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: «Шаблони «Composite»,
«Flyweight», «Interpreter», «Visitor»

Варіант №18

Виконав:
студент групи ІА-24
Гуменюк К. Е.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Contents

Тема.
Шаблони «Composite»,«Flyweight», «Interpreter», «Visitor»
Мета.
Метою даної лабораторної роботи є ознайомлення з шаблонами проєктування, зокрема з шаблоном " Interpreter", та їх практичне застосування при розробці програмного забезпечення.

Завдання.
1. Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми

Обрана тема.

..18 Shell (total commander) (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі - перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Короткі теоретичні відомості.

Даний шаблон використовується для подання граматики і інтерпретатора для вибраної мови (наприклад, скриптової). Граматика мови представлена термінальними і нетермінальними символами, кожен з яких інтерпретується в контексті використання. Клієнт передає контекст і сформовану пропозицію в використовувану мову в термінах абстрактного синтаксичного дерева (деревоподібна структура, яка однозначно визначає ієрархію виклику підвиразів), кожен вираз інтерпретується окремо з використанням контексту. У разі наявності дочірніх виразів, батьківський вираз інтерпретує спочатку дочірні (рекурсивно), а потім обчислює результат власної операції.

Шаблон зручно використовувати в разі невеликої граматики (інакше розростеться кількість використовуваних класів) і відносно простого контексту (без взаємозалежностей і т.п.).

Даний шаблон визначає базовий каркас інтерпретатора, який за допомогою рекурсії повертає результат обчислення пропозиції на основі результатів окремих елементів.

При використанні даного шаблону дуже легко реалізовується і розширюється граматика, а також додаються нові способи інтерпретації виразів.

Хід роботи.

