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 № 8

Технології розроблення програмного забезпечення

«Патерни проектування»

Виконала:
студентка групи ІА-32
Чайковська С. В.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати їх в реалізації програмної системи.

1 Теоретичні відомості

Шаблон «Composite»

Призначення: Патерн "Composite" використовується для представлення і роботи з деревовидними структурами, де окремі об’єкти (листи) і групи об’єктів (вузли) обробляються однаково.

Проблема: Патерн "Composite" корисний для роботи з деревовидною структурою. Наприклад, продукти й коробки, які можуть містити інші коробки. Задача: обчислити загальну ціну замовлення, не знаючи наперед структури вкладеності.

Рішення: "Composite" уніфікує роботу з продуктами й коробками через спільний інтерфейс. Продукт повертає свою ціну, коробка обчислює суму вартості свого вмісту, незалежно від рівнів вкладеності.

Переваги:

- Спрощує роботу зі складними структурами.
- Полегшує додавання нових компонентів.
- Створює загальнодизайнерське рішення.

Недоліки:

- Може бути складним для реалізації.
- Перебір вмісту може вимагати значних ресурсів.

Шаблон «Flyweight»

Призначення: Шаблон використовується для зменшення кількості об'єктів шляхом поділу загальних даних між екземплярами. Наприклад, у графічних системах або текстових редакторах.

Проблема: В іграх або додатках із великою кількістю однакових об'єктів (наприклад, кулі, частинки, текстові символи) обсяг оперативної пам’яті стає критичним. Кожен об'єкт містить багато даних, які можуть дублюватися.

Рішення: Розділяти стан об’єктів на: - Внутрішній – загальні дані, які зберігаються в спільному об'єкті. - Зовнішній – унікальні дані, пов’язані з конкретним контекстом використання. Це дозволяє значно скоротити використання пам'яті.

Переваги:

- Економія оперативної пам'яті.
- Уніфікація структури даних.

Недоліки:

- Зростає складність коду.
- Витрачається час на пошук спільних даних.

Структура патерну зображена на Рис. 1:

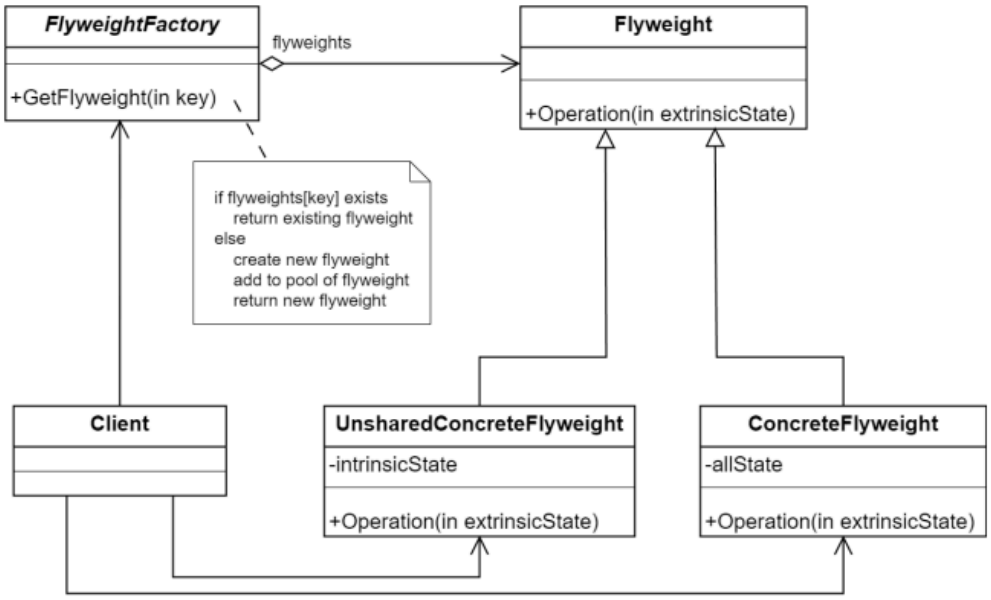


Рис. 1 – Структура патерну Flyweight (Легковаговик)

Шаблон «Intepreter»

Призначення: Шаблон використовується для подання граматики і створення інтерпретатора для мови (наприклад, скриптової), що дозволяє працювати з термінальними та нетермінальними символами у вигляді деревовидної структури.

Проблема: Необхідно автоматизувати розбір виразів або завдань, які часто змінюються, і реалізувати зрозумілу систему обчислення.

Рішення: Створюється абстрактне синтаксичне дерево, вузли якого представлені класами: - Термінальний вираз – відповідає за базу рекурсії. - Нетермінальний вираз – реалізує обробку підвиразів (рекурсивно). Кожен вузол виконує операцію розбору з урахуванням переданого контексту.

Переваги:

- Легка розширюваність граматики.
- Простота змін у способі обчислення виразів.

Недоліки:

- Ускладнення підтримки граматики з великою кількістю правил.

