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

із дисципліни «*Технології розроблення програмного забезпечення*»

Патерни проектування

Аудіоредактор

Виконала
студентка групи ІА–34
Кузьменко В. С.

Перевірив
викладач
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.

Тема роботи:

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

Принципи SOLID – це набір рекомендацій для проєктування програмного забезпечення, які допомагають створювати код, що легко підтримується, розширюється та тестується. Вони спрямовані на те, щоб зробити систему більш модульною і стійкою до змін. SOLID – це абревіатура, яка об'єднує п'ять основних принципів.

1. Принцип єдиного обов'язку (Single Responsibility Principle, SRP)

Цей принцип стверджує, що кожен клас має виконувати лише одну логічну задачу або відповідати за одну частину функціональності. У класу має бути лише одна причина для змін. Наприклад, якщо у вас є клас, що обробляє дані і одночасно відповідає за їх відображення, це порушує SRP. Краще розділити його на два класи: один для обробки даних, а інший – для їх відображення.

2. Принцип відкритості/закритості (Open/Closed Principle, OCP)

Програма має бути відкритою для розширення, але закритою для змін. Це означає, що ви повинні мати змогу додавати нову функціональність, не змінюючи вже існуючий код. Цього можна досягти через використання абстракцій, інтерфейсів або базових класів. Наприклад, замість того щоб модифікувати метод сортування в існуючому класі, можна створити новий клас, який реалізує цей метод, використовуючи спільний інтерфейс.

3. Принцип підстановки Барбари Лісков (Liskov Substitution Principle, LSP)

Кожен підклас або похідний клас має мати можливість замінити базовий клас без порушення функціональності програми. Іншими словами, класи-нащадки повинні підтримувати поведінку, визначену базовим класом. Наприклад, якщо у вас є клас "Фігура" з методом площа(), підкласи, такі як "Круг" або "Прямокутник", повинні коректно реалізовувати цей метод і поводитися передбачувано. Якщо підклас змінює поведінку базового класу так, що програма починає працювати некоректно, LSP порушується.

4. Принцип розділення інтерфейсу (Interface Segregation Principle, ISP)

Інтерфейси мають бути невеликими і спеціалізованими. Клієнти не повинні залежати від методів, які вони не використовують. Наприклад, якщо у вас є великий інтерфейс, який включає методи для малювання, збереження та друку, різні класи, що реалізують цей інтерфейс, можуть бути змушені реалізовувати методи, які їм не потрібні. Замість цього краще розбити інтерфейс на декілька менших, кожен з яких відповідає окремій задачі.

5. Принцип інверсії залежностей (Dependency Inversion Principle, DIP)

Високоуровневі модулі не повинні залежати від низькорівневих модулів. Обидва повинні залежати від абстракцій. Крім того, абстракції не повинні залежати від деталей, а деталі повинні залежати від абстракцій. Це означає, що залежності між класами слід організовувати через інтерфейси або абстрактні класи, а не через конкретні реалізації. Наприклад, замість того щоб клас "Робот" напряму використовував клас "Лазер", він повинен працювати через інтерфейс "Зброя". Це дозволяє легко замінити лазер на будь-яку іншу зброю, не змінюючи код класу "Робот".

Ці принципи тісно пов’язані між собою і часто використовуються разом. Їх застосування дозволяє уникнути проблем із залежностями, дублюванням коду та ускладненнями при внесенні змін до програмного забезпечення.

Abstract Factory (Абстрактна фабрика)

Абстрактна фабрика допомагає створювати сімейства взаємопов'язаних об'єктів без прив'язки до їхніх конкретних класів. Наприклад, якщо ваша програма підтримує кілька операційних систем, цей шаблон дозволить створювати специфічні елементи інтерфейсу, такі як кнопки та вікна, для кожної ОС, залишаючи клієнтський код незмінним. Завдяки цьому можна легко адаптувати програму до нових платформ.

Factory Method (Фабричний метод)

Фабричний метод визначає інтерфейс для створення об'єктів, але дозволяє підкласам самостійно вирішувати, який саме об'єкт створити. Наприклад, у грі

можна мати клас зброї, де метод createProjectile() створює відповідні снаряди. Клас «Лук» може створювати стріли, а клас «Гармата» – ядра. Це дозволяє змінювати типи об’єктів, не змінюючи загальної логіки програми.

Memento (Збереження)

Цей шаблон дозволяє зберігати та відновлювати стан об'єкта без порушення його інкапсуляції. Наприклад, у текстовому редакторі можна реалізувати функцію «Скасувати» за допомогою збереження стану документа в об'єкти-збереження. Коли користувач хоче повернутися до попередньої версії, програма просто відновлює потрібний стан.