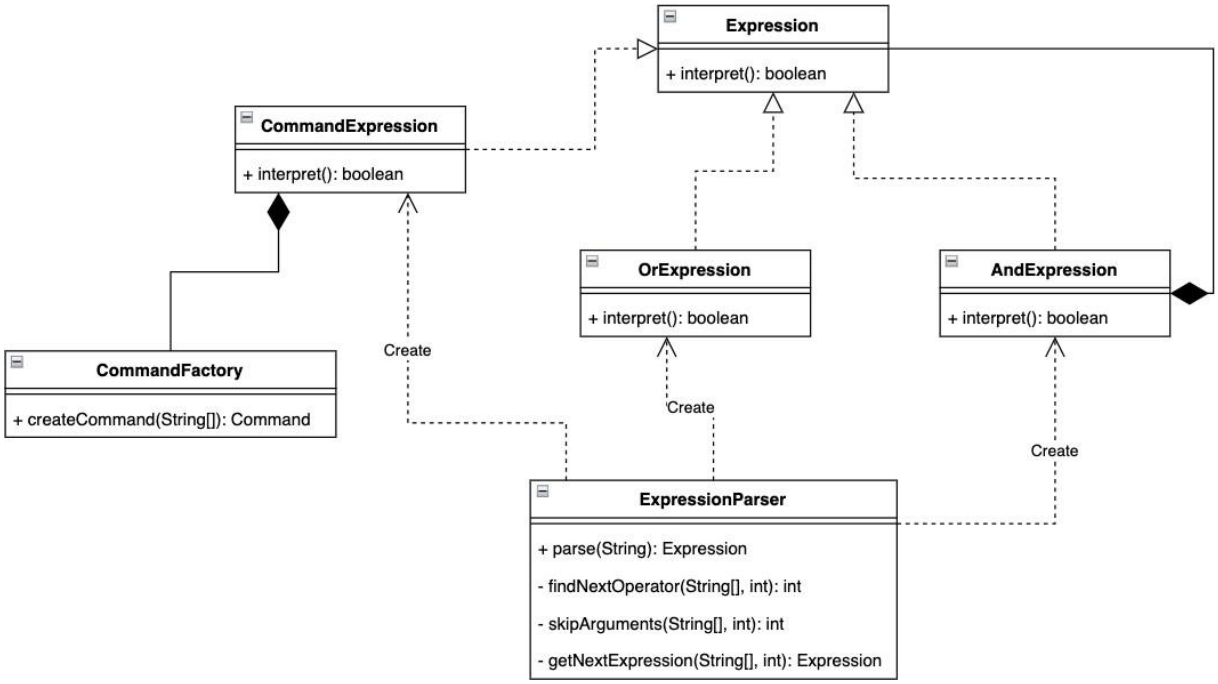

Рисунок 1.1 – UML діаграма шаблону Interpreter

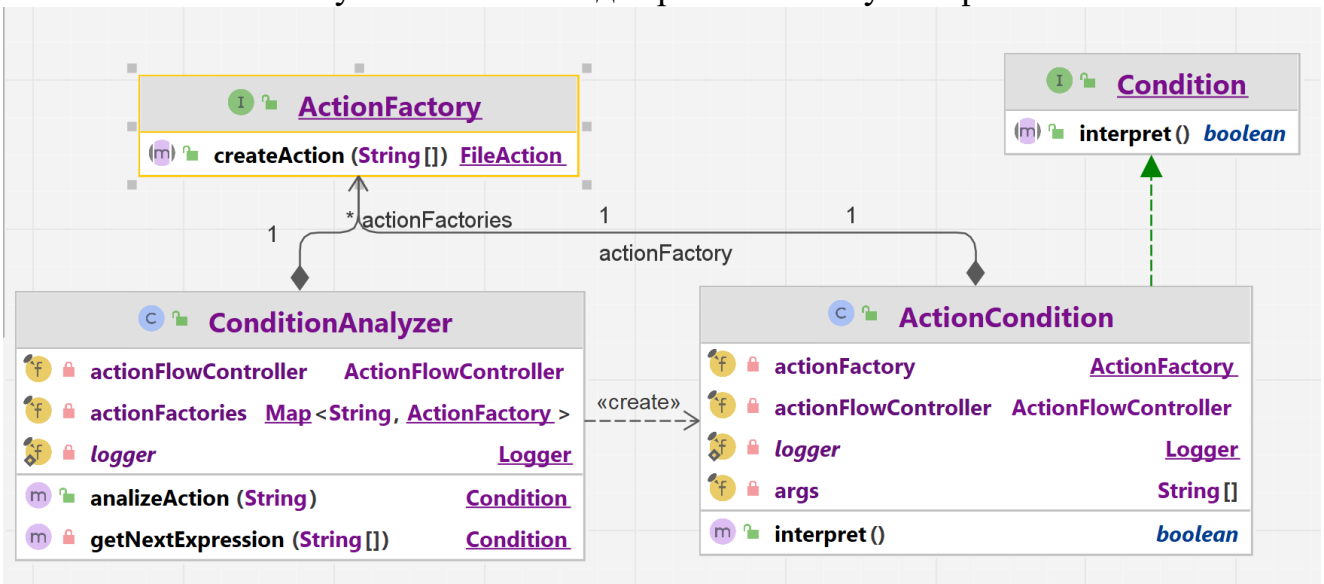

Рисунок 1.2 – Діаграма класів , згенерована IDE

Інтерфейс Condition:

- **Роль:** Базовий інтерфейс для всіх умов, визначає контракт на перевірку умови.
- **Методи:**
  - interpret() : boolean - Перевіряє умову та повертає результат (true або false).

2. Клас ActionCondition:

- **Роль:** Представляє умову, яка залежить від виконання певної дії, створеної за допомогою ActionFactory.
- **Реалізує:** Інтерфейс Condition.
- **Поля:**
  - args : String[] - Масив аргументів для створення дії.
  - actionFactory : ActionFactory - Фабрика дій, що створює потрібну дію.
  - logger: Logger - Для логування дій.
  - actionFlowController: ActionFlowController - Контролер для виконання дії.
- **Методи:**
  - interpret() : boolean - Створює дію за допомогою actionFactory, виконує її через actionFlowController та повертає true, якщо дія успішно виконана.

3. Клас ConditionAnalyzer:

- **Роль:** Аналізує введений рядок та повертає об'єкт Condition.
- **Поля:**
  - actionFactories : Map<String, ActionFactory> - Карта фабрик дій, де ключ - назва дії, а значення - відповідна фабрика.
  - logger: Logger - Для логування дій.
  - actionFlowController: ActionFlowController - Контролер для виконання дії.
- **Методи:**
  - analizeAction(String) : Condition - Аналізує введений рядок та повертає об'єкт Condition, який представляє собою дію.
  - getNextExpression(String[]) : Condition - Парсить наступний вираз з

масиву рядків.

