Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота №6
Технологія розробки програмного забезпечення
«Патерни проектування»

Виконала
студентка групи ІА–32:
Ткачук М. С.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

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.

Аудіо редактор (singleton, adapter, observer, mediator, composite, client-server)

Аудіо редактор повинен володіти наступним функціоналом: представлення аудіо даних будь-якого формату в WAVE-формі, вибір і подальші операції копіювання / вставки / вирізання / деформації по сегменту аудіозапису, можливість роботи з декількома звуковими доріжками, кодування в найбільш поширених форматах (ogg, flac, mp3).

Теоретичні відомості

Шаблон «Abstract Factory»

Шаблон Abstract Factory використовується для створення цілих сімейств пов’язаних об’єктів без прив’язки до їх конкретних класів. Він визначає спільний інтерфейс фабрики, а конкретні фабрики реалізують створення продуктів певного типу. Патерн зручний, коли потрібно забезпечити узгодженість між об’єктами одного стилю або конфігурації. Основний недолік це складність розширення фабрик при додаванні нових типів продуктів

Шаблон «Factory Method»

Factory Method визначає інтерфейс для створення об’єктів, але дозволяє підкласам вирішувати, який саме об’єкт створити. Завдяки цьому можна підміняти базові типи їхніми підтипами, не змінюючи клієнтський код. Патерн спрощує додавання нових продуктів, але часто призводить до появи великих паралельних ієрархій класів.

Шаблон «Memento»

Патерн Memento зберігає і відновлює внутрішній стан об’єкта без порушення інкапсуляції. Об’єкт-Originator створює знімок свого стану, а Caretaker зберігає ці знімки. Це дозволяє реалізувати скасування дій та історію станів. Основний недолік це можливе значне споживання пам’яті при частому створенні знімків.

Шаблон «Observer»

Observer створює залежність «один до багатьох», коли зміна стану одного об’єкта автоматично повідомляє всі пов’язані об’єкти. Суб’єкт веде список підписників і надсилає їм сповіщення. Патерн забезпечує слабке зв’язування та зручну систему оновлень, але не гарантує контрольованої черговості отримання повідомлень.

Шаблон «Decorator»

Decorator дозволяє динамічно розширювати функціональність об’єкта за допомогою обгорток. Декоратор реалізує той самий інтерфейс, що й базовий об’єкт, і додає нові можливості, не змінюючи вихідний клас. Це гнучкіша альтернатива наслідуванню, хоча велика кількість дрібних декораторів може ускладнювати структуру програми.

