

Національний технічний університет України «Київський
політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №6

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:

студент групи IA-33

Грицай Андрій

Перевірив:

асистент кафедри ICT

Мягкий Михайло Юрійович

Тема: Патерни проектування

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

Тема лабораторного циклу:

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 редактування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіоформатів, евалайзер.

Теоретичні відомості Абстрактна фабрика (Abstract Factory)

Шаблон "Абстрактна фабрика" надає інтерфейс для створення групи взаємозалежних об'єктів без визначення їх конкретних класів. Використовується, коли система повинна працювати з кількома наборами об'єктів, які пов'язані між собою. Абстрактна фабрика визначає методи для створення кожного типу об'єктів, а конкретні реалізації фабрик створюють ці об'єкти. У Java цей шаблон зазвичай реалізується шляхом створення абстрактного класу або інтерфейсу фабрики, а потім конкретних фабрик, які створюють необхідні об'єкти. Це дозволяє легко змінювати цілі набори об'єктів, забезпечуючи незалежність клієнтського коду від конкретних реалізацій.

Фабричний метод (Factory Method)

Шаблон "Фабричний метод" визначає інтерфейс для створення об'єктів, але дозволяє підкласам вирішувати, який саме об'єкт буде створено. Він надає гнуучкість у виборі класів, які потрібно створити, і сприяє відкритості до розширень. У Java цей шаблон часто реалізується за допомогою абстрактного класу або інтерфейсу з одним методом, який повертає створений об'єкт, а конкретні підкласи реалізують цей метод для створення потрібних об'єктів. Це корисно, коли система повинна створювати об'єкти, типи яких невідомі заздалегідь.

Збереження стану (Memento)

Шаблон "Memento" дозволяє зберігати і відновлювати внутрішній стан об'єкта без порушення його інкапсуляції. Це корисно для реалізації функцій "Скасувати" або "Повернутися до попереднього стану". У шаблоні є три учасники: "Одержанувач" (об'єкт, стан якого зберігається), "Опікун" (керує збереженими станами) і "Сувенір" (зберігає стан). У Java це зазвичай реалізується через класи, де сувенір є внутрішнім класом об'єкта. Це дозволяє зберігати історію змін і повертатися до попереднього стану, забезпечуючи контроль за змінами об'єкта.

Спостерігач (Observer)

Шаблон "Спостерігач" визначає залежність "один-до-багатьох", коли зміна стану одного об'єкта повідомляє всім його підписникам (спостерігачам). Це забезпечує синхронізацію об'єктів і дозволяє автоматично оновлювати їх у разі змін. У Java цей шаблон реалізується через інтерфейси, такі як `Observer` і `Observable`, або за допомогою механізму слухачів (Listeners). Клас-спостережуваний повідомляє всіх зареєстрованих спостерігачів про зміни. Це корисно для систем, де дані змінюються динамічно, наприклад, GUI або обробка подій.

Декоратор (Decorator)

Шаблон "Декоратор" дозволяє динамічно додавати нові функції об'єкту, не змінюючи його структури. Він обгортає оригінальний об'єкт у додатковий об'єкт-декоратор, який реалізує ту ж саму базову функціональність, але додає нову поведінку. У Java цей шаблон часто реалізується за допомогою абстрактного класу або інтерфейсу, який реалізують і базовий клас, і декоратори. Це дозволяє створювати гнучкі системи, де об'єкти можуть бути розширені без множинного успадкування, зберігаючи прозорий інтерфейс для клієнта.

Хід роботи

У цій роботі я використовую патерн "Знімок" (Memento). Цей патерн було обрано як архітектурне рішення для реалізації механізму збереження та відновлення стану музичного програвача без порушення інкапсуляції.

У процесі роботи над проектом виникла необхідність зберігати поточні налаштування плеєра (гучність, режим перемішування, поточний трек, позицію відтворення та налаштування евалайзера) для можливості їх відновлення після перезапуску програми або повернення до попереднього стану.

Без використання патерну Memento, реалізація такої логіки вимагала б розкриття внутрішньої структури класу `MusicPlayer`. Зовнішнім класам, що відповідають за збереження (наприклад, у базу даних), довелося б напряму звертатися до приватних полів плеєра, що порушує принцип інкапсуляції. Це призвело б до жорсткої залежності коду збереження від внутрішньої реалізації плеєра: будь-яка зміна в атрибутах `MusicPlayer` вимагала б переписування логіки збереження в інших частинах системи.

