Decorator Pattern

Prof. Jonathan Lee (李允中)
Department of CSIE
National Taiwan University

Design Aspect of Decorator

Responsibilities of an object without subclassing

Outline

- FileViewer Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Decorator Pattern Structure
- NTU Coffee Shop: Another Example
- Homework

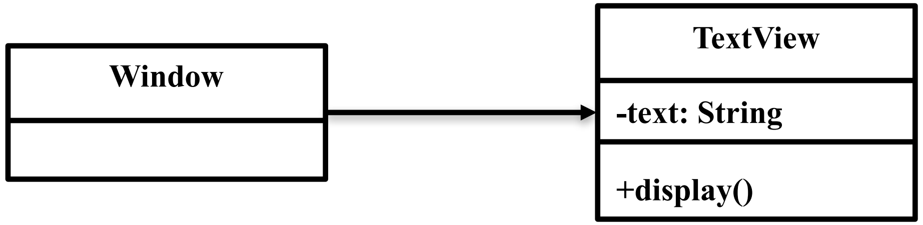
FileViewer (Decorator)

Prof. Jonathan Lee (李允中)

Department of Computer Science and
Information Engineering
National Taiwan University

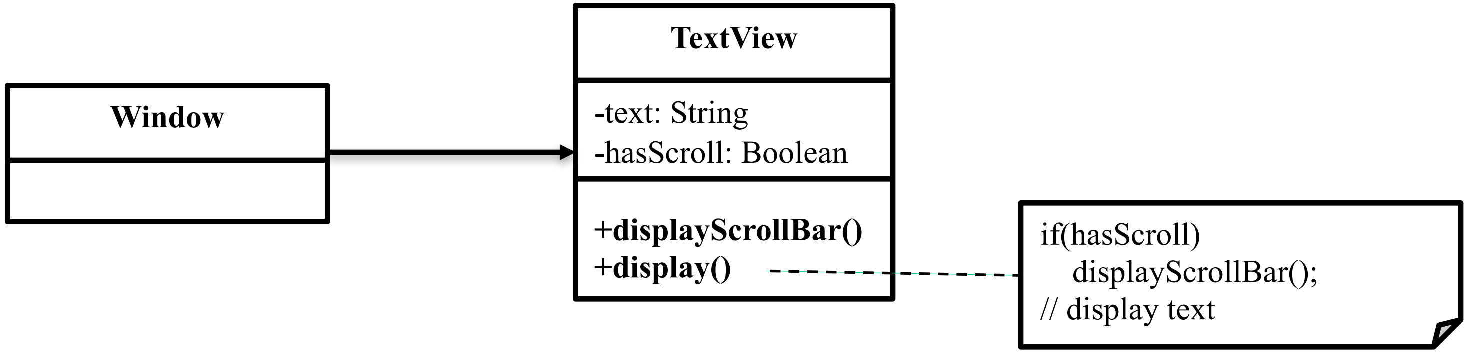
Requirements Statements$_1$

- In FileViewer,
  - We have a TextView object that displays text in a window.



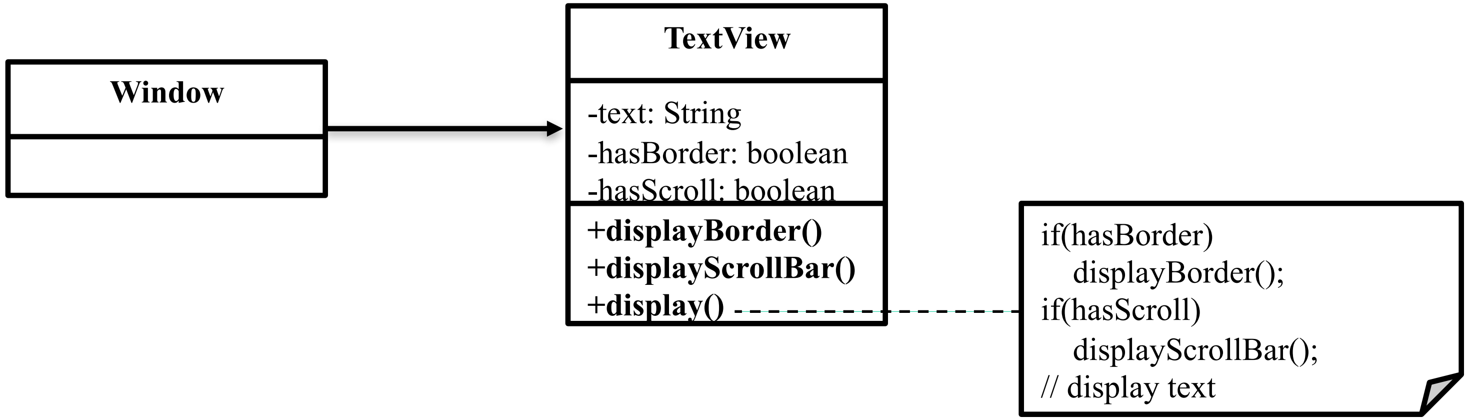
Requirements Statements$_2$

- In FileViewer,
  - TextView has no scroll bars by default, because we might not always need them.



Requirements Statements$_3$

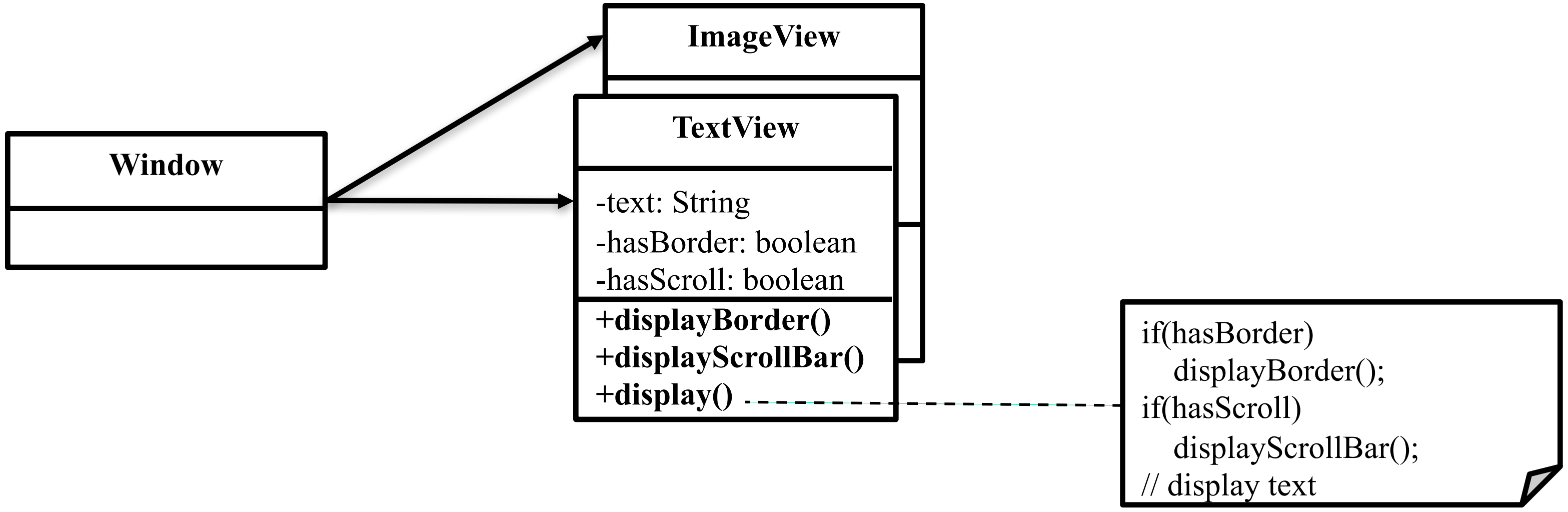
- In FileViewer,
  - We can also add a thick black border around the TextView.



```
if(hasBorder)
   displayBorder();
if(hasScroll)
   displayScrollBar();
// display text
```

Requirements Statements$_4$

- In FileViewer,
  - It is highly likely that we will support various file formats (views) for display in the future.



Window

ImageView

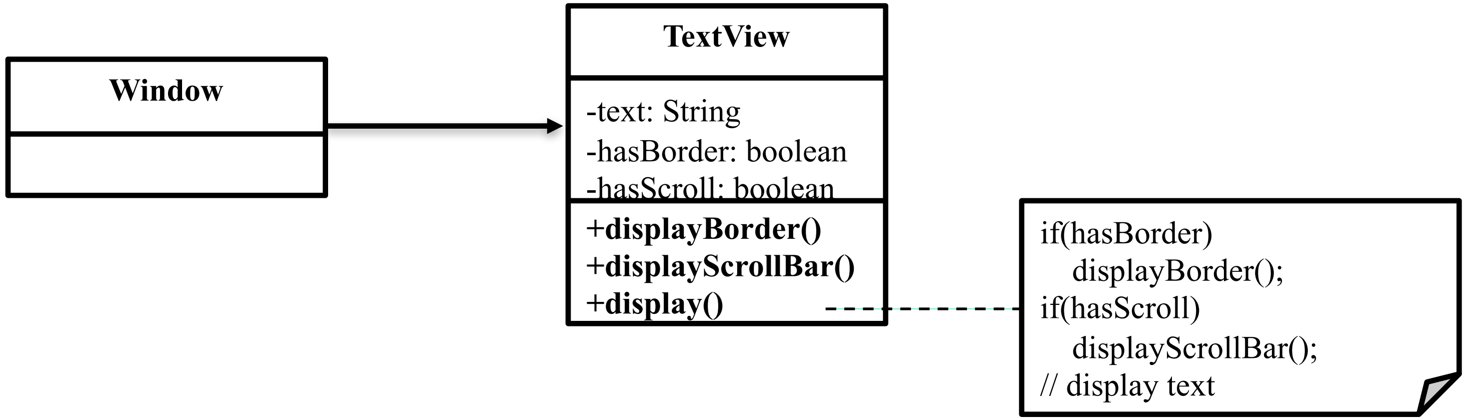
TextView

-text: String
-hasBorder: boolean
-hasScroll: boolean

+displayBorder()
+displayScrollBar()
+display()

