a concrete Decorator

```
public class Border extends Decorator {
    private int thickness;

   public Border(Widget w, int thick) {
        super(w);
        thickness = thick;
   }

   @Override
   public void display() {
        widget.display();
        System.out.print(". It's border thickness is: " + thickness);
   }
}
```



Steps 4: Creating More Decorators

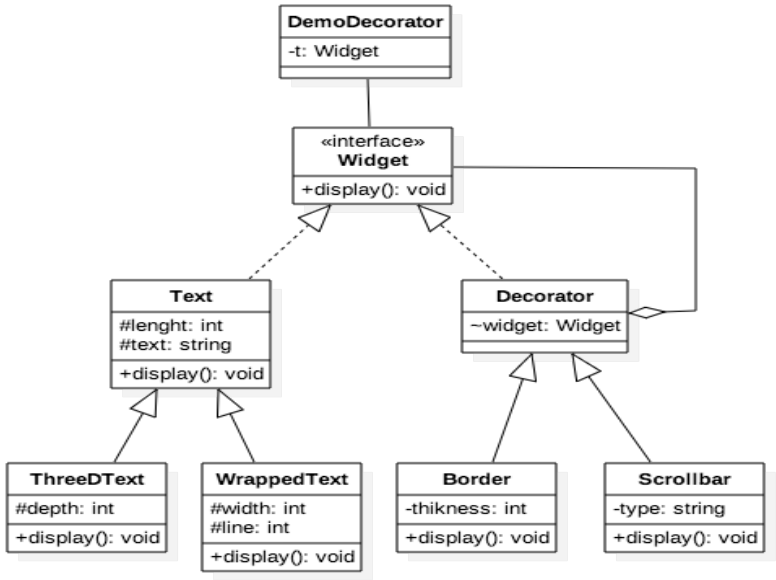
```
public class Scrollbar extends Decorator {
    private String type;

    public Scrollbar(Widget w, String st) {
        super(w);
        type = st;
    }

    @Override
    public void display() {
        widget.display();
        System.out.print(". Its scrollbar type is: " + type );
    }
}
```



Step 5: Let's create one or more concrete Components

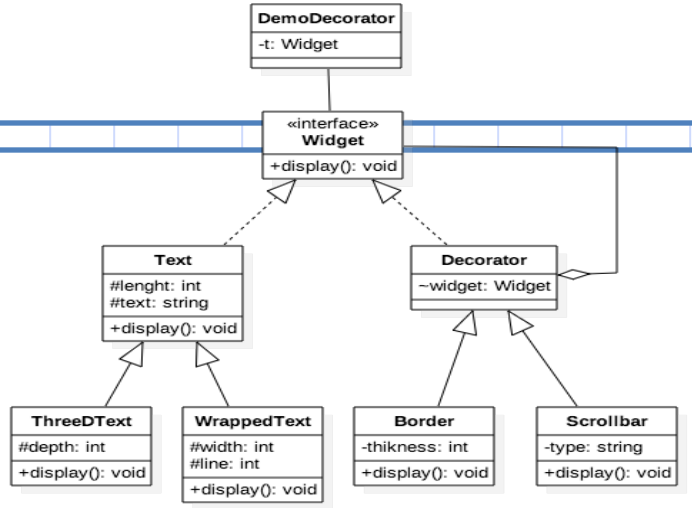
We start by a concrete component that creates some text on the screen:

```
class Text implements Widget {
  protected int length;
  String text;
  public Text( String s) {
      text  =  s;
      length = text.length();
}
  public void display() {
      System.out.print("This is a plain text: " + text + ", and its length is: " + length);
  }
}
```



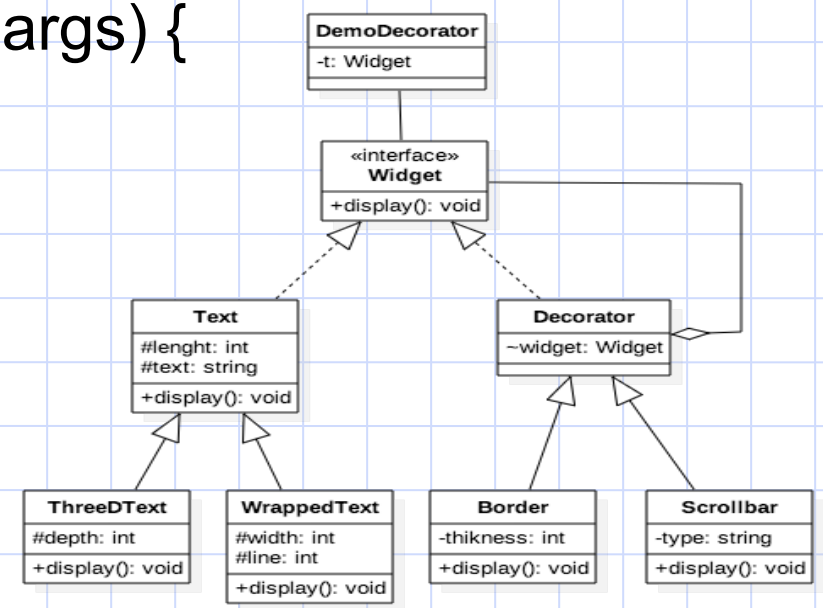
Step 6: more components

```
public class WrappedText extends Text {
        private int width, lines;
        public WrappedText (String s, int w, int h) {
                super(s);
                width = w;
                lines = length / width;
        }
         public void display() {
                System.out.print("This is a wrapped Text: " + text + ", and its length is: "
                                + length + " Its width is " + width + " and its height is: " + lines);
        }
}
```




```
public class ThreeDText extends Text {
  protected int depth;
  public ThreeDText(String s, int d){
        super(s);   depth = d;
   }
 public void display() {
        System.out.print("3-D text, " + length + "character long " + depth + "pixel depth");
   }
}
```

```
public class DemoDecorator {
    public static void main(String[] args) {
        Widget t = new Text ("TXT");
        t.display();
        System.out.println();
        t = new Border(t, 2);
        t.display();
        System.out.println();
        t = new Scrollbar(t, "vertical");
        t.display();

        // ASSUME MORE CODE TO ADD MORE DECORATORS
    }
}
```