Патерн Memento вирішує цю проблему елегантно. Він дозволяє об'єкту `MusicPlayer` (Originator) самому створювати знімок свого стану у вигляді спеціального об'єкта `PlayerMemento`.

Об'єкт `PlayerMemento` зберігає всі необхідні дані, але доступ до них має лише сам `Originator`. Інші компоненти, такі як `Caretaker` (або система збереження в БД), можуть лише зберігати цей об'єкт, не маючи можливості змінювати його вміст чи досліджувати внутрішню структуру. Коли необхідно відновити стан,

Мemento передається назад у MusicPlayer, який витягує з нього дані та відновлює свої налаштування. Це забезпечує надійне збереження стану та ізоляє внутрішню структуру плеєра від механізмів персистентності.

Реалізація класів

Реалізація шаблону “Memento” вимагає створення окремих класів для реалізації кожного конкретного алгоритму.

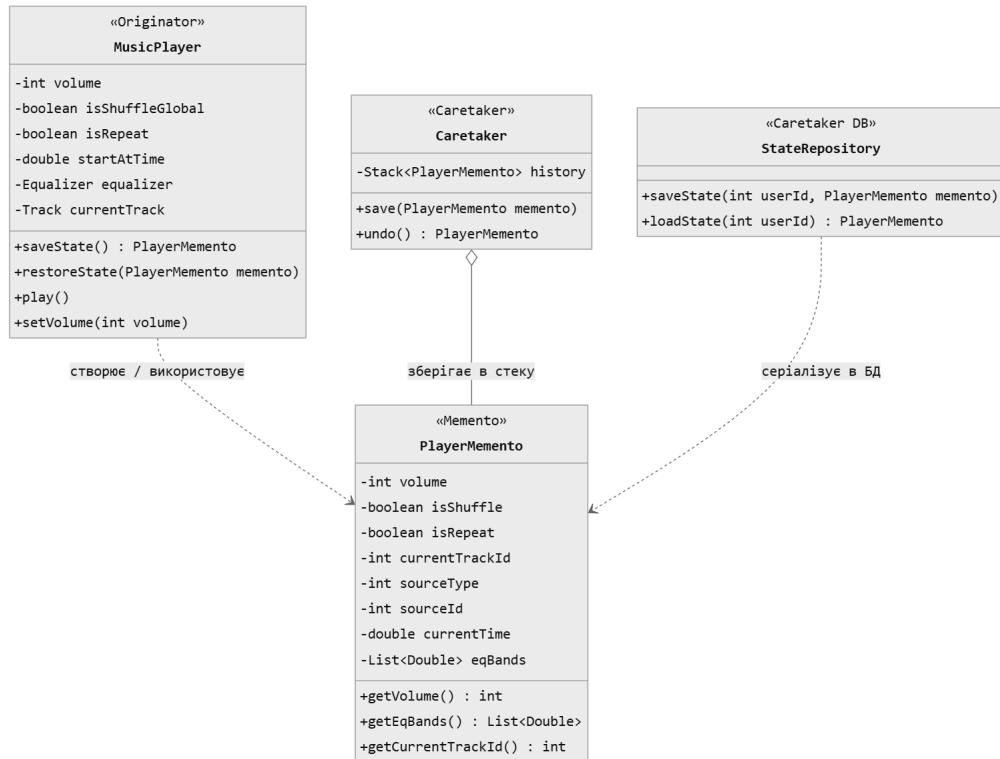


Рисунок 1. - Діаграма класів, яка представляє використання шаблону в реалізації системи

Опис створених об'єктів:

PlayerMemento (Знімок / Memento). Цей клас є пасивним контейнером даних, що зберігає внутрішній стан об'єкта MusicPlayer у певний момент часу. Він містить поля для зберігання гучності, пропорцій перемішування/повторення, ідентифікатора поточного треку, позиції відтворення та налаштувань еквалайзера (eqBands). Клас не містить бізнес-логіки й надає доступ до даних лише через геттери, забезпечуючи незмінність збереженого стану.

```

public class PlayerMemento implements Serializable {
    private static final long serialVersionUID = 1L;

    private final int volume;
    private final boolean isShuffle;
    private final boolean isRepeat;
    private final int currentTrackId;
    private final int sourceType;
  