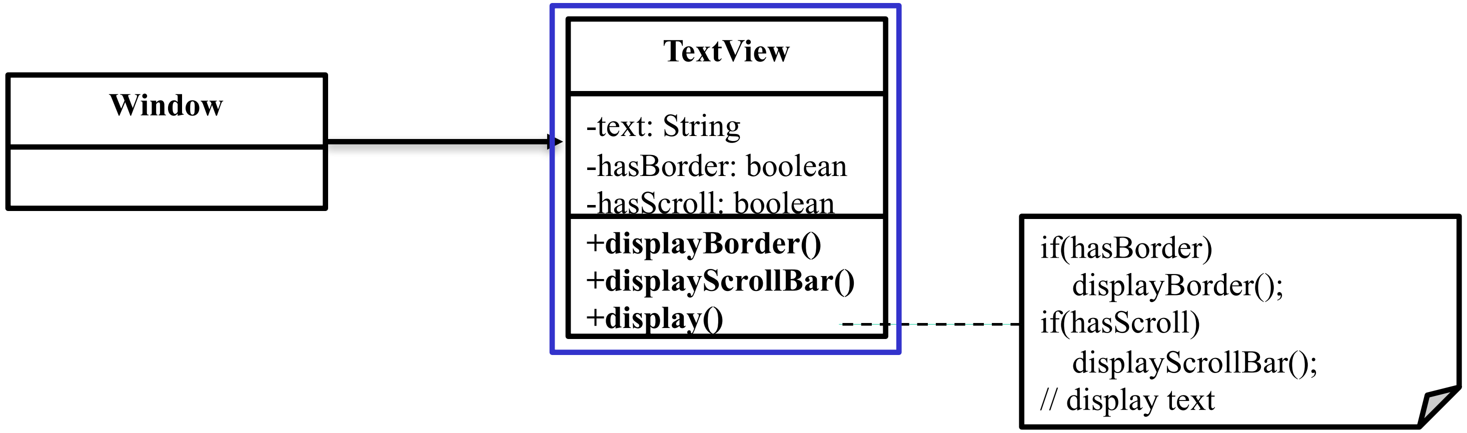
```
if(hasBorder)
    displayBorder();
if(hasScroll)
    displayScrollBar();
// display text
```

Initial Design



```
if(hasBorder)
   displayBorder();
if(hasScroll)
   displayScrollBar();
// display text
```

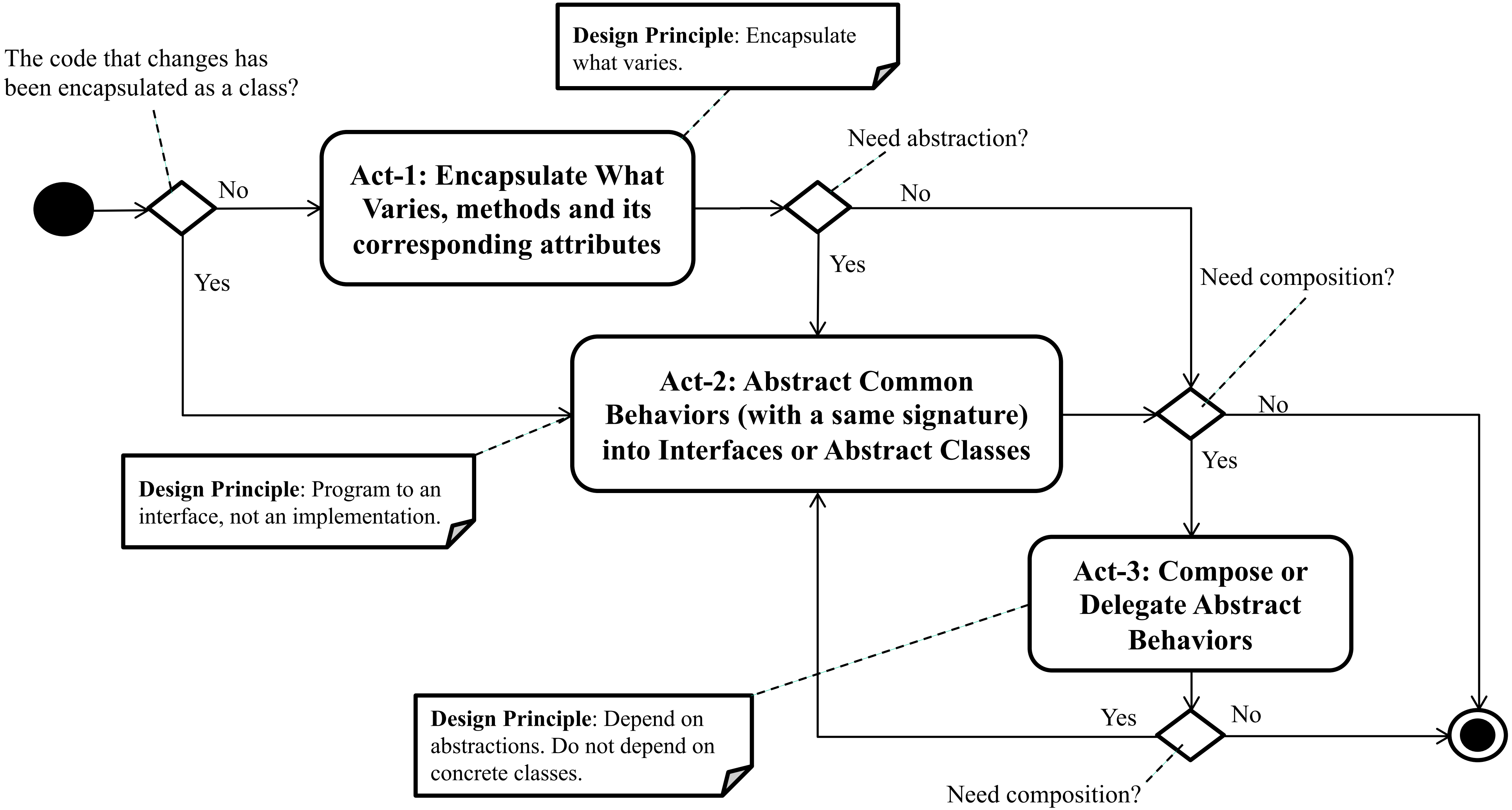
Problems with Initial Design



Problem 1: If we want not only scroll bars or thick borders but many other UI components, such as toolbar, we need re-open TextView for modification to meet the new requirement.

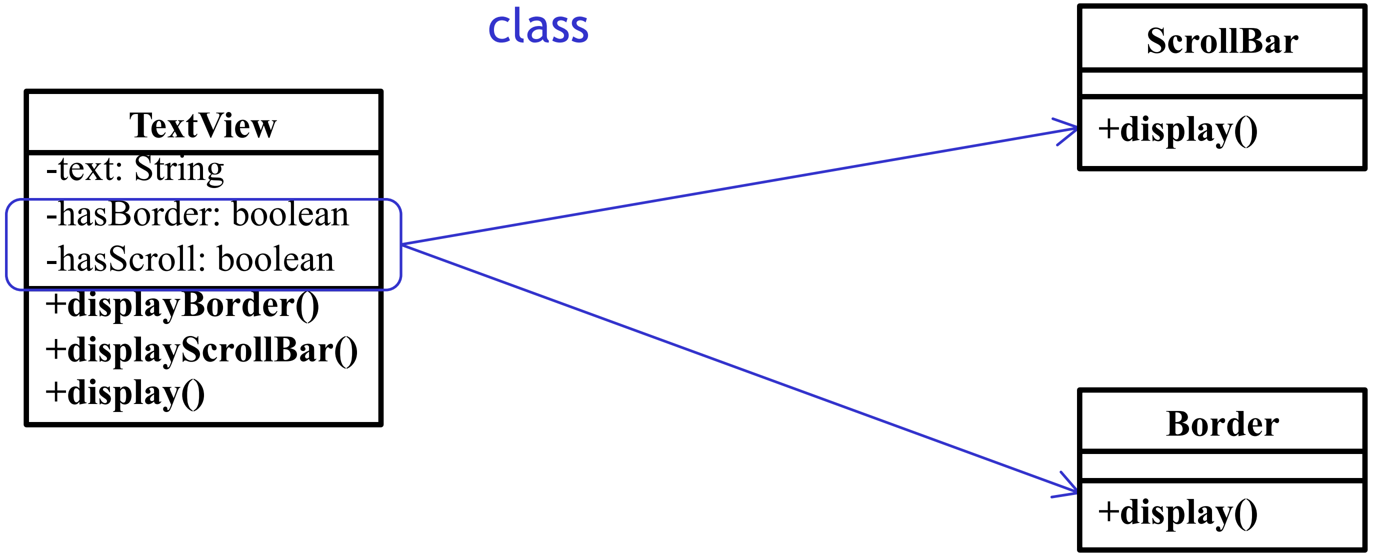
Problem 2: At a later time, if we want to support various kinds of file formats, like image, we need to duplicate drawBorder() and drawScrollBar().