Observer (Спостерігач)

Шаблон «Спостерігач» використовується для автоматичного сповіщення об'єктів про зміну стану іншого об'єкта. Наприклад, у мобільному додатку для новин, коли сервер оновлює список новин, усі підписані компоненти (зокрема, інтерфейс користувача) автоматично отримують оновлену інформацію. Це дозволяє підтримувати синхронізацію без прямого зв’язку між об'єктами.

Decorator (Декоратор)

Декоратор дозволяє динамічно додавати нову функціональність об'єкту, не змінюючи його структуру. Наприклад, у програмі для оформлення замовлення кави базовий клас напою можна доповнювати такими декораторами, як додавання молока, вершків або сиропу. Таким чином, можна створювати різні комбінації функцій без роздування кількості класів.

Ці шаблони проєктування допомагають створювати код, який легко адаптується до змін і забезпечує гнучкість у розробці. Вони знижують зв’язність між компонентами системи та спрощують розширення її функціональності.

Хід роботи

1. Ознайомитися з теоретичними відомостями.
2. Реалізувати частину функціоналу програми через класи та їх взаємодію.
3. Використати один з шаблонів проектування для реалізації обраної теми.
4. Створити не менше 3 класів відповідно до теми.
5. Скласти звіт про виконану роботу.

Шаблон Observer

Після ознайомлення з теоретичними матеріалами було вирішено реалізувати шаблон проєктування *Observer (Спостерігач)*. Цей поведінковий шаблон належить до категорії шаблонів, що описують спосіб взаємодії між об’єктами, і призначений для організації механізму «один до багатьох». Його суть полягає в тому, що при зміні стану одного об’єкта - суб’єкта (Subject) - усі залежні від нього об’єкти - спостерігачі (Observers) - автоматично отримують сповіщення та оновлюють свій стан відповідно.

У контексті розробки мого застосунку даний шаблон є особливо корисним, адже дозволяє ефективно організувати взаємодію між компонентами інтерфейсу та логікою програми. Наприклад, коли користувач натискає кнопку або виконує певну дію мишкою, система має миттєво реагувати - оновлювати візуальні елементи, змінювати стан об’єкта в пам’яті. Завдяки шаблону *Observer* ці реакції можуть бути реалізовані без необхідності створювати складні прямі залежності між об’єктами.

Такий підхід робить код більш гнучким, масштабованим і зрозумілим, оскільки дозволяє легко додавати або видаляти спостерігачів без втручання у внутрішню логіку суб’єкта. Крім того, він сприяє розділенню відповідальностей - кожен компонент системи виконує власну роль, реагуючи лише на ті події, які його безпосередньо стосуються.

Реалізація шаблону *Observer* в аудіоредакторі забезпечує зручне та структуроване керування подіями, що виникають під час взаємодії користувача із застосунком.

