

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ
СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №6
з курсу
«Технології розроблення програмного забезпечення»

Виконавець:
Нікітченко Наталя Олегівна
студентка групи ІА-33
залікова книжка № ІА-3318

«14» 11 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

6. ЛАБОРАТОРНА РОБОТА № 6

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати їх в реалізації програмної системи.

Завдання:

- Ознайомитись з короткими теоретичними відомостями.
- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
- Реалізувати один з розглянутих шаблонів за обраною темою.
- Реалізувати не менше 3-х класів відповідно до обраної теми.
- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Теоретичні відомості:

Шаблон «Decorator»

Призначення: Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється

полоса прокрутки. Початкові функції елемента (наприклад, рядки статусу) залишаються незмінними.

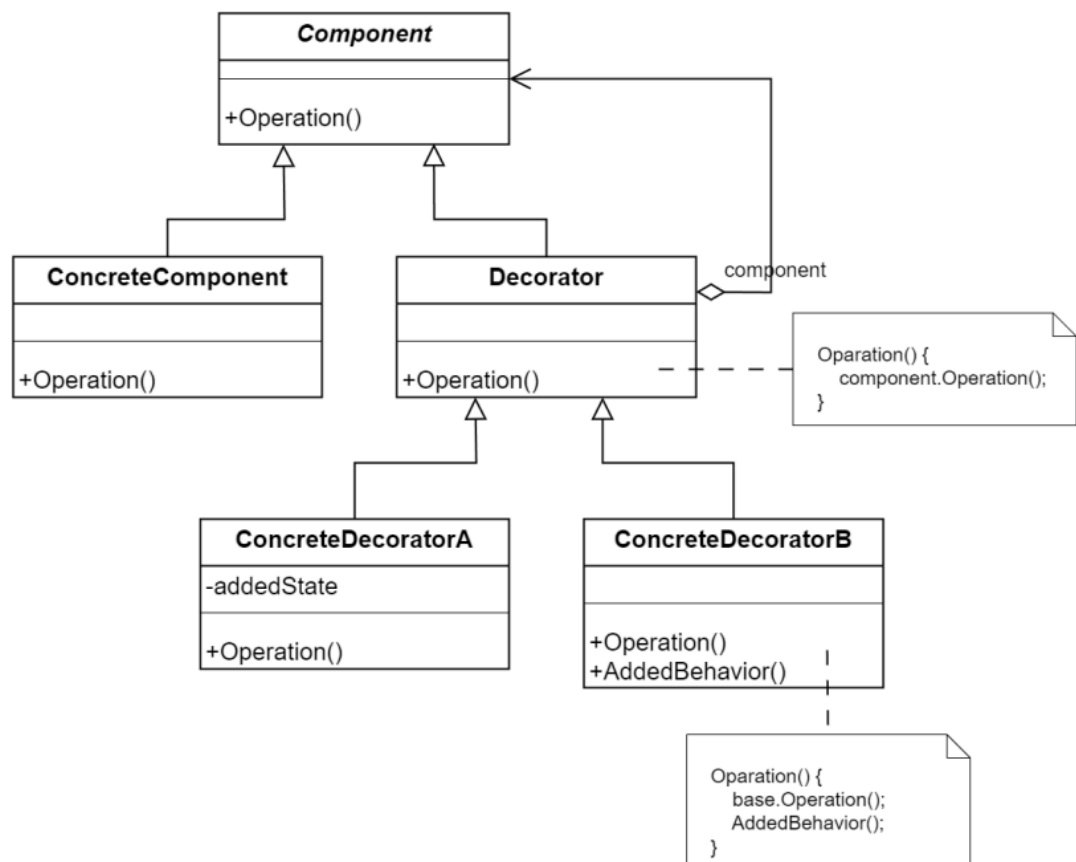


Рис.1 - Структура патерну «Decorator»

Переваги та недоліки:

- + Дозволяє мати кілька дрібних об'єктів, замість одного об'єкта «на всі випадки життя».
- + Дозволяє додавати обов'язки «на льоту».
- + Більша гнучкість, ніж у спадкування.
- Велика кількість крихлих класів.
- Важко конфігурувати об'єкти, які загорнуто в декілька обгортки одночасно.

Хід роботи:

Фрагменти коду по реалізації цього шаблону:

```
1 package com.example.graphiceditor.model.decorator;  
2  
3 public interface RenderLayer { no usages 5 implementations new *  
4     void render(); no usages 5 implementations new *  
5 }  
6
```

Рис.2 - Інтерфейс RenderLayer

RenderLayer — це спільний контракт (інтерфейс) для всіх “шарів” рендерингу.