Шаблон «Visitor»

Призначення: Патерн дозволяє вказувати операції над елементами без зміни їхньої структури. Групує однотипні операції, застосовувані до різних об'єктів.

Проблема: Уявіть програму з графами (вузли — міста, локації тощо). Завдання — реалізувати експорт графа в XML без змін у стабільному коді вузлів, але при цьому дозволити додавання нових операцій.

Рішення: Нову поведінку реалізують через окремий клас "Відвідувач". Об'єкти передають себе до методу відвідувача для виконання операцій. Це дозволяє додавати нові операції без змін у класах вузлів.

Переваги:

- Додає нові операції без змін у коді об'єктів.
- Групує логіку, що застосовується до різних об'єктів.

Недоліки:

- Додає складність у реалізації.
- Необхідно підтримувати багато відвідувачів для різних операцій.

2 Тема

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).

3 Аргументація вибору патерну «Flyweight»

Патерн *Flyweight* було обрано для реалізації механізму ефективного управління символами в текстовому редакторі. Його застосування дозволяє суттєво зменшити споживання пам’яті під час роботи з великими обсягами тексту, повторно використовуючи однакові об’єкти замість створення нових копій.

Основні аргументи вибору патерну:

- *Оптимізація використання ресурсів.* Flyweight забезпечує повторне використання вже створених об’єктів із однаковим внутрішнім станом (символи, що повторюються), що дозволяє мінімізувати кількість створюваних екземплярів та зменшити навантаження на пам’ять.
- *Розділення внутрішнього та зовнішнього стану.* Патерн чітко відокремлює сталі характеристики об’єкта від змінних. Це забезпечує незалежність логіки від конкретного контексту використання.
- *Централізоване керування об’єктами.*Фабрика Flyweight створює, зберігає та повторно надає об’єкти клієнтам, що гарантує відсутність дублювання і єдиний контроль за створенням символів.
- *Покращення продуктивності.* Завдяки зменшенню кількості об’єктів і уникненню дублювання, програма працює швидше та ефективніше оперує великими текстами.

Таким чином, застосування патерну *Flyweight* дозволяє побудувати легку та ефективну архітектуру, у якій зменшується кількість створюваних об’єктів, оптимізується використання пам’яті, а управління станом стає більш контрольованим і гнучким.

4 Реалізація частини функціоналу робочої програми

Структура проекту зображена на Рисунку 2:

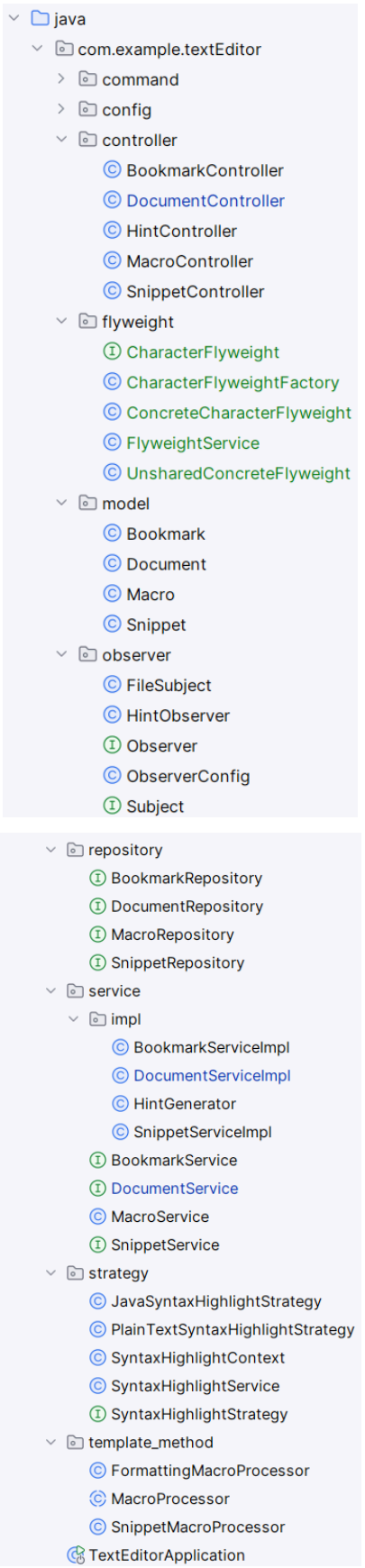


Рис. 2 – Структура проекту з використанням патерну «Flyweight»

Структура проєкту TextEditor:

Архітектура побудована за принципом багаторівневого розподілу, що забезпечує чітке розділення відповідальностей:

- *flyweight* – реалізація патерну Flyweight для оптимізації використання пам’яті під час роботи з великою кількістю однотипних об’єктів (символів тексту). Основні класи:
  - *CharacterFlyweight* — це інтерфейс, який визначає контракт для всіх об’єктів-символів. Він задає спільну поведінку через методи *getSymbol()* та *operation(int position),* що дозволяє розділяти внутрішній і зовнішній стан символів.
  - *ConcreteCharacterFlyweight* — реалізує інтерфейс *CharacterFlyweight* і представляє розділюваний об’єкт, який містить лише внутрішній стан.
  - *CharacterFlyweightFactory* — фабрика, що керує пулом створених об’єктів *Flyweight*. Вона надає клієнтам доступ до вже існуючих символів або створює нові, якщо таких ще немає в кеші.
  - *UnsharedConcreteFlyweight* — реалізує інтерфейс *CharacterFlyweight*, але не ділиться своїм станом із іншими об’єктами. Використовується для складних або унікальних символів.
  - *FlyweightService* — сервісний клас, який демонструє практичне використання фабрики *Flyweight* у процесі обробки тексту.
- *template_method* – реалізація патерну Template Method для стандартизації процесу обробки макросів, дозволяючи нащадкам визначати специфічну логіку кроків. Основні класи:
  - *MacroProcessor* — це абстрактний клас, який визначає контракт для всіх обробників макросів. Він встановлює

уніфікований процес обробки *(process())*, який мають дотримуватися всі нащадки.
    - *SnippetMacroProcessor* — реалізує абстрактні методи (*validateInput, execute* тощо) для обробки макросів, специфічних для вставки сніпетів коду.
    - *FormattingMacroProcessor* — реалізує абстрактні методи для обробки макросів форматування тексту. Його *execute(String)* застосовує стилі тексту або змінює регістр (великий/малий).
- *observer* – реалізація патерну Observer (спостерігач), який дозволяє автоматично оновлювати підказки при зміні тексту документа. Основні класи:
    - *Subject* – інтерфейс, що визначає методи *attach(), detach(), notifyObservers()*.
    - *Observer* – інтерфейс, що має метод *update(String content)*.
    - *FileSubject* – конкретний суб'єкт *(ConcreteSubject)*, який зберігає поточний стан тексту *(content)* і повідомляє всіх підписаних спостерігачів про зміни.
    - *HintObserver* – конкретний спостерігач *(ConcreteObserver)*, який реагує на зміни вмісту документа, генеруючи нові підказки для користувача.
    - *ObserverConfig* – клас конфігурації Spring, який автоматично реєструє спостерігача *(HintObserver)* у суб'єкта *(FileSubject)* під час запуску застосунку.
- *command* – реалізація патерну «Command» для організації дій із документами через інкапсуляцію запитів у вигляді об'єктів. Забезпечує гнучке керування командами, підтримку історії дій. Основні класи:
    - *Command* – інтерфейс, що визначає метод *execute(),* який реалізується конкретними командами.

- *OpenDocumentCommand, SaveDocumentCommand, UploadDocumentCommand, EditDocumentCommand* – конкретні команди, які інкапсулюють дії з відкриття, збереження, завантаження та редагування документів відповідно.
- *CommandInvoker* – клас, що виконує команди, зберігає їх в історії та може реалізовувати скасування дій.
- *DocumentServiceImpl* виступає отримувачем (receiver), який безпосередньо реалізує бізнес-логіку команд.
- *DocumentController* є клієнтом, який створює конкретні команди та передає їх інвокеру для виконання.

- *controller* – містить контролери, які приймають запити від користувача, викликають бізнес-логіку та повертають відповіді на UI: *DocumentController, BookmarkController, HintController, MacroController, SnippetController* – керують операціями з відповідними сутностями (документи, закладки, підказки, макроси, фрагменти коду).
- *model* – включає моделі даних (сутності), що відображають таблиці бази даних: *Document, Bookmark, Hint, Macros, Snippet*. Кожна з моделей описує структуру даних, які зберігаються у БД.
- *repository* – шар доступу до даних. Містить інтерфейси для роботи з БД: *DocumentRepository, BookmarkRepository, HintRepository, MacrosRepository, SnippetRepository*. Завдяки Spring Data JPA забезпечується збереження, пошук та вибірка даних без написання SQL-запитів вручну.
- *service* – шар бізнес-логіки. Інтерфейси: *DocumentService, BookmarkService, MacroService, SnippetService.*

  Реалізації (impl): *DocumentServiceImpl, BookmarkServiceImpl, HintGenerator, SnippetServiceImpl.* У цьому шарі виконується логіка обробки даних, яка відділяє контролери від репозиторіїв.

- *strategy* – реалізація патерну «Strategy» для підсвічування синтаксису. Містить інтерфейс, контекст і конкретні стратегії:
    - *SyntaxHighlightStrategy* – інтерфейс, що визначає метод *highlight(String text)* для різних алгоритмів підсвічування.
    - *JavaSyntaxHighlightStrategy* – конкретна стратегія для підсвічування синтаксису мови Java за допомогою регулярних виразів.
    - *PlainTextSyntaxHighlightStrategy* – стратегія, яка залишає текст без змін (використовується для випадків без підсвічування).
    - *SyntaxHighlightContext* – контекст, який зберігає поточну стратегію та викликає її методи.
    - *SyntaxHighlightService* – сервіс, який визначає, яку саме стратегію застосувати до тексту залежно від розширення файлу.
- *TextEditorApplication* – головний клас, що запускає Spring Boot застосунок.