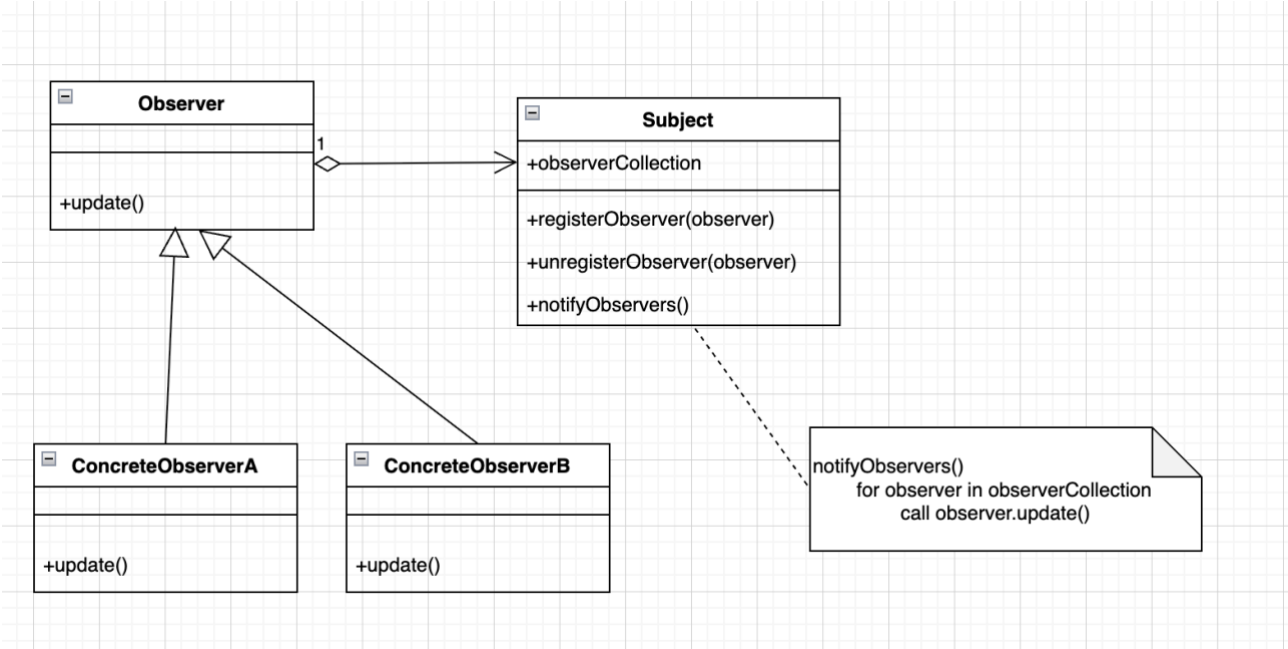


Рис. 1 Загальна структура шаблону Observer

Оскільки розробка застосунку ведеться мовою програмування *Java*, для створення графічного інтерфейсу використовується бібліотека *Swing*, яка вже містить у своїй архітектурі реалізацію шаблону *Observer (Спостерігач)*. Цей механізм лежить в основі роботи компонентів інтерфейсу - наприклад, коли користувач натискає кнопку, відповідні події автоматично надсилаються зареєстрованим «слухачам» (listeners), які реагують на них без необхідності прямого зв’язку між елементами.

Враховуючи це, для частини функціональності, пов’язаної саме з інтерактивними компонентами графічного інтерфейсу, доцільно використовувати вбудовану реалізацію шаблону *Observer*, що вже оптимізована під *Swing*. Це дозволяє уникнути дублювання коду та забезпечує стабільну взаємодію між елементами GUI. Наприклад, натискання кнопки «Відкрити

файл» може викликати подію, яку обробляє спеціальний об’єкт, що реагує на неї, - без потреби напряму викликати методи інших частин програми.

Однак бібліотека Swing обмежується роботою лише з подіями графічного інтерфейсу, тому для решти частин застосунку, зокрема тих, що відповідають за логіку обробки аудіо, збереження файлів, конвертацію, була розроблена власна реалізація шаблону *Observer*. У цій реалізації кожна подія в системі (наприклад збереження файлу, помилка конвертації) може бути зафіксована спеціальним об’єктом-спостерігачем, який реагує відповідно до заданої логіки.

Таким чином, поєднання вбудованого механізму спостерігачів у *Swing* із власною реалізацією шаблону *Observer* у внутрішній логіці програми дозволяє створити добре структуровану, подієво орієнтовану архітектуру, де всі частини системи взаємодіють узгоджено та незалежно одна від одної.