Design Process for Change



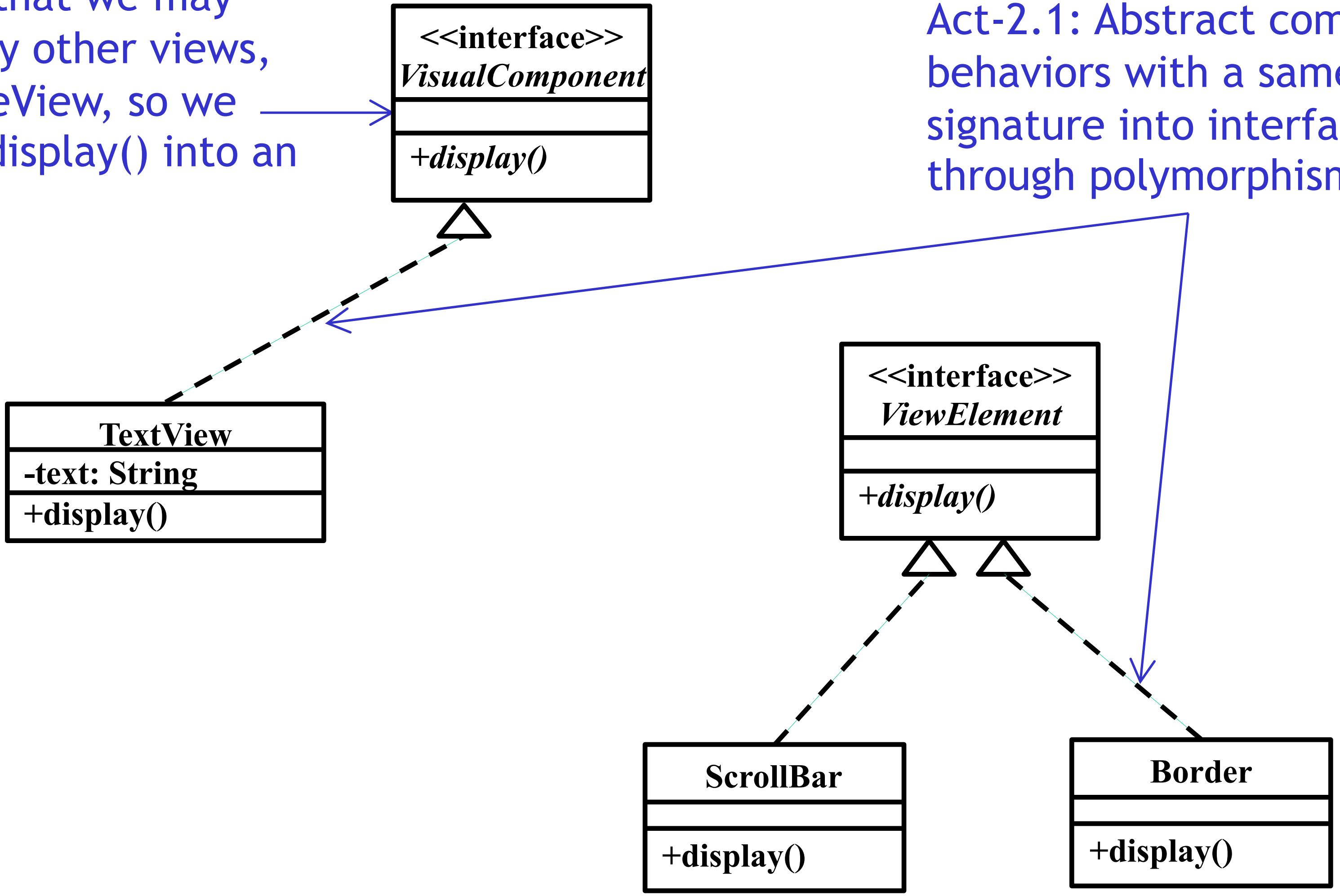
The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Design Principle: Encapsulate what varies.
Need abstraction?
No
Yes
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
Design Principle: Program to an interface, not an implementation.
Need composition?
No
Yes
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

Act-1: Encapsulate What Varies



Act-1.1: Encapsulate an attribute into a concrete class

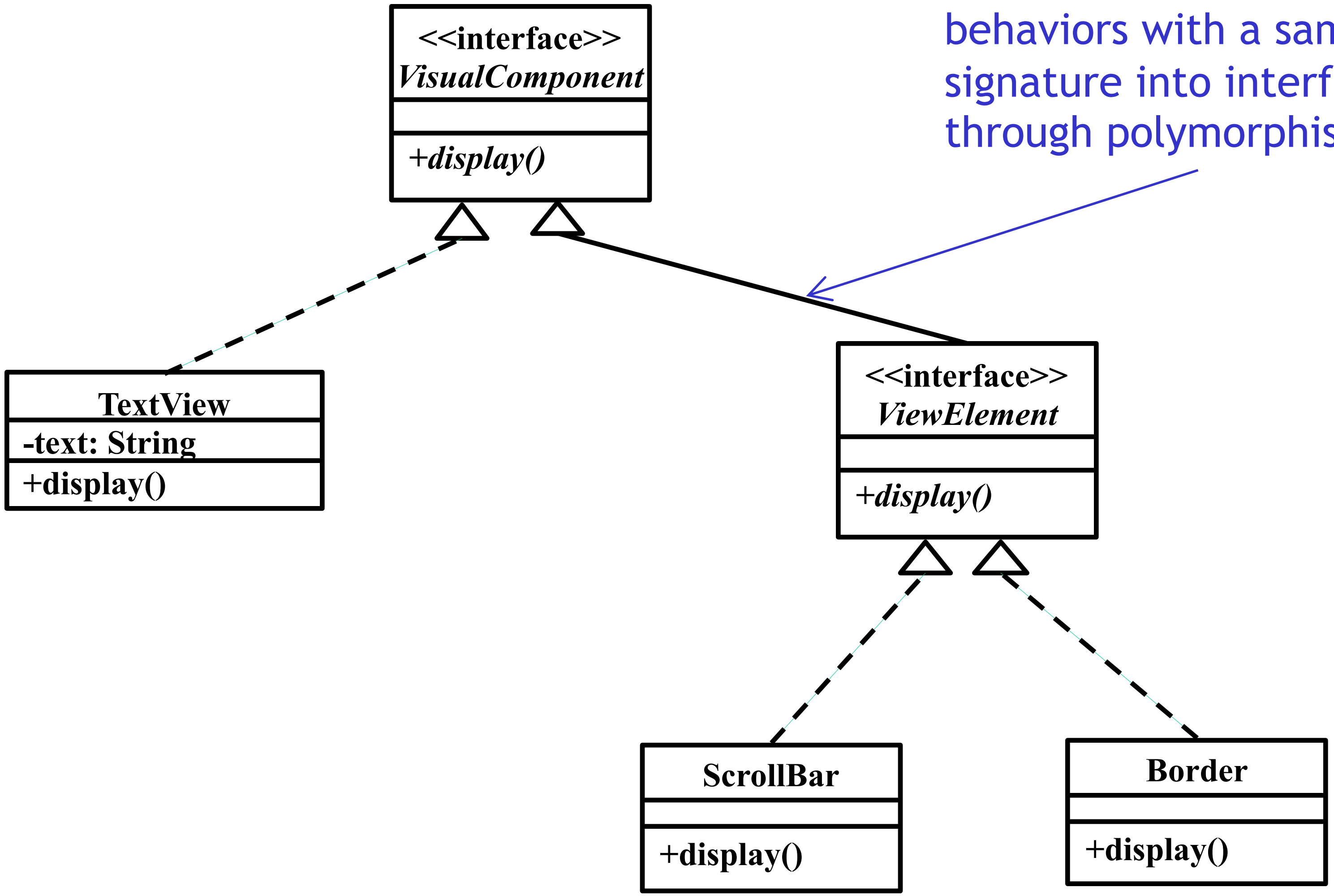
Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes



Consider that we may have many other views, like ImageView, so we abstract display() into an interface.