Така структура проєкту дає змогу легко підтримувати й розширювати систему, оскільки контролери, бізнес-логіка та доступ до даних розділені по різних шарах.

5 Реалізація патерну «Flyweight»

1. ***Інтерфейс CharacterFlyweight*** представлений на Рисунку 3:

```java
package com.example.textEditor.flyweight;

public interface CharacterFlyweight {
    char getSymbol();
    void operation(int position);
}
```

Рис. 3 – Інтерфейс CharacterFlyweight

Інтерфейс *CharacterFlyweight*, наведений на рис. 3, визначає спільний контракт для всіх об’єктів, які реалізують патерн *Flyweight* у системі текстового редактора. Основне призначення цього інтерфейсу — задати єдину поведінкову модель для всіх символів, як спільних, так і унікальних.

Інтерфейс містить два методи:

- *getSymbol()* — повертає внутрішній стан символу, який залишається незмінним і може розділятися між багатьма контекстами.
- *operation(int position)* — реалізує операцію з зовнішнім станом, який передається ззовні, наприклад, позицією символу у тексті.

Інтерфейс *CharacterFlyweight* забезпечує слабке зв’язування між внутрішнім станом символу та його контекстом використання, дозволяючи мінімізувати кількість створюваних об’єктів під час обробки тексту.

2. ***Клас CharacterFlyweightFactory*** представлений на Рисунку 4:

```java
package com.example.textEditor.flyweight;

import java.util.HashMap;
import java.util.Map;

public class CharacterFlyweightFactory {  5 usages  new *
    private static final CharacterFlyweightFactory INSTANCE = new CharacterFlyweightFactory();
    private final Map<Character, CharacterFlyweight> pool = new HashMap<>();  3 usages

    private CharacterFlyweightFactory() {}  1 usage  new *

    public static CharacterFlyweightFactory getInstance() {  1 usage  new *
        return INSTANCE;
    }

    public CharacterFlyweight getFlyweight(char symbol) {  1 usage  new *
        return pool.computeIfAbsent(symbol, ConcreteCharacterFlyweight::new);
    }
```

```
20          public int getFlyweightCount() {  1 usage  new *
21              return pool.size();
22          }
23
24          public Map<Character, CharacterFlyweight> getPool() {  no usages  new *
25              return pool;
26          }
27      }
```

Рис. 4 – Клас CharacterFlyweightFactory

Клас *CharacterFlyweightFactory*, наведений на рис. 4, реалізує фабрику спільних об’єктів *Flyweight*. Його основне призначення — керувати пулом (кешем) створених екземплярів *ConcreteCharacterFlyweight* і гарантувати, що для кожного унікального символу створюється лише один об’єкт.

У класі визначено:

- колекцію *Map<Character, CharacterFlyweight> pool*, де зберігаються усі створені flyweight-об’єкти;
- метод *getFlyweight(char symbol)*, який або повертає вже існуючий екземпляр символу, або створює новий та додає його до пулу;
- методи *getFlyweightCount()* і *getPool()* для контролю стану фабрики.

Клас *CharacterFlyweightFactory* реалізує централізоване управління об’єктами *Flyweight*, що дозволяє ефективно контролювати пам’ять і уникати дублювання екземплярів при роботі з великим текстом.

3. ***Клас ConcreteCharacterFlyweight*** представлений на Рисунку 5:

```
1   package com.example.textEditor.flyweight;
2
3   public class ConcreteCharacterFlyweight implements CharacterFlyweight {
4       private final char symbol;  2 usages
5
6       public ConcreteCharacterFlyweight(char symbol) { this.symbol = symbol; }
9
10      @Override
11      public char getSymbol() { return symbol; }
14
15      @Override
16      public void operation(int position) {
17          System.out.println("Символ '" + getSymbol() + "' використано на позиції " + position);
18      }
19  }
```

Рис. 5 – Клас ConcreteCharacterFlyweight

Клас *ConcreteCharacterFlyweight*, наведений на рис. 5, є конкретною реалізацією інтерфейсу *CharacterFlyweight* і відповідає за зберігання спільного (розділюваного) стану символів у системі. Його основне призначення — представляти один екземпляр для кожного унікального символу, який може використовуватися повторно багатьма клієнтами без дублювання об’єктів.

У класі визначено поле *symbol* — внутрішній стан, який є незмінним і спільним для всіх контекстів використання даного символу.

Метод *operation(int position)* виконує обробку символу з урахуванням зовнішнього стану — позиції у тексті.

Клас *ConcreteCharacterFlyweight* реалізує ключову ідею патерну — розподіл спільного стану між багатьма об’єктами, що суттєво зменшує споживання пам’яті при роботі з великими обсягами тексту.