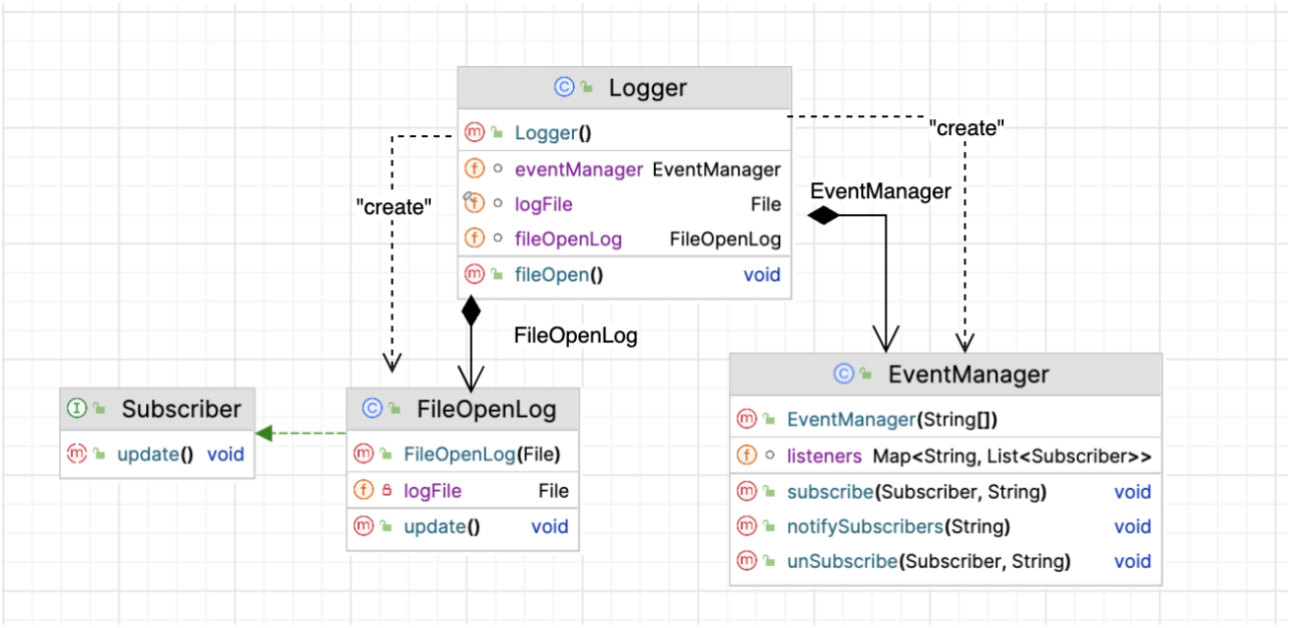


Рис. 2 Структура класів реалізації шаблону Observer

Клас **EventManager** виконує роль центрального елемента системи сповіщень і реалізує функціональність шаблону *«Спостерігач» (Observer)*. Його основне завдання - керувати підписниками та забезпечувати обмін повідомленнями між об’єктами, не створюючи між ними прямої залежності. У класі визначено три основні методи:

- **subscribe()** - додає певного підписника (об’єкт, який реалізує інтерфейс Subscriber) до списку слухачів конкретної події. Це дозволяє об’єкту «підписатися» на певну дію, наприклад, відкриття файлу.
- **unSubscribe()** - виконує протилежну операцію, тобто видаляє підписника зі списку слухачів певної події. Завдяки цьому можна динамічно керувати тим, хто отримує сповіщення про події.
- **notifySubscribers()** - відповідає за повідомлення всіх об’єктів, які підписані на конкретну подію. Коли подія відбувається, метод викликає функцію update() для кожного підписника, передаючи інформацію про цю подію.

Таким чином, **EventManager** виконує роль «посередника» між ініціатором події та її отримувачами, забезпечуючи слабке зв’язування компонентів та можливість легкого розширення системи.

Клас **FileOpenLog** реалізує інтерфейс **Subscriber** і виступає конкретним підписником, який реагує на сповіщення від **EventManager**. Він перевизначає метод **update()**, у якому описана логіка дій, що виконуються при настанні певної події. У цьому випадку метод update() реалізує запис у журнал подій - фіксує дію користувача (відкриття файлу) та час її виконання. Це дозволяє створити історію дій користувача для подальшого аналізу або відстеження. Отже, кожного разу, коли користувач відкриває файл, **FileOpenLog** отримує сповіщення від **EventManager** і додає відповідний запис до журналу дій.

Клас **Logger** об’єднує всю логіку запису подій у єдину систему. Він має агрегацію з класом **EventManager**, а також містить об’єкти класів, що відповідають за реєстрацію окремих типів подій (наприклад, FileOpenLog). У конструкторі класу **Logger** створюється об’єкт **EventManager**, який ініціалізується множиною можливих подій (наприклад, «openFile», «saveFile», «deleteFile» тощо). Далі відбувається підписка відповідних класів логування на потрібні події: зокрема, об’єкт fileOpenLog підписується на подію **"openFile"**.

Коли користувач викликає метод **fileOpen()** у класі **Logger**, це породжує подію «openFile». У цей момент **EventManager** активується та викликає метод **notifySubscribers()**, який надсилає повідомлення всім підписникам цієї події. Серед них - **FileOpenLog**, який отримує сповіщення й виконує власний метод update(), фіксуючи у журналі інформацію про дію користувача.

Загалом, така архітектура дозволяє розділити відповідальність між компонентами:

- **Logger** - ініціює події;
- **EventManager** - розповсюджує повідомлення про події;
- **FileOpenLog** - реагує на них відповідним чином.

Цей підхід забезпечує гнучкість, масштабованість і розширюваність системи, оскільки нові типи подій або нові підписники можна легко додати без зміни існуючої логіки.