Хід Роботи

Діаграма класів

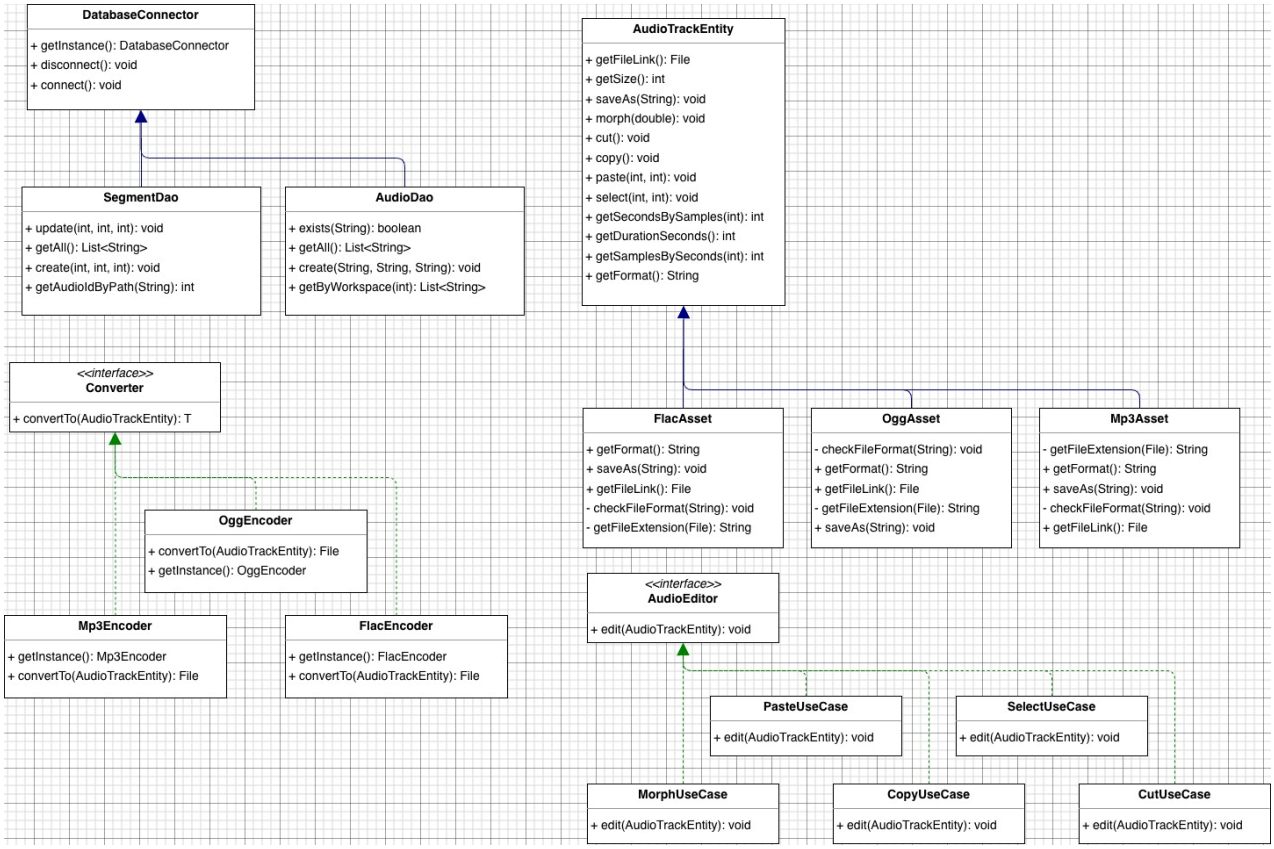


Рисунок 1 – Діаграма класів

На рисунку 1 накведено діаграму класів аудіоредактора, що відображає логічну структуру програмного забезпечення, основні класи, їхні атрибути, методи та взаємозв’язки між ними.

Центральним елементом системи є клас AudioTrackEntity, який описує сутність аудіотреку та містить методи для виконання основних операцій редагування cut(), copy(), paste(), morph(), а також методи для роботи з форматом файлу, отримання семплів і тривалості (getDurationSeconds(), getFormat(), getSampleBySeconds()).

Цей клас є базовим для роботи з різними типами звукових файлів і виступає ядром усієї системи.

Від AudioTrackEntity наслідуються класи Mp3Asset, OggAsset та FlacAsset, що реалізують підтримку різних аудіоформатів. Кожен із цих класів має власні методи для збереження (saveAs()), перевірки формату (checkFileFormat()), а також отримання розширення файлу (getFileExtension()), що дозволяє системі працювати з популярними форматами MP3, OGG та FLAC.

Окремий клас AudioEditor реалізує високорівневу логіку редагування файлів. Він містить метод edit(), який викликає відповідні операції залежно від сценарію редагування: копіювання, вставлення, вирізання або деформації звуку. Для реалізації цих сценаріїв використовуються окремі класи CutUseCase, CopyUseCase, SelectUseCase і MorphUseCase, кожен із яких виконує власну операцію редагування через метод edit(AudioTrackEntity).

Для кодування звукових доріжок у різні формати використовується абстрактний клас Converter<T>, який містить універсальний метод convertTo(AudioTrackEntity). Від нього наслідуються конкретні реалізації Mp3Encoder, OggEncoder та FlacEncoder, які забезпечують перетворення аудіо у відповідний формат.