```

```

    private final int sourceId;
    private final double currentTime;
    private final List<Double> eqBands; // Нове поле

    public PlayerMemento(int volume, boolean isShuffle,
    boolean isRepeat,
                           int currentTrackId, int sourceType,
    int sourceId,
                           double currentTime, List<Double>
    eqBands) {
        this.volume = volume;
        this.isShuffle = isShuffle;
        this.isRepeat = isRepeat;
        this.currentTrackId = currentTrackId;
        this.sourceType = sourceType;
        this.sourceId = sourceId;
        this.currentTime = currentTime;
        this.eqBands = eqBands;
    }

    public int getVolume() { return volume; }
    public boolean isShuffle() { return isShuffle; }
    public boolean isRepeat() { return isRepeat; }
    public int getCurrentTrackId() { return currentTrackId; }
    public int getSourceType() { return sourceType; }
    public int getSourceId() { return sourceId; }
    public double getCurrentTime() { return currentTime; }
    public List<Double> getEqBands() { return eqBands; }
}

```

MusicPlayer (Ініціатор / Originator). Це основний клас бізнес-логіки, стан якого необхідно зберігати. Він володіє повною інформацією про поточне відтворення. Клас реалізує два ключові методи патерну: `saveState()`, який створює та повертає новий об'єкт `PlayerMemento` з поточними параметрами , та `restoreState(PlayerMemento memento)`, який приймає знімок і відновлює налаштування плеєра (гучність, евалайзер, позицію треку) до збережених значень.

```

public class MusicPlayer {
    private int volume;
    private boolean isRepeat;
    private boolean isShuffleGlobal;
    private Track currentTrack;
    private Playable currentPlayable;
    private MediaPlayer mediaPlayer;
    private final Equalizer equalizer;
    private final TrackService trackService;

```

```

private final PlaylistService playlistService;

public MusicPlayer(TrackService trackService, PlaylistService playlistService)
{
    this.trackService = trackService;
    this.playlistService = playlistService;
    this.volume = 50;
    this.equalizer = new Equalizer();
}

public PlayerMemento saveState() {
    int trackId = (currentTrack != null) ? currentTrack.getTrackID() : -1;
    int sourceType = 0;
    int sourceId = 0;
    if (currentPlayable instanceof Playlist) {
        sourceType = 1;
        sourceId = ((Playlist) currentPlayable).getPlaylistID();
    }
    double currentTime = 0.0;
    if (mediaPlayer != null) {
        currentTime = mediaPlayer.getCurrentTime().toSeconds();
    }
    return new PlayerMemento(this.volume, this.isShuffleGlobal, this.isRepeat,
        trackId, sourceType, sourceId, currentTime, equalizer.getGains() );
}

public void restoreState(PlayerMemento memento) {
    if (memento == null) return;
    this.setVolume(memento.getVolume());
    this.setShuffle(memento.isShuffle());
    if (this.isRepeat != memento.isRepeat()) this.toggleRepeat();
    equalizer.restoreGains(memento.getEqBands());
    if (memento.getSourceType() == 1) {
        Playlist pl = playlistService.getPlaylistByID(memento.getSourceId());
        if (pl != null) this.setPlayableSource(pl);
    } else {
        Library lib = new Library();
        this.setPlayableSource(lib);
    }

    if (memento.getCurrentTrackId() != -1) {
        Track t = trackService.getTrackByID(memento.getCurrentTrackId());

```

```

        if (t != null) {
            this.playTrack(t);
        }
    }

public void setVolume(int volume) {
    this.volume = volume;
    if (mediaPlayer != null) mediaPlayer.setVolume(volume / 100.0);
}

public void setShuffle(boolean enable) {
    this.isShuffleGlobal = enable;
    if (currentPlayable != null) currentPlayable.setShuffle(enable);
}

public void toggleRepeat() {
    this.isRepeat = !this.isRepeat;
}
}

```

Caretaker (Опікун). Цей клас відповідає за зберігання та управління життєвим циклом знімків. Він використовує колекцію Stack<PlayerMemento> для зберігання історії станів. Caretaker запитує збереження стану в MusicPlayer перед виконанням операцій (наприклад, перемиканням треку) і може повернути попередній стан через метод undo(), не знаючи внутрішньої структури збережених об'єктів.

```

public class Caretaker {
    private final Stack<PlayerMemento> history = new
Stack<>();

    public void save(PlayerMemento memento) {
        history.push(memento);
    }

    public PlayerMemento undo() {
        if (!history.isEmpty()) {
            return history.pop();
        }
        return null;
    }
}

