МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

імені ІГОРЯ СІКОРСЬКОГО»

ЛАБОРАТОРНА РОБОТА №6

ТЕМА: «Патерни проектування»

Виконала:

Студентка групи ІА-34

Потейчук С.А.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Теоретичні відомості

Шаблон «Abstract Factory» (Абстрактна Фабрика)

Призначення: Шаблон «Абстрактна фабрика» надає інтерфейс для створення сімейств пов'язаних або залежних об'єктів без вказівки їх конкретних класів. Він дозволяє гарантувати, що створені об'єкти будуть сумісними та узгодженими між собою.

Проблема: При розробці системи генерації кімнат для гри виникає потреба створювати різні стилі кімнат (хайтек, модерн, класичний), де всі елементи кімнати (стіни, двері, меблі) мають належати до одного стилю. Без використання патерну код генерації швидко перетворюється на набір умовних операторів, що ускладнює додавання нових стилів та підтримку існуючих.

Рішення: Створюється інтерфейс абстрактної фабрики, який оголошує методи для створення кожного типу об'єктів (наприклад, createChair(), createTable()). Для кожного стилю реалізується конкретна фабрика, яка повертає відповідні об'єкти. Клієнтський код працює тільки з інтерфейсом фабрики, що гарантує узгодженість стилів усіх створених об'єктів.

Переваги: гарантія узгодженості створюваних об'єктів, спрощення додавання нових сімейств продуктів, відокремлення коду створення об'єктів від бізнес-логіки.

Недоліки: складність додавання нових типів продуктів (вимагає змін у всіх фабриках), загальне ускладнення коду.

Шаблон «Factory Method» (Фабричний Метод)

Призначення: Шаблон «Фабричний метод» визначає інтерфейс для створення об'єкта, але залишає підкласам вирішення питання про те, який саме клас створювати. Це дозволяє перенести створення об'єктів на рівень підкласів, зберігаючи спільну логіку роботи в базовому класі.

Проблема: У системі, що працює з мережевими пакетами, є спільна логіка обробки, але створення конкретних типів пакетів (TCP, UDP) має відрізнятися. Жорстка прив'язка до конкретних класів у коді обробки ускладнює розширення системи новими типами пакетів.

Рішення: Базовий клас визначає абстрактний фабричний метод для створення об'єктів. Кожен підклас реалізує цей метод, повертаючи потрібний тип об'єкта. Таким чином, код у базовому класі може працювати з об'єктами через їх спільний інтерфейс, не знаючи про їхні конкретні типи.

Переваги: послаблення зв'язку між кодом та конкретними класами, централізація коду створення об'єктів, спрощення додавання нових типів продуктів.

Недоліки: може призвести до створення великої кількості підкласів для кожного типу продукту, що ускладнює структуру програми.

Шаблон «Memento» (Знімок)

Призначення: Шаблон «Знімок» дозволяє зберігати та відновлювати попередні стани об'єктів, не розкриваючи деталі їхньої реалізації. Він забезпечує збереження стану об'єкта в зовнішньому сховищі з можливістю подальшого відновлення.

Ключові компоненти:

- Originator (Творець): Об'єкт, стан якого потрібно зберегти.
- Memento (Знімок): Об'єкт, що зберігає стан творця.
- Caretaker (Опікун): Об'єкт, що відповідає за зберігання та управління знімками.

Переваги: не порушує інкапсуляцію, спрощує структуру основного об'єкта.

Недоліки: може вимагати значних ресурсів пам'яті при частому створенні знімків.

Шаблон «Observer» (Спостерігач)

Призначення: Шаблон визначає залежність "один-до-багатьох" між об'єктами, коли зміна стану одного об'єкта автоматично сповіщає всіх залежних від нього об'єктів.

Реалізація: Суб'єкт підтримує список спостерігачів і надає методи для їх додавання та видалення. При зміні стану суб'єкт сповіщає всіх зареєстрованих спостерігачів.

Приклади використання: система сповіщень YouTube, повідомлення про наявність товару в інтернет-магазинах, механізм прив'язок даних у WPF.

Переваги: слабкий зв'язок між об'єктами, можливість динамічно додавати та видаляти спостерігачів.