Компонент DatabaseConnector відповідає за взаємодію з локальною базою даних SQLite. Він реалізує підключення, відключення та отримання екземпляра підключення (getInstance(), connect(), disconnect()). З ним працюють два DAO-класи AudioDao і SegmentDao, які реалізують доступ до даних аудіофайлів і сегментів.
AudioDao забезпечує методи create(), getAll(), exists() і getByWorkspace(), а SegmentDao: операції create(), update() та getAudioIdByPath(). Це дає змогу системі зберігати інформацію про проєкти, сегменти та аудіофайли у структурованому вигляді.

Діаграма класів демонструє, як система аудіоредактора розділена на логічні шари:

- рівень даних (DAO-класи, DatabaseConnector),
- рівень бізнес-логіки (AudioEditor, UseCase-класи),
- рівень моделей (AudioTrackEntity і похідні класи для форматів),
- рівень перетворень (Converter і Encoder-класи).

Патерн observer

Патерн Observer це підхід, який дозволяє організувати механізм підписки між об’єктами системи. Один об’єкт виступає у ролі джерела подій, а інші отримують оновлення щоразу, коли в ньому відбуваються зміни. Спостерігачі і джерело подій слабо пов’язані між собою, тому їх можна додавати або вилучати без впливу на іншу логіку. Цей патерн часто використовується в подієвих системах, інтерфейсах користувача, сервісах логування та системах реального часу

У контексті аудіоредактора Observer застосовано для реалізації механізму відстеження подій та ведення журналу дій користувача. До таких подій належить відкриття аудіофайлу, редагування сегментів, збереження проєкту та інші операції, які повинні бути зафіксовані для подальшого аналізу або технічної підтримки.

Центральним елементом є клас EventsHub, який виступає джерелом подій. Він дозволяє підписувати та відписувати об’єкти, що реалізують інтерфейс EventSink, а також сповіщати їх у момент виникнення події. Коли відбувається дія, EventsHub викликає метод notifySubscribers, передаючи інформацію про конкретну подію.

Класи AuditLog і OpenOperationLogger виконують роль спостерігачів. Вони підписуються на EventsHub і отримують відповідні повідомлення через метод

handleEvent. Так AuditLog відповідає за запис інформації про відкриття файлів у журнал, а OpenOperationLogger може логувати додаткові операції, наприклад перетворення форматів, завантаження або структурні зміни в аудіодоріжці.

Використання Observer забезпечує такі переваги.
По-перше, модуль логування відокремлений від основної логіки аудіоредактора, тому його можна легко розширити, додати нові типи спостерігачів або змінити формат ведення журналу.
По-друге, EventsHub дозволяє підключати різні підсистеми без зміни коду ядра програми, що підвищує гнучкість архітектури.
По-третє, така структура дає змогу легко відстежувати і аналізувати послідовність дій користувача, що важливо для відлагодження та покращення UX.