4. ***Клас UnsharedConcreteFlyweight*** представлений на Рисунку 6:

```
1   package com.example.textEditor.flyweight;
2
3   public class UnsharedConcreteFlyweight implements CharacterFlyweight {  1 usage  new *
4       private final String complexSymbol;  3 usages
5       private final String color;  2 usages
6       private final boolean bold;  2 usages
7       private final int fontSize;  2 usages
8
9       public UnsharedConcreteFlyweight(String complexSymbol, String color, boolean bold, int fontSize) {
10          this.complexSymbol = complexSymbol;
11          this.color = color;
12          this.bold = bold;
13          this.fontSize = fontSize;
14      }
15
16      @Override  2 usages  new *
17      public char getSymbol() {
18          return complexSymbol.charAt(0);
19      }
20
21      @Override  3 usages  new *
22      public void operation(int position) {
23          System.out.println("Нерозділюваний символ \"" + complexSymbol + "\" на позиції " + position
24                  + " (color=" + color + ", bold=" + bold + ", fontSize=" + fontSize + ")");
25      }
26  }
```

Рис. 6 – Клас UnsharedConcreteFlyweight

Клас *UnsharedConcreteFlyweight,* наведений на рис. 6, реалізує інтерфейс *CharacterFlyweight*, однак представляє унікальні об’єкти, які мають власний,

складніший внутрішній стан. Його основне призначення — забезпечити зберігання і обробку символів або конструкцій, які не можуть бути спільно використані між контекстами.

Клас містить кілька внутрішніх параметрів:

- *complexSymbol* — текстовий вираз або комбінація символів;
- *color, bold, fontSize* — атрибути форматування, що визначають зовнішній вигляд символу.

Метод *operation(int position)* відображає інформацію про нерозділюваний символ з урахуванням позиції та параметрів стилю.

Клас *UnsharedConcreteFlyweight* демонструє можливість розширення патерну *Flyweight* для об'єктів зі складним або унікальним станом, які не підлягають кешуванню у фабриці.

5. ***Клас FlyweightService*** представлений на Рисунку 7:

```java
package com.example.textEditor.flyweight;

import org.springframework.stereotype.Service;

@Service  3 usages  new *
public class FlyweightService {
    private final CharacterFlyweightFactory factory = CharacterFlyweightFactory.getInstance();

    public void processText(String text) {  2 usages  new *
        if (text == null) return;

        int position = 0;
        for (char ch : text.toCharArray()) {
            CharacterFlyweight flyweight = factory.getFlyweight(ch);

            if (flyweight instanceof ConcreteCharacterFlyweight concrete) {
                concrete.operation(position);
                System.out.println("Concrete Flyweight: " + concrete.getSymbol());
            } else {
                flyweight.operation(position);
            }

            position++;
        }

        CharacterFlyweight unshared = new UnsharedConcreteFlyweight( complexSymbol: "[END]",  color: "red"
        unshared.operation(position);
    }

    public int getUniqueSymbolCount() { return factory.getFlyweightCount(); }
```

Рис. 7 – Клас FlyweightService

Клас *FlyweightService*, зображений на рис. 5, виступає клієнтом патерну *Flyweight*, який демонструє практичне використання об’єктів *Flyweight* у процесі обробки тексту. Його основне призначення — організувати послідовне отримання символів із фабрики, виклик методів обробки та демонстрацію економії пам’яті за рахунок спільного використання об’єктів.

Метод *processText(String text)* виконує такі дії:

1. Ітерується по кожному символу вхідного рядка.
2. Отримує спільний Flyweight-об’єкт із фабрики.
3. Викликає метод *operation(int position)* для виконання дії з урахуванням позиції символу.
4. Додає наприкінці спеціальний *UnsharedConcreteFlyweight ([END])*, який є унікальним об’єктом без кешування.

Метод *getUniqueSymbolCount()* дозволяє оцінити кількість створених спільних об’єктів Flyweight.

Клас *FlyweightService* реалізує контекстну логіку клієнта, що показує взаємодію між фабрикою, спільними та унікальними Flyweight-об’єктами у межах патерну.

6 Діаграма класів для патерну «Flyweight»

Діаграма класів для патерну «Flyweight» зображена на Рисунку 8:

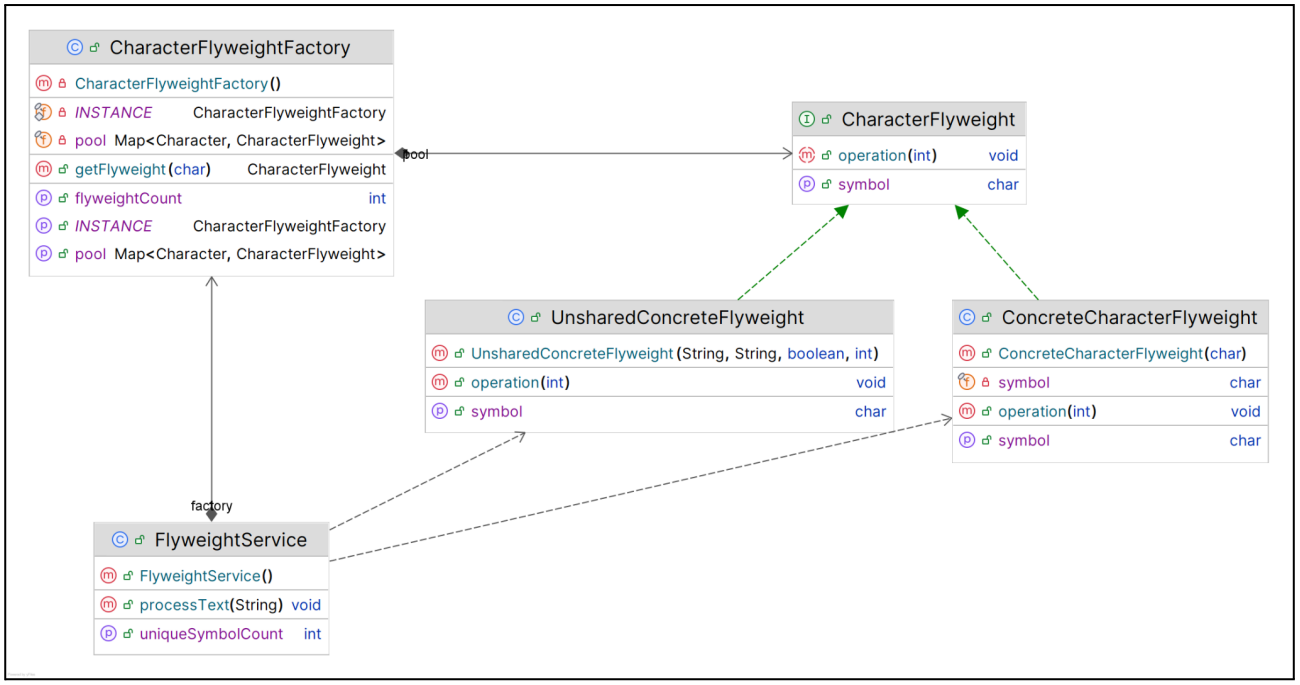


Рис. 8 – Діаграма класів для патерну «Flyweight»

Патерн *Flyweight* дозволяє ефективно використовувати пам’ять за рахунок повторного використання вже створених об’єктів із спільним внутрішнім станом. Завдяки цьому велика кількість однотипних об’єктів може оброблятися з мінімальними витратами ресурсів.

Інтерфейс:

CharacterFlyweight — визначає спільний контракт для всіх об’єктів-символів. Він містить методи:

- *char getSymbol()* — повертає внутрішній стан, який є спільним для багатьох екземплярів;
- *void operation(int position)* — приймає зовнішній стан і виконує відповідну операцію.

Конкретні реалізації:

- *ConcreteCharacterFlyweight* — реалізує спільний об’єкт. Містить лише внутрішній стан — символ, який не змінюється між різними контекстами. Такі об’єкти повторно використовуються в різних частинах тексту, що суттєво економить пам’ять.

- *UnsharedConcreteFlyweight* — представляє нерозділюваний об’єкт *Flyweight*, який має власні властивості. Він використовується для складних або унікальних елементів тексту, які не можна розділяти між іншими частинами програми.

Фабрика Flyweight:

CharacterFlyweightFactory — відповідає за створення, кешування та повторне використання об’єктів *Flyweight*. Забезпечує централізоване управління розділюваними об’єктами.

Містить:

- *Map<Character, CharacterFlyweight> pool* — пул збережених символів;
- *getFlyweight(char)* — повертає вже існуючий об’єкт із пулу або створює новий, якщо такий ще не існує;
- *getFlyweightCount()* — показує кількість унікальних створених символів.

Клієнт:

FlyweightService — клас, який демонструє практичне використання патерну. Він отримує об’єкти *Flyweight* із фабрики, виконує над ними операції, підраховує унікальні символи. Сервіс виступає посередником між бізнес-логікою програми та механізмом повторного використання об’єктів.

Посилання на GitHub: https://github.com/chaikovsska/textEditor

7 Висновки

У ході виконання лабораторної роботи було реалізовано патерн проєктування «Flyweight» для оптимізації використання пам’яті під час роботи з великою кількістю однотипних об’єктів у текстовому редакторі. Завдяки цьому патерну вдалося розділити внутрішній і зовнішній стан об’єктів, що дозволило повторно використовувати однакові символи замість створення нових екземплярів. Реалізація фабрики керування пулом об’єктів забезпечила централізоване створення та повторне використання спільних екземплярів,

