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

«Патерни проектування»

Виконав:
студент групи ІА–32
Лось Ярослав

Київ 2025

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

Зміст

Теоретичні відомості 3
Хід роботи 4
1. Загальний опис роботи 4
2. Опис класів програмної системи 5
3. Опис результатів коду 12
4. Діаграми класів 13
Висновки 14
Контрольні запитання 15

Теоретичні відомості

Шаблон «Memento»

Призначення: 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 мементо об'єктів в системі.

Таким чином вдається досягти наступних цілей:

- зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;
- передача об'єктів «Memento» лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;
- збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

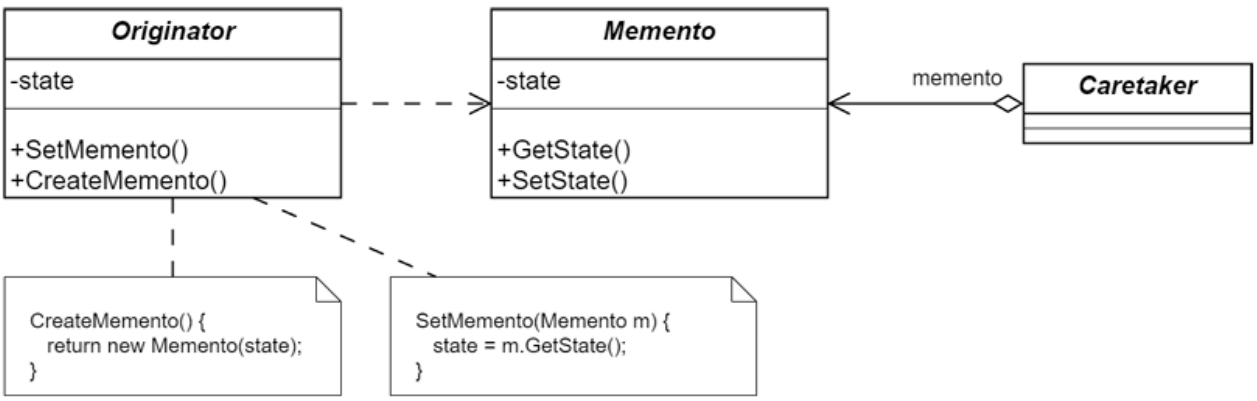


Рисунок 6.3. Структура шаблону «Знімок»

Шаблон «Мементо» дуже зручно використати разом з шаблоном «Команда» для реалізації «скасовних» дій – дані про дію зберігаються в мементо, а команда має можливість вважати і відновити початкове положення відповідних об'єктів.

Переваги та недоліки:

+ Не порушує інкапсуляцію вихідного об’єкта.

+ Спрощує структуру вихідного об’єкта. Не потрібно зберігати історію версій свого стану.

- Вимагає багато пам’яті, якщо клієнти дуже часто створюють знімки.

- Може спричинити додаткові витрати пам’яті, якщо об’єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

Хід роботи

1. **Музичний програвач** (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам’ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

1. Загальний опис роботи

Основна мета роботи полягає в реалізації механізму збереження та відновлення внутрішнього стану об'єкта без порушення його інкапсуляції. У нашій предметній області «Музичний програвач» часто виникає потреба скасувати зміни у плейлисті (наприклад, якщо користувач випадково видалив пісню або невдало змінив порядок треків). Для цього використаємо патерн **Memento** (Знімок).

Система дозволяє зберігати "знімки" стану плейлиста перед внесенням змін. Якщо користувача не влаштовує поточний стан, він може повернутися до попереднього версії (операція Undo).

Програмна система складається з трьох ключових компонентів патерну:

- **Originator (Ініціатор)**: Клас Playlist, який містить поточний стан (список пісень) і вміє створювати свої знімки та відновлюватися з них.
- **Memento (Знімок)**: Клас PlaylistMemento, що є пасивним контейнером для збереження копії стану плейлиста.
- **Caretaker (Опікун)**: Клас HistoryKeeper, який відповідає за зберігання історії знімків, але не має права змінювати їх вміст або заглядати всередину.

2. Опис класів програмної системи

Клас PlaylistMemento (Memento)

Опис:

Клас PlaylistMemento представляє собою незмінний знімок стану плейлиста в конкретний момент часу. Його єдина задача - зберігати дані. Згідно з філософією патерну, внутрішній вміст цього класу має бути доступним лише для Originator-a (Playlist).