Недоліки: відсутність гарантій щодо порядку обробки сповіщень.

Шаблон «Decorator» (Декоратор)

Призначення: Шаблон дозволяє динамічно додавати нову функціональність об'єктам, обертаючи їх в спеціальні об'єкти-декоратори.

Принцип роботи: Декоратор містить посилання на об'єкт того ж типу, що й декораований об'єкт, і перехоплює виклики методів, додаючи власну логіку до основних операцій.

Приклади використання: додавання смуг прокрутки до візуальних елементів, розширення функціональності потоків вводу-виводу, додавання логування або шифрування до базових операцій.

Переваги: гнучкість у додаванні функціональності, альтернатива спадкуванню.

Недоліки: велика кількість дрібних класів, складність конфігурації при багаторівневому декоратуванні.

Тема: Патерни проектування.

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

Хід роботи

7. Редактор зображень (state, prototype, memento, facade, composite, client server)

Редактор зображень має такі функціональні можливості: відкриття/збереження зображень у найпопулярніших форматах, застосування ефектів, наприклад поворот, розтягування, стиснення, кадрування зображення, можливість створення колажів шляхом «нашарування» зображень.

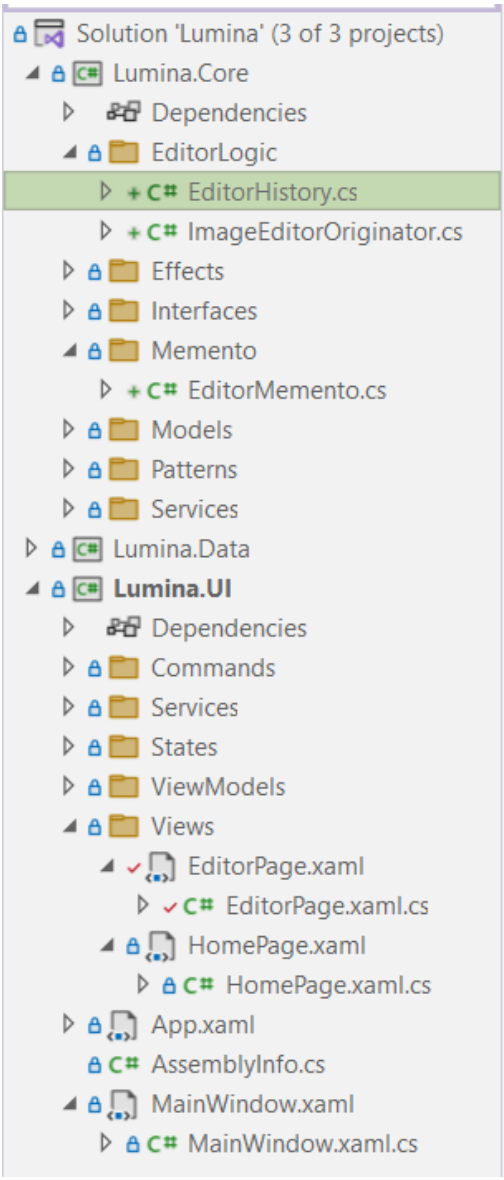


Рисунок 1 – Структура проєкту

Загальна структура проєкту:

- Lumina.Core – Бізнес-логіка, сервіси, моделі.
- Lumina.Data – Робота з базою даних, реалізації репозиторіїв, контекст EF Core, сутності та мапери.
- Lumina.UI – Графічний інтерфейс (WPF), користувацькі дії, візуальне відображення, керування станами.

Lumina.UI:

- App.xaml / App.xaml.cs – Точка входу. Ініціалізація головного вікна.
- MainWindow.xaml / MainWindow.xaml.cs – Головне вікно, у якому завантажуються сторінки (HomePage, EditorPage).
- HomePage.xaml / HomePage.xaml.cs – Головна сторінка з кнопками “Open Image”, “Create Collage”, “Go to Editor”, “Exit”.
- EditorPage.xaml / EditorPage.xaml.cs – Головна сторінка редактора: полотно, панель інструментів, ефекти, стани.
- ViewModels/ – ViewModel-и для зв’язку UI та Core.
- States/ – Реалізація патерну State для поведінки редактора.