Код системи

Лістинг 1- EventManager

```
package org.example.logs;

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

public class EventManager {
    Map<String, List<Subscriber>> listeners = new HashMap<>();

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

    public void subscribe(Subscriber subscriber, String event) {
        List<Subscriber> users = listeners.get(event);
        users.add(subscriber);
    }

    public void unSubscribe(Subscriber subscriber, String event) {
        List<Subscriber> users = listeners.get(event);
        users.remove(subscriber);
    }

    public void notifySubscribers(String event) {
```

```
        List<Subscriber> users = listeners.get(event);
        for (Subscriber listener : users) {
            listener.update();
        }
    }
}
```

Лістинг 2- FileOpenLog

```
package org.example.logs;

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

public class FileOpenLog implements Subscriber {
    private File logFile;

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

    @Override
    public void update() {
        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);
        }
    }
}
```

Лістинг 3- class Logger

```
package org.example.logs;

import java.io.File;

public class Logger {
    EventManager eventManager;
    final File logFile = new File("logs/log.txt");
    FileOpenLog fileOpenLog;

    // ...
    public Logger() {
        eventManager = new EventManager("openFile");

        fileOpenLog = new FileOpenLog(logFile);
        eventManager.subscribe(fileOpenLog, "openFile");

        //...
    }

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

Лістинг 4- interface Subscriber

```
package org.example.logs;