Він має один метод:

render() — викликається коли потрібно відобразити шар.

```
package com.example.graphiceditor.ui;  
  
import com.example.graphiceditor.model.decorator.*;  
  
public class DemoDecorator { new *  
  
    public static void main(String[] args) { new *  
  
        RenderLayer base = new CanvasLayer( name: "portrait.png");  
  
        RenderLayer withFilters =  
            new BorderDecorator(  
                new BlurFilter(  
                    new GrayscaleFilter(base)  
                ),  
                color: "#00FF00"  
            );  
  
        System.out.println("=== Render with filters ===");  
        withFilters.render();  
    }  
}
```

Рис.3 - **DemoDecorator** — демонстрація роботи

```

package com.example.graphiceditor.model.decorator;

public class CanvasLayer implements RenderLayer { no usages new *

    private final String name; 2 usages

    public CanvasLayer(String name) { no usages new *
        this.name = name;
    }

    @Override no usages new *
    public void render() {
        System.out.println("[Layer] Base image: " + name);
    }
}

```

Рис.4 - **CanvasLayer** - базовий компонент

CanvasLayer — звичайний шар зображення.

Він:

- зберігає назву файлу зображення,
- у методі render() просто друкує:
"[Layer] Base image: portrait.png"

```

package com.example.graphiceditor.model.decorator;

public abstract class LayerDecorator implements RenderLayer {

    protected final RenderLayer inner;

    protected LayerDecorator(RenderLayer inner) {
        this.inner = inner;
    }

    @Override
    public void render() {
        inner.render();
    }
}

```

Рис.5 - LayerDecorator

LayerDecorator — це абстрактний декоратор, який:

- реалізує той самий інтерфейс, що й базові шари (RenderLayer);
- зберігає всередині інший RenderLayer (тобто шар, який він "обгортає");
- передає виклики render() внутрішньому шару.

```

package com.example.graphiceditor.model.decorator;

public class GrayscaleFilter extends LayerDecorator {

    public GrayscaleFilter(RenderLayer inner) {
        super(inner);
    }

    @Override
    public void render() {
        super.render();
        System.out.println(" + apply GRAYSCALE filter");
    }
}

```

Рис.6 - GrayscaleFilter

GrayscaleFilter:

Викликає render() в базового шару

Додає фільтр "чорно-білий".

```
package com.example.graphiceditor.model.decorator;  
  
public class BlurFilter extends LayerDecorator { no usages new *  
  
    public BlurFilter(RenderLayer inner) { no usages new *  
        super(inner);  
    }  
  
    @Override 3 usages new *  
    public void render() {  
        super.render();  
        System.out.println("    + apply BLUR filter");  
    }  
}
```

Рис.7 - BlurFilter

BlurFilter: Обгортає GrayscaleFilter (або будь-який інший).

```
package com.example.graphiceditor.model.decorator;  
  
public class BorderDecorator extends LayerDecorator { no usages new *  
  
    private final String color; 2 usages  
  
    public BorderDecorator(RenderLayer inner, String color) { no usages new *  
        super(inner);  
        this.color = color;  
    }  
  
    @Override 4 usages new *  
    public void render() {  
        super.render();  
        System.out.println("    + draw BORDER with color " + color);  
    }  
}
```

Рис.8 - BorderDecorator

BorderDecorator:

Обгортає попередній фільтр або базовий шар;

Малює рамку.

Діаграму класів, яка представляє використання шаблону в реалізації системи:

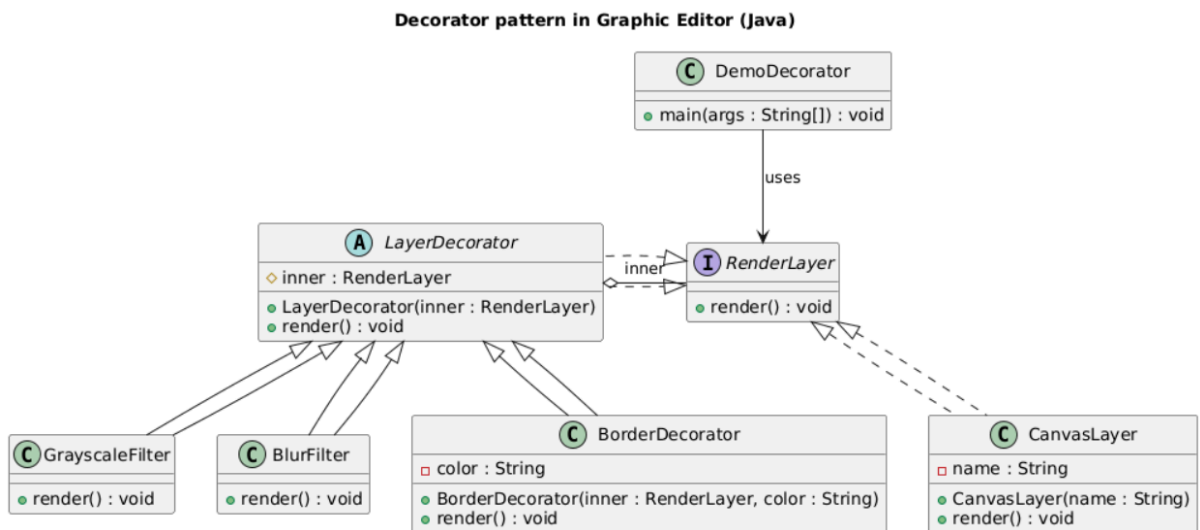


Рис.9 - Діаграму класів реалізованого шаблону

Пояснення до діаграми:

RenderLayer – компонент (інтерфейс), спільний для всіх шарів.

CanvasLayer – базовий шар зображення (ConcreteComponent).

LayerDecorator – абстрактний декоратор, який:

- реалізує RenderLayer,
- зберігає посилання inner : RenderLayer.

GrayscaleFilter, BlurFilter, BorderDecorator – конкретні декоратори, що наслідують LayerDecorator і додають свої ефекти.

DemoDecorator – клієнт, який створює CanvasLayer, обгортає його декораторами і викликає render().

Висновки:

У ході роботи було реалізовано шаблон проектування Decorator, призначений для динамічного розширення поведінки об'єктів без зміни їхнього коду. Це дозволило створити гнучку структуру рендерингу графічних шарів у графічному редакторі.

Основною ідеєю патерну є те, що декоратор і базовий об'єкт мають спільний інтерфейс, тому їх можна взаємозамінювати. Реалізовані класи GrayscaleFilter, BlurFilter та BorderDecorator обгортають базовий шар (CanvasLayer) та додають власні ефекти, не втручаючись у його роботу.

На діаграмі видно дві ключові залежності:

1. Реалізація інтерфейсу RenderLayer

Декоратори реалізують той самий інтерфейс, що й базовий шар. Це забезпечує поліморфізм і дозволяє застосовувати декоратори у будь-якій комбінації.

2. Композиція (вкладений RenderLayer усередині декоратора)

Кожен декоратор містить посилання на інший RenderLayer, завдяки чому ефекти можна накладати ланцюжком. Саме ця властивість дозволяє отримувати необмежено гнучку структуру ефектів.

Комбінація цих двох механізмів створює повноцінну поведінкову модель, у якій об'єкти можуть оброблятися багаторазово, послідовно та незалежно від основного коду базового шару.

Таким чином, реалізація патерну Decorator у системі графічного редактора є обґрунтованою й ефективною. Вона:

- підтримує розширюваність без модифікації існуючих класів;

