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

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 «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати їх в реалізації програмної системи

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

Короткі теоретичні відомості Компонувальник (Composite)

Шаблон "Компонувальник" дозволяє працювати з групами об'єктів так само, як з одним об'єктом, організовуючи їх у деревоподібну структуру. Використовується, коли потрібно представити ієрархію частина-ціле і надати єдиний інтерфейс для роботи з окремими об'єктами та їх групами. У Java цей шаблон реалізується шляхом створення базового інтерфейсу або абстрактного класу для всіх об'єктів у структурі та підкласів для представлення як "листків" (окремих об'єктів), так і "гілок" (груп об'єктів). Це дозволяє клієнтському коду працювати з усією структурою без перевірок її деталей.

Легковаговик (Flyweight)

Шаблон "Легковаговик" оптимізує використання пам'яті шляхом спільного використання об'єктів, які поділяють однаковий стан.

Використовується, коли система створює велику кількість однотипних об'єктів, що можуть розділяти загальний внутрішній стан. У Java цей шаблон зазвичай реалізується через фабрику, яка управляє пулом спільних об'єктів. Зовнішній стан об'єктів зберігається поза ними, щоб уникнути надмірного споживання пам'яті. Це дозволяє значно знизити накладні витрати в системах із великою кількістю об'єктів.

Інтерпретатор (Interpreter)

Шаблон "Інтерпретатор" визначає спосіб представлення граматики ієрархії мови і надає інтерпретатор для її виконання. Використовується для роботи з мовами, які мають чітко визначену граматику, наприклад, для створення калькуляторів або розбору виразів. У Java цей шаблон реалізується через створення класів, які представляють правила граматики,

та методу interpret(), що виконує операції. Це забезпечує зручність у роботі з граматикою, проте може бути неефективним для складних мов через зростання кількості класів.

Відвідувач (Visitor)

Шаблон "Відвідувач" дозволяє визначати нові операції для об'єктів без зміни їхніх класів. Використовується, коли необхідно виконати кілька різних операцій над об'єктами складної структури, але їх класи не можна змінювати. У Java цей шаблон реалізується шляхом створення інтерфейсу Visitor, який має методи для кожного типу елементів, та їх реалізацій для виконання конкретних операцій. Об'єкти структури реалізують метод accept(Visitor visitor), який викликає відповідний метод відвідувача. Це дозволяє додавати нові операції, зберігаючи класи об'єктів незмінними.

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 використано Патерн "Пристосуванець" (Flyweight), обраний як ключове архітектурне рішення для оптимізації використання оперативної пам'яті (Memory Overhead) під час роботи з великою кількістю дрібних об'єктів — символів тексту.

У процесі роботи над проєктом стало зрозуміло, що текстовий документ може містити тисячі або навіть мільйони символів. Хоча кожен символ є унікальним за своєю позицією в тексті (зовнішній стан), більшість із них мають спільні атрибути форматування (шрифт, розмір, колір).

Без використання патерну "Пристосуванець", створення повноцінного об'єкта для кожного символу з повним набором даних про його вигляд призвело б до:

1. Надмірних витрат пам'яті: Зберігання тисяч дублікатів одних і тих самих рядків (наприклад, "Arial", "12pt", "Black") у пам'яті.
2. Зниження продуктивності: Значне уповільнення роботи системи та збільшення навантаження на збірку сміття (Garbage Collection), що зробило б застосунок нездатним ефективно обробляти великі документи.

Патерн "Пристосуванець" елегантно вирішує цю проблему, розділяючи стан об'єкта на внутрішній (Intrinsic State), який є спільним і незмінним, та зовнішній (Extrinsic State), який є унікальним для кожного контексту.

У моїй реалізації, відображеній на UML-схемі, цей поділ реалізовано наступними класами:

1. CharacterStyle (Пристосуванець / Concrete Flyweight)

- Внутрішній стан (Intrinsic): Визначає спільні, незмінні атрибути форматування: fontName (назва шрифту), fontSize (розмір) та color (колір).
- Екземпляри цього класу є незмінними (immutable) і можуть безпечно використовуватися багатьма об'єктами одночасно, забезпечуючи економію пам'яті.

2. FlyweightFactory (Фабрика Пристосуванців)

- Управління пулом: Відповідає за створення та керування пулом об'єктів CharacterStyle через внутрішній словник (stylePool: Map<String, CharacterStyle>).
- Метод createKeyFont(): Коли клієнт (TextDocument) запитує певний стиль, фабрика перевіряє, чи вже існує об'єкт CharacterStyle з цими атрибутами. Якщо так — повертається існуючий екземпляр; якщо ні — створюється новий і додається до пулу.
- Це гарантує, що ідентичний стиль зберігається в пам'яті лише в одному екземплярі.