public interface Subscriber {
    public void update();
}
```

Висновок

У ході виконання даної лабораторної роботи було детально розглянуто та проаналізовано п’ять ключових шаблонів проєктування: Abstract Factory**,** Factory Method**,** Memento**,** Observer **та** Decorator. У процесі дослідження було не лише вивчено їхню теоретичну основу, але й практично реалізовано приклад використання, що дозволило краще зрозуміти роль у побудові гнучких і масштабованих програмних систем.

Отримані результати показали, що загалом застосування шаблонів проєктування значно покращує архітектуру програмного забезпечення, робить код більш структурованим, легким для підтримки та розширення. Зокрема:

- **Abstract Factory** і **Factory Method** забезпечують незалежність коду від конкретних реалізацій і сприяють розширюваності;
- **Memento** дозволяє ефективно реалізовувати механізми збереження станів і відкату;
- **Observer** спрощує організацію взаємодії між об’єктами, зменшуючи зв’язність системи;
- **Decorator** забезпечує можливість динамічного розширення функціональності без зміни структури класів.

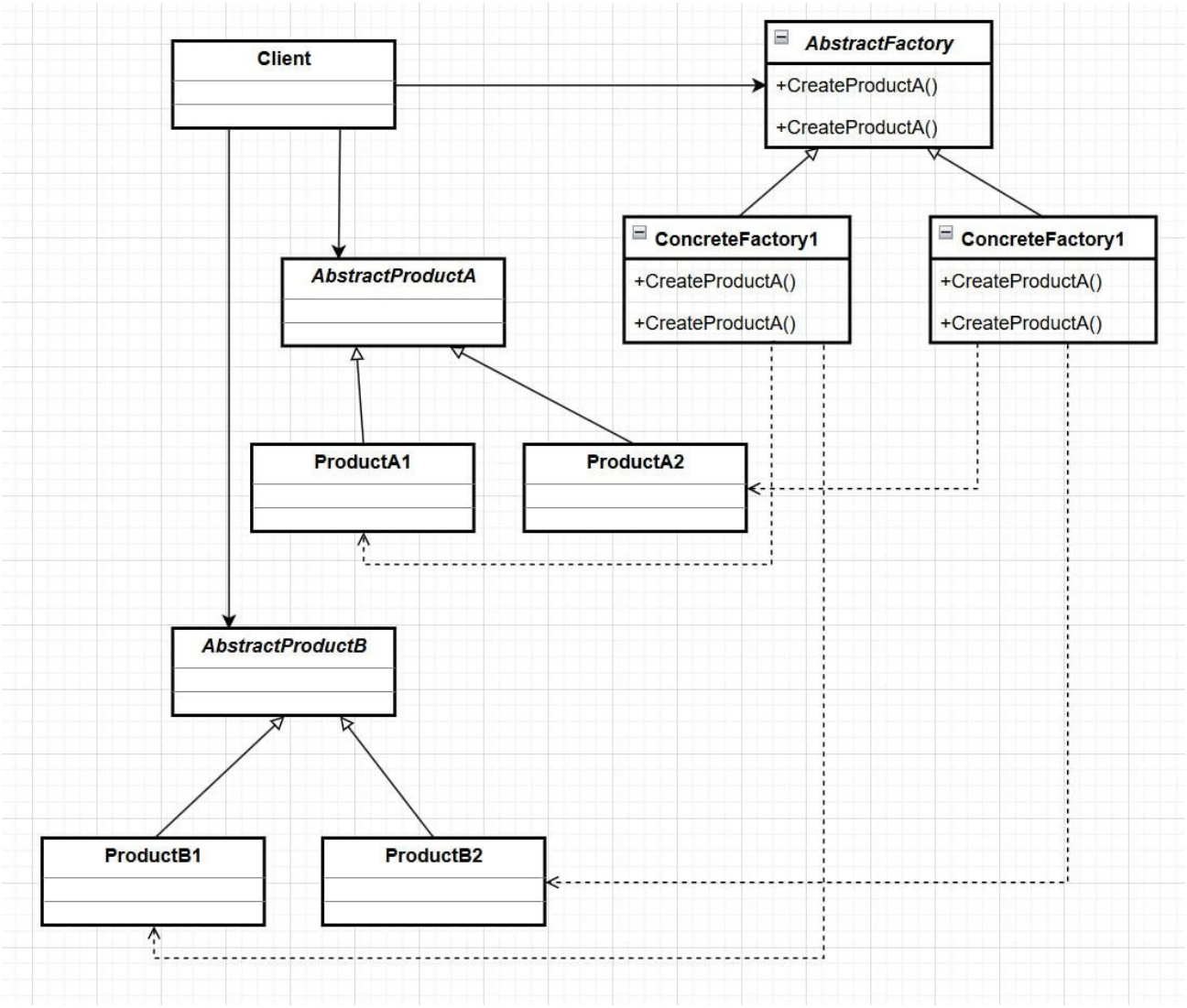
Загалом, робота продемонструвала практичну цінність шаблонів проєктування як важливого інструменту в розробці програмних систем. Їхнє використання допомагає створювати надійні, модульні та масштабовані рішення, що відповідають принципам об’єктно-орієнтованого програмування та сучасним вимогам до якості програмного забезпечення.

Відповіді на теоретичні питання

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон *«Абстрактна фабрика» (Abstract Factory)* призначений для створення сімейств взаємопов'язаних або взаємозалежних об'єктів без необхідності вказувати їхні конкретні класи. Він забезпечує гнучкість системи, дозволяючи легко змінювати типи об'єктів, які створюються, не змінюючи клієнтський код.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