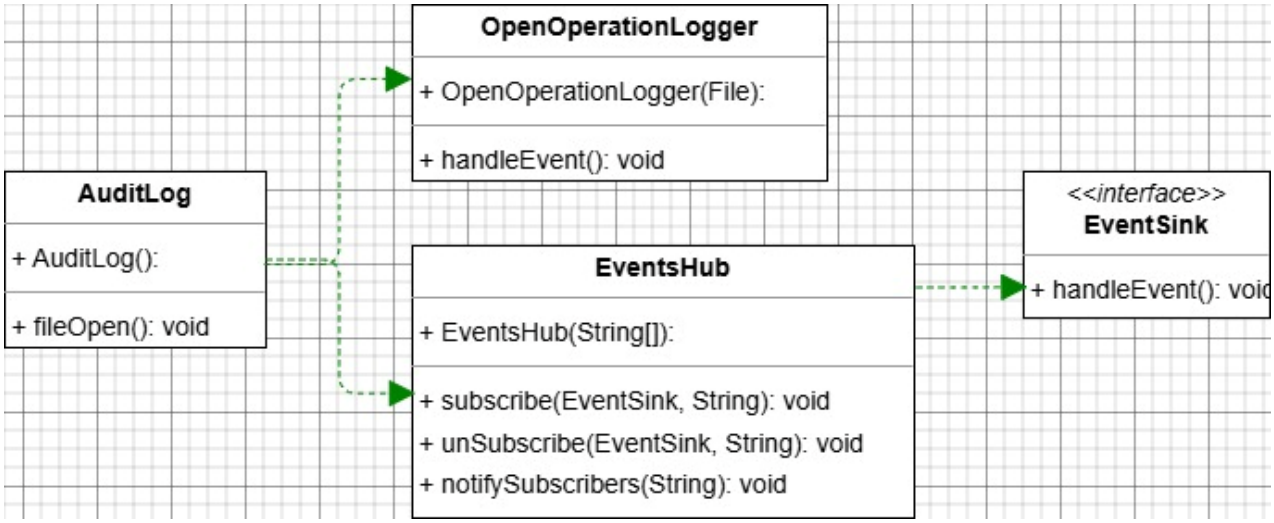


Рисунок 2 – Схема патерну observer

Фрагменти коду:

AuditLog.java

```
package app.soundlab.audit;


public class AuditLog {
    private final EventsHub eventsHub;


    public AuditLog() {
        this.eventsHub = new EventsHub("openFile");
    }


    public void subscribeToOpen(EventSink sink) {
        eventsHub.subscribe(sink, "openFile");
    }


    public void unsubscribeFromOpen(EventSink sink) {
        eventsHub.unSubscribe(sink, "openFile");
    }


    public void fileOpen() {
        eventsHub.notifySubscribers("openFile");
    }
```

```
}
```

ConsoleOperationLogger.java

```
package app.soundlab.audit;

import java.time.LocalDateTime;

public class ConsoleOperationLogger implements EventSink {
    private final String observerName;

    public ConsoleOperationLogger(String observerName) {
        this.observerName = observerName;
    }

    @Override
    public void handleEvent() {
        System.out.println(observerName + " recorded file opening at " +
LocalDateTime.now());
    }
}
```

EventsHub.java

```
package app.soundlab.audit;
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class EventsHub {
    private final Map<String, List<EventSink>> listeners = new HashMap<>();

    public EventsHub(String... operations) {
        for (String operation : operations) {
            listeners.put(operation, new ArrayList<>());
        }
    }

    public void subscribe(EventSink eventSink, String event) {
        listeners.computeIfAbsent(event, key -> new ArrayList<>()).add(eventSink);
    }

    public void unSubscribe(EventSink subscriber, String event) {
        List<EventSink> users = listeners.get(event);
        if (users != null) {
            users.remove(subscriber);
```

```
        }
    }

    public void notifySubscribers(String event) {
        List<EventSink> users = listeners.get(event);
        if (users == null) {
            return;
        }
        for (EventSink listener : users) {
            listener.handleEvent();
        }
    }
}
```

EventSink.java

```
package app.soundlab.audit;

public interface EventSink {
    void handleEvent();
}
```

OpenOperationLogger.java

```
package app.soundlab.audit;
```

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.time.LocalDate;
import java.time.LocalTime;

public class OpenOperationLogger implements EventSink {
    private final File logFile;

    public OpenOperationLogger(File logFile) {
        this.logFile = logFile;
    }

    @Override
    public void handleEvent() {
        File parent = logFile.getParentFile();
        if (parent != null && !parent.exists()) {
            parent.mkdirs();
        }
        try (FileWriter writer = new FileWriter(logFile, true)) {
            writer.write("File was opened at " + LocalTime.now() + " " +
LocalDate.now() + "\n");
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