У графічному редакторі виникає необхідність повертати попередні стани зображення (наприклад, після масштабування, переміщення, обрізки чи застосування ефектів). Користувач повинен мати можливість виконати Undo (скасування) та Redo (повторення дій). Для реалізації такої поведінки оптимально підходить шаблон проектування Memento.

Патерн дозволяє зберегти внутрішній стан об’єкта (позиція, розмір, шлях до файлу, ефекти), відновити попередній стан без порушення інкапсуляції та організувати історію станів для Undo/Redo.

```
9 references
public class EditorMemento
{
    2 references
    public string ImagePath { get; }
    2 references
    public double X { get; }
    2 references
    public double Y { get; }
    2 references
    public double Width { get; }
    2 references
    public double Height { get; }

    1 reference
    public EditorMemento(string imagePath, double x, double y, double width, double height)
    {
        ImagePath = imagePath;
        X = x;
        Y = y;
        Width = width;
        Height = height;
    }
}
```

Рисунок 2 – клас EditorMemento.cs

EditorMemento (Memento) – контейнер для збереження стану.

Він містить позицію X/Y, розмір (ширину/висоту) та шлях до зображення.

EditorMemento не має логіки, не змінюється, не обробляється – це чистий збережений стан в момент часу.

```
4 references
public class ImageEditorOriginator
{
    6 references
    public string CurrentImagePath { get; private set; } = "";
    5 references
    public double PosX { get; private set; }
    5 references
    public double PosY { get; private set; }
    5 references
    public double Width { get; private set; } = 400;
    5 references
    public double Height { get; private set; } = 300;

    2 references
    public void SetState(string path, double x, double y, double width, double height)
    {
        CurrentImagePath = path;
        PosX = x;
        PosY = y;
        Width = width;
        Height = height;
    }

    2 references
    public EditorMemento Save()
    {
        return new EditorMemento(CurrentImagePath, PosX, PosY, Width, Height);
    }

    2 references
    public void Restore(EditorMemento memento)
    {
        CurrentImagePath = memento.ImagePath;
        PosX = memento.X;
        PosY = memento.Y;
        Width = memento.Width;
        Height = memento.Height;
    }
}
```

Рисунок 3 – клас ImageEditorOriginator.cs

ImageEditorOriginator (Originator) – це центральний об’єкт, який:

- містить поточний стан редактора (позиція X/Y, ширина/висота зображення, шлях до файлу);
- створює об’єкт Memento;
- відновлює стан із Memento.

Основні обов’язки включають: SetState (приймає нові параметри зображення), Save (створює Memento) та Restore (відновлює стан із Memento).

ImageEditorOriginator знає всі внутрішні деталі редактора, але не розкриває їх іншим класам, що забезпечує інкапсуляцію.

```
2 references
public class EditorHistory
{
    private readonly Stack<EditorMemento> _undoStack = new();
    private readonly Stack<EditorMemento> _redoStack = new();

    2 references
    public void Push(EditorMemento state)
    {
        _undoStack.Push(state);
        _redoStack.Clear();
    }

    1 reference
    public EditorMemento? Undo()
    {
        if (_undoStack.Count <= 1)
            return null;

        var current = _undoStack.Pop();
        _redoStack.Push(current);

        return _undoStack.Peek();
    }

    1 reference
    public EditorMemento? Redo()
    {
        if (_redoStack.Count == 0)
            return null;

        var state = _redoStack.Pop();
        _undoStack.Push(state);
        return state;
    }
}
```

Рисунок 4 – клас EditorHistory.cs

EditorHistory (Caretaker) – клас-«історія», що керує стеками Undo та Redo.

Обов’язки включають:

- Push(memento) – додає стан в історію;
- Undo() – повертає один крок назад;
- Redo() – повертає один крок вперед.

EditorHistory не знає, що містить Memento, він лише зберігає і повертає його.