Ключові характеристики:

- Іммутабельність: Стан записується лише один раз через конструктор. Сеттерів немає.
- Глибоке копіювання: У конструкторі відбувається створення нового списку (new ArrayList<>(songs)). Це ключовий момент: якщо просто зберегти посилання на оригінальний список, то зміни в плейлисті змінять і "збережений" стан, що зруйнує ідею патерну.

```java
public class PlaylistMemento {
    2 usages
    private final List<Song> songsState;

    1 usage
    public PlaylistMemento(List<Song> songs) {
        this.songsState = new ArrayList<>(songs);
    }

    1 usage
    public List<Song> getSavedSongs() {
        return songsState;
    }
}
```

Рисунок 1 – Код класу PlaylistMemento

Інтерфейс Originator

Опис:

Інтерфейс Originator визначає контракт для об'єктів, які мають здатність зберігати свій стан та відновлюватися з нього. Це гарантує, що будь-який клас, який реалізує цей інтерфейс, обов'язково матиме методи для створення "знімка" (save) та відкату змін (restore).

Характеристики:

- Оголошує метод save(), який має повертати об'єкт-знімок (Memento).
- Оголошує метод restore(), який приймає знімок і застосовує його.

```java
public interface Originator {
    1 usage  1 implementation
    PlaylistMemento save();

    2 usages  1 implementation
    void restore(PlaylistMemento memento);
}
```

Рисунок 2 – Код інтерфейсу Originator

Класи Song

Опис:

Клас Song є базовим інформаційним об'єктом системи, з якого складається стан Ініціатора (у контексті нашого патерну Memento).

Характеристики:

- Інформаційна сутність: Зберігає назву треку (title) та ім'я виконавця (artist).

```java
public class Song {
    4 usages
    private String title;
    4 usages
    private String artist;

    4 usages
    public Song(String title, String artist) {
        this.title = title;
        this.artist = artist;
    }

    @Override
    public String toString() {
        return title + " - " + artist;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Song song = (Song) obj;
        return title.equals(song.title) && artist.equals(song.artist);
    }
}
```

Рисунок 3 – Код класу Song

Клас Playlist (Originator)

Опис:

Клас Playlist виступає в ролі Ініціатора (Originator). Це основний клас бізнес-логіки, який містить важливі дані (список пісень), що змінюються з часом. Він не займається збереженням історії самостійно, але надає інструменти для створення знімка (save) та відновлення з нього (restore).

Ключові характеристики:

- Управління станом**:** Методи addSong() та removeSong() змінюють поточний стан об'єкта.
- Метод save(): Створює та повертає новий об'єкт PlaylistMemento, передаючи йому поточний список пісень.
- Метод restore (Memento): Приймає об'єкт знімка, дістає з нього збережений список і замінює ним свій поточний стан. Це миттєво повертає плейлист у минуле.

Код класу Playlist

```
public class Playlist implements Originator {
    private List<Song> songs = new ArrayList<>();

    public void addSong(Song song) {
        songs.add(song);
        System.out.println("Додано: " + song);
    }

    public void removeSong(Song song) {
        if (songs.remove(song)) {
            System.out.println("Видалено: " + song);
        } else {
            System.out.println("Пісню не знайдено: " + song);
        }
    }
```

```
    @Override
    public PlaylistMemento save() {
        System.out.println("[Originator] Створення знімка стану");
        return new PlaylistMemento(songs);
    }

    @Override
    public void restore(PlaylistMemento memento) {
        this.songs = memento.getSavedSongs();
        System.out.println("[Originator] Стан відновлено");
    }

    @Override
    public String toString() {
        return "Плейлист: " + songs.toString();
    }
}
```

Клас HistoryKeeper (Caretaker)

Опис:

Клас HistoryKeeper виконує роль Опікуна. Він знає, коли робити знімки і коли їх відновлювати, але не знає, що саме міститься всередині знімка.

Ключові характеристики:

- Стек історії: Використовує структуру даних Stack<PlaylistMemento> для реалізації принципу LIFO (Last In, First Out). Останній збережений стан буде першим при скасуванні дій.
- Метод backup(): Звертається до плейлиста, просить його зберегтися (playlist.save()) і кладе отриманий знімок у стек.
- Метод undo(): Перевіряє, чи є збережені стани. Якщо так - дістає верхній знімок зі стека (pop()) і передає його плейлисту для відновлення (playlist.restore()).

