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**

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Тема роботи: 3.Текстовий редактор

Виконав
студент групи ІА–33
Супик Андрій Олександрович

Київ 2025

**Тема**: Патерни проектування

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

Посилання на репозиторій: https://github.com/insxlll/trpzlab

**Короткі 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 гнучкість 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: "Одержувач" (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

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

У цій роботі я використовую патерн "Спостерігач" (Observer). Цей патерн було обрано як архітектурне рішення для реалізації синхронізації між моделлю даних (текстовим документом) та його відображенням у графічному інтерфейсі.

У процесі роботи над проєктом стало зрозуміло, що зміни в тексті повинні миттєво відображатися не лише в основному вікні редактора, але й потенційно в інших компонентах системи (наприклад, у рядку стану, лічильнику слів або вікні попереднього перегляду).

Без використання патерну Observer, реалізація такої логіки призвела б до сильної зв'язаності (tight coupling) компонентів. Класу, що відповідає за зберігання тексту (Document), довелося б зберігати посилання на всі елементи інтерфейсу (EditorView, StatusBar тощо) і напряму викликати їхні методи оновлення при кожній зміні символу. Це порушило б принцип єдиної відповідальності, оскільки бізнес-логіка (обробка тексту) залежала б від логіки відображення (GUI), а додавання будь-якого нового візуального компонента вимагало б зміни коду самого Документа.

Патерн Observer вирішує цю проблему елегантно. Він дозволяє встановити механізм підписки "один-до-багатьох", де об'єкт даних не залежить від конкретних класів, що його відображають. У моєму проєкті клас Document виступає в ролі Видавця (Subject). Він надає методи subscribe() та unsubscribe() для реєстрації слухачів.

Компоненти інтерфейсу (Підписники/Observers) реалізують спільний інтерфейс з методом update(). Коли користувач вносить зміни в текст (вставка або видалення), Document автоматично викликає метод notify(), який проходить по списку підписників і повідомляє їм про зміну. Важливо, що Document не знає, які саме класи його слухають і як вони будуть обробляти цю подію — він лише повідомляє факт зміни стану. Це дозволяє легко розширювати функціонал редактора, додаючи нові віджети статистики чи відображення без втручання в ядро системи.

Реалізація паттерну:

```
public interface DocumentSubject {
    void addObserver(DocumentObserver observer);
    void removeObserver(DocumentObserver observer);
}
```

Рисунок 1. - реалізація інтерфейсу DocumentSubject

```
public interface DocumentObserver {

    void documentChanged(DocumentEvent event);
}
```

Рисунок 2.- реалізація інтерфейсу DocumentObserver

```
public interface DocumentSaveHandler {

    void save(ObservableDocument document);
}
```

Рисунок 3.- реалізація інтерфейсу DocumentSaveHandler

```java
public class ChangeCounterObserver implements DocumentObserver {
    private int changeCount = 0;

    @Override
    public void documentChanged(DocumentEvent event) {
        if (event.getType() == DocumentEvent.Type.CONTENT_CHANGED) {
            changeCount++;
        }
    }


    public int getChangeCount() {
        return changeCount;
    }

    public void reset() {
        changeCount = 0;
    }
}
```

Рисунок 4. - реалізація класу ChangeCounterObserver

```java
public class AutoSaveObserver implements DocumentObserver {
    private final ObservableDocument document;
    private final DocumentSaveHandler saveHandler;
    private final ScheduledExecutorService scheduler;
    private final long delayMs;
    private ScheduledFuture<?> future;


    public AutoSaveObserver(ObservableDocument document, DocumentSaveHandler saveHandler, long delayMs) {
        this.document = Objects.requireNonNull(document);
        this.saveHandler = Objects.requireNonNull(saveHandler);
        this.delayMs = Math.max(100L, delayMs);
        this.scheduler = Executors.newSingleThreadScheduledExecutor(r -> {
            Thread t = new Thread(r, "AutoSaveObserver");
            t.setDaemon(true);
            return t;
        });
    }

    @Override
    public synchronized void documentChanged(DocumentEvent event) {
        if (event.getType() == DocumentEvent.Type.CONTENT_CHANGED || event.getType() == DocumentEvent.Type.CLEARED) {

            if (future != null && !future.isDone()) {
                future.cancel(false);
            }
            future = scheduler.schedule(() -> {
                try {
                    saveHandler.save(document);
                } catch (Exception ex) {
                    System.err.println("[AutoSaveObserver] save handler failed: " + ex.getMessage());
                    ex.printStackTrace();
                }
            }, delayMs, TimeUnit.MILLISECONDS);
        }
    }

    public void shutdown() {
        try {
            scheduler.shutdownNow();
        } catch (Exception ignored) {}
    }
}
```