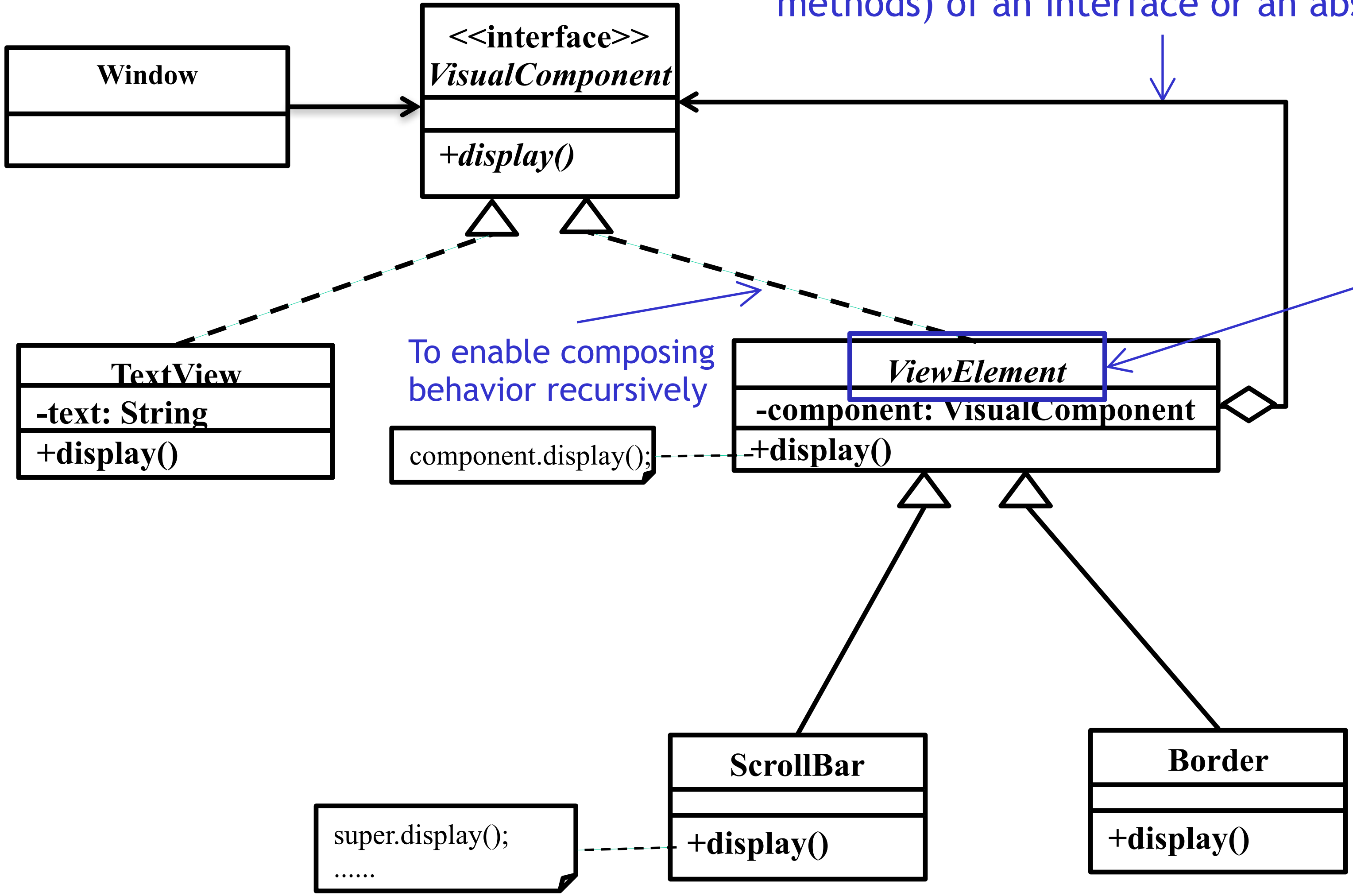
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-2: Abstract Common Behaviors into Interfaces/Abstract Classes



Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-3: Compose Abstract Behaviors

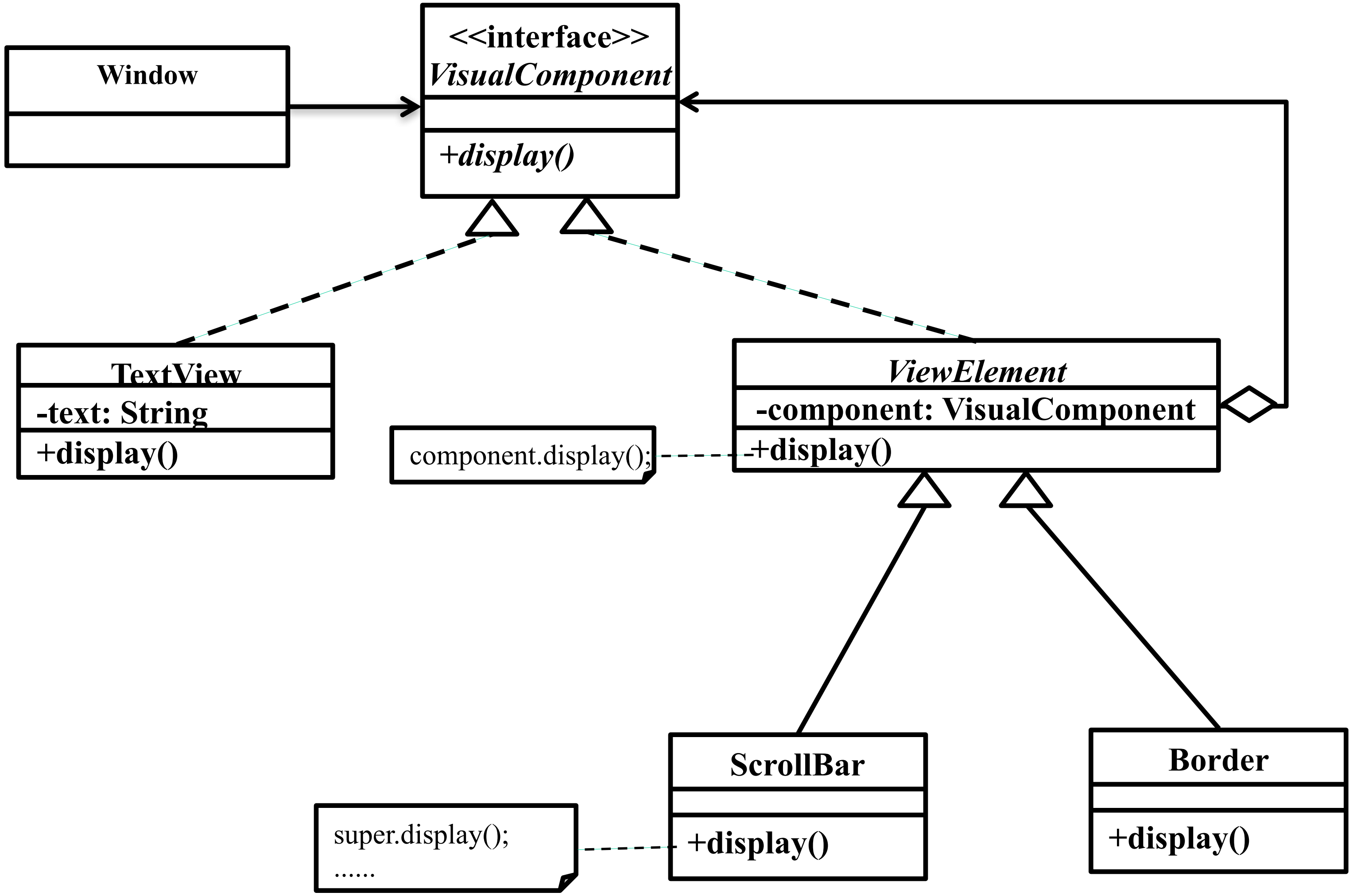


Act-3.1: Compose behaviors (multiple methods) of an interface or an abstract class

To enable composing behavior recursively

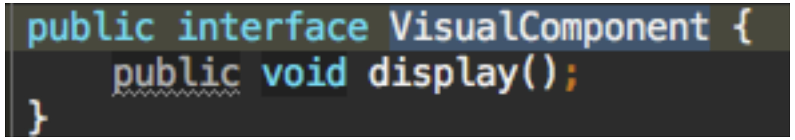
The interface is changed into an abstract class in order to keep the composition relation with the visual component attribute (hold reference to)

Refactored Design after Design Process



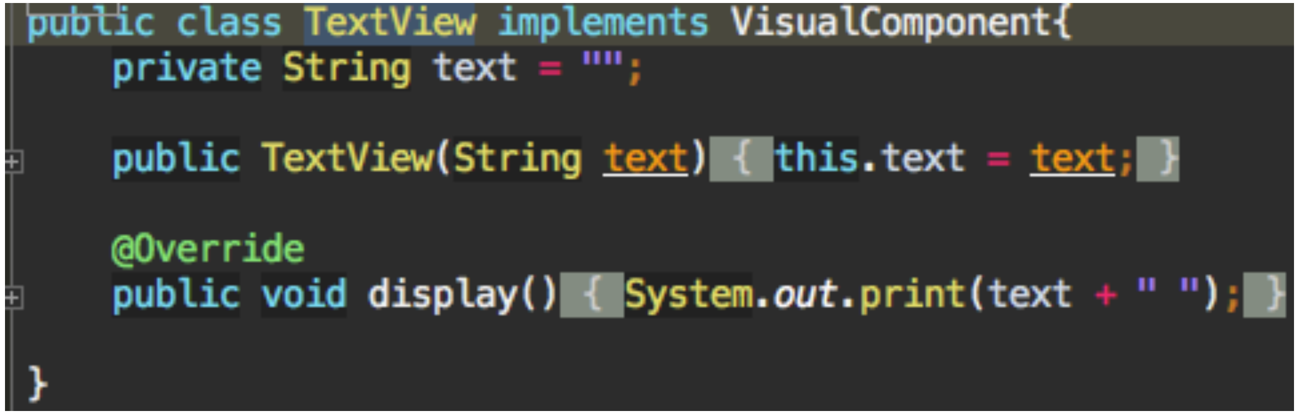
Source code

Visualcomponent Class



```java
public interface VisualComponent {
    public void display();
}
```