Код класу HistoryKeeper

```
public class HistoryKeeper {
    private Stack<PlaylistMemento> history = new Stack<>();
    private Playlist playlist;

    public HistoryKeeper(Playlist playlist) {
        this.playlist = playlist;
    }

    public void backup() {
        history.push(playlist.save());
    }

    public void undo() {
        if (!history.isEmpty()) {
            history.pop();
            if (!history.isEmpty()) {
                playlist.restore(history.peek());
            } else {
                System.out.println("Історія порожня!");
            }
        } else {
            System.out.println("Нічого скасовувати!");
        }
    }

    public void restoreLastSave() {
        if (!history.isEmpty()) {
            playlist.restore(history.pop());
        } else {
            System.out.println("Немає збережених станів!");
        }
    }
}
```

Клас MementoClient (Client)

Опис:

Клас MementoClient демонструє сценарій роботи користувача. Він створює плейлист та менеджер історії, додає пісні, робить збереження, вносить деструктивні зміни (видалення) і демонструє успішне відновлення втрачених даних.

```
public class MementoClient {

    public static void main(String[] args) {
        Playlist myPlaylist = new Playlist();
        HistoryKeeper history = new HistoryKeeper(myPlaylist);

        myPlaylist.addSong(new Song( title: "Song 1",  artist: "Artist A"));
        myPlaylist.addSong(new Song( title: "Song 2",  artist: "Artist B"));
        System.out.println(myPlaylist);

        history.backup();

        myPlaylist.addSong(new Song( title: "Song 3",  artist: "Artist C"));
        myPlaylist.removeSong(new Song( title: "Song 1",  artist: "Artist A"));

        System.out.println("\nПоточний (зіпсований) стан:");
        System.out.println(myPlaylist);

        System.out.println("\nСкасування змін...");
        history.restoreLastSave();

        System.out.println("\nСтан після відновлення:");
        System.out.println(myPlaylist);
    }

}
```

Рисунок 4 – Код класу MementoClient

3. Опис результатів коду

Після запуску клієнтського класу ми отримуємо лог, який показує, як змінювався стан об'єкта в часі.

```
Додано: Song 1 - Artist A
Додано: Song 2 - Artist B
Плейлист: [Song 1 - Artist A, Song 2 - Artist B]
[Originator] Створення знімка стану
Додано: Song 3 - Artist C
Видалено: Song 1 - Artist A

Поточний (зіпсований) стан:
Плейлист: [Song 2 - Artist B, Song 3 - Artist C]

Скасування змін...
[Originator] Стан відновлено

Стан після відновлення:
Плейлист: [Song 1 - Artist A, Song 2 - Artist B]
```

Рисунок 5 – Результат виконання програми

Аналіз результатів:

1. Спочатку плейлист мав пісні 1 та 2. Цей стан було збережено.
2. Потім було додано пісню 3 і видалено пісню 1. Плейлист змінився.
3. Після виклику відновлення, плейлист повернувся точно до того стану, який був зафіксований у момент збереження (пісні 1 та 2 на своїх місцях).
4. Це підтверджує, що об'єкт PlaylistMemento коректно зберіг копію даних, а Playlist зміг використати цю копію для самовідновлення.

4. Діаграми класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами побудуємо UML-діаграми класів.

