Problem No: 07

Problem Name: Implement and design Decorator design pattern (for design use UML/user_defined class)

Objective:

The objective of this lab is to understand and implement the Decorator Design Pattern. The pattern is used to add responsibilities to objects dynamically without modifying their base code. It promotes code reusability and adherence to the Open-Closed Principle.

Theory:

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. Instead of creating many subclasses, decorators provide a flexible alternative to extend functionality.

Key Components:

1. **Component:** Defines the interface for objects.
2. **ConcreteComponent:** The base implementation of the component.
3. **Decorator:** Maintains a reference to a component object and implements the component interface.
4. **ConcreteDecorator:** Adds new responsibilities dynamically to the component.

This pattern is useful for designing flexible and extensible systems.

Application:

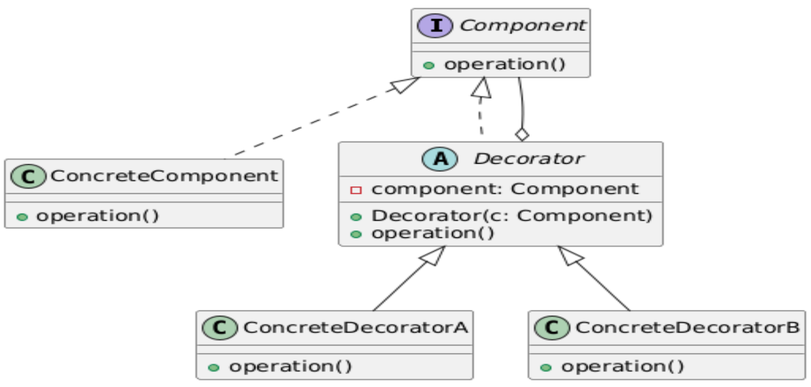
The decorator pattern is used when:

1. You need to add or remove responsibilities at runtime.
2. Subclassing would lead to an explosion of classes.
3. You want to combine multiple behaviors flexibly.

Examples:

1. Java I/O Streams (e.g., BufferedReader, InputStreamReader).
2. GUI frameworks (adding scrollbars, borders, colors dynamically).
3. Logging frameworks (adding timestamp, formatting, etc.).

UML Design:



Implementation in C++:

```
#include <iostream>
// Concrete Component
class ConcreteComponent : public Component {
public:
void operation() override {
cout << "Base operation" << endl;
}
};

// Decorator base class
class Decorator : public Component {
protected:
Component* component;
public:
Decorator(Component* c) : component(c) {}
void operation() override {
if (component)
component->operation();
}
};

// Concrete Decorator A
class ConcreteDecoratorA : public Decorator {
public:
ConcreteDecoratorA(Component* c) : Decorator(c) {}
void operation() override {
Decorator::operation();
cout << " + Added feature A" << endl;
}
};

// Concrete Decorator B
class ConcreteDecoratorB : public Decorator {
public:
ConcreteDecoratorB(Component* c) : Decorator(c) {}
void operation() override {
Decorator::operation();
cout << " + Added feature B" << endl;
}
};

// Demo
int main() {
ConcreteComponent base;
ConcreteDecoratorA decoratedA(&base);
ConcreteDecoratorB decoratedB(&decoratedA);

cout << "Final decorated operation:" << endl;
decoratedB.operation();

return 0;
}
```

Result Discussion:

Sample Output:

```
Final decorated operation:
Base operation
+ Added feature A
+ Added feature B
```

Discussion:

1. The ConcreteComponent performs the base operation.
2. ConcreteDecoratorA adds extra behavior A.
3. ConcreteDecoratorB further enhances the object with behavior B.
4. We can add responsibilities in different combinations dynamically.

Conclusion:

The Decorator Design Pattern provides a flexible alternative to subclassing for extending functionality. In this lab:

1. We designed a component and decorated it with multiple decorators.
2. We demonstrated a runtime combination of different behaviors.
3. This pattern ensures code remains extensible while adhering to the Open-Closed Principle.

Thus, the decorator pattern is essential for scalable, reusable, and maintainable object-oriented software design.