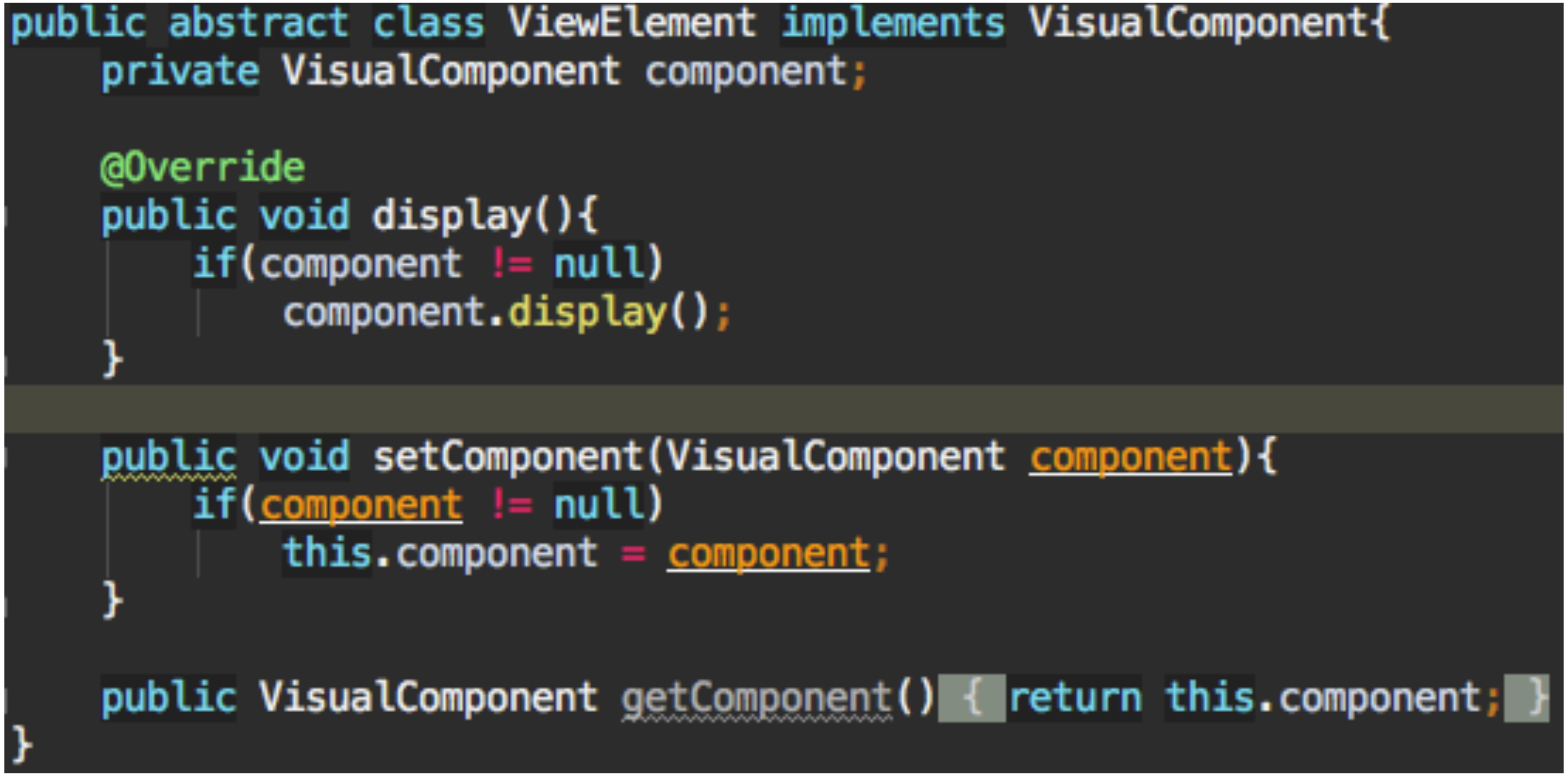
TextView Class



```java
public class TextView implements VisualComponent{
    private String text = "";

    public TextView(String text) { this.text = text; }

    @Override
    public void display() { System.out.print(text + " "); }

}
```

Source code

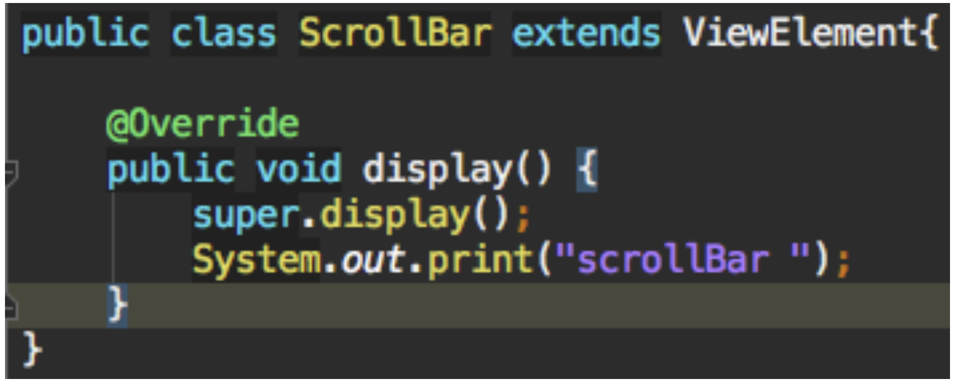
ViewElement Class



```java
public abstract class ViewElement implements VisualComponent{
    private VisualComponent component;

    @Override
    public void display(){
        if(component != null)
            component.display();
    }

    public void setComponent(VisualComponent component){
        if(component != null)
            this.component = component;
    }

    public VisualComponent getComponent() { return this.component; }
}
```

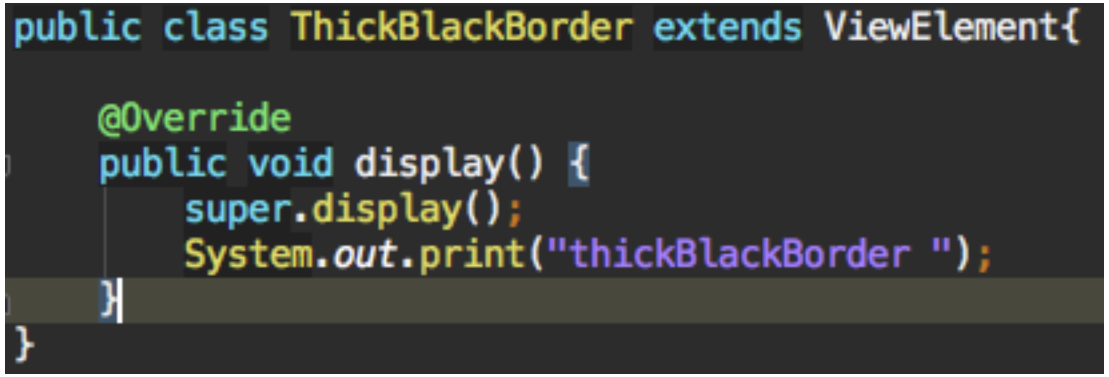
Source code

ScrollBar Class



```
public class ScrollBar extends ViewElement{

    @Override
    public void display() {
        super.display();
        System.out.print("scrollBar ");
    }
}
```

ThickblackBorder Class



```
public class ThickBlackBorder extends ViewElement{

    @Override
    public void display() {
        super.display();
        System.out.print("thickBlackBorder ");
    }
}
```

Input

- [TextView_name] [TextView_name's text]
- [TextView_name] add [view_element] …
- [TextView_name] display

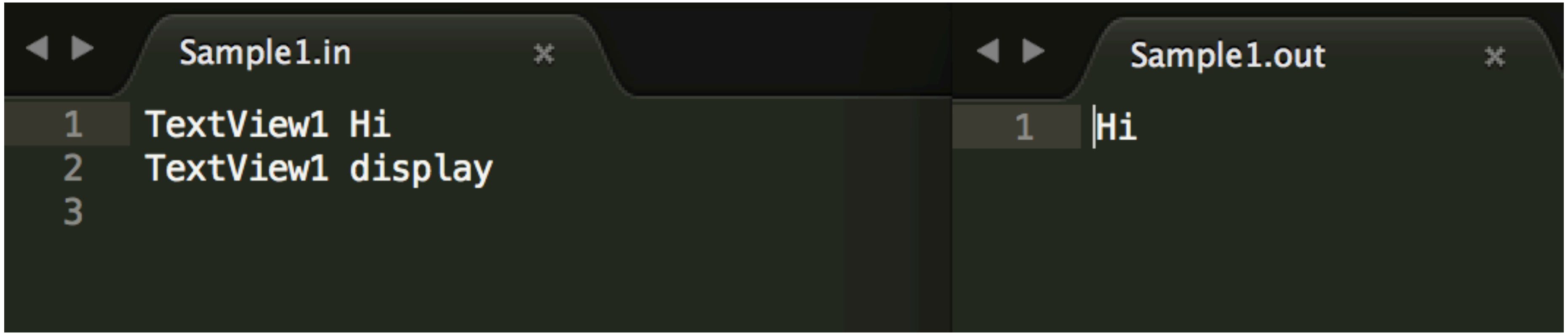
Output

- [TextView_name’s text] [view_element]…

Test cases

- TestCase 1: Only TextView
- TestCase 2: Add with ScrollBar
- TestCase 3: Add with ThickBlackBorder
- TestCase 4: Add with ScrollBar and ThickBlackBorder
- TestCase 5: More than one TextView
- TestCase 6: Display before add ViewComponent

