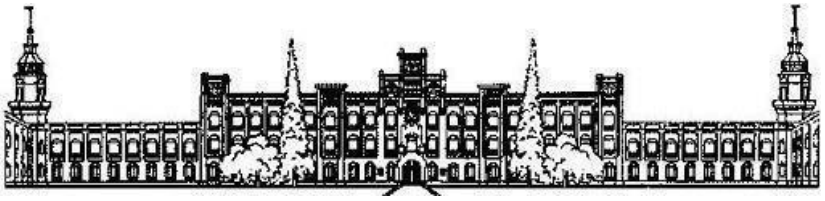


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №8

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування.»

Виконав:
студент групи ІА-32
Діденко Я.О

Перевірив:
Мягкий М.Ю

Київ 2025

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

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.

Теоретичні Відомості

Розглянуті патерни проєктування

1. Патерн «Компонувальник» (Composite)

Призначення: Об'єднує об'єкти в деревоподібну структуру для представлення ієрархії «частина-ціле».
Ідея: Патерн дозволяє клієнтському коду однаково працювати як з окремими об'єктами (листками дерева), так і з групами об'єктів (вузлами/композитами), оскільки всі вони реалізують спільний інтерфейс. Класичним прикладом є файлова система, де файл - це простий елемент, а папка - це композит, що може містити як файли, так і інші папки.

2. Патерн «Пристосуванець» (Flyweight)

Призначення: Дозволяє вмістити велику кількість об'єктів в обмежений обсяг оперативної пам'яті.
Ідея: Замість створення безлічі однакових екземплярів об'єктів створюється один спільний екземпляр, який використовується в різних контекстах. Для цього стан об'єкта розділяється на *внутрішній* (спільний для всіх, зберігається в самому пристосуванці) та *зовнішній* (унікальний для кожного контексту, передається ззовні). Наприклад, у текстовому редакторі кожна буква може бути пристосуванцем, де символ і шрифт - це внутрішній стан, а координати на сторінці - зовнішній.

3. Патерн «Інтерпретатор» (Interpreter)

Призначення: Визначає граматику простої мови та надає інтерпретатор для розбору та виконання виразів цією мовою.
Ідея: Для кожного правила граматики (термінального або нетермінального) створюється окремий клас. Речення мови представляється у вигляді синтаксичного дерева (AST), складеного з екземплярів цих класів. Патерн є ефективним, коли граматика мови відносно проста. **Саме цей патерн було реалізовано в даній лабораторній роботі** для обробки команд shell.

4. Патерн «Відвідувач» (Visitor)

Призначення: Дозволяє додавати нові операції до існуючих класів, не змінюючи їхню структуру.
Ідея: Нова функціональність виноситься в окремий клас - *відвідувач*. Об'єкти, над якими виконується операція, мають метод accept(visitor), що дозволяє відвідувачу "зайти" і виконати свою роботу. Це зручно, коли є складна структура об'єктів (наприклад, документ) і потрібно виконувати над нею різнорідні операції

Шаблон «Interpreter»

Призначення: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 взаємних залежностей і т.п.).
Даний шаблон визначає базовий каркас інтерпретатора, який за допомогою рекурсії повертає результат обчислення пропозиції на основі результатів окремих елементів.