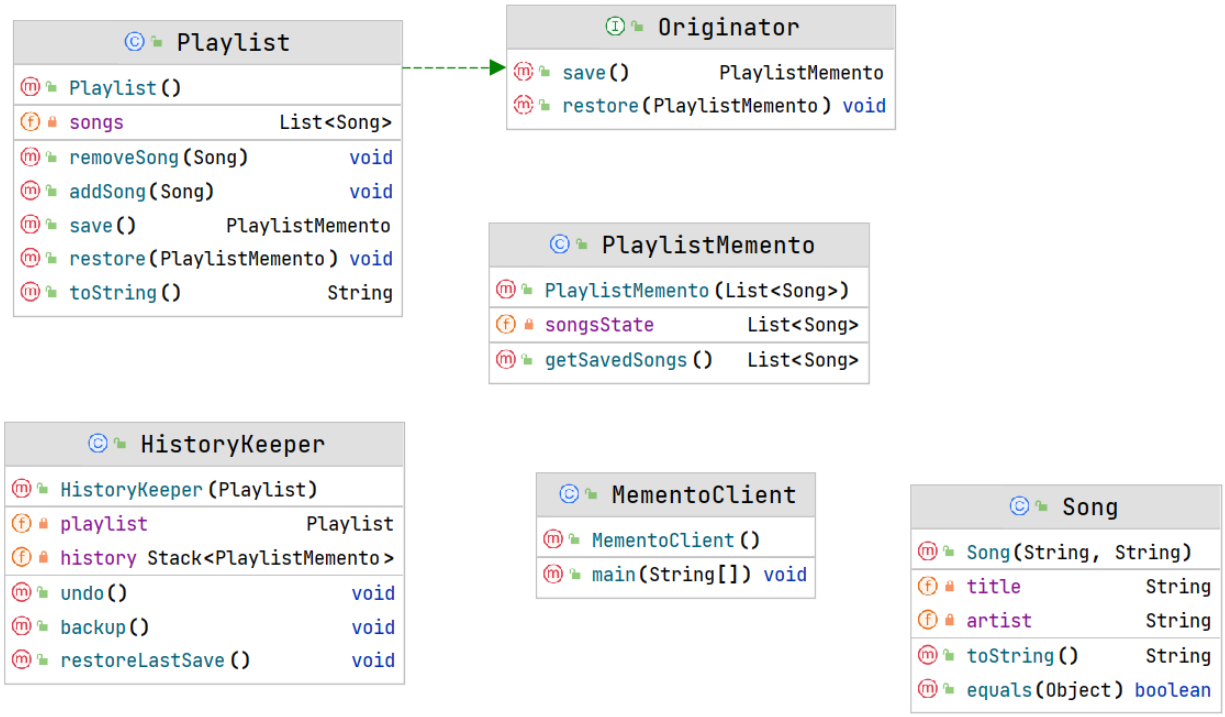


Рисунок 6 – Діаграма класів (спрощена)