```

StateRepository (Опікун БД / Persistent Caretaker). Цей клас виконує роль опікуна для довготривалого зберігання. Він отримує об'єкт PlayerMemento і серіалізує його дані в таблицю бази даних playback_state. Це дозволяє відновлювати стан плеєра (включно з налаштуваннями евалайзера) навіть після перезапуску програми, завантажуючи дані з БД назад у об'єкт PlayerMemento.

```
public class StateRepository {  
    public void saveState(int userId, PlayerMemento memento) {  
        String sql = "INSERT OR REPLACE INTO playback_state  
(user_id, volume, is_shuffle, is_repeat, current_track_id,  
source_type, source_id, current_time, eq_settings) VALUES (?, ?,  
?, ?, ?, ?, ?, ?);  
  
        try (Connection conn = DatabaseConnection.connect();  
             PreparedStatement pstmt =  
conn.prepareStatement(sql)) {  
  
            pstmt.setInt(1, userId);  
            pstmt.setInt(2, memento.getVolume());  
            pstmt.setInt(3, memento.isShuffle() ? 1 : 0);  
            pstmt.setInt(4, memento.isRepeat() ? 1 : 0);  
            pstmt.setInt(5, memento.getCurrentTrackId());  
            pstmt.setInt(6, memento.getSourceType());  
            pstmt.setInt(7, memento.getSourceId());  
            pstmt.setDouble(8, memento.getCurrentTime());  
  
            String eqStr = "";  
            if (memento.getEqBands() != null &&  
!memento.getEqBands().isEmpty()) {  
                eqStr = memento.getEqBands().stream()  
                    .map(String::valueOf)  
                    .collect(Collectors.joining(", "));  
            }  
            pstmt.setString(9, eqStr);  
  
            pstmt.executeUpdate();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public PlayerMemento loadState(int userId) {  
        String sql = "SELECT volume, is_shuffle, is_repeat,  
current_track_id, source_type, source_id, current_time,  
eq_settings FROM playback_state WHERE user_id = ?";  
  
        try (Connection conn = DatabaseConnection.connect();  
             PreparedStatement pstmt =  
conn.prepareStatement(sql)) {  
            pstmt.setInt(1, userId);  
            ResultSet rs = pstmt.executeQuery();  
            if (rs.next()) {  
                PlayerMemento memento = new PlayerMemento();  
                memento.setVolume(rs.getInt("volume"));  
                memento.setShuffle(rs.getBoolean("is_shuffle"));  
                memento.setRepeat(rs.getBoolean("is_repeat"));  
                memento.setCurrentTrackId(rs.getInt("current_track_id"));  
                memento.setType(rs.getString("source_type"));  
                memento.setId(rs.getInt("source_id"));  
                memento.setCurrentTime(rs.getDouble("current_time"));  
                memento.setEqBands(rs.getString("eq_settings").split(", "));  
                return memento;  
            }  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

        PreparedStatement pstmt =
conn.prepareStatement(sql)) {

    pstmt.setInt(1, userId);
    ResultSet rs = pstmt.executeQuery();

    if (rs.next()) {
        List<Double> eqBands = new ArrayList<>();
        String eqStr = rs.getString("eq_settings");

        if (eqStr != null && !eqStr.isEmpty()) {
            for (String val : eqStr.split(",")) {
                try {

eqBands.add(Double.parseDouble(val));
                } catch (NumberFormatException e) {
                    eqBands.add(0.0);
                }
            }
        }
        return new PlayerMemento(
            rs.getInt("volume"),
            rs.getBoolean("is_shuffle"),
            rs.getBoolean("is_repeat"),
            rs.getInt("current_track_id"),
            rs.getInt("source_type"),
            rs.getInt("source_id"),
            rs.getDouble("current_time"),
            eqBands
        );
    }
} catch (SQLException e) {
    e.printStackTrace();
}
return null;
}
}

```

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з принципами побудови архітектури програмного забезпечення на основі патернів проєктування, зокрема "Знімок" (Memento).

Особливу увагу я приділив реалізації шаблону Memento, детально описавши його основну логіку та функціональність у контексті моєго проєкту музичного програвача.

Цей патерн було обрано як архітектурне рішення для забезпечення можливості збереження та відновлення стану MusicPlayer (Originator) без порушення його

інкапсуляції. Я зрозумів, наскільки цей патерн спрощує процес управління станом, дозволяючи винести відповідальність за зберігання історії в окремі класи (Caretaker та StateRepository).