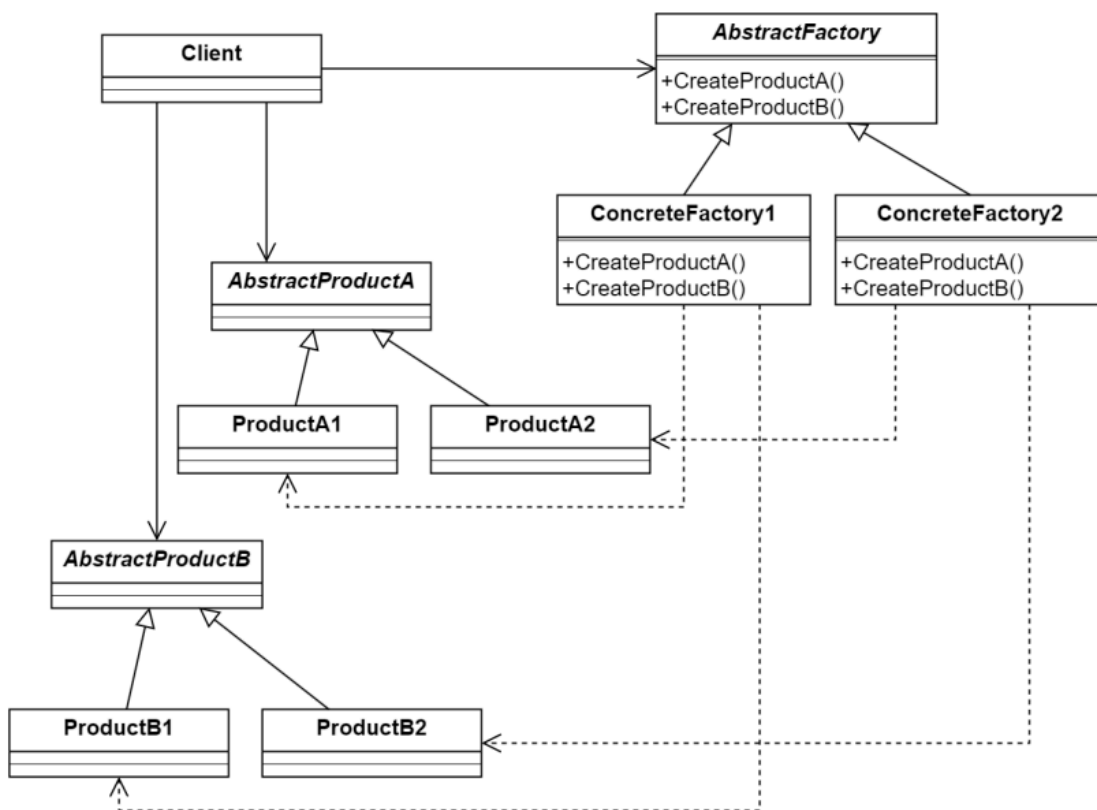
- забезпечує чисту архітектуру згідно принципів SOLID (зокрема Open/Closed Principle);
- дозволяє легко додавати нові ефекти рендерингу;
- забезпечує можливість комбінування ефектів у будь-якому порядку.

Контрольні питання:

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

Класи з діаграми:

- Client;
- AbstractFactory;
- ConcreteFactory1, ConcreteFactory2;
- AbstractProductA, AbstractProductB;
- ProductA1, ProductA2 (конкретні продукти типу A);
- ProductB1, ProductB2 (конкретні продукти типу B);

Яка між ними взаємодія:

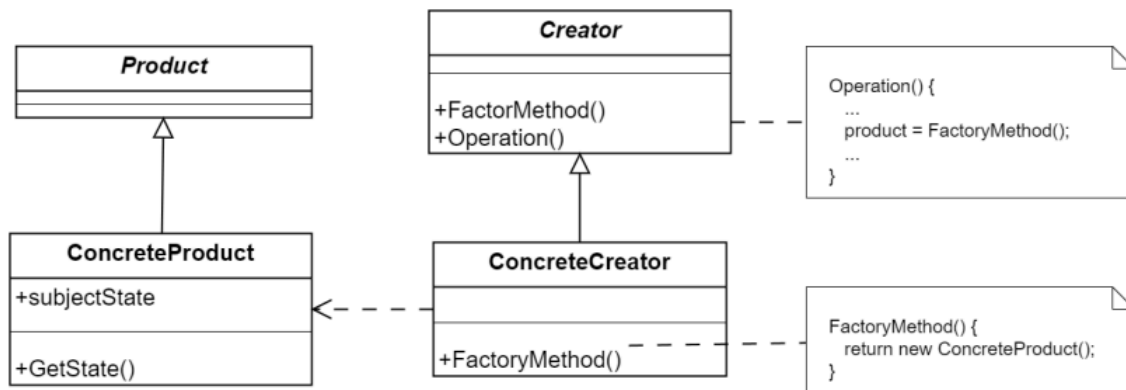
1. Client працює тільки з інтерфейсами: AbstractFactory, AbstractProductA, AbstractProductB.
2. Client має посилання на AbstractFactory і викликає її методи CreateProductA() та CreateProductB().
3. Насправді під AbstractFactory знаходиться одна з ConcreteFactory (1 або 2), яка створює відповідну пару продуктів:
CreateProductA() → ProductA1 або ProductA2;
CreateProductB() → ProductB1 або ProductB2.
4. ProductA1/A2 наслідують AbstractProductA, ProductB1/B2 наслідують AbstractProductB — тому Client бачить їх як абстрактні продукти, а не як конкретні класи.
5. Таким чином одна ConcreteFactory виробляє узгоджене сімейство продуктів A+B.