Sample Output

This is a plain text: TXT, and its length is: 3

This is a plain text: TXT, and its length is: 3. It's border thickness is: 2

This is a plain text: TXT, and its length is: 3. It's border thickness is: 2. Its scrollbar type is: vertical

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

Singleton Pattern

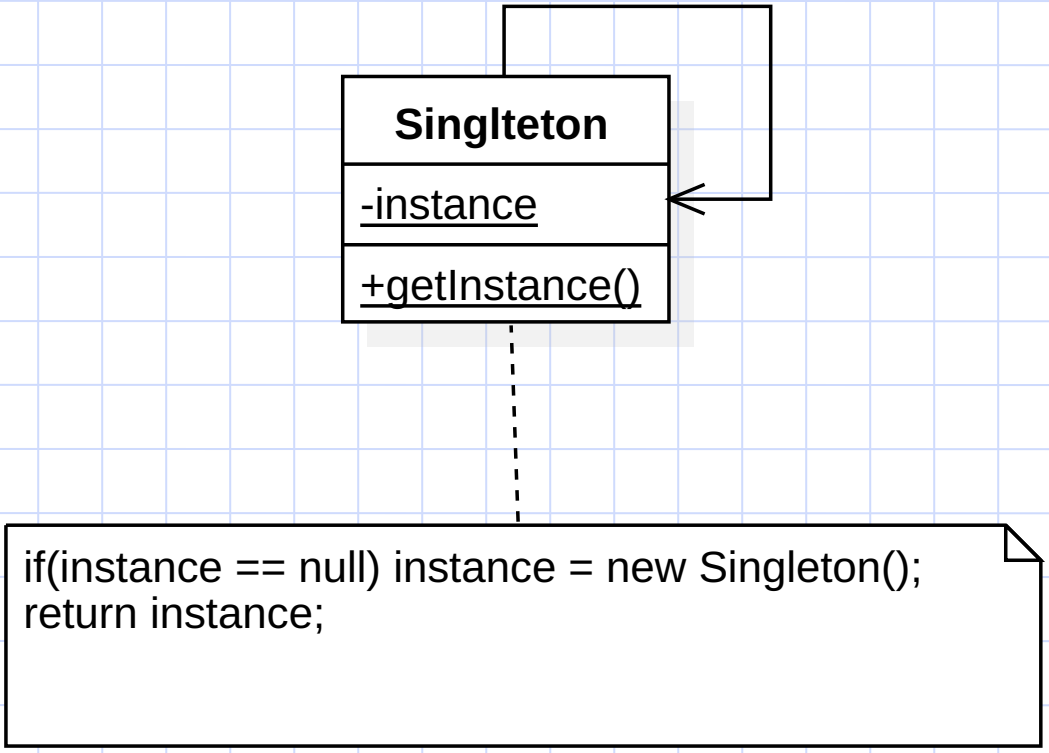
What is Singleton Pattern

- It is highly desirable if we can use some Design Pattern to control the access to that shared resource.
  - A good example is the login process
  - Another example is debugging the shared sources