Використання шаблону Memento забезпечує:

- Інкапсуляцію: Внутрішня структура MusicPlayer (гучність, евалайзер, черга) залишається прихованою від зовнішніх об'єктів, що зберігають стан.
- Спрощення Originator-a: Код плеєра не перевантажений логікою управління версіями станів або взаємодією з БД для збереження.
- Відновлення стану: Легке повернення до попередніх налаштувань відтворення або відновлення сесії після перезапуску.

Впровадження патерну Memento дозволило створити надійну та гнучку архітектуру програми, де механізм збереження даних відокремлений від основної бізнес-логіки відтворення, дотримуючись Принципу єдиної відповідальності.

Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» використовується для створення груп взаємопов'язаних об'єктів без необхідності вказувати їх конкретні класи. Він надає інтерфейс для створення об'єктів певної тематики або сімейства, при цьому клієнтський код не знає, які саме класи стоять за цими об'єктами. Завдяки цьому досягається гнучкість у зміні або розширенні системи — наприклад, можна легко підмінити одну реалізацію іншою, не змінюючи основний код програми. Основна перевага цього шаблону полягає в ізоляції процесу створення об'єктів від їх використання. Це дозволяє централізовано керувати створенням різних компонентів, забезпечуючи узгодженість між ними. «Абстрактна фабрика» часто застосовується в системах, де потрібно підтримувати кілька варіантів оформлення інтерфейсу або різні конфігурації об'єктів, зберігаючи при цьому чисту архітектуру та принцип незалежності від конкретних реалізацій.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

До шаблону «Абстрактна фабрика» зазвичай входять кілька основних типів класів, які взаємодіють між собою в узгодженій структурі:

AbstractFactory (Абстрактна фабрика) – це інтерфейс або абстрактний клас, який визначає набір методів для створення різних типів об'єктів (наприклад, `createButton()`, `createWindow()`). Він не створює об'єкти самостійно, а лише задає контракт, як це має робитися.

ConcreteFactory (Конкретна фабрика) – це клас, який реалізує методи абстрактної фабрики. Він відповідає за створення конкретних об'єктів певного сімейства (наприклад, об'єктів у стилі Windows чи MacOS).

AbstractProduct (Абстрактний продукт) – інтерфейс або абстрактний клас, який описує загальні властивості або поведінку продуктів, що створюються фабрикою.

ConcreteProduct (Конкретний продукт) – це конкретна реалізація абстрактного продукту, яка відповідає певному типу фабрики.

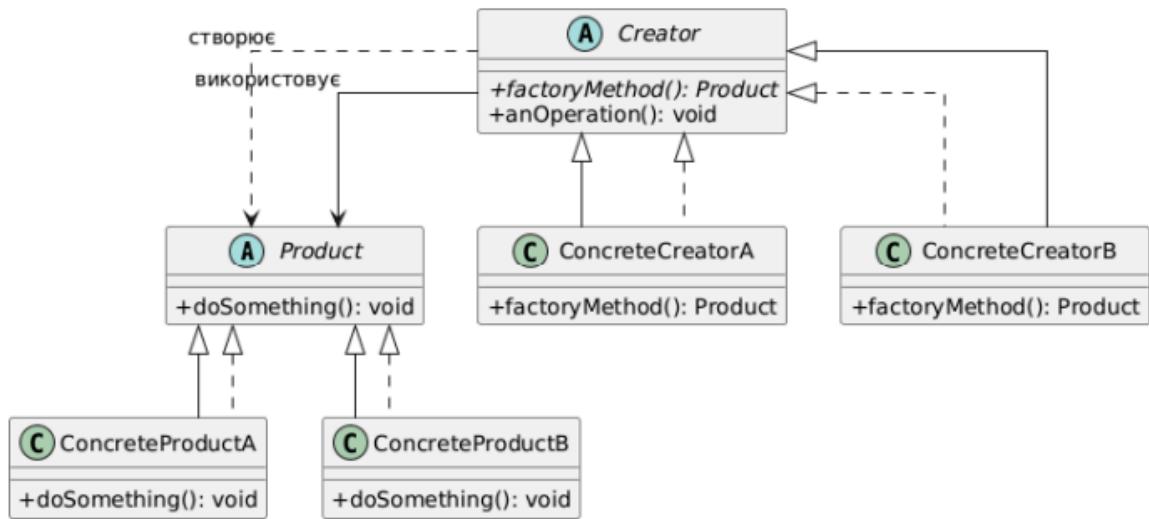
Client (Клієнт) – це клас, який використовує фабрику для створення об'єктів. Клієнт працює тільки з абстрактними інтерфейсами, тому не залежить від конкретних реалізацій об'єктів. Взаємодія між ними відбувається так: клієнт звертається до абстрактної фабрики, щоб створити об'єкти певного типу. Конкретна фабрика реалізує цей запит і повертає відповідні конкретні продукти. У результаті клієнт може використовувати створені об'єкти, не знаючи, які саме класи стоять за ними, що забезпечує гнучкість і зручність розширення системи.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» (Factory Method) призначений для делегування створення об'єктів підкласам, щоб основний клас не залежав від конкретних типів створюваних об'єктів. Тобто замість того, щоб напряму створювати об'єкти через оператор `new`, базовий клас викликає спеціальний метод —