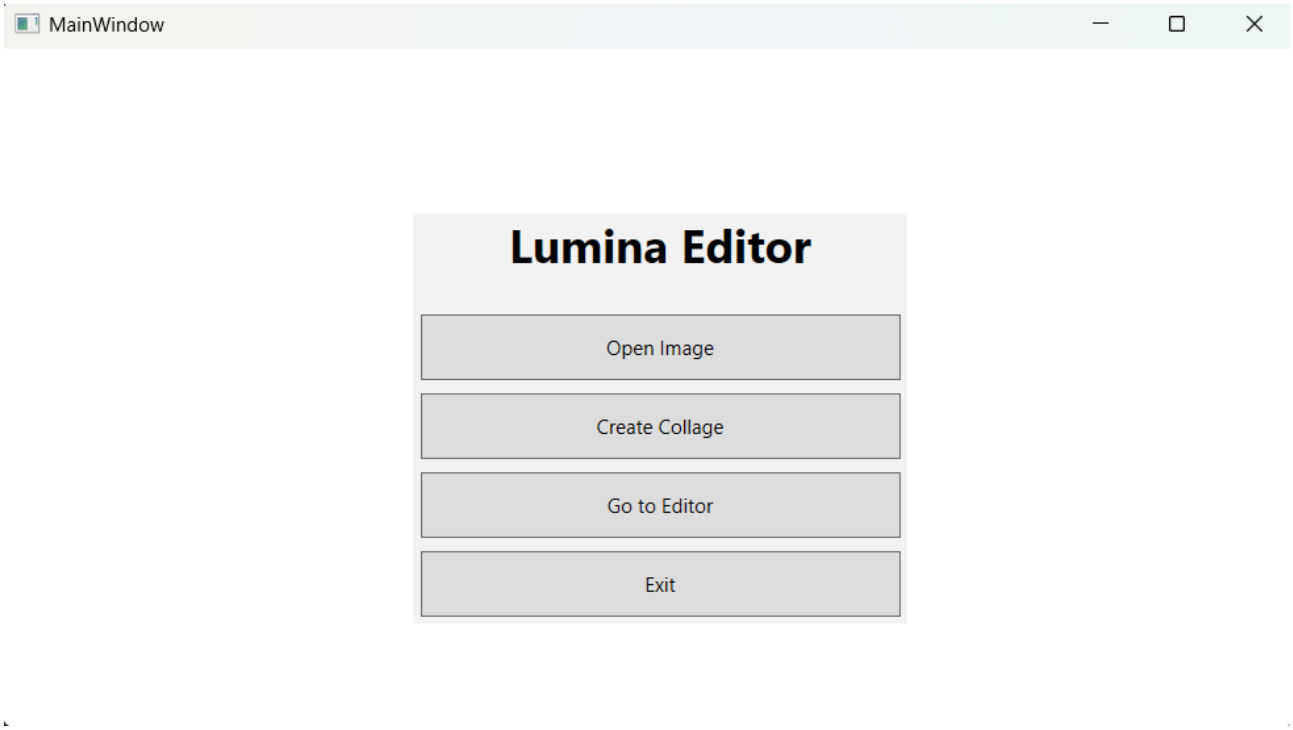


Рисунок 5 – Головна сторінка

Рисунок 6 – Сторінка редактора з кнопками Undo та Redo

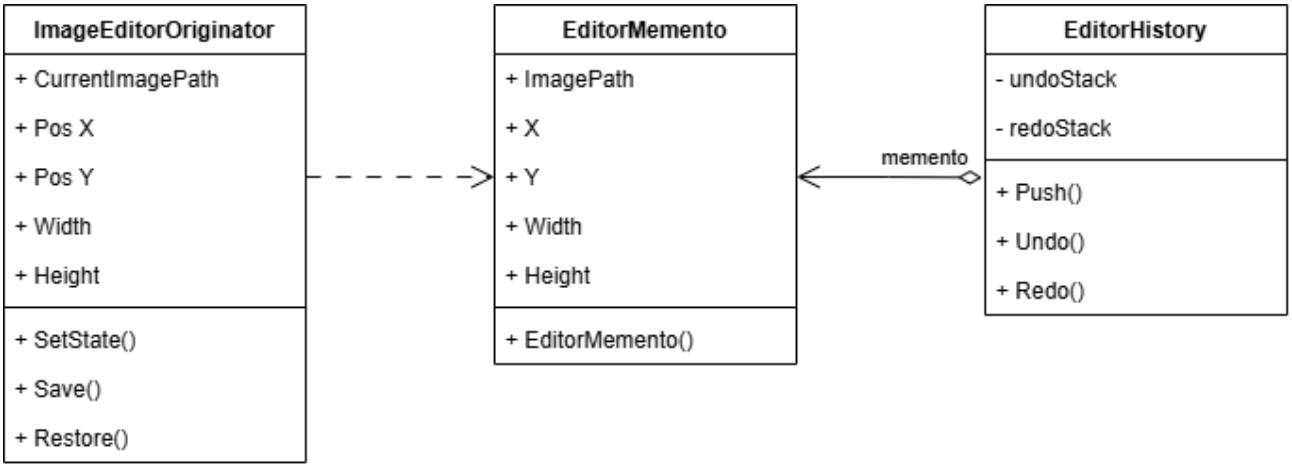


Рисунок 7 – Діаграма класів, яка представляє використання шаблону в реалізації системи

Переваги використання Memento в даному проєкті:

- Повноцінна підтримка Undo/Redo (редактор може відкотити будь-яку дію, яка змінює стан зображення).
- Інкапсуляція (жоден інший клас не отримує доступ до внутрішніх полів стану редактора).
- Простота масштабування (до Memento легко додати тип ефекту, список шарів та параметри трансформацій).
- Безпека (Memento не дозволяє іншим класам змінювати свій вміст, що гарантує, що історія змін не буде пошкоджена або переписана).
- Незалежність від UI (патерн реалізований у логіці редактора, UI лише натискає кнопки Undo/Redo).

Висновки

У ході виконання лабораторної роботи було реалізовано патерн Memento, що забезпечує збереження та відновлення станів об’єкта без порушення інкапсуляції. На прикладі редактора зображень було створено три основні компоненти: Originator (ImageEditorOriginator), який містить внутрішній стан зображення, Memento, який зберігає цей стан у незмінному вигляді, та Caretaker (EditorHistory), що керує історією операцій і надає можливості Undo та Redo.