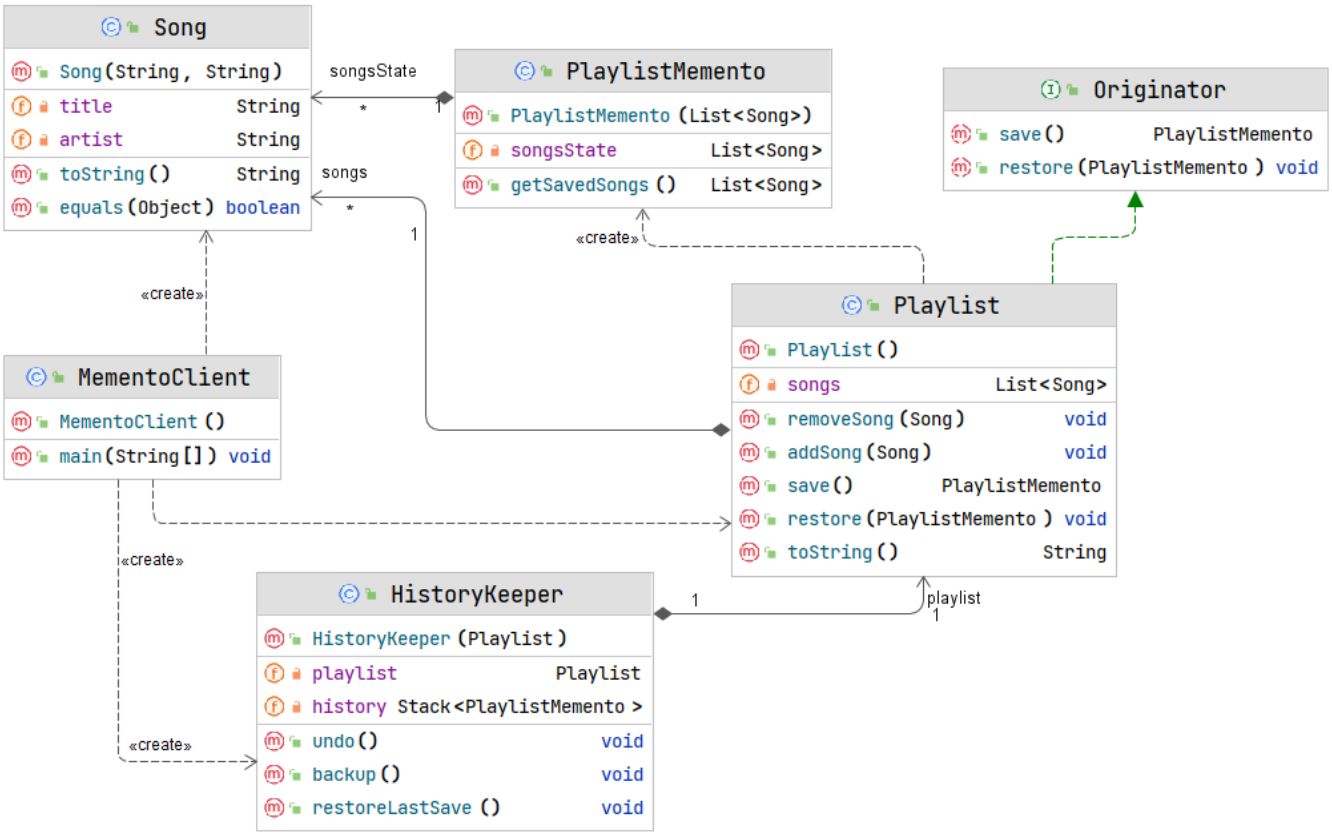


Рисунок 7 – Діаграма класів (повна)

Висновки

В ході виконання роботи було реалізовано патерн «Memento» для забезпечення можливості скасування дій у музичному програвачі.

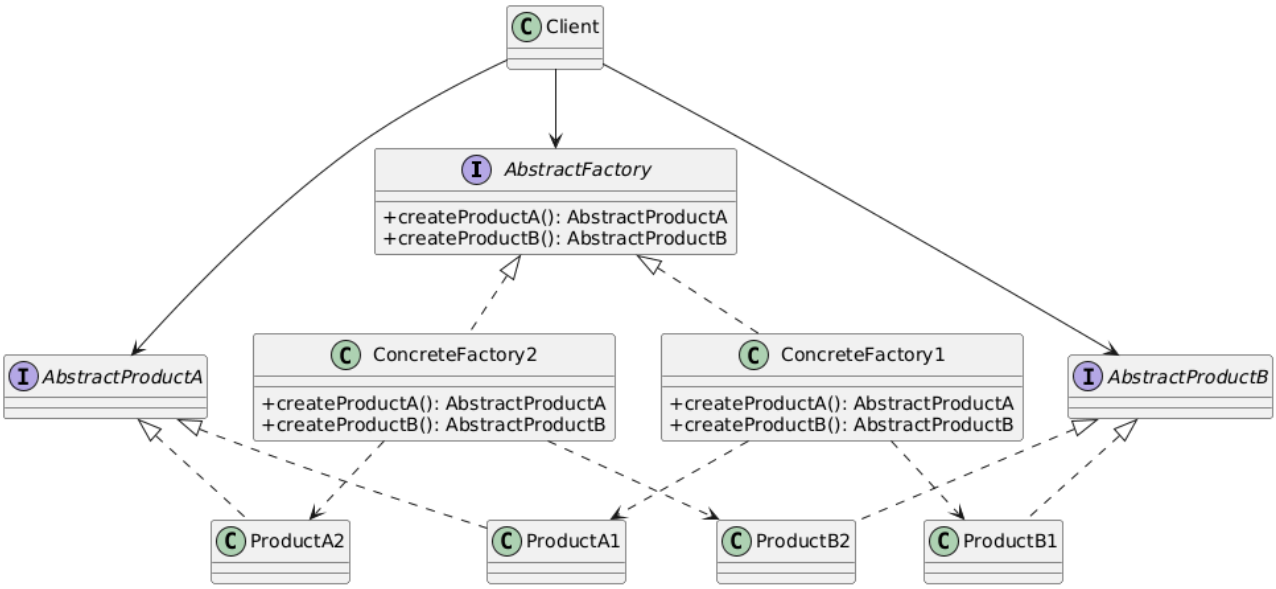
Реалізація продемонструвала важливість глибокого копіювання при створенні знімка (для посилальних типів даних, таких як списки), щоб уникнути побічних ефектів. Головною перевагою використання патерну стало дотримання принципу інкапсуляції: хоча стан об'єкта зберігається десь зовні (у HistoryKeeper), ніхто, крім самого Playlist, не має доступу до зміни або читання цих збережених даних.

Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?

Призначення: Шаблон «Абстрактна фабрика» (Abstract Factory) надає інтерфейс для створення сімейств взаємопов'язаних або залежних об'єктів без зазначення їхніх конкретних класів. Він дозволяє клієнту працювати з фабрикою абстрактно, не знаючи, яку саме серію продуктів (наприклад, стиль інтерфейсу "Windows" чи "MacOS") вона створює.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

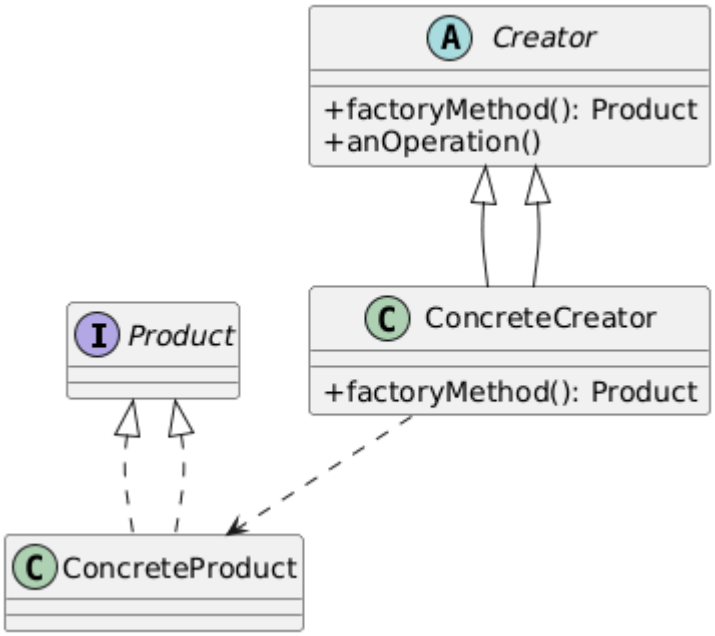
- **AbstractFactory:** Інтерфейс, що оголошує методи створення абстрактних продуктів.
- **ConcreteFactory:** Реалізує методи створення конкретних продуктів певної серії (сімейства).
- **AbstractProduct:** Інтерфейс для типу продукту.
- **ConcreteProduct:** Конкретна реалізація продукту.
- **Client:** Використовує тільки інтерфейси фабрики та продуктів.

- **Взаємодія:** Клієнт викликає методи фабрики для створення об'єктів. Фабрика повертає конкретні продукти, сумісні між собою.

4. Яке призначення шаблону «Фабричний метод»?

Призначення: Шаблон «Фабричний метод» (Factory Method) визначає інтерфейс для створення об'єкта, але дозволяє підкласам вирішувати, який клас інстанціювати. Фабричний метод дозволяє класу делегувати створення об'єктів своїм підкласам.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

- **Product:** Інтерфейс об'єктів, які створює метод.
- **ConcreteProduct:** Реалізація продукту.
- **Creator:** Оголошує фабричний метод, який повертає об'єкт типу Product. Може містити базову реалізацію.
- **ConcreteCreator:** Перевизначає фабричний метод, повертаючи екземпляр ConcreteProduct.

- **Взаємодія:** Клієнт звертається до Creator. Creator у своїй логіці викликає factoryMethod(). Реалізація цього методу в підкласі (ConcreteCreator) створює і повертає конкретний об'єкт.

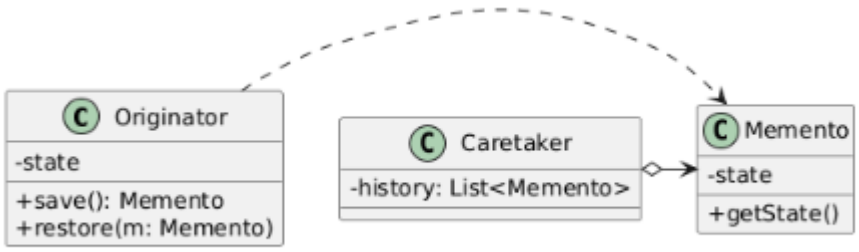
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

- **Рівень абстракції:** «Фабричний метод» працює з одним продуктом (або ієрархією одного продукту), а «Абстрактна фабрика» створює сімейства пов'язаних продуктів.
- **Спосіб реалізації:** «Фабричний метод» базується на **спадкуванні** (підкласи вирішують, що створювати), а «Абстрактна фабрика» базується на **композиції** (клієнт має посилання на об'єкт фабрики, який створює продукти).
- Часто методи всередині «Абстрактної фабрики» реалізуються як «Фабричні методи».

