**Decorator Design Pattern Concept**

Imagine you have a plain burger. It's good, but you might want to add some toppings to enhance its flavour. You can choose lettuce, tomato, cheese, etc., without altering the basic structure of the burger. This concept, in the realm of software design, is akin to the Decorator Design Pattern. ***It allows for adding new functionalities to an object dynamically, without changing its structure.***

**What is Decorator Pattern**

In Java, the Decorator Pattern is a structural design pattern that lets you attach additional responsibilities to an object at runtime. It is a flexible alternative to subclassing for extending functionality. This pattern involves a set of decorator classes that are used to wrap concrete components.

### Components of the Decorator Pattern

1. **Component:** An interface or abstract class defining the methods that will be implemented.

// Component

public interface TextField {

void draw();

}

1. **Concrete Component:** A class that implements the Component interface.

// ConcreteComponent

public class BasicTextField implements TextField {

public void draw() {

System.out.println("TextField: Basic");

}

}

1. **Decorator:** An abstract class that wraps a Component and implements the Component interface.

// Decorator

public abstract class TextFieldDecorator implements TextField {

protected TextField wrappedTextField;

public TextFieldDecorator(TextField textField) {

this.wrappedTextField = textField;

}

public void draw() {

wrappedTextField.draw();

}

}

1. **Concrete Decorator:** A class that extends the Decorator and adds additional features.

// ConcreteDecorator

public class SpellCheckDecorator extends TextFieldDecorator {

public SpellCheckDecorator(TextField textField) {

super(textField);

}

public void draw() {

super.draw();

addSpellCheck();

}

private void addSpellCheck() {

System.out.println("Decorator: Adding Spell Check");

}

}

//Use these as follows:

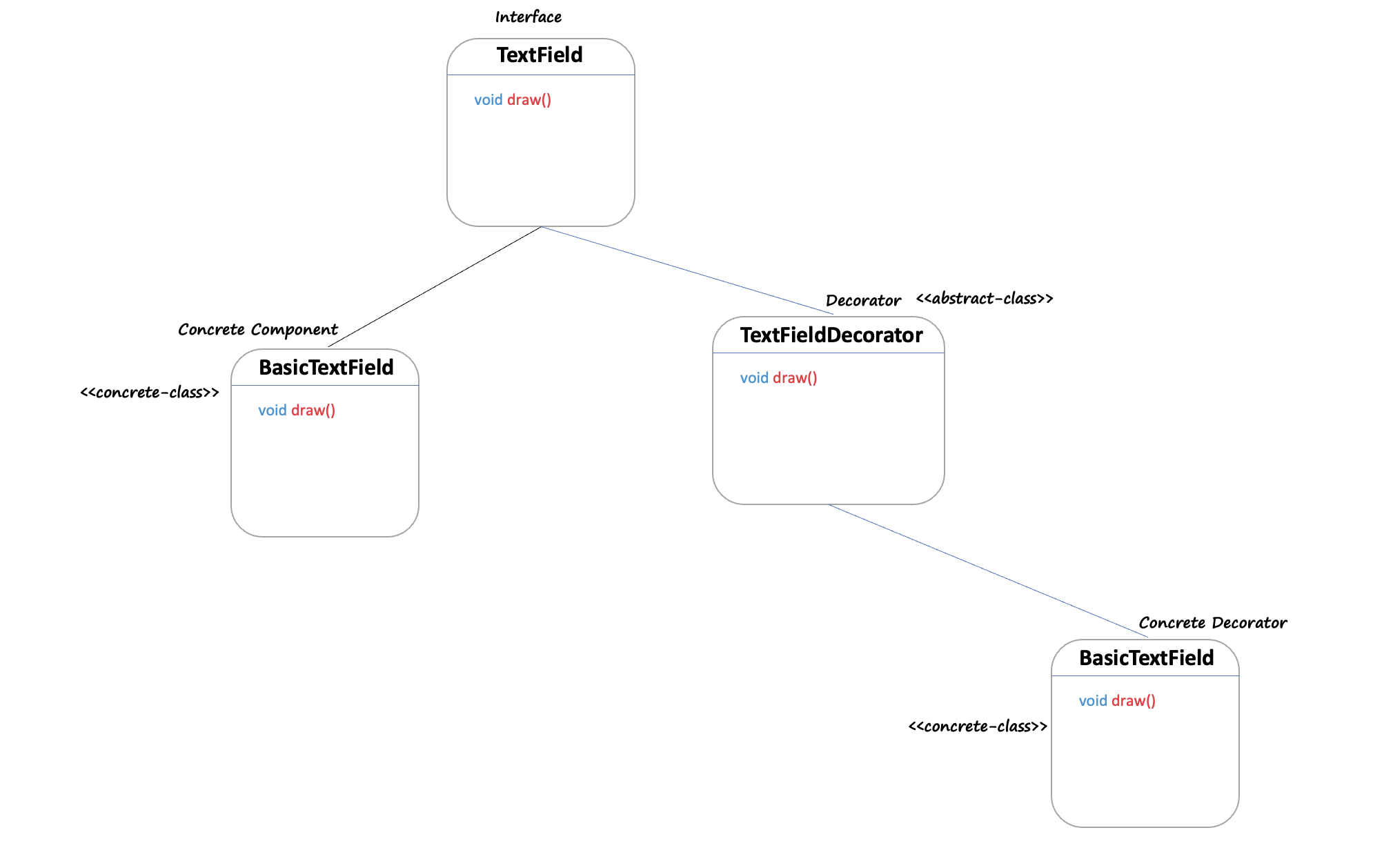
TextField textField = new BasicTextField();

TextField spellCheckedField = new SpellCheckDecorator(textField);

spellCheckedField.draw();

// Outputs: TextField: Basic Decorator: Adding Spell Check

**Class Diagram**



**What is Decorator Pattern**

* **Dynamic Extension:** When you need to extend the functionality of objects dynamically and transparently without affecting other objects.
* **Avoid Class Explosion:** When subclassing leads to an excessive number of classes.
* **Functionality Variation:** When you need to add responsibilities to individual objects, not to an entire class.

**When Not to Use**

* **Simple Extension:** For cases where a simple inheritance is sufficient and more manageable.
* **Performance Critical Situations:** The pattern can introduce a complexity that leads to performance issues, especially if there are many layers of decoration.
* **Complex Dependencies:** If the decorators can create complicated dependencies, it's better to reconsider using this pattern.