4. Яке призначення шаблону «Фабричний метод»?

Доручити підкласам вирішувати, який конкретний об'єкт створювати.

Замість new в базовому класі використовується віртуальний метод createProduct(), який перевизначається в підкласах.

5. Нарисуйте структуру шаблону «Фабричний метод».



6. Які класи входять в шаблон «Фабричний метод», та яка між ними взаємодія?

Creator (абстрактний) - визначає фабричний метод `factoryMethod()` і може використовувати результат у своїх алгоритмах.

ConcreteCreator - перевизначає `factoryMethod()` і створює конкретний продукт.

Product (інтерфейс) – інтерфейс/базовий клас продукції.

ConcreteProduct – конкретні реалізації продукту.

Яка в них взаємодія:

Product – базовий інтерфейс/клас продукту.

ConcreteProduct – конкретна реалізація (наслідує Product).

Creator – абстрактний творець, містить:

- метод `FactoryMethod() : Product` (фабричний метод);
- метод `Operation()` – бізнес-логіка, яка всередині викликає `FactoryMethod()` і працює з повернутим Product.

ConcreteCreator наслідує **Creator** і перевизначає **FactoryMethod()**, створюючи конкретний ConcreteProduct.

Client викликає `Operation()` у Creator'а, не знаючи, який саме ConcreteProduct буде створено — це вирішує ConcreteCreator.

7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

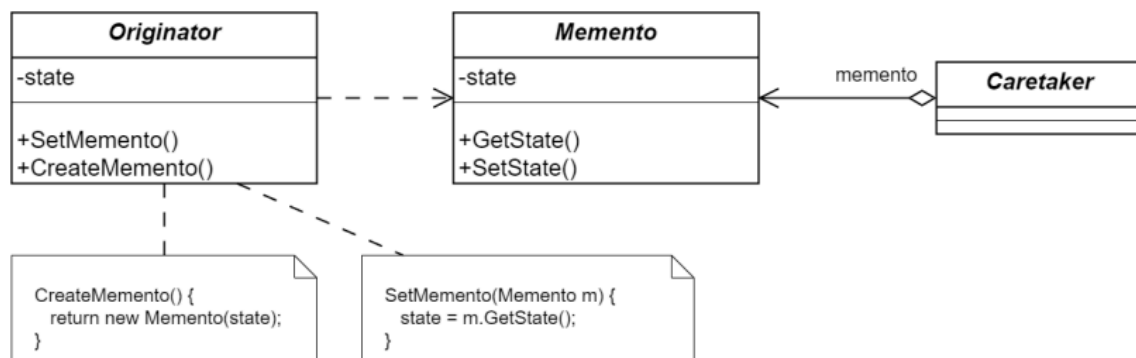
Фабричний шаблонний метод – створює один продукт певного типу, делегуючи вибір конкретного класу підкласам.

Абстрактна фабрика шаблонний метод – створює ціле сімейство узгоджених продуктів (кілька типів), причому всі вони відповідають одному варіанту (тема, платформа тощо).

8. Яке призначення шаблону «Знімок»?

Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції [6]. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту «Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання мemento об'єктів в системі.

9. Нарисуйте структуру шаблону «Знімок».



10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Originator – об'єкт, стан якого треба зберігати.

Memento – об'єкт-знімок; зберігає стан **Originator** (часто зроблений *inner*-класом, щоб приховати деталі).

Caretaker – тримає посилання на **Memento** (або стек/список), викликає `restore`, але не змінює сам знімок.

Яка між ними взаємодія:

Originator має внутрішній стан `state` і два методи:

- `CreateMemento()` – створює новий `Memento`, передаючи йому свій поточний стан;
- `SetMemento(Memento m)` – відновлює свій стан із переданого `memento`.

Memento зберігає копію `state` та надає методи `GetState()` / `SetState()` (у реальних реалізаціях часто приховані від `Caretaker`'а).

