Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

# Лабораторна робота №6

ШАБЛОНИ «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

## Варіант 4

Виконав
студент групи ІА – 13:
Запотоцький І.А

Перевірив:
М’який М.Ю

## Завдання:

1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми.

## Варіант:

4)
Графічний редактор (proxy, prototype, decorator, bridge, flyweight, SOA) Графічний редактор повинен вміти створювати / редагувати растрові (або векторні на розсуд студента) зображення в 2-3 основних популярних форматах (bmp, png, jpg), мати панель інструментів для створення графічних примітивів, вибору кольорів, нанесення тексту, додавання найпростіших візуальних ефектів (ч/б растр, інфрачервоний растр, 2-3 на вибір учня), роботи з шарами.

## Хід роботи

Основні учасники шаблону "Декоратор":

Component (Компонент): Абстрактний клас або інтерфейс, який визначає базовий інтерфейс для конкретних компонентів.

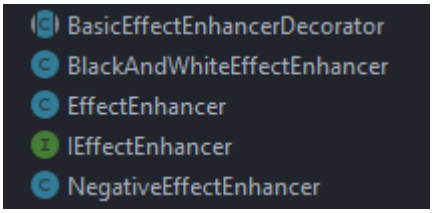
ConcreteComponent (Конкретний компонент): Клас, який реалізує інтерфейс Component і представляє базовий об'єкт, до якого можна додавати нові функціональності.

Decorator (Декоратор): Абстрактний клас або інтерфейс, який також реалізує інтерфейс Component і містить посилання на об'єкт типу Component. Це базовий клас для всіх конкретних декораторів.

ConcreteDecorator (Конкретний декоратор): Клас, який реалізує інтерфейс Decorator та розширює його функціональність. Використовується для додавання конкретної функціональності до компонента.

Принциповий механізм роботи шаблону полягає в тому, що компонент може бути обгорнутий декількома декораторами, кожен з яких додає певну функціональність. Кожен декоратор містить посилання на об'єкт типу Component, а сам компонент може бути як базовим, так і обгорнутим декораторами.

В моему випадку шаблон реалізован для поєднання різних ефектів на зображення:




```
8 ages  4 implementations
public interface IEffectEnhancer {
    4 usages  4 implementations
    BufferedImage enhanceEffect(BufferedImage image);

}
```

```
usages
ublic class EffectEnhancer implements IEffectEnhancer{


    4 usages
    @Override
    public BufferedImage enhanceEffect(BufferedImage image)
        //some pre-effect enhancing logic
        return image;
    }
```

```
2 usages  2 inheritors
public abstract class BasicEffectEnhancerDecorator implements IEffectEnhancer {

    2 usages
    private final IEffectEnhancer wrapper;

    2 usages
    public BasicEffectEnhancerDecorator(IEffectEnhancer wrapper) { this.wrapper = wrapper; }

    4 usages  2 overrides
    @Override
    public BufferedImage enhanceEffect(BufferedImage image) { return wrapper.enhanceEffect(image); }
}
```

І два об’єкти декоратора

```java
2 usages
public class BlackAndWhiteEffectEnhancer extends BasicEffectEnhancerDecorator{

    1 usage
    public BlackAndWhiteEffectEnhancer(IEffectEnhancer wrapper) { super(wrapper); }

    4 usages
    @Override
    public BufferedImage enhanceEffect(BufferedImage image){
        image = super.enhanceEffect(image);
        return enhanceBlackAndWhiteEffect(image);
    }

    1 usage
    private BufferedImage enhanceBlackAndWhiteEffect(BufferedImage originalImage){
        BufferedImage grayscaleImage = new BufferedImage(
                originalImage.getWidth(),
                originalImage.getHeight(),
                originalImage.getType()
        );

        for (int y = 0; y < originalImage.getHeight(); y++) {
            for (int x = 0; x < originalImage.getWidth(); x++) {
                int rgb = originalImage.getRGB(x, y);

                int grayscaleValue = calculateGrayscale(rgb);

                grayscaleImage.setRGB(x, y, grayscaleValue);
            }
        }

        return grayscaleImage;
    }
1 usage
private static int calculateGrayscale(int rgb) {
    int red = (rgb >> 16) & 0xFF;
    int green = (rgb >> 8) & 0xFF;
    int blue = rgb & 0xFF;

    int average = (red + green + blue) / 3;


    return (average << 16) | (average << 8) | average;
}
```

```java
public class NegativeEffectEnhancer extends BasicEffectEnhancerDecorator{


    1 usage
    public NegativeEffectEnhancer(IEffectEnhancer wrapper) { super(wrapper); }

    4 usages
    @Override
    public BufferedImage enhanceEffect(BufferedImage image){
        image =  super.enhanceEffect(image);
        return enhanceNegativeEffect(image);
    }

    1 usage
    private BufferedImage enhanceNegativeEffect(BufferedImage originalImage) {
        int width = originalImage.getWidth();
        int height = originalImage.getHeight();

        BufferedImage negativeImage = new BufferedImage(width, height, originalImage.getType());

        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                int rgb = originalImage.getRGB(x, y);

                int invertedRGB = invertColors(rgb);

                negativeImage.setRGB(x, y, invertedRGB);
            }
        }

        return negativeImage;
    }

    1 usage
    private static int invertColors(int rgb) {
        int alpha = (rgb >> 24) & 0xFF;
        int red = 255 - ((rgb >> 16) & 0xFF);
        int green = 255 - ((rgb >> 8) & 0xFF);
        int blue = 255 - (rgb & 0xFF);

        return (alpha << 24) | (red << 16) | (green << 8) | blue;
```

Шаблон проектування "Декоратор" має кілька переваг, що роблять його привабливим для використання в програмному проекті:

Гнучкість та розширюваність: Декоратор дозволяє динамічно додавати нові функціональності об'єктам, не змінюючи їхньої структури. Це робить систему більш гнучкою та розширюваною.

Легка зміна порядку декораторів: Комбінування різних декораторів дозволяє створювати різноманітні конфігурації об'єктів. Порядок декораторів може бути легко змінений, щоб отримати різні результати.

Збереження відкритого інтерфейсу: Інтерфейс компоненту залишається відкритим для клієнтського коду. Клієнти можуть взаємодіяти з об'єктами так само, як і раніше, навіть якщо ті об'єкти були декоровані.

Підтримка принципу відкритості/закритості: Декоратор виконує принцип відкритості/закритості (Open/Closed Principle), що робить його сумісним із засадами об'єктно-орієнтованого програмування.

Видалення відомостей про декоратори з компонентів: Компоненти не повинні знати про наявність декораторів і навіть можуть бути невтручаними. Це забезпечує відокремлення між класами декораторів та компонентами.

Спрощення об'єктів: Декоратор дозволяє розділити одну велику класову ієрархію на прості компоненти та декоратори, що робить структуру програми більш зрозумілою.

Можливість додавання або вилучення функціональності на льоту: Так як декоратори можуть динамічно додавати або вилучати функціональність, це дозволяє легко маневрувати зовнішнім виглядом або поведінкою об'єктів.

Шаблон "Декоратор" дозволяє отримати велику кількість можливостей, забезпечуючи при цьому чистоту коду та його легкість розширення.