“фабричний”, який визначається або перевизначається у підкласах. Такий підхід дозволяє гнучко змінювати або розширювати систему, додаючи нові типи об'єктів без зміни існуючого коду. Шаблон забезпечує принцип «відкритості/закритості» — код відкритий для розширення, але закритий для модифікації. Його часто використовують, коли клас не може заздалегідь знати, який саме тип об'єкта потрібно створити, або коли створення об'єкта повинно залежати від конкретних умов чи контексту.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

До шаблону «Фабричний метод» входять кілька основних класів, які мають чітко визначені ролі та взаємодію між собою:

Product (Продукт) – це інтерфейс або абстрактний клас, який описує спільні властивості та поведінку об'єктів, що створюються фабрикою.

ConcreteProduct (Конкретний продукт) – це конкретна реалізація інтерфейсу Product. Кожен підтип продукту має свою реалізацію, що відповідає певній потребі або умовам створення.

Creator (Творець або фабрика) – це абстрактний клас або інтерфейс, який містить оголошення фабричного методу (createProduct()), але не визначає його конкретну реалізацію. Також він може мати загальну логіку, що використовує створені об'єкти.

ConcreteCreator (Конкретний творець) – це клас, який перевизначає фабричний метод, створюючи певний тип продукту (ConcreteProduct). Взаємодія між ними відбувається так: клієнт звертається до творця (Creator), щоб отримати продукт. Замість того, щоб створювати об'єкт напряму, він викликає фабричний метод. Конкретний творець (ConcreteCreator) визначає, який саме продукт створити, і повертає відповідний об'єкт. Завдяки цьому клієнт працює лише з абстракціями (Product і Creator), не залежачи від

конкретних реалізацій, що забезпечує гнучкість і легкість розширення програми.

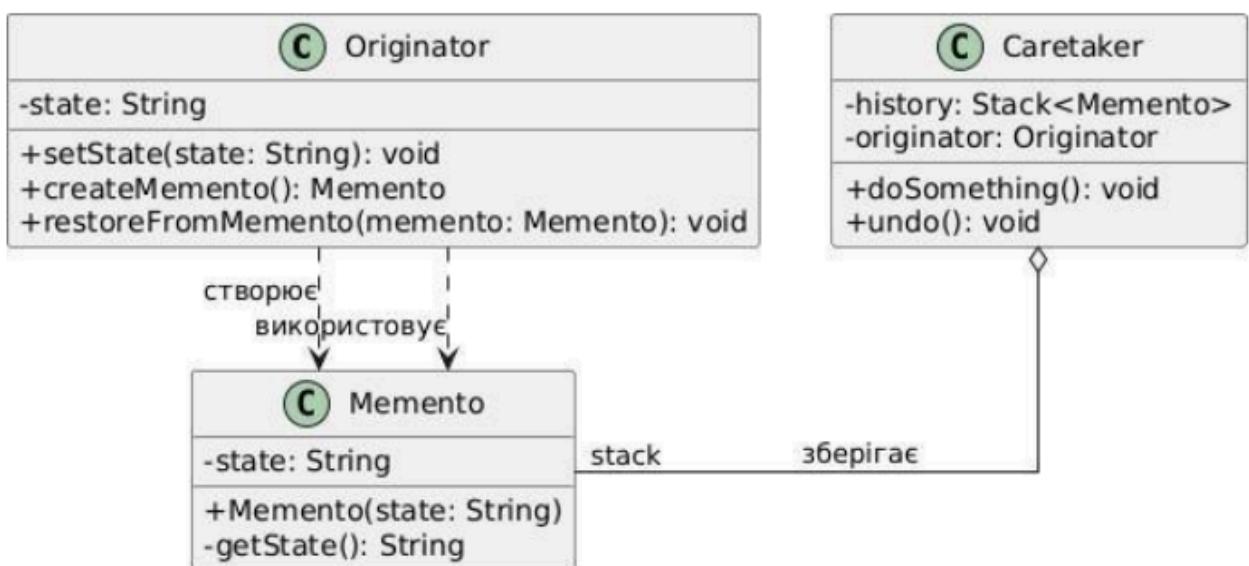
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