Реалізація патерну дозволила розробити механізм відкату змін зображення, що є критично важливим для будь-якого реального графічного редактора. У балансі між інкапсуляцією та гнучкістю Memento дає змогу зберегти цілісність внутрішніх даних редактора і водночас забезпечити швидке та безпечне відновлення попередніх станів. Історія змін упорядкована і повністю контролюється caretaker-класом, що спрощує реалізацію послідовного Undo/Redo.

Перевага Memento у цьому застосуванні полягає в тому, що логіка роботи редактора та логіка керування історією змін розділені, що суттєво підвищує модульність і розширюваність системи. Додавання нових ефектів або операцій редагування зображення не потребує змін у механізмі Undo/Redo – достатньо щоразу створювати знімок стану.

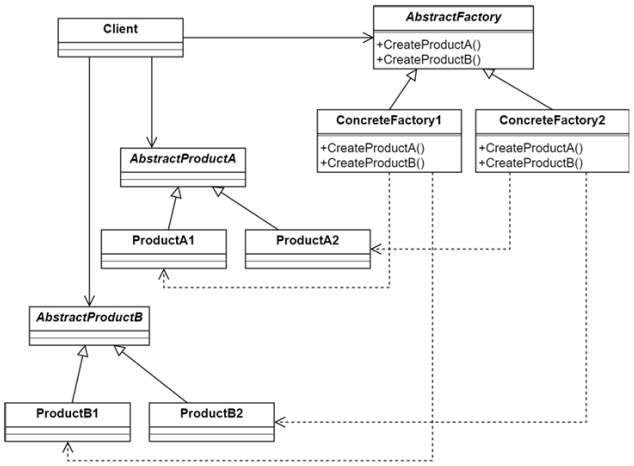
У результаті розроблений редактор зображень отримав повноцінний механізм історії, який може масштабуватися та адаптуватися до складніших сценаріїв. Використання патерну Memento продемонструвало свою ефективність у забезпеченні стабільності, керованості та зручності роботи користувача з графічними даними.

Контрольні питання

1. Яке призначення шаблону «Абстрактна фабрика»?