ObserverDemo.java

```
package app.soundlab;

import app.soundlab.audit.AuditLog;
import app.soundlab.audit.ConsoleOperationLogger;
import app.soundlab.audit.EventSink;
import app.soundlab.audit.OpenOperationLogger;

import java.io.File;

public class ObserverDemo {
    public static void main(String[] args) {
        AuditLog auditLog = new AuditLog();

        EventSink persistentObserver = new OpenOperationLogger(new
File("audit/logs.txt"));
```

```
        EventSink consoleObserver = new ConsoleOperationLogger("Console
observer");


        auditLog.subscribeToOpen(persistentObserver);
        auditLog.subscribeToOpen(consoleObserver);


        System.out.println("First open notifies both observers.");
        auditLog.fileOpen();


        auditLog.unsubscribeFromOpen(consoleObserver);
        System.out.println("Second open notifies only the file logger.");
        auditLog.fileOpen();
    }
}
```

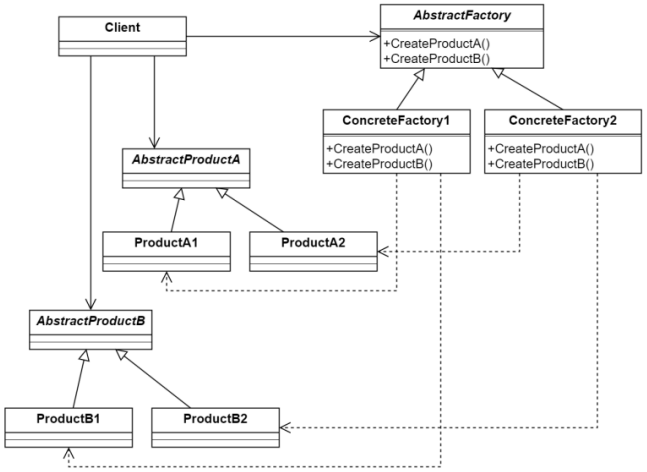
Вихідний код:
https://github.com/mandarinchik21/AudioEditor/tree/main/lab6

Висновок: У ході виконання лабораторної роботи було розглянуто та проаналізовано основні шаблони проектування «Abstract Factory», «Factory Method», «Memento», «Observer» та «Decorator», а також досліджено їх застосування у складних програмних системах. Реалізація частини функціоналу аудіоредактора показала, що використання цих патернів дозволяє суттєво спростити архітектуру, розділити відповідальності між компонентами та підвищити гнучкість системи. Завдяки введенню чітких інтерфейсів, можливості створення узгоджених об’єктів, зберіганню стану, динамічному розширенню поведінки та організації механізму підписки аудіоредактор став більш модульним, масштабованим і зручним для подальшого розширення. Виконана робота підтвердила, що комплексне застосування шаблонів проектування є ефективним підходом для побудови сучасного програмного забезпечення із високою підтримуваністю та зрозумілою структурою.

Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?
Абстрактна фабрика призначена для створення сімейств пов’язаних об’єктів (продуктів) без прив’язки до їх конкретних класів, забезпечуючи узгодженість між цими об’єктами (один стиль / конфігурація).

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

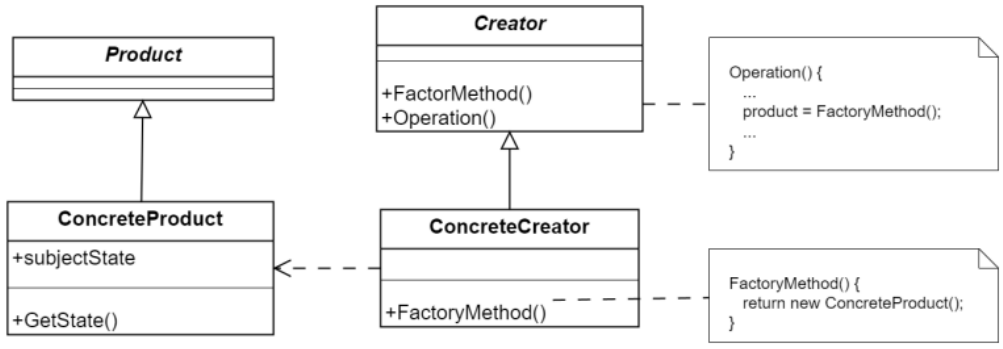
- AbstractFactory – оголошує інтерфейс для створення продуктів (CreateProductA, CreateProductB тощо).
- ConcreteFactory – конкретні фабрики, які створюють продукти певного сімейства (HighTechFactory, ModernFactory і т.п.).
- AbstractProductA, AbstractProductB – інтерфейси (або базові класи) для різних типів продуктів.
- ConcreteProductA1, ConcreteProductB1, ... – конкретні реалізації продуктів одного стилю / сімейства.
- Client – працює тільки з абстракціями фабрики та продуктів.