Рисунок 5. - реалізація класу AutoSaveObserver

```java
public class VersionHistoryObserver implements DocumentObserver {
    private final Deque<String> history;
    private final int maxSize;


    public VersionHistoryObserver(int maxSize) {
        this.history = new ArrayDeque<>(maxSize + 1);
        this.maxSize = Math.max(1, maxSize);
    }

    @Override
    public synchronized void documentChanged(DocumentEvent event) {
        if (event.getType() == DocumentEvent.Type.CONTENT_CHANGED) {
            String oldText = event.getOldText() != null ? event.getOldText() : "";

            if (history.isEmpty() || !Objects.equals(history.peekLast(), oldText)) {
                history.addLast(oldText);
                if (history.size() > maxSize) {
                    history.removeFirst();
                }
            }
        }
    }


    public synchronized List<String> getVersions() {
        return new ArrayList<>(history);
    }


    public synchronized String restoreLast(ObservableDocument document) {
        if (history.isEmpty()) return null;
        String last = history.peekLast();
        if (last != null) {
            document.setText(last);
        }
        return last;
    }


    public synchronized void clear() {
        history.clear();
    }
}
```

Рисунок 6. - реалізація класу VersionHistoryObserver

```java
public class WordCountObserver implements DocumentObserver {
    private int charCount = 0;
    private int wordCount = 0;

    @Override
    public void documentChanged(DocumentEvent event) {
        if (event.getType() == DocumentEvent.Type.CONTENT_CHANGED) {
            String text = event.getNewText() != null ? event.getNewText() : "";
            charCount = text.length();

            String trimmed = text.trim();
            if (trimmed.isEmpty()) {
                wordCount = 0;
            } else {
                wordCount = trimmed.split("\\s+").length;
            }
        }
    }

    public int getCharCount() {
        return charCount;
    }

    public int getWordCount() {
        return wordCount;
    }
}
```

Рисунок 7. - реалізація класу WordCountObserver

```java
public class UnsavedChangesObserver implements DocumentObserver {
    private volatile boolean dirty = false;

    @Override
    public void documentChanged(DocumentEvent event) {
        if (event.getType() == DocumentEvent.Type.CONTENT_CHANGED
                || event.getType() == DocumentEvent.Type.CLEARED) {
            dirty = true;
        }
        if (event.getType() == DocumentEvent.Type.TITLE_CHANGED) {

            dirty = true;
        }
    }

    public boolean isDirty() {
        return dirty;
    }


    public void markSaved() {
        dirty = false;
    }
}
```

Рисунок 8. - реалізація класу UnsavedChangesObserver

```java
public class ObservableDocument extends Document implements DocumentSubject {
    private final List<DocumentObserver> observers = new ArrayList<>();

    public ObservableDocument() {
        super();
    }

    public ObservableDocument(String title, String[] content) {
        super(title, content);
    }

    @Override
    public synchronized void setContent(String[] content) {
        String oldText = DocumentTextHelper.getText(this);
        super.setContent(content);
        String newText = DocumentTextHelper.getText(this);
        if (!Objects.equals(oldText, newText)) {
            notifyObservers(new DocumentEvent(DocumentEvent.Type.CONTENT_CHANGED, oldText, newText));
        }
    }


    public synchronized void setText(String text) {
        String oldText = DocumentTextHelper.getText(this);
        DocumentTextHelper.setText(this, text);
        String newText = DocumentTextHelper.getText(this);
        if (!Objects.equals(oldText, newText)) {
            notifyObservers(new DocumentEvent(DocumentEvent.Type.CONTENT_CHANGED, oldText, newText));
        }
    }

    @Override
    public synchronized void setTitle(String title) {
        String oldText = getTitle();
        super.setTitle(title);
        String newText = getTitle();
        if (!Objects.equals(oldText, newText)) {
            notifyObservers(new DocumentEvent(DocumentEvent.Type.TITLE_CHANGED, oldText, newText));
        }
    }

    @Override
    public synchronized void addObserver(DocumentObserver observer) {
        if (observer != null && !observers.contains(observer)) {
            observers.add(observer);
        }
    }

    @Override
    public synchronized void removeObserver(DocumentObserver observer) {
        observers.remove(observer);
    }

    private synchronized void notifyObservers(DocumentEvent event) {

        List<DocumentObserver> snapshot = new ArrayList<>(observers);
        for (DocumentObserver obs : snapshot) {
            try {
                obs.documentChanged(event);
            } catch (Exception ex) {

                System.err.println("[ObservableDocument] Observer threw: " + ex.getMessage());
                ex.printStackTrace();
            }
        }
    }
}
```