Призначення шаблону «Абстрактна фабрика» – надати інтерфейс для створення сімейств пов'язаних або залежних об'єктів без прив'язки до їхніх конкретних класів. Він гарантує, що створені об'єкти будуть узгодженими між собою (наприклад, всі елементи інтерфейсу в одному стилі або всі компоненти для однієї бази даних).

2. Нарисуйте структуру шаблону «Абстрактна фабрика»



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

Класи та їхня взаємодія:

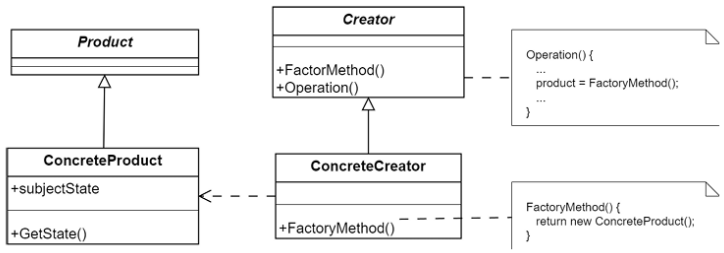
- Клієнт (Client): Отримує конкретну фабрику (наприклад, через конструктор). Не знає про конкретні класи продуктів.
- Абстрактна Фабрика (AbstractFactory): Визначає "меню" доступних продуктів.
- Конкретна Фабрика (ConcreteFactory): Клієнт викликає її методи створення (createProductA()). Фабрика інстанціює та повертає конкретний продукт.
- Абстрактний Продукт (AbstractProduct): Забезпечує спільний інтерфейс, через який Клієнт працює з продуктами.
- Конкретний Продукт (ConcreteProduct): Створюється фабрикою і використовується клієнтом.

Взаємодія: Клієнт, маючи посилання на AbstractFactory, викликає методи створення продуктів. ConcreteFactory, отримавши виклик, створює екземпляр ConcreteProduct і повертає його як AbstractProduct. Клієнт працює з усіма продуктами через їх абстрактні інтерфейси, гарантуючи узгодженість.

4. Яке призначення шаблону «Фабричний метод»?

Призначення шаблону «Фабричний метод» – надати підкласам можливість вирішувати, який саме клас створювати. Він визначає інтерфейс для створення об'єкта, але залишає вибір конкретного типу цього об'єкта за підкласами. Це дозволяє делегувати створення об'єктів нащадкам, зберігаючи спільну логіку в базовому класі.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

Класи та їхня взаємодія:

- Creator (Творець): Містить код, який використовує продукт. Він викликає свій фабричний метод, щоб отримати цей продукт, але не знає його конкретного типу.
- ConcreteCreator (Конкретний Творець): Відповідає за створення конкретного об'єкта. Перевизначає factoryMethod() і повертає новий екземпляр ConcreteProduct.
- Product (Продукт): Загальний інтерфейс для всіх об'єктів, які може створити Творець.
- ConcreteProduct (Конкретний Продукт): Реалізація продукту, яку повертає ConcreteCreator.

Взаємодія: Клієнт викликає метод базового класу Creator, який всередині себе використовує factoryMethod(). Оскільки цей метод перевизначений у ConcreteCreator, він повертає потрібний ConcreteProduct. Базова логіка Creator тепер працює з цим продуктом через інтерфейс Product.

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

Мета:

- Фабричний метод: Створює один продукт, делегуючи вибір його конкретного типу підкласам.
- Абстрактна фабрика: Створює сімейство пов'язаних продуктів. Клас фабрики явно визначає, які саме продукти створювати.

Спосіб реалізації:

- Фабричний метод: Використовує спадкування та віртуальні методи.
- Абстрактна фабрика: Використовує композицію – окремий об'єкт-фабрика передається клієнту.

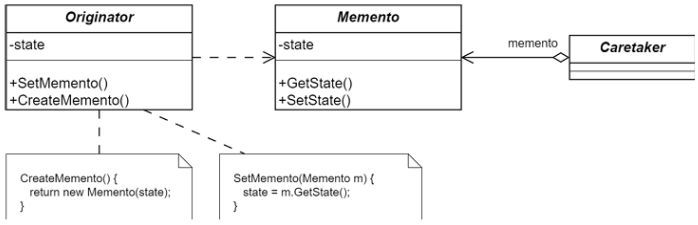
Масштаб:

- Фабричний метод – це один метод.
- Абстрактна фабрика – це цілий об'єкт з кількома методами.