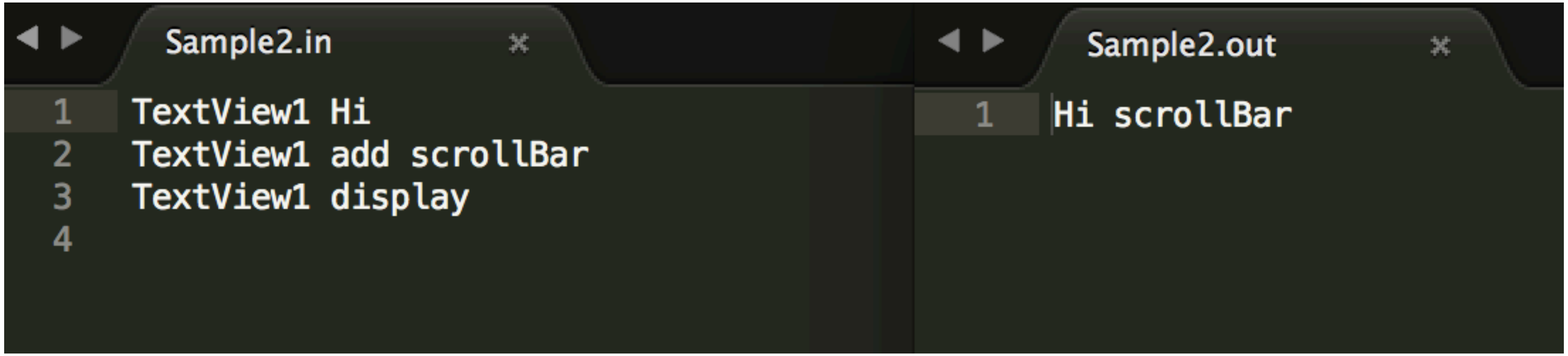
Test case1



```
TextView1 Hi
TextView1 display

```

```
Hi
```

Test case2



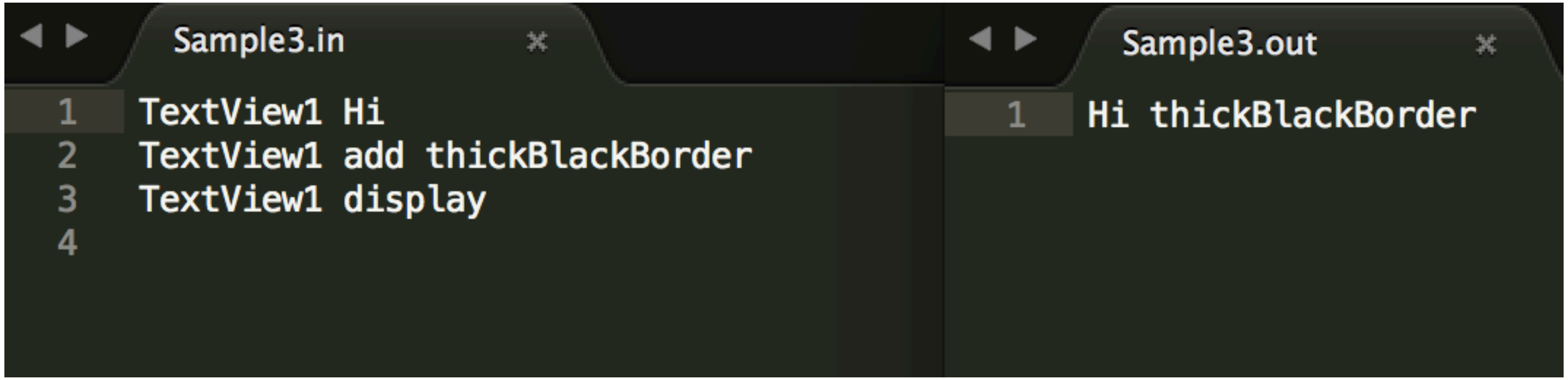
Sample2.in

```
TextView1 Hi
TextView1 add scrollBar
TextView1 display

```

Sample2.out

```
Hi scrollBar
```

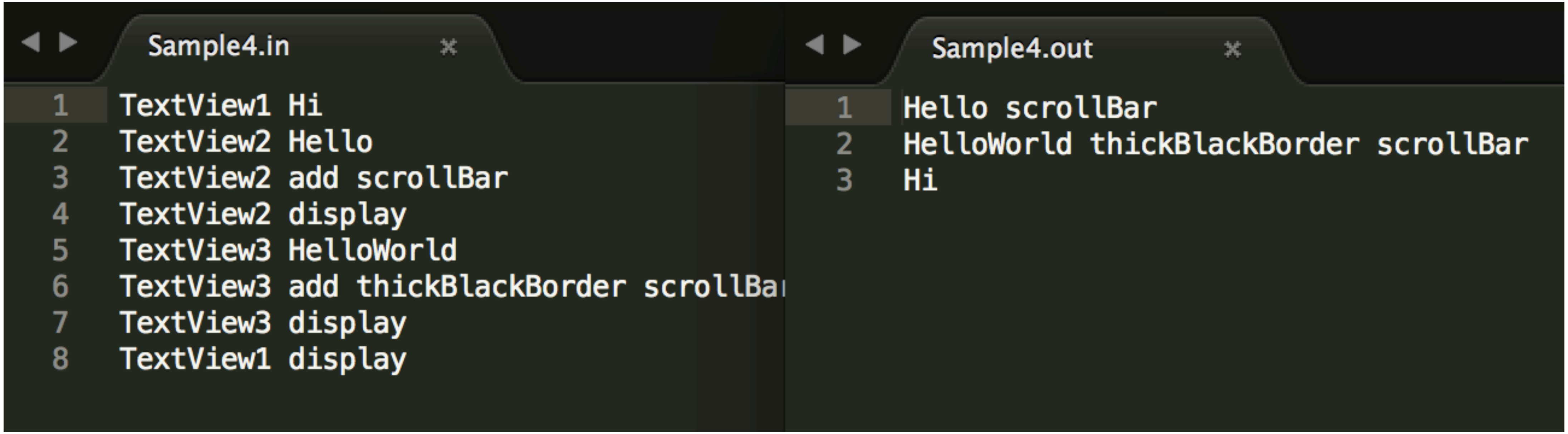
Test case3



```
TextView1 Hi
TextView1 add thickBlackBorder
TextView1 display

```

```
Hi thickBlackBorder
```

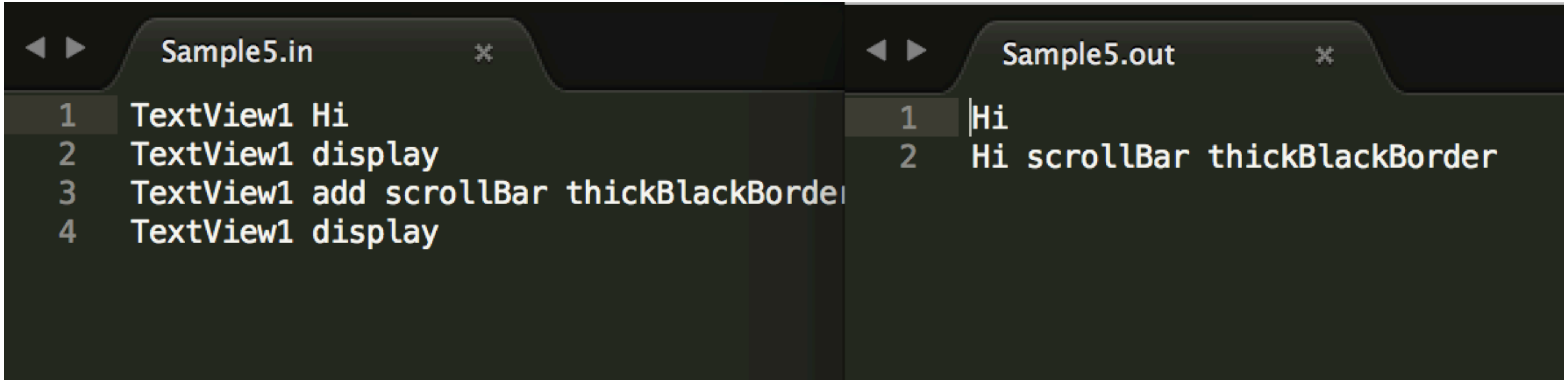
Test case4



```
TextView1 Hi
TextView2 Hello
TextView2 add scrollBar
TextView2 display
TextView3 HelloWorld
TextView3 add thickBlackBorder scrollBar
TextView3 display
TextView1 display
```

```
Hello scrollBar
HelloWorld thickBlackBorder scrollBar
Hi
```

Test case5



```
TextView1 Hi
TextView1 display
TextView1 add scrollBar thickBlackBorde
TextView1 display
```

```
Hi
Hi scrollBar thickBlackBorder
```

Recurrent Problem$_1$

- A class will be modified if you want to attach additional responsibilities (decorators) to an object dynamically.
  - Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit.
  - For example, should let you add properties like borders or behaviors like scrolling to any user interface component.

Recurrent Problem$_2$

- One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance.

- This is inflexible, however, because the choice of border is made statically.