3. TextCharacter (Контекст / Client)

- Зовнішній стан (Extrinsic): Містить унікальні дані: сам символ (char), його координати на екрані (x, y, w, h), які залежать від контексту використання.
- Посилання: Містить посилання на спільний об'єкт CharacterStyle (style: CharacterStyle).
- Клас TextCharacter не зберігає дані про форматування безпосередньо, а делегує це пристосуванцю, комбінуючи унікальну позицію символу зі спільним стилем під час відмалювання (render()).

4. TextDocument (Клієнтський код)

- Клас, що оперує символами, зберігає список об'єктів TextCharacter (characters: List<TextCharacter>). Завдяки використанню фабрики, тисячі об'єктів TextCharacter посилаються лише на невелику кількість спільних об'єктів CharacterStyle, що забезпечує істотну економію ресурсів і підвищує ефективність роботи текстового редактора.

Діаграма класів

Рисунок 1 – Діаграма класів, яка представляє використання патерну Flyweight

Код класів:

```
public class TextDocument {
private final List<TextCharacter> characters;
private final FlyweightFactory styleFactory;
private final String name;

public TextDocument(String name) {
this.name = name;
this.characters = new ArrayList<>();
this.styleFactory = new FlyweightFactory();
}

public void addCharacter(char character, int x, int y,
String fontName, int fontSize, String color) {
CharacterStyle style = styleFactory.getStyle(fontName, fontSize, color);
TextCharacter ch = new TextCharacter(character, x, y, style);
characters.add(ch);
```

```
}


public void addText(String text, int startX, int startY,
String fontName, int fontSize, String color) {
int x = startX;
int y = startY;
for (char c : text.toCharArray()) {
if (c == '\n') {
y += fontSize + 5;
x = startX;
} else {
addCharacter(c, x, y, fontName, fontSize, color);
x += fontSize / 2;
}
}
}


public void render() {
System.out. println("\n=== Рендеринг документа: " + name + " ===");
for (TextCharacter character : characters) {
character. render();
}
}


public void printMemoryStatistics() {
System.out.println("\n=== Статистика документа: " + name + " ===");
System.out.println("Загальна кількість символів: " + characters.size());
styleFactory.printStatistics();
long withoutFlyweight = characters.size() * 100;
long withFlyweight = characters.size() * 20 +
styleFactory.getStyleCount() * 80;
long savedMemory = withoutFlyweight - withFlyweight;
System.out.println("\nОцінка використання пам'яті:");
System.out.println("Без Flyweight: ~" + withoutFlyweight + " байт");
System.out. println("З Flyweight: ~" + withFlyweight + " байт");
System.out.println("Заощаджено: ~" + savedMemory + " байт (" +
(savedMemory * 100 / withoutFlyweight) + "%)");
}

public int getCharacterCount() {
return characters.size();
}

public String getName() {
return name;
}
```

```
public List<TextCharacter> getCharacters() {
return characters;
}
public FlyweightFactory getStyleFactory() {
return styleFactory;
}

public void clear() {
characters.clear();
styleFactory.clear();
}
}
```

Код класу TextDocument

```
public class TextCharacter {
private final char character;
private final int x;
private final int y;
private final CharacterStyle style;

public TextCharacter(char character, int x, int y, CharacterStyle style)
{
this.character = character;
this.x = x;
this.y = y;
this.style = style;
}

public char getCharacter() {
return character;
}

public int getX() {
return x;
}

public int getY() {
return y;
}

public CharacterStyle getStyle() {
return style;
}


public void render() {
style.display(character, x, y);
}
```

```
@Override
public String toString() {
return "TextCharacter{char='" + character + "', position=(" + x + ", " +
y +
"), style=" + style + "}";
}
}
```