Рисунок 9-10. - реалізація класу ObservableDocument

## Діаграма класів

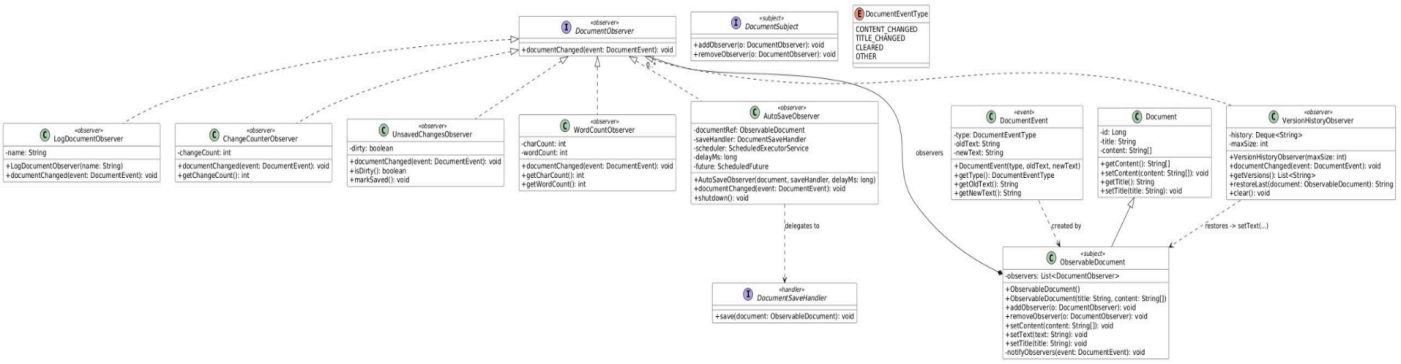



Рисунок 11. - Діаграма класів паттерну Observer

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

Особливу увагу я приділив реалізації шаблону Observer, детально описавши його основну логіку та функціональність у контексті мого проєкту текстового редактора.

Цей патерн було обрано як архітектурне рішення для десинхронізації та відокремлення моделі даних (ObservableDocument) від логіки реакції на її зміни (Конкретних Спостерігачів, таких як AutoSaveObserver, VersionHistoryObserver та WordCountObserver). Я зрозумів, наскільки цей патерн спрощує процес управління залежностями, забезпечуючи слабку зв'язаність компонентів.

Використання шаблону Observer забезпечує:

- Слабка зв'язаність: Модель не знає про конкретні класи, які її використовують.
- Розширюваність: Легке додавання нових функціональних можливостей (наприклад, нового лічильника статистики) без зміни основного коду документа.
- Ефективна синхронізація: Автоматичне оповіщення всіх залежних об'єктів про зміну стану.

Впровадження патерну Observer дозволило створити модульну та гнучку архітектуру програми, де зміни в тексті обробляються численними компонентами незалежно, дотримуючись Принципу єдиної відповідальності.