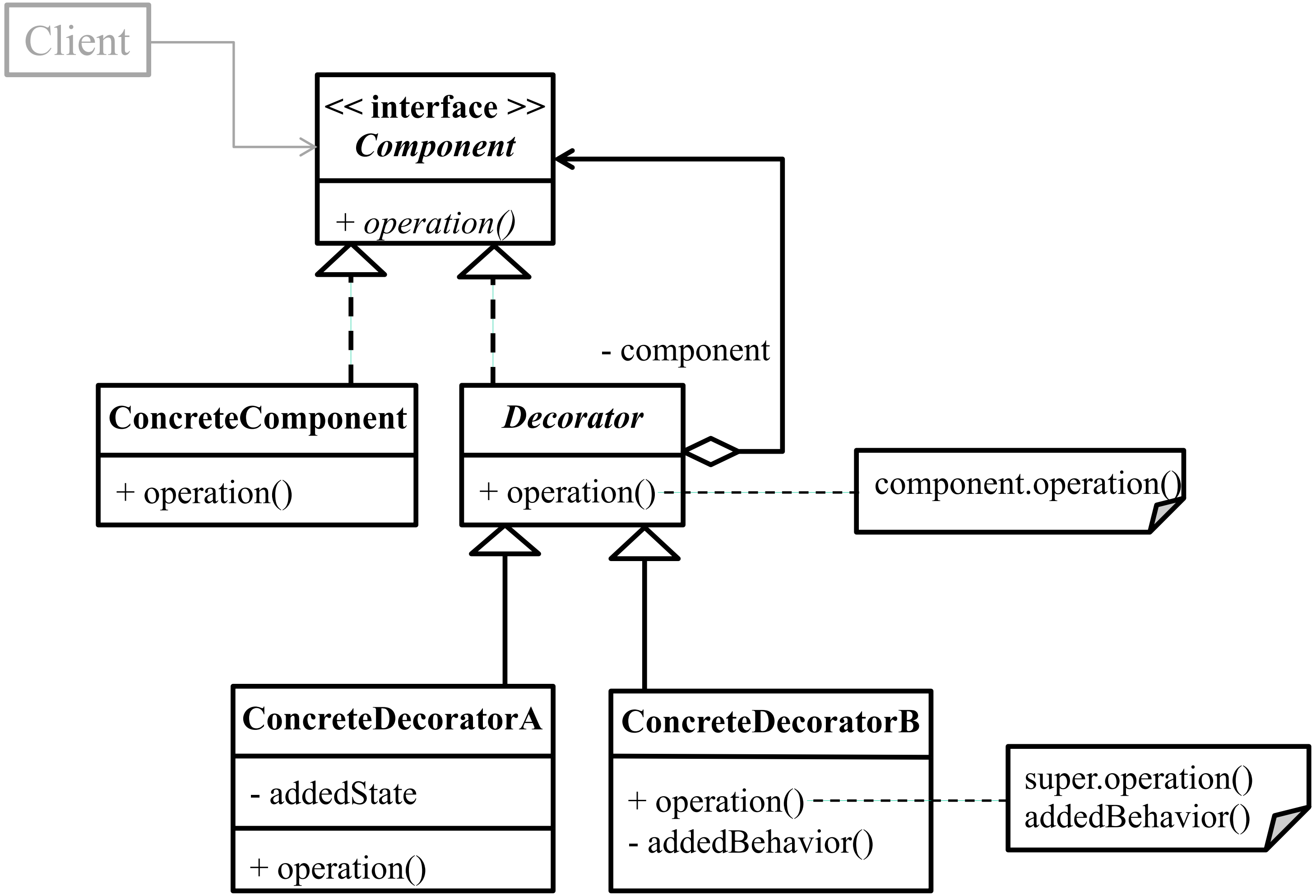
- A client can't control how and when to decorate the component with a border.

Intent

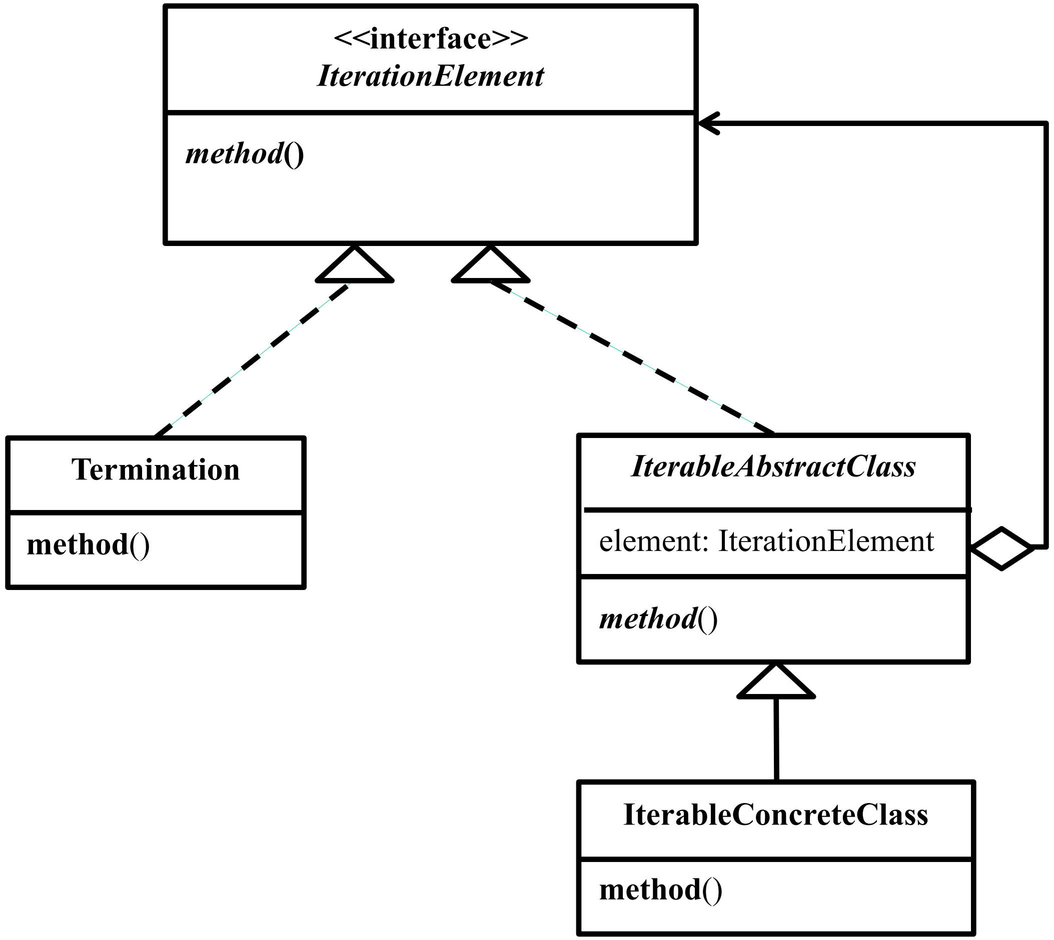
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- Extending responsibilities via subclassing forces developers to consider that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis.