Код класу TextCharacter

```
public class CharacterStyle {
private final String fontName;
private final int fontSize;
private final String color;

public CharacterStyle(String fontName, int fontSize, String color) {
this.fontName = fontName;
this.fontSize = fontSize;
this.color = color;
}

public String getFontName() {
return fontName;
}

public int getFontSize() {
return fontSize;
}

public String getColor() {
return color;
}

public void display(char character, int x, int y) {
System.out.println("Символ '" + character + "' на позиції (" + x + ", " +
y +
") з шрифтом: " + fontName + ", розміром: " + fontSize +
", кольором: " + color);
}

@Override
public boolean equals(Object obj) {
if (this == obj) return true;
if (obj == null || getClass() != obj.getClass()) return false;
CharacterStyle that = (CharacterStyle) obj;
return fontSize == that.fontSize &&
fontName.equals(that.fontName) &&
color.equals(that.color);
}
```

```
@Override
public int hashCode() {
int result = fontName.hashCode();
result = 31 * result + fontSize;
result = 31 * result + color. hashCode();
return result;
}

@Override
public String toString() {
return "CharacterStyle{fontName='" + fontName + "', fontSize=" +
fontSize +
", color='" + color + "'}";
}
}
```

Код класу CharacterStyle

```
public class FlyweightFactory {
private final Map<String, CharacterStyle> stylePool;

public FlyweightFactory() {
this.stylePool = new HashMap<>();
}

public CharacterStyle getStyle(String fontName, int fontSize, String
color) {
String key = createKey(fontName, fontSize, color);
CharacterStyle style = stylePool.get(key);
if (style == null) {
style = new CharacterStyle(fontName, fontSize, color);
stylePool.put(key, style);
System.out.println("Створено новий стиль: " + style);
} else {
System.out.println("Використано існуючий стиль: " + style);
}
return style;
}


private String createKey(String fontName, int fontSize, String color) {
return fontName + "_" + fontSize + "_" + color;
}


public int getStyleCount() {
return stylePool.size();
}
```

```
public void printStatistics() {
System.out.println("=== Статистика FlyweightFactory ===");
System.out.println("Кількість унікальних стилів у пулі: " +
stylePool.size());
System.out.println("Стилі в пулі:");
for (CharacterStyle style : stylePool.values()) {
System.out. println(" - " + style);
}
}

public void clear() {
stylePool.clear();
}
}
```

Код класу FlyweightFactory

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з такими патернами, як «Composite», «Flyweight», «Interpreter» та «Visitor».

Особливу увагу я приділив реалізації шаблону "Пристосуванець" (Flyweight), детально описавши його основну логіку та функціональність у контексті мого проєкту текстового редактора. Цей патерн було обрано як архітектурне рішення для оптимізації використання оперативної пам'яті, щоб уникнути дублювання даних при зберіганні великої кількості символів, які мають спільні атрибути форматування (шрифт, колір, розмір).

Під час реалізації шаблону "Пристосуванець" я зрозумів, наскільки цей патерн елегантно вирішує проблему ресурсоємності системи при роботі з масивами даних. Він дозволяє чітко розділити стан об'єкта на внутрішній (спільний для багатьох об'єктів) та зовнішній (унікальний контекст), зберігаючи внутрішній стан у спеціальній фабриці. Це робить застосунок значно продуктивнішим та стійкішим до навантажень, оскільки замість створення тисяч "важких" об'єктів для кожного символу, система оперує легковаговими посиланнями на вже існуючі спільні екземпляри стилів.

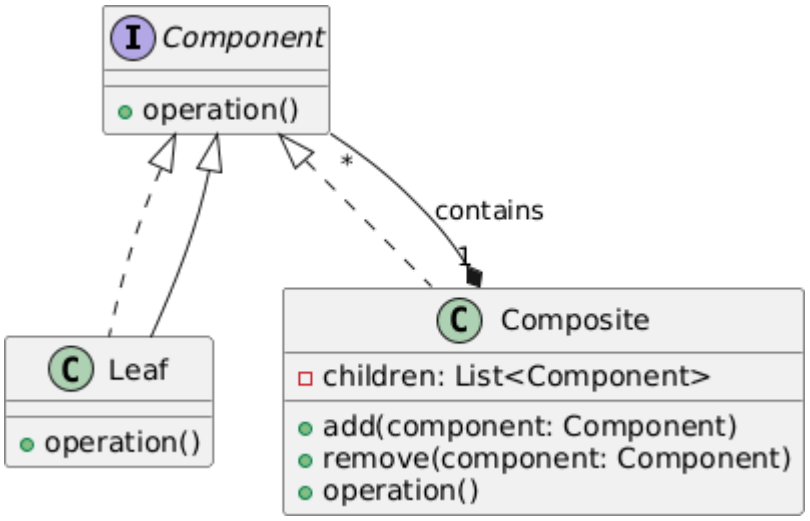
Контрольні запитання

1. Яке призначення шаблону «Композит»?

Шаблон «Композит» (Composite) призначений для того, щоб єдиним способом працювати як з окремими об’єктами, так і з групами об’єктів.