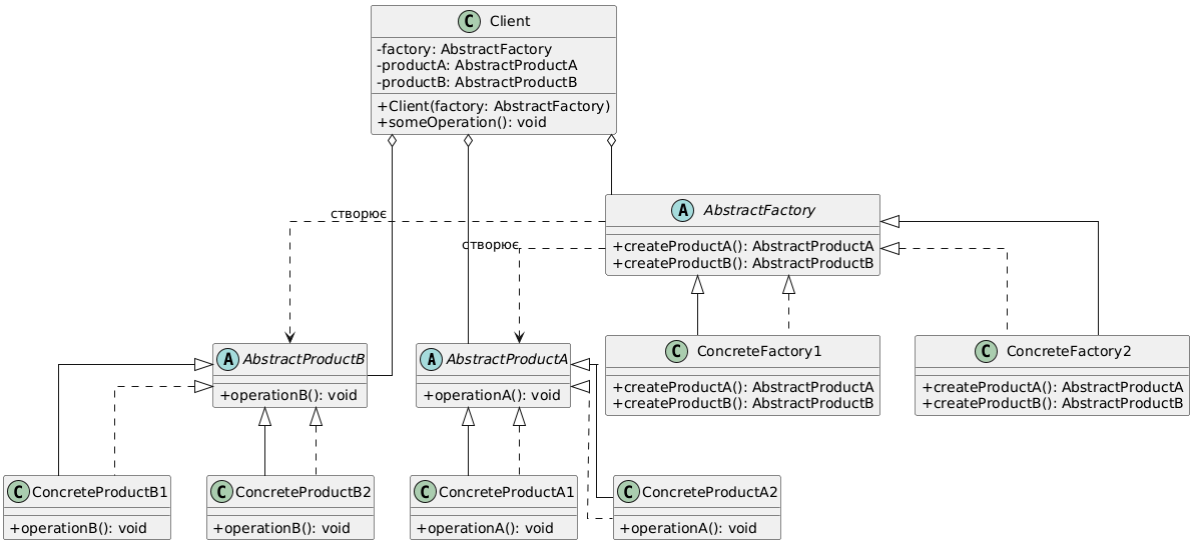
## Контрольні запитання

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.

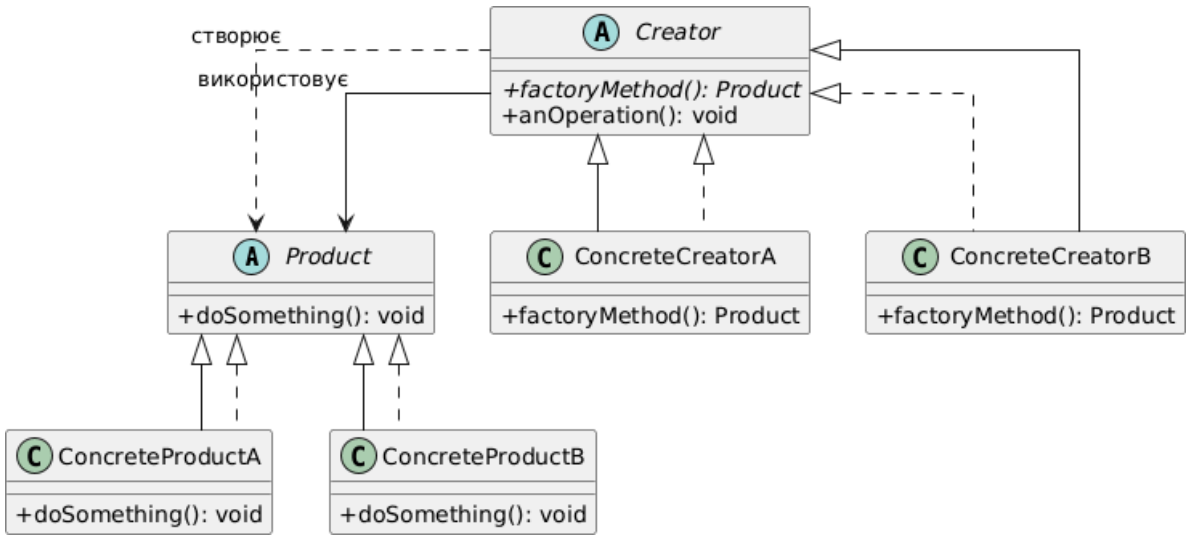
Взаємодія між ними відбувається так: клієнт звертається до абстрактної фабрики, щоб створити об’єкти певного типу. Конкретна фабрика реалізує цей запит і повертає відповідні конкретні продукти. У результаті клієнт може використовувати створені об’єкти, не знаючи, які саме класи стоять за ними, що забезпечує 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.