Взаємодія: Client отримує посилання на AbstractFactory (конкретна фабрика підставляється під час ініціалізації) і через її методи створює об’єкти продуктів. Конкретна фабрика гарантує, що всі створені продукти належать до одного сімейства (одного стилю).

4. Яке призначення шаблону «Фабричний метод»?

Фабричний метод визначає інтерфейс для створення об’єктів певного базового типу, але дозволяє підкласам вирішувати, який саме конкретний клас створювати. Він усуває жорстку прив’язку коду до конкретних типів продуктів.

5. Нарисуйте структуру шаблону «Фабричний метод»



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

- Product – абстрактний продукт (інтерфейс або базовий клас).
- ConcreteProduct – конкретні реалізації продуктів.
- Creator – базовий клас, що містить фабричний метод factoryMethod() і загальну логіку роботи з продуктами.
- ConcreteCreator – перевизначає factoryMethod() і повертає конкретні продукти.

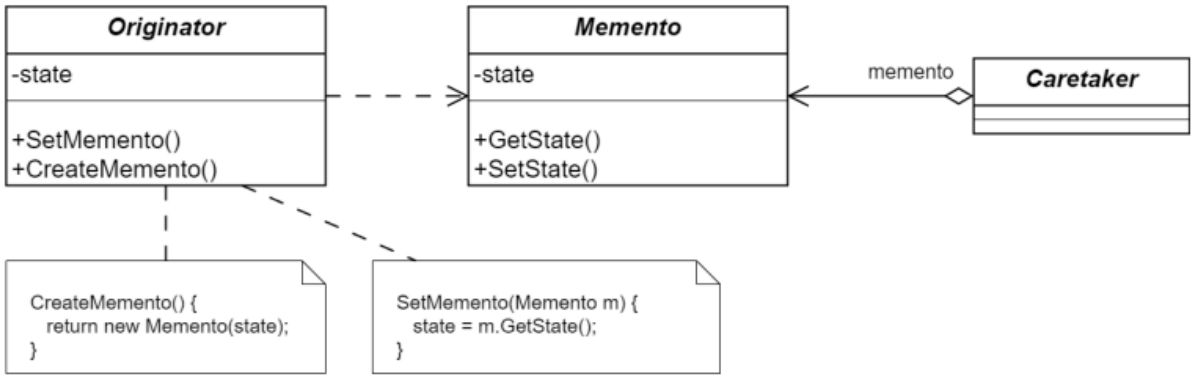
Взаємодія: клієнт працює з Creator або ConcreteCreator, викликає методи, які всередині звертаються до factoryMethod(). Конкретний кріейтор вирішує, який саме ConcreteProduct буде створено.

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

- Масштаб:
    - Абстрактна фабрика створює цілі сімейства продуктів (кілька типів об’єктів, узгоджених між собою).
    - Фабричний метод зосереджується на створенні одного типу продукту (одного ієрархічного сімейства).
- Структура:
    - Абстрактна фабрика зазвичай реалізується як інтерфейс з кількома фабричними методами (по одному на кожен вид продукту).
    - Фабричний метод – це один віртуальний метод у базовому класі, який перевизначається в підкласах.
- Призначення:
    - Абстрактна фабрика вирішує задачу узгодженості стилю/сімейства об’єктів.
    - Фабричний метод вирішує задачу гнучкої підміни конкретного класу продукту.