Шаблон «Фабричний метод» визначає спосіб створення одного об'єкта через спеціальний метод, який можна перевизначити у підкласах. Його головна мета — дати змогу підкласам вирішувати, який саме об'єкт створювати. Іншими словами, він зосереджений на розширенні одного процесу створення об'єкта. Натомість «Абстрактна фабрика» використовується для створення цілих сімейств взаємопов'язаних об'єктів, які повинні узгоджено працювати між собою. Вона надає інтерфейс для створення кількох видів об'єктів без зазначення їх конкретних класів. Таким чином, «Абстрактна фабрика» часто використовує фабричні методи всередині себе для реалізації створення різних продуктів.

8. Яке призначення шаблону «Знімок»?

Шаблон «Знімок» (Memento) призначений для збереження та відновлення попереднього стану об'єкта без порушення принципу інкапсуляції. Він дозволяє створити “знімок” внутрішнього стану об'єкта, який можна зберегти для подальшого відновлення, не розкриваючи деталей реалізації цього об'єкта іншим класам. Цей шаблон часто використовується у випадках, коли потрібно реалізувати функцію “Скасувати” (Undo) або повернення до попереднього стану системи. «Знімок» дає змогу безпечно відкотити зміни, відновивши об'єкт до збереженого стану, при цьому інші частини програми не мають доступу до його внутрішніх полів чи логіки.

9. Нарисуйте структуру шаблону «Знімок».



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

До шаблону «Знімок» (Memento) входять три основні класи, кожен з яких виконує свою роль у процесі збереження та відновлення стану об'єкта:

Originator (Джерело) – це об'єкт, стан якого потрібно зберігати. Він створює знімок свого поточного стану та може пізніше відновити цей стан із переданого знімка. Originator знає, які саме дані потрібно зберегти, але не надає до них прямого доступу іншим класам.

Memento (Знімок) – це клас, який зберігає внутрішній стан об'єкта. Він зазвичай має два інтерфейси: 1) публічний — доступний лише для Caretaker, щоб передавати або зберігати знімки; 2) приватний — доступний тільки для Originator, який може записувати і читати свій стан.

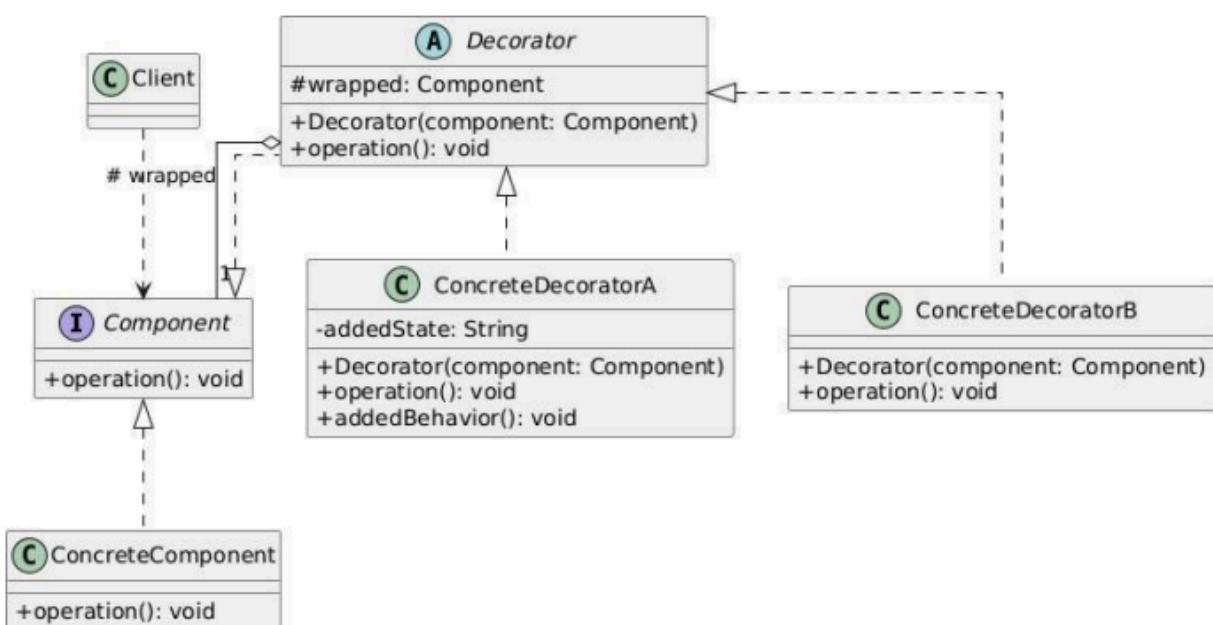
Caretaker (Опікун) – це клас, який керує знімками: зберігає їх, передає назад до Originator, коли потрібно відновити стан, але не має доступу до вмісту самого знімка.