Caretaker:

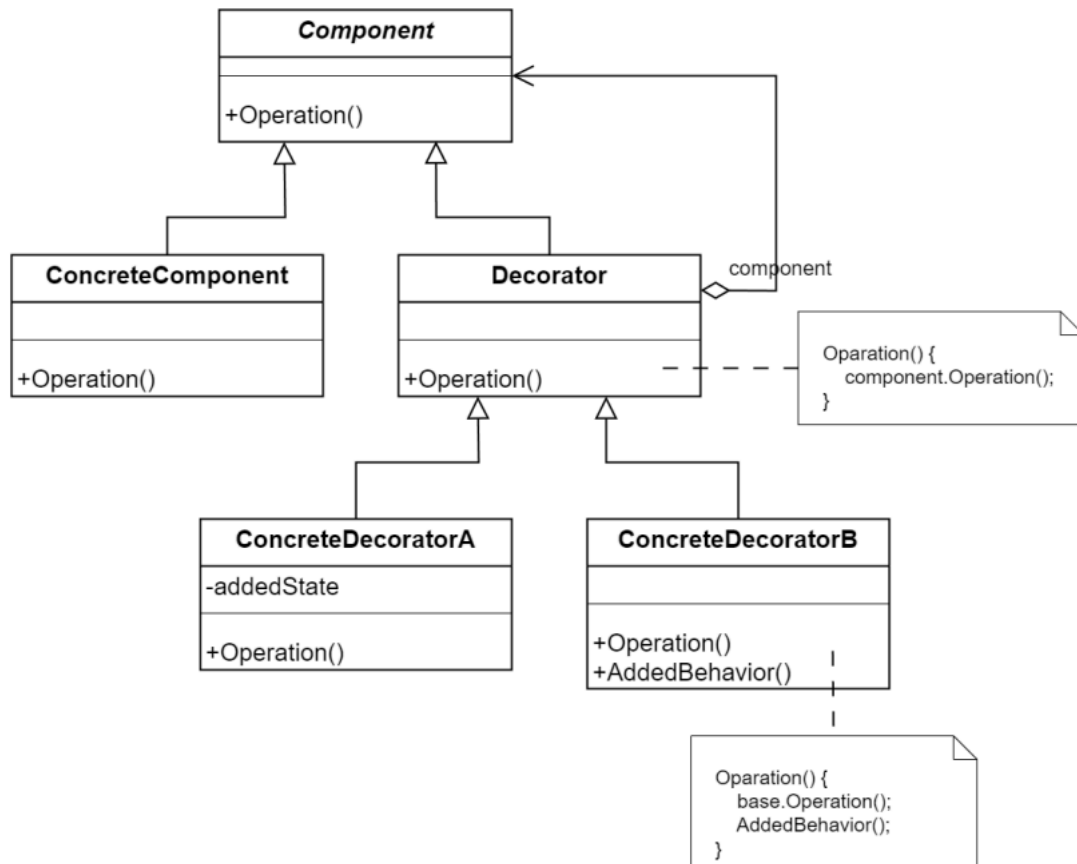
- отримує від `Originator` об'єкт `Memento` і зберігає його (наприклад, у стеку);
- пізніше передає цей `Memento` назад `Originator`'у, коли треба зробити `Undo`.

`Caretaker` не змінює вміст `Memento`, тільки зберігає та повертає його → інкапсуляція стану `Originator` не порушується.

11. Яке призначення шаблону «Декоратор»?

Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми [6]. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

Component – спільний інтерфейс для всіх об'єктів (оригінал + декоратори).

ConcreteComponent – базова реалізація функціоналу.

Decorator – має посилання на **Component** і реалізує той самий інтерфейс; зазвичай просто делегує виклики.

ConcreteDecoratorA

ConcreteDecoratorB

Яка між ними взаємодія:

Component – базовий інтерфейс із методом `Operation()`.

ConcreteComponent – звичайна реалізація **Component**, яка містить базову поведінку.

Decorator:

- ❖ також реалізує **Component**;
- ❖ містить поле `component` типу **Component** (композиція);
- ❖ у `Operation()` викликає `component.Operation()` (делегує виклик).

ConcreteDecoratorA і **ConcreteDecoratorB** наслідують **Decorator**:

- ❖ перевизначають `Operation()`;
- ❖ додають свою поведінку до або після виклику `base.Operation()` (тобто делегованої операції).

Client працює з інтерфейсом **Component**, може обгортати базовий **ConcreteComponent** у один або кілька декораторів, не змінюючи класу компонента.

14. Які є обмеження використання шаблону «декоратор»?

Ускладнення структури: багато дрібних класів-декораторів, композиція з декількох шарів може бути важко відстежуваною.

Налагодження складніше: важко зрозуміти, через які саме декоратори пройшов виклик.

Не підходить, коли потрібна «глобальна» зміна класу для всіх місць одразу.

Не зручно, якщо інтерфейс має дуже багато методів — кожен декоратор змушений їх дублювати і делегувати.