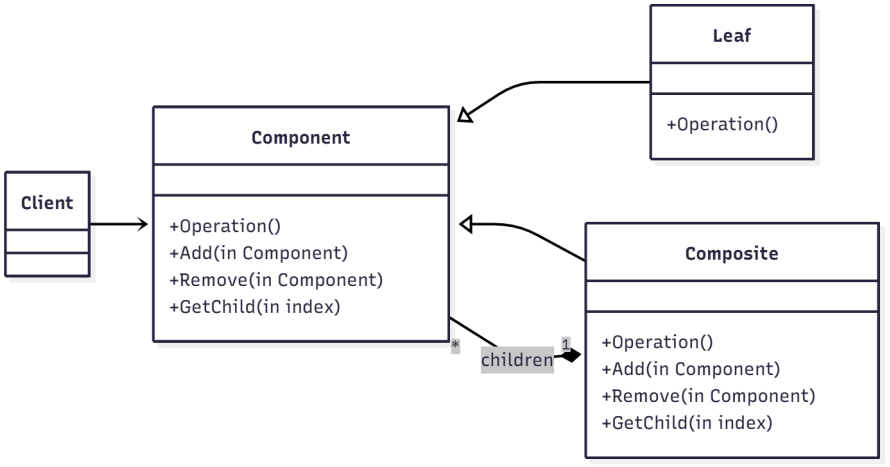
зменшуючи навантаження на систему. Використання Flyweight підвищило ефективність, знизило споживання ресурсів та покращило продуктивність програми. Застосування цього патерну зробило архітектуру системи більш гнучкою, масштабованою та придатною до розширення, забезпечивши можливість обробки великих обсягів текстових даних без значних витрат пам’яті.

8 Відповіді на питання

1. Яке призначення шаблону «Композит»?

Шаблон «Composite» використовується для представлення ієрархічних деревоподібних структур типу «частина-ціле». Він дозволяє однаково обробляти як окремі об’єкти, так і групи об’єктів, спрощуючи роботу з рекурсивними структурами.

2. Нарисуйте структуру шаблону «Композит».



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

- Component — спільний інтерфейс для всіх об’єктів дерева. Може містити базові методи (наприклад, Operation()).
- Leaf — кінцевий елемент без дочірніх об’єктів. Реалізує поведінку компонентів нижнього рівня.
- Composite — об’єкт, який містить інші компоненти (як листи, так і інші композити). Реалізує методи для роботи з колекцією дочірніх елементів (наприклад, Add(), Remove(), GetChild()).

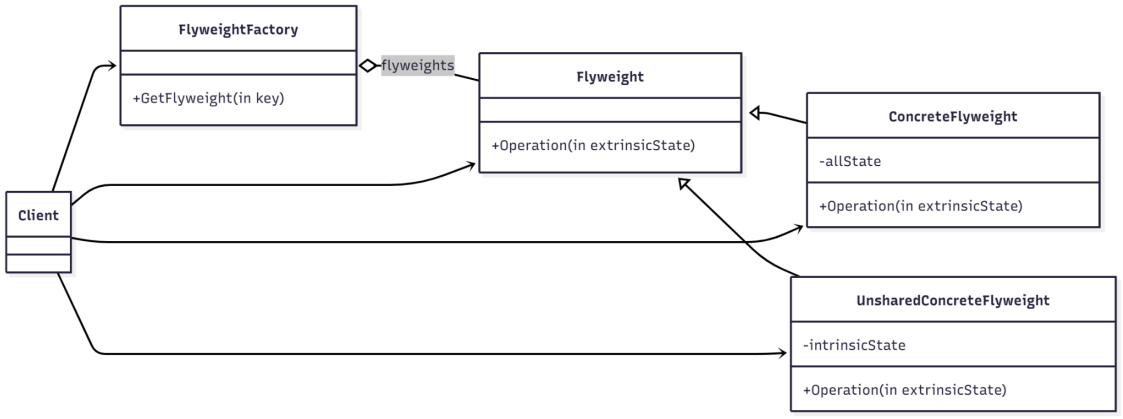
- Client — працює з об’єктами через інтерфейс Component, не розрізняючи окремі об’єкти та їх групи.

Взаємодія: Клієнт викликає методи Component. Якщо об’єкт — композит, він делегує виконання дочірнім об’єктам. Таким чином, усі елементи обробляються рекурсивно однаковим способом.

4. Яке призначення шаблону «Легковаговик»?

Шаблон «Flyweight» використовується для зменшення кількості об’єктів у програмі шляхом спільного використання однакових об’єктів. Він розділяє внутрішній стан (незмінна спільна інформація) та зовнішній стан (залежний від контексту використання).

5. Нарисуйте структуру шаблону «Легковаговик».



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

- Flyweight – Абстрактний клас або інтерфейс, який визначає метод operation(ExtrinsicState). Він задає інтерфейс для використання як внутрішнього, так і зовнішнього стану.
- ConcreteFlyweight – Реалізує інтерфейс Flyweight і містить внутрішній стан, спільний для багатьох контекстів. Ці об’єкти зберігаються та повторно використовуються фабрикою.
- UnsharedConcreteFlyweight – Клас, який також реалізує інтерфейс Flyweight, але не є спільним (унікальний для кожного клієнта). Використовується, коли деякі об’єкти мають індивідуальний стан, який не можна ділити.