- **AbstractFactory** — інтерфейс, що оголошує методи створення об'єктів.

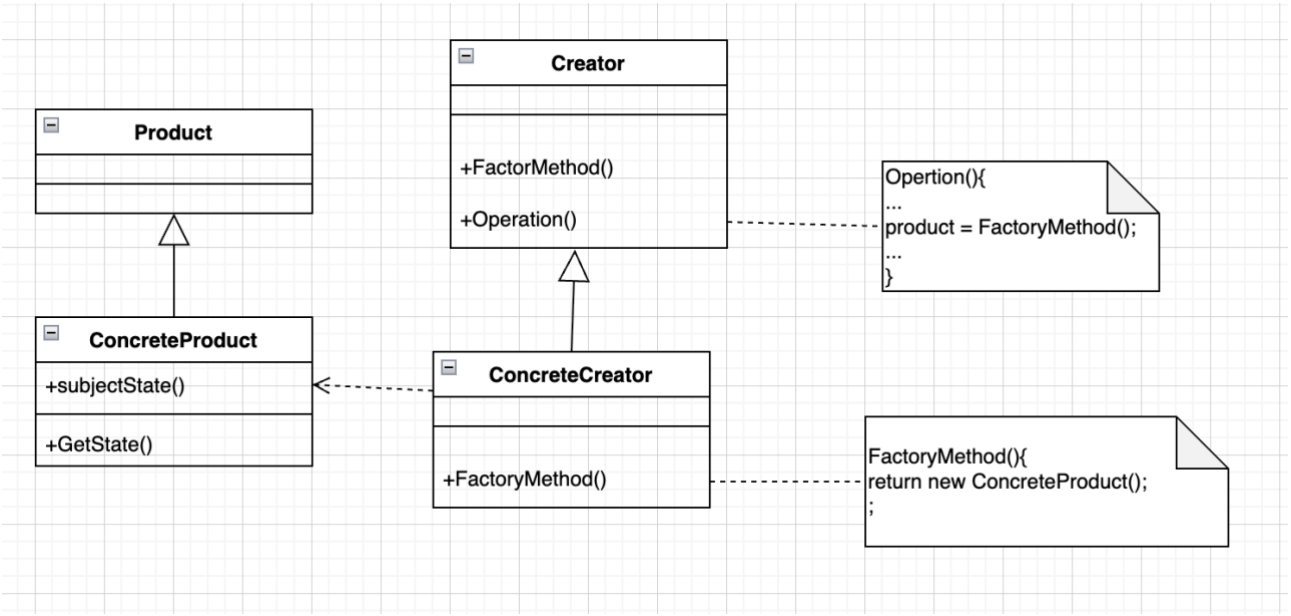
- **ConcreteFactory** — конкретні фабрики, які реалізують методи для створення певних варіацій об’єктів.
- **AbstractProduct** — інтерфейс або абстрактний клас для продуктів, що створюються фабрикою.
- **ConcreteProduct** — конкретна реалізація продуктів.
- **Client** — використовує фабрику для створення об’єктів, не знаючи їхніх конкретних класів.

Взаємодія: клієнт звертається до абстрактної фабрики, яка створює об’єкти конкретних типів, що між собою узгоджені.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон *«Фабричний метод» (Factory Method)* визначає інтерфейс для створення об’єктів, але дозволяє підкласам вирішувати, який саме клас створювати. Таким чином, створення об’єктів делегується підкласам, що забезпечує розширюваність і гнучкість архітектури.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

- **Product** — загальний інтерфейс для всіх об’єктів, які створює фабрика.
- **ConcreteProduct** — конкретна реалізація продукту.
- **Creator** — базовий клас, який визначає метод factoryMethod().
- **ConcreteCreator** — підкласи, які реалізують factoryMethod() для створення конкретних об’єктів.

Взаємодія: клієнт викликає метод operation(), який у свою чергу використовує factoryMethod() для створення потрібного продукту.

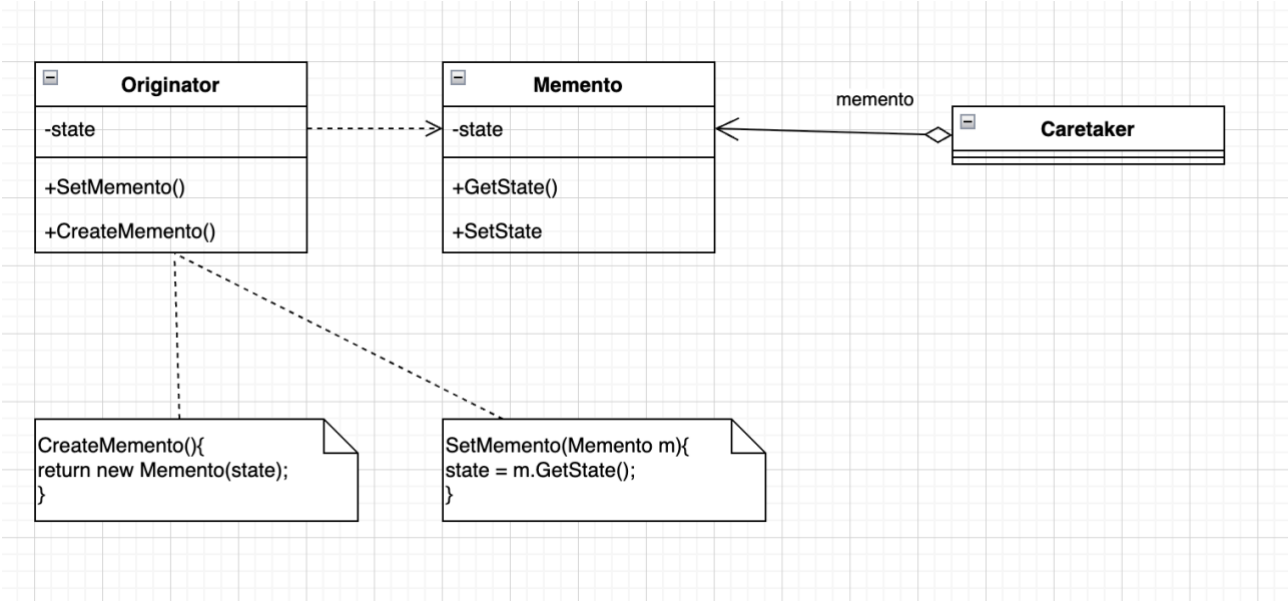
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

Ознака	**Abstract Factory**	**Factory Method**
Призначення	Створює сімейства взаємопов’язаних об’єктів	Створює окремі об’єкти певного типу
Реалізація	Містить кілька методів створення	Один метод для створення об’єкта
Рівень абстракції	Вищий — групує фабрики	Нижчий — реалізує створення конкретного об’єкта
Приклад	GUI для різних ОС (Windows, macOS)	Створення конкретного типу документа

8. Яке призначення шаблону «Знімок»?