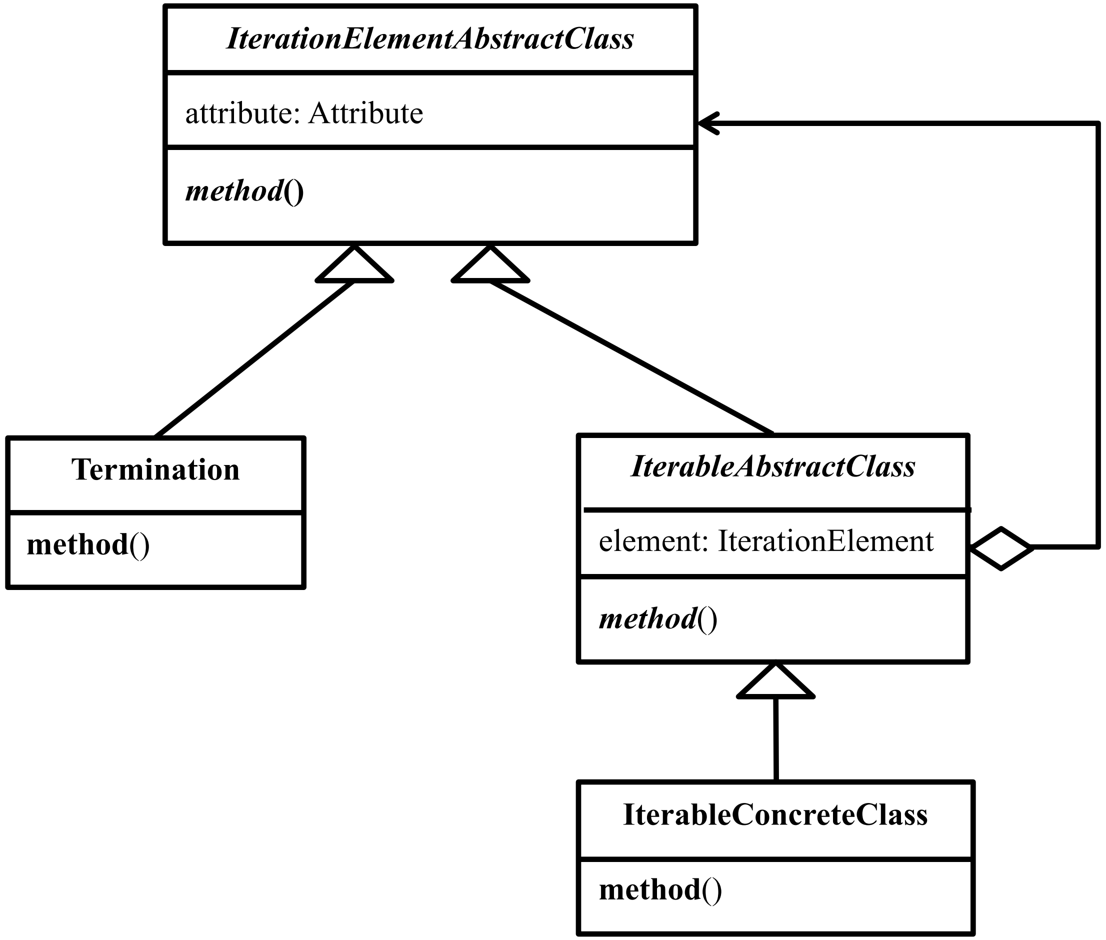
Decorator Pattern Structure$_1$



Recursive Design$_1$



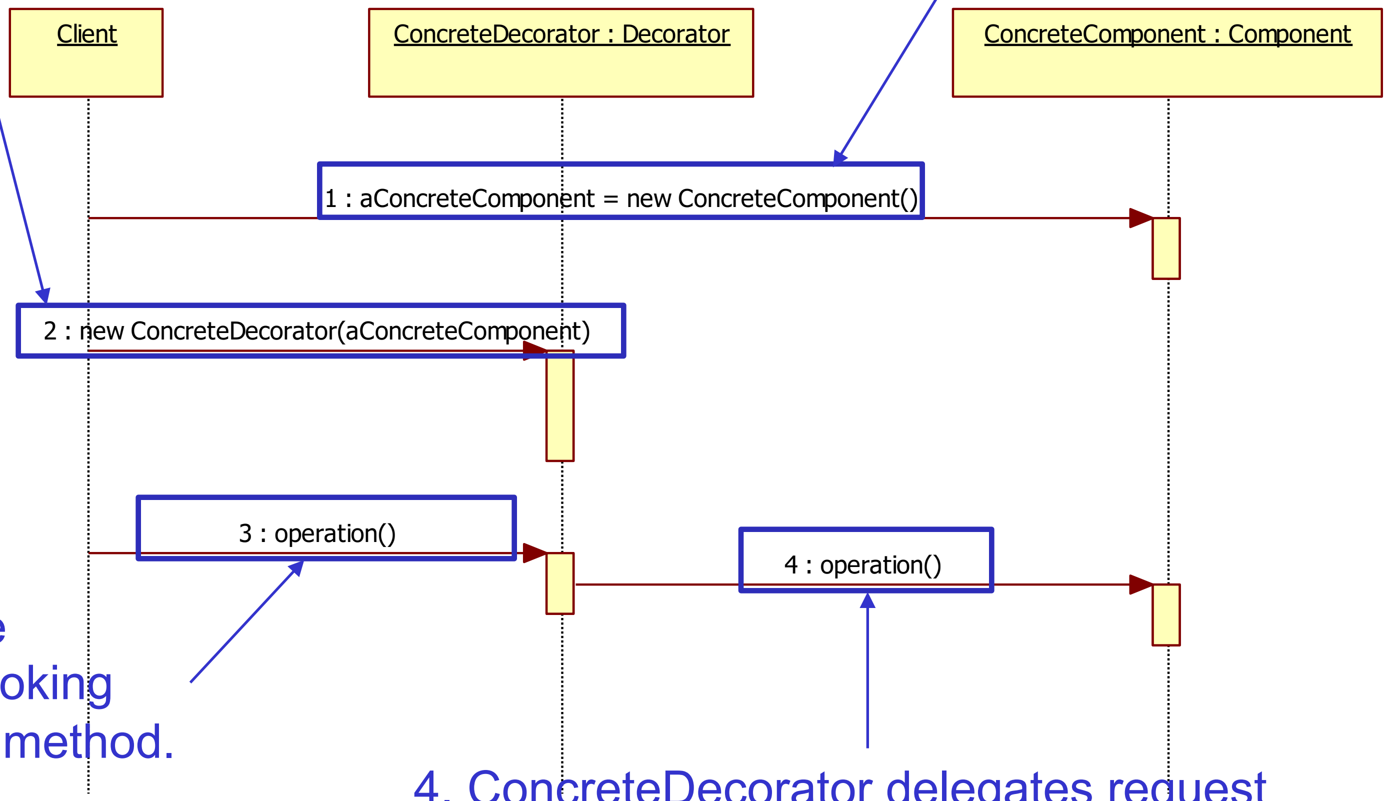
Recursive Design$_2$



Decorator Pattern Structure$_2$

2. Client is responsible for creating ConcreteDecorator's instance with ConcreteComponent's reference.

1. Client is responsible for creating ConcreteComponent's instance.


Client
ConcreteDecorator : Decorator
ConcreteComponent : Component
1 : aConcreteComponent = new ConcreteComponent()
2 : new ConcreteDecorator(aConcreteComponent)
3 : operation()
4 : operation()

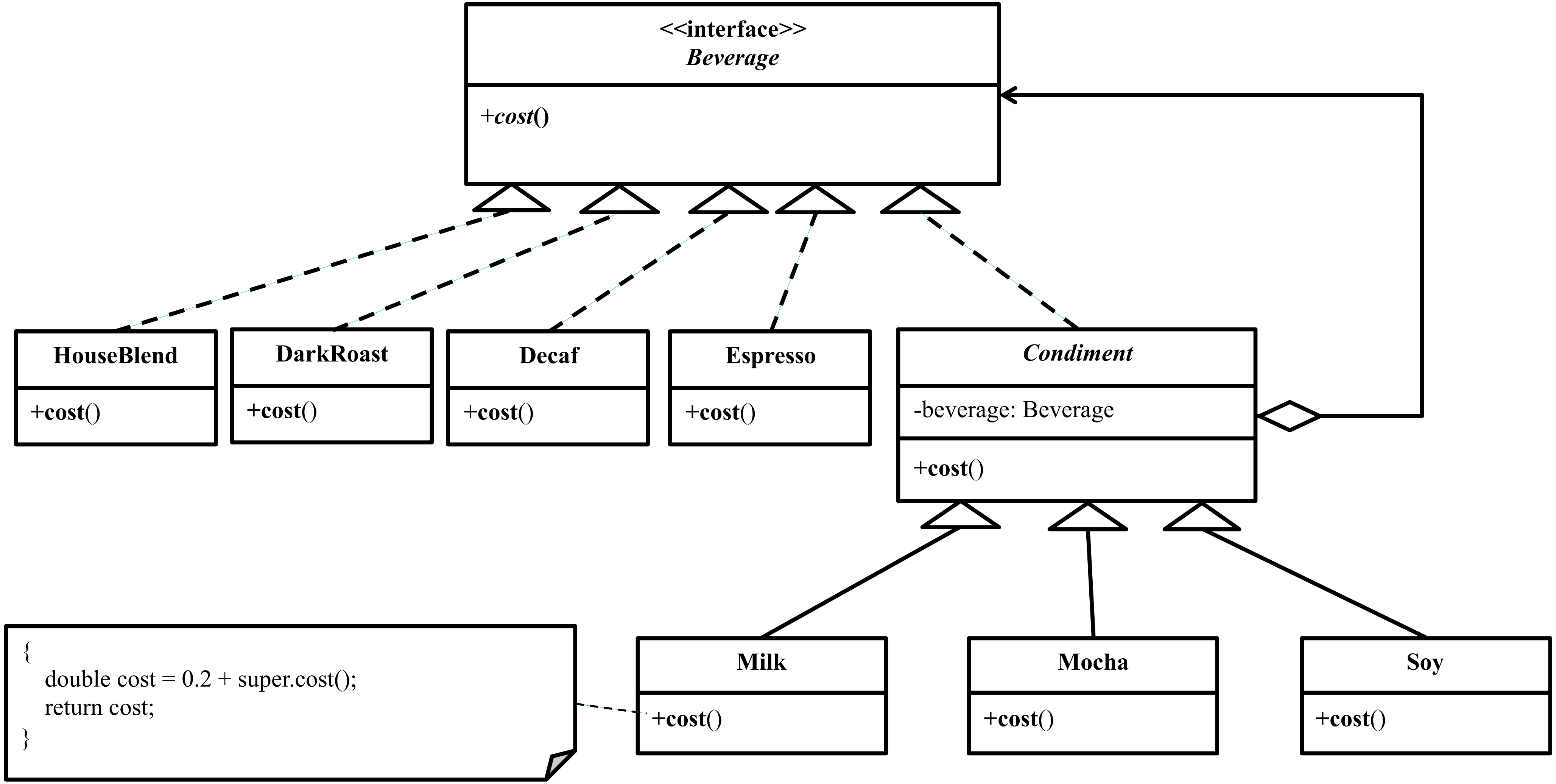
3. Client performs the operation through invoking ConcreteDecorator's method.

4. ConcreteDecorator delegates request to what it wrapped(ConcreteComponent).

Decorator Pattern Structure$_3$

	Instantiation	Use	Termination
Client	Other class except classes in the decorator pattern	Other class except classes in the decorator pattern	Other class except classes in the decorator pattern
Component	X	Client and ConcreteDecorator use this interface to invoke ConcreteComponent’s and ConcreteDecorator’s operation through polymorphism	X
Concrete Component	The client class or other class except classes in the decorator pattern	Client and ConcreteDecorator uses this class to invoke the operation implementation through polymorphism	Classes who hold the reference of ConcreteComponent
Decorator	X	ConcreteDecorator use this abstract class to compose another ConcreteDecorator and ConcreteComponent dynamically	X
Concrete Decorator	The client class or other class except classes in the decorator pattern	Another ConcreteDecorator uses this class to invoke the operation implementation through polymorphism	Classes who hold the reference of ConcreteDecorator

Refactored Design after Design Process



```
{
    double cost = 0.2 + super.cost();
    return cost;
}
```