8. Яке призначення шаблону «Знімок» (Memento)?
Шаблон «Знімок» призначений для збереження та подальшого відновлення внутрішнього стану об’єкта без порушення інкапсуляції. Він дозволяє реалізувати історію змін, скасування дій (undo) тощо.

9. Нарисуйте структуру шаблону «Знімок».



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

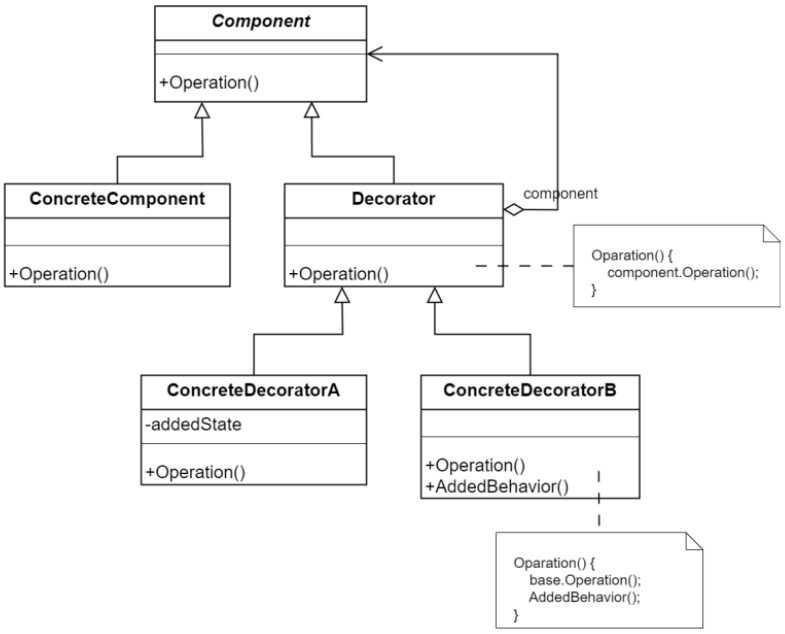
- Originator – об’єкт, стан якого потрібно зберігати та відновлювати; створює Memento і відновлюється з нього.
- Memento – знімок стану Originator; інкапсулює дані стану та не розкриває їх іншим класам.
- Caretaker – зберігає один або кілька об’єктів Memento (історія), але не читає внутрішній вміст.

Взаємодія: Originator створює Memento і передає його Caretaker на зберігання. Коли потрібно відкотитися назад, Caretaker повертає Memento назад до Originator, який відновлює свій стан.

11. Яке призначення шаблону «Декоратор»?

Декоратор призначений для динамічного додавання нової функціональності об’єкту без зміни його класу. Він «обгортає» базовий об’єкт і додає поведінку до або після викликів його методів.

12. Нарисуйте структуру шаблону «Декоратор».



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

- Component – загальний інтерфейс або базовий клас для всіх об’єктів, які можуть бути «декоровані».
- ConcreteComponent – базовий об’єкт, до якого додається додаткова функціональність.
- Decorator – базовий клас декоратора, реалізує Component і зберігає посилання на Component всередині себе.
- ConcreteDecorator – конкретні декоратори, які додають додаткову поведінку до викликів методів компонента.

Взаємодія: Client працює через інтерфейс Component, але фактично може отримати ланцюжок декораторів, де кожен декоратор викликає внутрішній Component і доповнює його поведінку.

14. Які є обмеження використання шаблону «Декоратор»?

Основні обмеження та недоліки:

- З’являється велика кількість дрібних класів (множина декораторів та їхніх комбінацій), що ускладнює розуміння структури.
- Важко відстежувати порядок обгортання, особливо коли над одним об’єктом застосовано кілька декораторів одночасно.
- Неправильна комбінація декораторів може призвести до неочікуваної поведінки.
- Не завжди зручно, якщо інтерфейс компонента дуже широкий — доводиться прокидувати багато методів через усі декоратори.