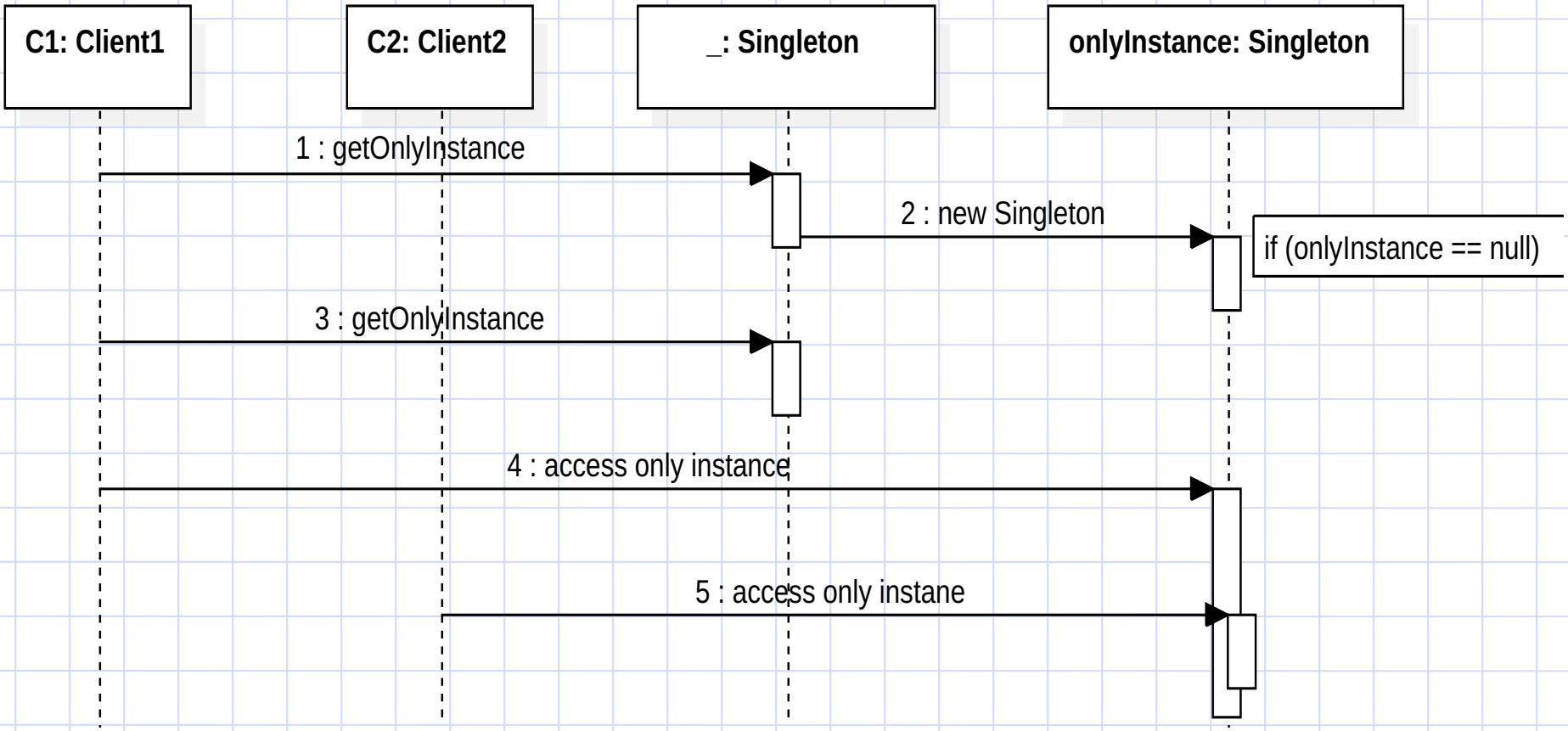
Singleton Pattern

- User shouldn't be allowed to create instances of the Singleton object.
- Need to have a private class-data-field of the Singleton.
- Need to have a public class-method to have access to private class-data-field

Singleton Pattern



Singleton Pattern Interactions



Singleton Design Pattern Example

Step 1: create Singleton Class

```
public class Singleton {
    private static Singleton onlyInstance;
    private ArrayList<String >usernameList;
    private ArrayList<String> passwordList;
    private ArrayList<String> nameList;
    // MORE CODE
}
```

Step 2: create instance

```
class Singleton {
        …
        …
      private Singleton(){
            usernameList = new ArrayList<String>();
            passwordList = new ArrayList<String>();
            nameList = new ArrayList<String>();
      }

      public static Singleton getOnlyInstance() {
            if(onlyInstance == null)
                  onlyInstance = new Singleton();
            return onlyInstance;
      }
        …
        …
}
```

Step 3: getter, setter, updaters, …

```
class Singleton {
        …
        …
          public static void setOnlyInstance(Singleton onlyInstance) {
                SingletonLogin.onlyInstance = onlyInstance;
        }

        public  void addUsername(String username) {
                usernameList.add(username);
        }

        public void setUsename(int index, String newUsername){
                usernameList.set(index, newUsername);
        }

        public void removeUsename(int index, String newUsername){

        }


}
```

Sep 4: using Singleton Patten

```
public class DemoSingletoPattern {

    public static void main(String[] args) {

        Singleton c1 = Singleton.getOnlyInstance();
        c1.addName("Jack Lemon");
        c1.addUsername("jlemon");
        c1.addPassword("jl1234");

        Singleton c2 = Singleton.getOnlyInstance();
        c2.addName("Merry Leu");
        c2.addUsername(mleu);
         c2.addPassword("orange1234);

    }
}
```

Benefits and Drawbacks of Singleton

- Singleton is a famous pattern as its known as simplest to be learned.
- It is useful for using a single copy of the shared resource.
- Drawbacks include:
  - Singleton classes cannot be sub classed.
  - Reduces testing flexibilities:
    - A good design advice is to minimize dependencies between classes. Particularly during unit testing this advice is very helpful.
    - With a singleton pattern this feature will be scarified. Because the object creation part is hidden, we cannot expect the singleton constructor to accept any parameters.