8. Яке призначення шаблону «Знімок»?

Призначення шаблону «Знімок» (Memento) — зберегти внутрішній стан об'єкта так, щоб пізніше можна було відновити об'єкт у цьому стані, не порушуючи принцип інкапсуляції (тобто не розкриваючи його внутрішньої структури).

9. Нарисуйте структуру шаблону «Знімок».



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

Класи та їхня взаємодія:

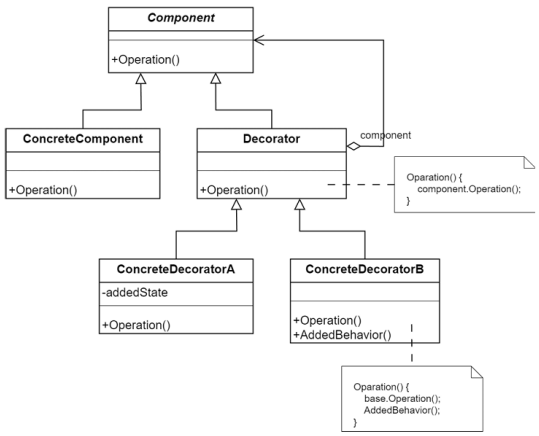
- Originator (Творець): Має методи createMemento(), який повертає новий об'єкт Memento, що містить поточний стан Творця та restore(memento), який відновлює свій внутрішній стан з отриманого об'єкта Memento.
- Memento (Знімок): Незмінний об'єкт, який зберігає "знімок" стану Творця. Його конструктор та методи доступу викликаються лише Творцем.
- Caretaker (Опікун): Зберігає колекцію об'єктів Memento (стек для реалізації undo). Він може запросити у Творця новий знімок або передати Творцю знімок для відновлення, але не "заглядає" всередину Memento.

Взаємодія: Опікун просить Творця створити знімок і зберігає його. Коли потрібно відновити стан, Опікун передає збережений знімок назад Творцю, який використовує його для відновлення власних полів.

11. Яке призначення шаблону «Декоратор»?

Призначення шаблону «Декоратор» – динамічно додавати об'єкту нові обов'язки (функціональність). Він є гнучкою альтернативою спадкуванню для розширення функціональності, дозволяючи "обгортати" об'єкти в різні декоратори, кожен з яких додає щось своє.

12. Нарисуйте структуру шаблону «Декоратор».



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

Класи та їхня взаємодія:

- Component (Компонент): Забезпечує єдиний інтерфейс для клієнта.
- ConcreteComponent (Конкретний Компонент): Об'єкт, який потрібно розширити.
- Decorator (Декоратор): Містить посилання (wrappedComponent) на об'єкт Component. Усі виклики методів він перенаправляє цьому об'єкту.
- ConcreteDecorator (Конкретний Декоратор): Перевизначає методи базового Decorator, додаючи власну логіку до виклику методу wrappedComponent.

Взаємодія: Клієнт працює з усіма об'єктами через інтерфейс Component. Він "обгортає" ConcreteComponent в один або кілька ConcreteDecorator. Коли клієнт викликає метод, виклик спочатку потрапляє до зовнішнього декоратора, який виконує свій код, потім викликає метод внутрішнього компонента (яким може бути інший декоратор або базовий об'єкт), і так далі.

14. Які є обмеження використання шаблону «декоратор»?

Складність конфігурації: Якщо об'єкт обгорнутий у кілька декораторів, важко зрозуміти його точну структуру та порядок виконання коду.

"Дизайн із сюрпризом": Клієнт, який очікує працювати з базовим компонентом, може бути збентежений його розширеною поведінкою, оскільки тип об'єкта (інтерфейс) не змінюється.

Велика кількість класів: Кожна нова функціональність потребує створення нового класу-декоратора, що може призвести до "роздування" кодової бази дрібними класами.

Складно видаляти функціональність: Неможливо просто "зняти" конкретний декоратор з середини ланцюжка. Зазвичай потрібно будувати ланцюжок заново.