4. Інтерфейс ActionFactory:

- **Роль:** Базовий інтерфейс для всіх фабрик, які створюють об'єкти дій з файлами (FileAction).
- **Методи:**
  - createAction(String[] args) : FileAction - Створює об'єкт дії.

Робота паттерну.

Виконаємо команду копіювання та видалення файлу, який копіювали

```
Input the action to perform or 'exit' to stop the program: copy test1.txt
2025-01-24T13:13:41.443+02:00  INFO 3056 --- [shell] [nio-8080-exec-3]
c.s.a.interpreter.ConditionAnalyzer      : Commencing analysis of command line input:
'copy test1.txt'
2025-01-24T13:13:41.443+02:00  INFO 3056 --- [shell] [nio-8080-exec-3]
c.s.a.interpreter.ConditionAnalyzer      : Action analysis completed with success
Not enough arguments. Require: 3
2025-01-24T13:13:41.444+02:00  WARN 3056 --- [shell] [nio-8080-exec-3]
c.s.a.factory.CopyFileActionFactory      : Action 'copy' wasn`t executed
```

Висновок.

Реалізація патерну "Інтерпретатор" у нашій системі значно розширила можливості умовного виконання дій:

- **Підтримка складних умов:** Патерн дозволив формалізувати умови та створити механізм для їхнього розбору, даючи змогу використовувати складніші вирази при виконанні дій.
- **Гнучкий розбір:** За допомогою класів Condition та ActionCondition ми створили систему, де кожен клас відповідає за свою частину інтерпретації умовних виразів, що робить їхнє розширення простішим.
- **Комбінування дій:** Користувачі можуть комбінувати дії, використовуючи логіку всередині ConditionAnalyzer, що робить систему потужнішою та гнучкішою.
- **Масштабованість та розширюваність:** Використання "Інтерпретатора" покращило архітектуру, зробивши систему більш масштабованою та адаптованою до майбутніх змін і вимог.

Таким чином, застосування "Інтерпретатора" дозволило нам створити гнучкий інструмент, який легко адаптується до різних сценаріїв і може бути розширений у майбутньому.

ДОДАТОК

```
public class ActionCondition implements Condition {
    private static final Logger logger =
LogUtils.getLogger(CopyFileActionFactory.class);

    private final String[] args;
    private final ActionFlowController actionFlowController;
    private final ActionFactory actionFactory;

    public ActionCondition(String[] args, ActionFlowController
actionFlowController, ActionFactory actionFactory) {
        this.args = args;
        this.actionFlowController = actionFlowController;
        this.actionFactory = actionFactory;
    }

    @Override
    public boolean interpret() {
        String userInput = args[0].toLowerCase();
        FileAction fileAction = actionFactory.createAction(args);
        if (fileAction != null) {
            logger.info("Executing action '{}'", userInput);
            actionFlowController.handleAction(fileAction);
            return true;
        } else {
            logger.warn("Action '{}' wasn`t executed", userInput);
            return false;
        }
    }
}
public class ConditionAnalyzer {
    private static final Logger logger =
LogUtils.getLogger(ConditionAnalyzer.class);

    private final ActionFlowController actionFlowController;
    private final Map<String, ActionFactory> actionFactories;

    public ConditionAnalyzer(ActionFlowController
actionFlowController, Map<String, ActionFactory> actionFactories) {
        this.actionFlowController = actionFlowController;
        this.actionFactories = actionFactories;
    }

    public Condition analizeAction(String commandLine) {
        logger.info("Commencing analysis of command line input:
'{}'", commandLine);

        try {
            String[] tokens = commandLine.trim().split("\\s+");

            if (tokens.length == 0) {
                throw new IllegalArgumentException("Empty command");
            }

            Condition result = getNextExpression(tokens);
```

```
            logger.info("Action analysis completed with success");
            return result;
        } catch (Exception e) {
            logger.error("Issue encountered while analyzing action:
{}", e.getMessage());
            throw e;
        }
    }

    private Condition getNextExpression(String[] tokens) {
        String userInput = tokens[0].toLowerCase();
        logger.debug("Recognized user input as action type: '{}'",
userInput);
        ActionFactory factory = actionFactories.get(userInput);

        if (factory == null) {
            logger.warn("Unrecognized action specified: '{}'",
userInput);
            throw new IllegalArgumentException("Unknown command: " +
userInput);
        }

        logger.debug("Generated command condition for action: '{}'",
userInput);
        return new ActionCondition(tokens, actionFlowController,
factory);
    }
}
```