Він дозволяє будувати деревоподібні структури (наприклад, елементи інтерфейсу, файлову систему, шари зображення) і викликати операції однаково для листків та компонувальників. Тобто клієнт не думає, має він справу з одним елементом чи з цілим набором елементів — інтерфейс у них спільний.

2. Нарисуйте структуру шаблону «Композит».



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

Шаблон «Композит» складається з трьох основних класів: Компонент, Листок і Компоновальник.

Компонент – це базовий інтерфейс або абстрактний клас, який визначає спільні операції для всіх елементів структури.

Листок – це простий елемент без дочірніх об’єктів, який просто виконує свою частину роботи.

Компоновальник — це елемент, який може містити інші компоненти. Він зберігає список дітей і викликає їхні операції, коли виконується його власна операція.

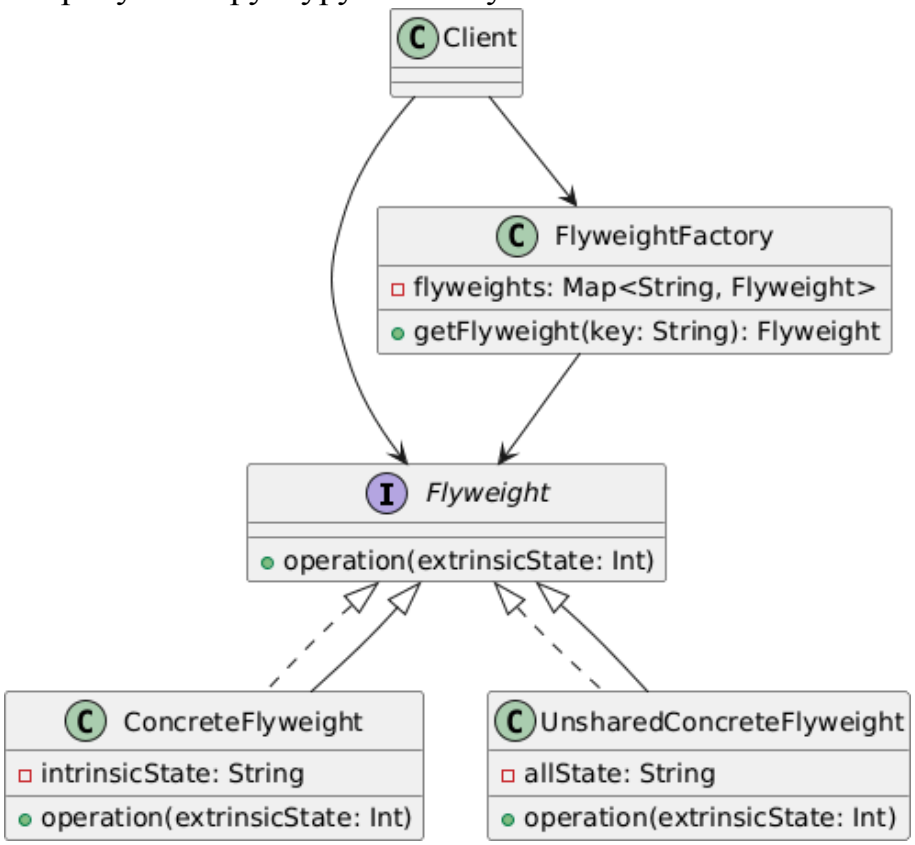
Взаємодія між ними така: клієнт звертається до всіх елементів через тип «Компонент». Листки обробляють запити напряму, а компоновальники передають їх своїм дочірнім елементам. Завдяки цьому клієнт не відрізняє, працює він з одним елементом чи з цілою групою.

4. Яке призначення шаблону «Легковаговик»?

Шаблон «Легковаговик» (Flyweight) призначений для зменшення використання пам’яті, коли у програмі потрібно створити дуже багато однотипних об’єктів.

Замість того щоб створювати тисячі однакових об’єктів, спільні дані виносять у один розділюваний об’єкт, а унікальні дані зберігають окремо. Це дозволяє значно економити ресурси та прискорювати роботу програми.

5. Нарисуйте структуру шаблону «Легковаговик».



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

У шаблон «Легковаговик» входять такі основні класи:

- Flyweight (Легковаговик) – об’єкт, який містить *внутрішній стан* (той, що спільний і не змінюється). Ці об’єкти можна багаторазово використовувати.
- ConcreteFlyweight – конкретна реалізація легковаговика з фіксованими внутрішніми даними.
- FlyweightFactory – створює та зберігає легковаговики. Якщо об’єкт з такими даними уже є, фабрика повертає існуючий, замість створення нового.
- Client – використовує легковаговики, передаючи їм *зовнішній стан* (унікальні дані, які не можна зберігати всередині легковаговика).