Взаємодія між ними відбувається так: Caretaker просить Originator створити знімок поточного стану й зберігає його. Коли потрібно відновити попередній стан, Caretaker передає збережений знімок назад Originator. Originator використовує дані із знімка, щоб повернутися до попереднього стану.

11. Яке призначення шаблону «Декоратор»?

Шаблон «Декоратор» (Decorator) призначений для динамічного розширення функціональності об'єкта без зміни його коду або створення нових підкласів. Він дозволяє «обгорнути» об'єкт у спеціальний клас-декоратор, який додає нову поведінку або змінює існуючу, при цьому зберігаючи інтерфейс початкового об'єкта. Такий підхід дає змогу гнучко комбінувати різні функції, створюючи різні варіації поведінки без множинного наслідування. Наприклад, замість створення десятків підкласів із різними поєднаннями можливостей, можна просто «декорувати» об'єкт потрібними обгортками. Це спрощує розширення програми й робить код більш гнучким і підтримуваним.

12. Нарисуйте структуру шаблону «Декоратор».



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

До шаблону «Декоратор» (Decorator) входять кілька основних класів, які працюють разом, щоб забезпечити динамічне розширення функціональності об'єкта:

Component (Компонент) – це інтерфейс або абстрактний клас, який визначає спільний інтерфейс для всіх об'єктів, що можуть бути декоровані. Він містить методи, які реалізують як базові класи, так і декоратори.

ConcreteComponent (Конкретний компонент) – це клас, який реалізує інтерфейс Component і містить основну функціональність об'єкта, яку можна розширювати за допомогою декораторів.

Decorator (Декоратор) – це абстрактний клас, який також реалізує інтерфейс Component, але містить посилання на інший об'єкт типу Component. Він делегує виклики методів цьому об'єкту, додаючи додаткову поведінку до або після виклику.

ConcreteDecorator (Конкретний декоратор) – це клас, який наслідує Decorator і додає нову поведінку або змінює існуючу. Таких декораторів може бути кілька, і їх можна комбінувати між собою.

Взаємодія між ними відбувається так: клієнт працює з об'єктами через інтерфейс Component, не знаючи, чи це звичайний компонент, чи обгорнутий декоратор. Коли викликається метод, декоратор може виконати додаткові дії (наприклад, логування, перевірку, візуальне оформлення), а потім передати виклик далі базовому компоненту. Таким чином, кілька декораторів можуть послідовно розширювати поведінку одного й того ж об'єкта, не змінюючи його код.

14. Які є обмеження використання шаблону «декоратор»?

1) Складність структури – при великій кількості декораторів і їх послідовному обгортанні код може стати важким для розуміння і супроводу, оскільки логіка поведінки розподіляється між багатьма класами.

2) Проблеми з ідентичністю об'єкта – декоратори обгортают об'єкт, і клієнт працює з посиланням на обгортку, а не на початковий об'єкт. Це може ускладнювати перевірку типу або порівняння об'єктів.

3) Ускладнене налагодження – через ланцюжок обгорток складніше відстежувати послідовність викликів методів і визначати, де саме відбувається певна зміна поведінки.

4) Не підходить для статичного розширення – якщо поведінка об'єкта має бути визначена на етапі компіляції, використання декораторів не дає переваги, оскільки вони працюють динамічно під час виконання.

5) Небажане надмірне використання – надто часте застосування декораторів може призвести до великої кількості класів і підвищити складність системи без реальної користі.