8. Яке призначення шаблону «Знімок»?

Призначення: Шаблон «Знімок» (Memento) дозволяє зберігати і відновлювати внутрішній стан об'єкта, не порушуючи його інкапсуляції. Це дозволяє повернути об'єкт до попереднього стану (undo).

9. Нарисуйте структуру шаблону «Знімок».



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

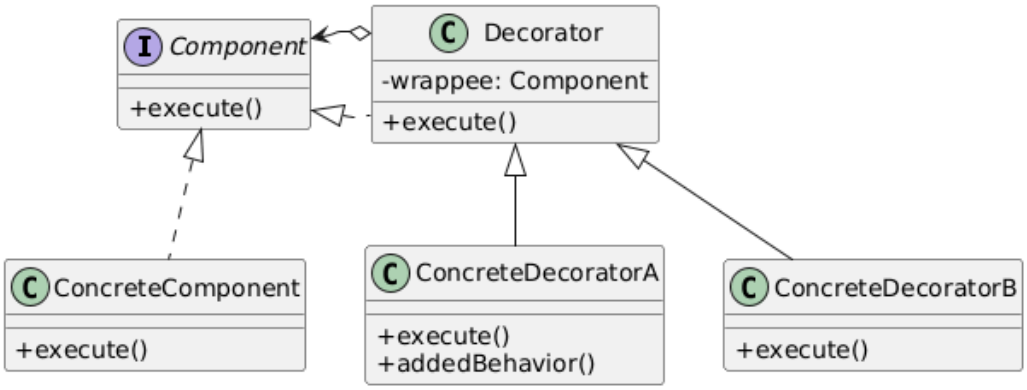
- **Originator (Ініціатор):** Об'єкт, стан якого треба зберігати. Створює Memento і відновлюється з нього.

- **Memento (Знімок):** Зберігає стан Originator-а. Захищений від доступу інших об'єктів (окрім Originator).
- **Caretaker (Опікун):** Відповідає за зберігання знімків (наприклад, у стеку історії), але не має права змінювати їх або читати вміст.
- **Взаємодія:** Caretaker просить Originator створити знімок. Originator створює Memento і віддає його Caretaker-у. Коли треба відновити стан, Caretaker повертає Memento Originator-у.

11. Яке призначення шаблону «Декоратор»?

Призначення: Шаблон «Декоратор» (Decorator) дозволяє динамічно додавати об'єктам нові обов'язки (функціональність), обгортаючи їх у корисні "обгортки". Це гнучка альтернатива успадкуванню для розширення функціональності.

12. Нарисуйте структуру шаблону «Декоратор».



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

- **Component:** Спільний інтерфейс для об'єктів, які можна декорувати.
- **ConcreteComponent:** Базовий клас, до якого додається функціонал.
- **Decorator:** Абстрактний клас, що реалізує інтерфейс Component і містить посилання на обгорнутий об'єкт Component. Передає виклики цьому об'єкту.
- **ConcreteDecorator:** Додає нову поведінку до або після виклику методу базового декоратора.

- **Взаємодія:** Клієнт працює з Component. Він може вкладати компоненти один в одного (матрьошка). Виклик методу проходить крізь ланцюжок декораторів, кожен з яких може додати свою логіку, і зрештою досягає базового компонента.

14. Які є обмеження використання шаблону «декоратор»?

- **Велика кількість дрібних об'єктів:** Система наповнюється безліччю маленьких об'єктів-обгорток, що може ускладнити налагодження (debugging).
- **Складність конфігурації:** Створення об'єкта, що обгорнутий у 5-6 декораторів, може вимагати громіздкого коду ініціалізації (якщо не використовувати Фабрику або Builder).
- **Залежність від порядку:** Важко реалізувати декоратор так, щоб його поведінка не залежала від того, в якому порядку він доданий в стек декораторів.
- **Втрата ідентичності:** Декоратор має той самий інтерфейс, що і компонент, але це інший об'єкт. Тому перевірки типу instanceof стосовно конкретного класу-компонента можуть не спрацювати.