Шаблон *«Знімок» (Memento)* дозволяє зберігати та відновлювати стан об’єкта, не порушуючи принципу інкапсуляції. Він використовується для реалізації функцій “Скасувати” (Undo) або “Відновити” (Redo).

9. Нарисуйте структуру шаблону «Знімок».



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

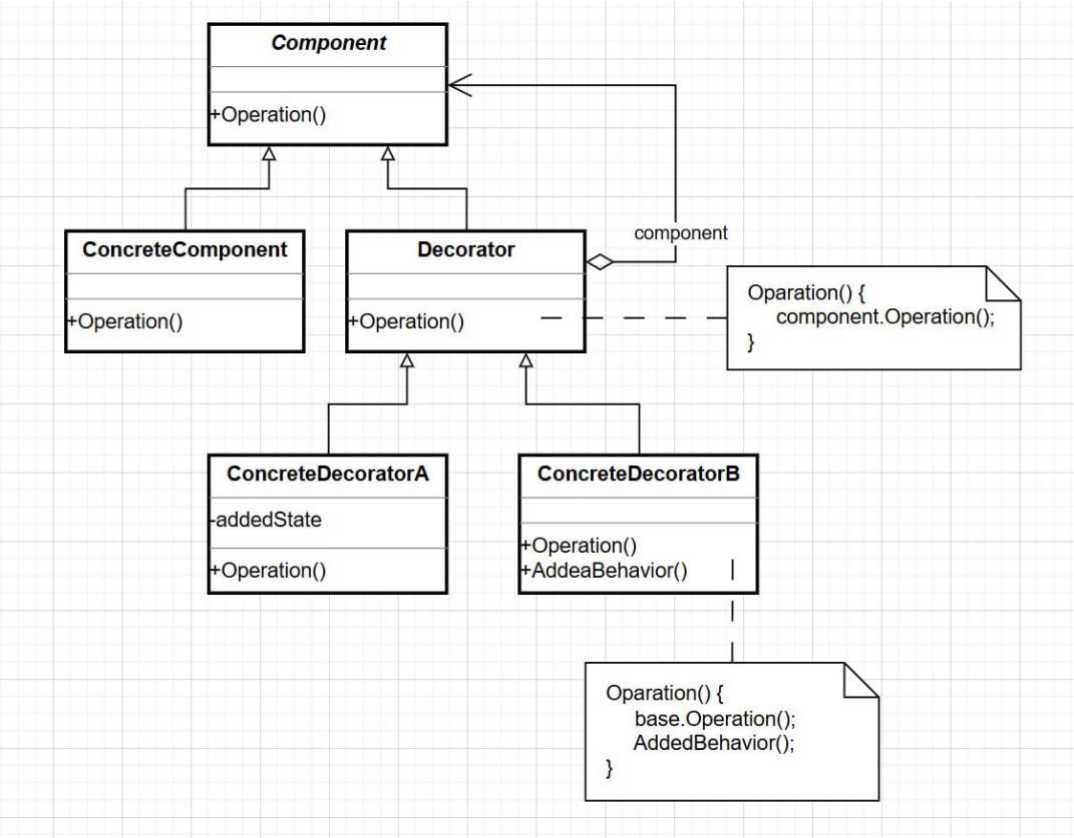
- **Originator** — об’єкт, стан якого потрібно зберігати. Створює Memento і може відновити свій стан.
- **Memento** — зберігає внутрішній стан Originator.
- **Caretaker** — зберігає знімки (Memento) та може ініціювати відновлення стану.

Взаємодія: Originator створює Memento, Caretaker зберігає його, а за потреби передає назад Originator для відновлення стану.

11. Яке призначення шаблону «Декоратор»?

Шаблон *«Декоратор» (Decorator)* використовується для динамічного додавання нової поведінки об’єктам, не змінюючи їхньої структури.
Він забезпечує гнучку альтернативу створенню підкласів для розширення функціональності.

12. Нарисуйте структуру шаблону «Декоратор».



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

- **Component** — базовий інтерфейс або абстрактний клас, який визначає спільний інтерфейс.
- **ConcreteComponent** — основний клас, до якого додається нова поведінка.
- **Decorator** — абстрактний клас, що реалізує той самий інтерфейс і містить посилання на об’єкт Component.
- **ConcreteDecorator** — додає нову поведінку, розширюючи функціональність без зміни основного класу.

Взаємодія: декоратор "обгортає" компонент, викликаючи його методи й додаючи власну логіку до або після виконання основної операції.

14. Які є обмеження використання шаблону «Декоратор»?

- Може ускладнювати структуру програми **через** велику кількість дрібних класів.

- Важко відстежувати порядок обгортання декораторів, коли їх багато.
- Не підходить для об’єктів, створення яких суворо контролюється, наприклад, через фабрику.
- Використання декораторів може ускладнити налагодження через багаторівневі виклики.