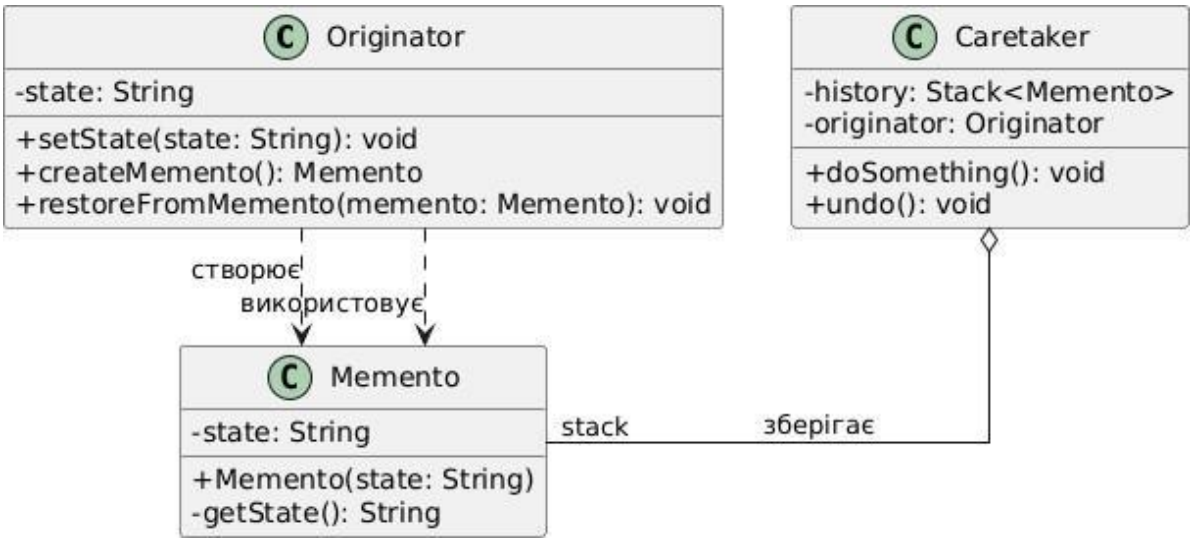
Натомість «Абстрактна фабрика» використовується для створення цілих сімейств взаємопов’язаних об’єктів, які повинні узгоджено працювати між собою. Вона надає інтерфейс для створення кількох видів об’єктів без зазначення їх конкретних класів. Таким чином, «Абстрактна фабрика» часто використовує фабричні 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

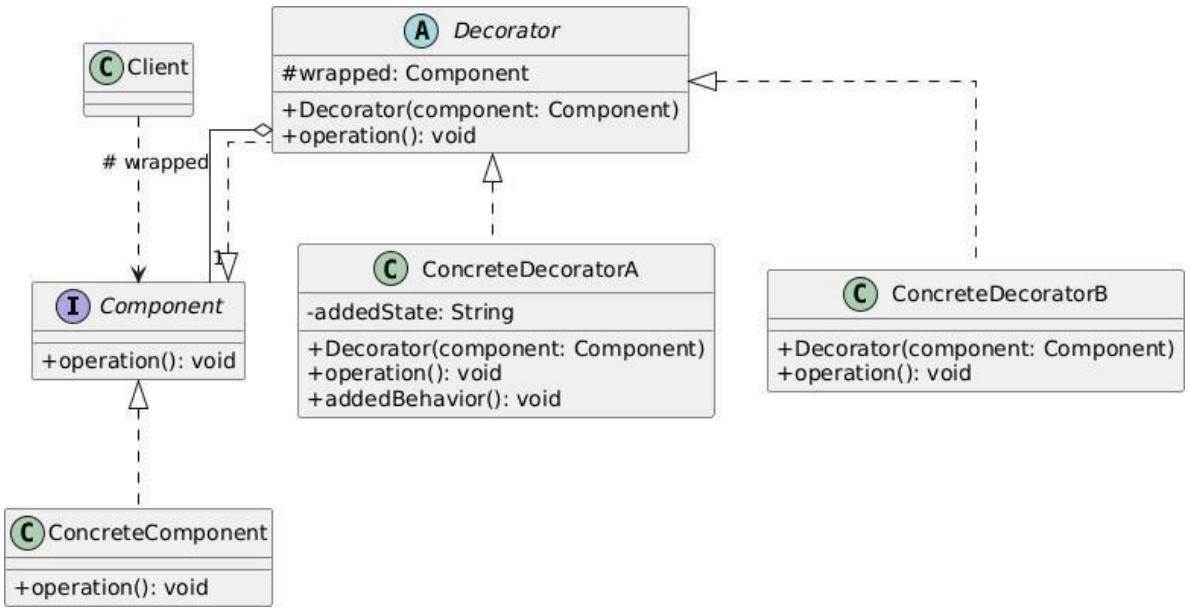
попередній стан, Caretaker передає збережений знімок назад Originator. Originator використовує дані із знімка, щоб повернутися до попереднього стану.

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 обгортають 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.