Клієнт звертається до фабрики, фабрика повертає вже існуючий легковаговик або створює новий. Легковаговик містить лише спільні дані, а унікальні дані клієнт передає йому під час використання. Це дозволяє значно економити пам’ять.

7. Яке призначення шаблону «Інтерпретатор»?

Шаблон «Інтерпретатор» (Interpreter) призначений для опису граматики мови та інтерпретації виразів цієї мови.

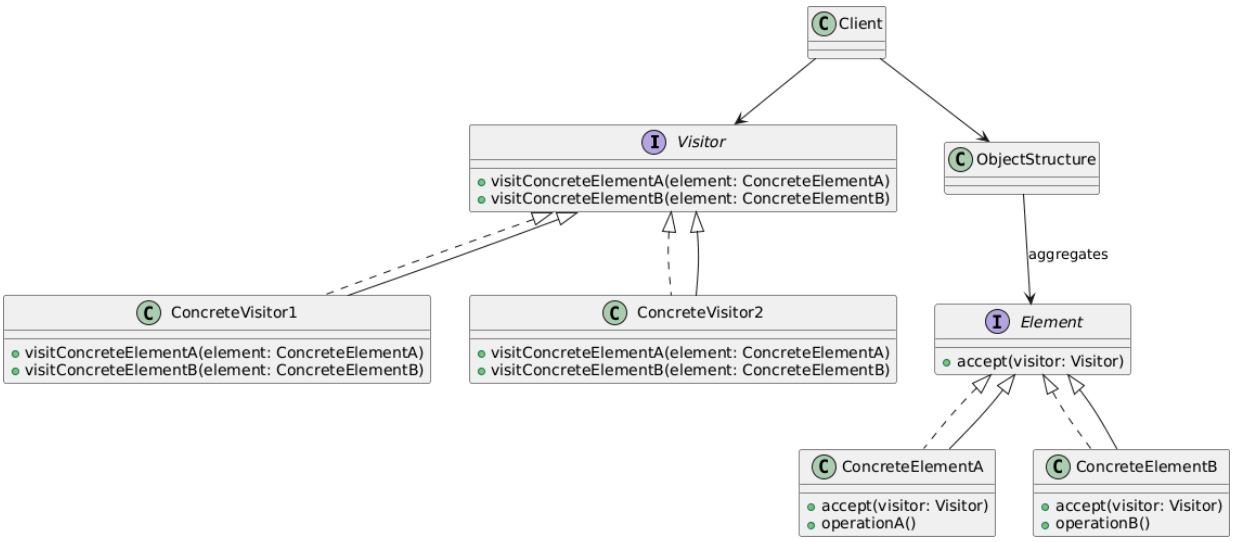
Він дозволяє створити невелику мову (наприклад, для формул, фільтрів, команд) та написати набір класів, які можуть читати та виконувати вирази цієї мови.

8. Яке призначення шаблону «Відвідувач»?

Шаблон «Відвідувач» (Visitor) призначений для того, щоб додавати нові операції до об’єктів складної структури, не змінюючи їхні класи.

Тобто він дозволяє винести логіку обробки об’єктів у окремий клас-відвідувач і “проходити” ним по елементах структури, виконуючи потрібні дії.

9. Нарисуйте структуру шаблону «Відвідувач».



10. Які класи входять в шаблон «Відвідувач», та яка між ними взаємодія? Шаблон «Відвідувач» складається з таких основних класів:

- Visitor (Відвідувач)

Оголошує методи для відвідування кожного типу елементів у структурі. Кожен метод відповідає конкретному класу елементів.

- ConcreteVisitor

Реалізує конкретні операції, які виконуються над елементами. Наприклад: підрахунок, вивід, перевірка, експорт тощо.

- Element (Елемент)

Базовий інтерфейс або абстрактний клас для всіх об’єктів структури. Містить метод accept(Visitor).

- ConcreteElement

Конкретні елементи, які реалізують метод accept, передаючи себе відвідувачу (visitor.visit(this)).

- ObjectStructure

Колекція або дерево елементів, які можна “обійти” за допомогою відвідувача.

Кожен елемент має метод accept, який приймає відвідувача. Відвідувач заходить у елемент, а елемент викликає відповідний метод відвідувача. Так відвідувач може виконувати нові операції над елементами, не змінюючи їхні класи.