- FlyweightFactory – Фабрика, що створює, зберігає та керує екземплярами ConcreteFlyweight. Якщо об’єкт уже існує в пулі — повертає його; інакше створює новий. Таким чином зменшується кількість дублюючих об’єктів.
- Client – Містить зовнішній стан, який передається у метод operation() при використанні Flyweight. Не створює об’єкти напряму — замість цього звертається до FlyweightFactory для отримання потрібного об’єкта.

Взаємодія:

- Client звертається до FlyweightFactory із запитом на отримання об’єкта Flyweight.
- FlyweightFactory перевіряє, чи існує вже такий об’єкт у пулі.
  - Якщо існує, то повертає існуючий.
  - Якщо ні, тоді створює новий ConcreteFlyweight або, за потреби, UnsharedConcreteFlyweight.
- Client передає зовнішній стан у метод operation() об’єкта Flyweight.
- Flyweight використовує свій внутрішній стан разом із зовнішнім станом клієнта для виконання операції.

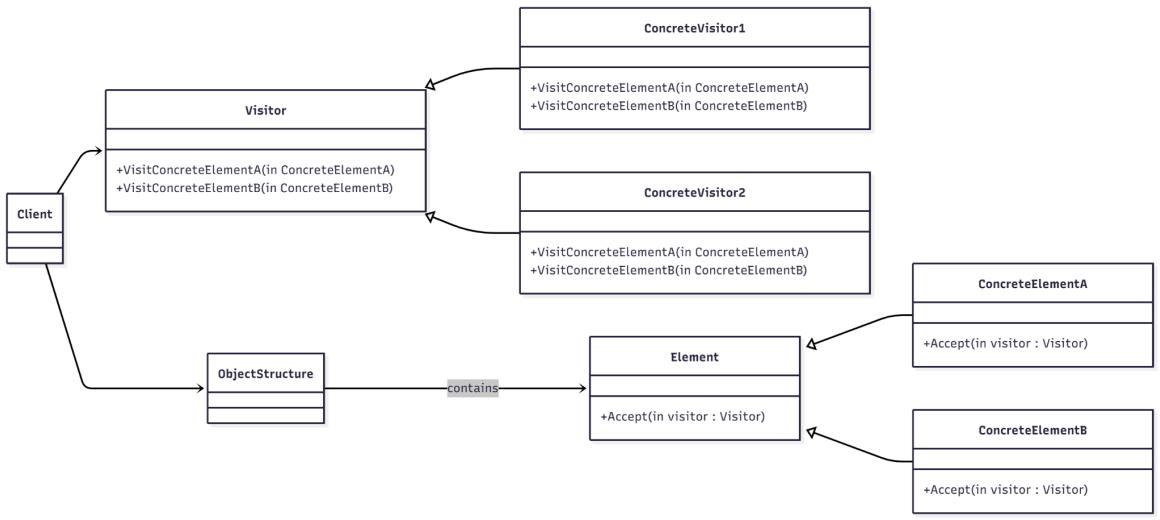
7. Яке призначення шаблону «Інтерпретатор»?

Шаблон використовується для подання граматики і створення інтерпретатора для мови, що дозволяє працювати з термінальними та нетермінальними символами у вигляді деревовидної структури. Кожен вузол дерева реалізує метод Interpret(context). Інтерпретатор рекурсивно обходить дерево, обчислюючи результат.

8. Яке призначення шаблону «Відвідувач»?

Шаблон «Visitor» дозволяє додавати нові операції до об’єктів, не змінюючи їх класів. Замість того, щоб додавати логіку всередину елементів, створюється окремий клас — відвідувач, який реалізує конкретні дії над різними типами елементів.

9. Нарисуйте структуру шаблону «Відвідувач».



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

- Visitor – Абстрактний клас або інтерфейс. Визначає методи для кожного типу елементів.
- ConcreteVisitor – Реалізує операції для кожного типу елементів.
- Element – Інтерфейс або абстрактний клас елемента. Містить метод: void Accept(Visitor visitor);
- ConcreteElementA / ConcreteElementB – Реалізують Accept(Visitor visitor), викликаючи відповідний метод відвідувача.
- ObjectStructure:
  - Містить колекцію елементів (List<Element>).
  - Забезпечує метод Accept(Visitor visitor), який проходить по всіх елементах і викликає для них Accept(visitor).
  - Може мати методи додавання/видалення елементів.
- Client – Створює відвідувача та ObjectStructure. Викликає structure.Accept(visitor) для застосування операцій до всіх елементів.

Взаємодія:

1. Client створює ConcreteVisitor та ObjectStructure.
2. ObjectStructure містить колекцію ConcreteElement.
3. Client викликає ObjectStructure.Accept(visitor).

4. ObjectStructure ітерує всі Element і викликає element.Accept(visitor) для кожного.

5. ConcreteElement передає себе відвідувачу, викликаючи відповідний метод.

6. ConcreteVisitor виконує операцію, знаючи конкретний тип елемента.