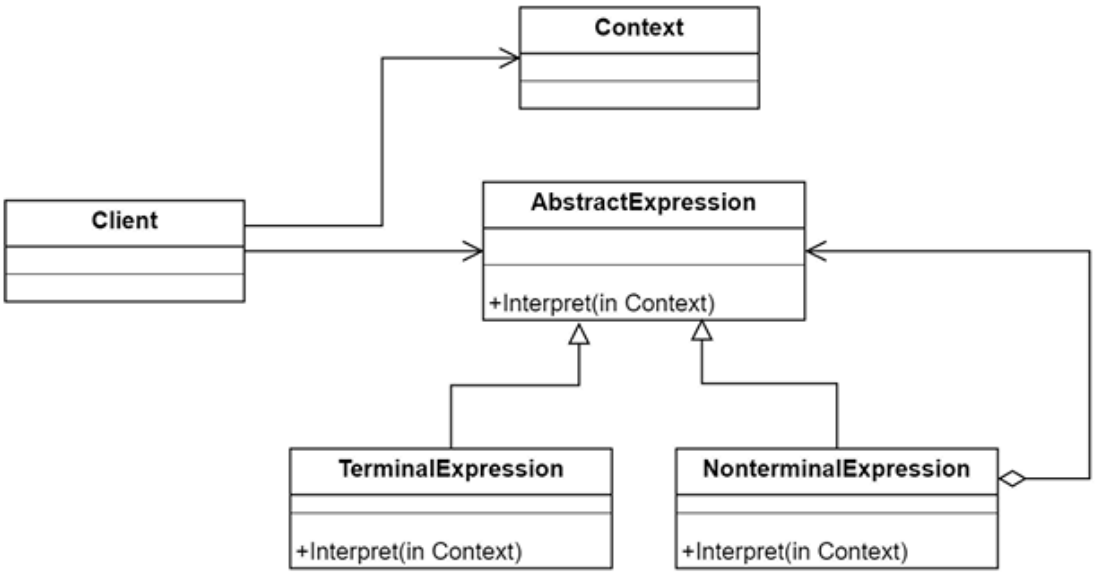


рис 1. Структура патерну Interpreter

Проблема: Стоїть задача пошуку рядків по зразку, як така, що часто зустрічається. Або ж загалом є якась задача, яка дуже часто змінюється.

Рішення: Може бути вирішена шляхом створення інтерпретатора, який визначає граматику мови. «Клієнт» будує речення у вигляді абстрактного синтаксичного дерева, у вузлах якого знаходяться об’єкти класів «НетермінальнийВираз» і «ТермінальнийВираз» (рекурсивне), потім «Клієнт» ініціалізує контекст і виражає операцію Розібрати(Контекст). На кожному вузлі типу «НетермінальнийВираз» визначається операція Розібрати для кожного підвиразу. Для класу «ТермінальнийВираз» операція Розібрати визначає базу рекурсії. «АбстрактнийВираз» визначає абстрактну операцію Розібрати, 94 загальну для всіх вузлів в абстрактному синтаксичному дереві. «Контекст» містить інформацію, глобальну по відношенню до інтерпретатора.

Переваги:
+ Граматику стає легко розширювати та змінювати, реалізації класів, що описують вузли абстрактного синтаксичного дерева схожі (легко кодуються).
+ Можна легко змінювати спосіб обчислення виразів.

Недолік:

- Супроводження граматики с великою кількістю правил є проблематичним.

Хід роботи

1. Загальний опис виконаної роботи

Основна мета полягала в розробці гнучкої системи для інтерпретації та виконання текстових команд користувача в контексті файлової оболонки «Shell». Замість жорсткого кодування логіки для кожної команди було застосовано патерн «Інтерпретатор», що дозволило представити кожну команду як окремий об'єкт, який можна динамічно створювати та виконувати.

Такий підхід дозволяє чітко відокремити логіку розбору команд від логіки їх виконання. Загальна інфраструктура для обробки вводу та управління станом (наприклад, поточною директорією) інкапсулюється в центральних компонентах, тоді як специфіка кожної операції реалізується в окремому, незалежному класі.

Програмна система складається з наступних компонентів:

- **Інтерфейс AbstractExpression**
  Це базовий контракт, що визначає єдиний метод interpret(). Усі конкретні команди реалізують цей інтерфейс, що дозволяє клієнтському коду працювати з ними уніфіковано, не знаючи деталей реалізації кожної окремої команди.
- **Конкретні класи-вирази (CopyCommandExpression, MoveCommandExpression та ін.)**
  Ці класи є реалізацією термінальних виразів граматики. Кожен клас відповідає за одну конкретну команду. Він зберігає параметри, необхідні для виконання (наприклад, вихідний та цільовий шляхи), та реалізує метод interpret(), в якому делегує фактичне виконання операції об'єкту контексту.
- **Клас CommandContext**
  Це центральний компонент, що виконує роль контексту. Він інкапсулює стан оболонки (поточну директорію) та всю складну логіку для взаємодії з файловою системою (копіювання,

переміщення, видалення тощо). Команди-вирази використовують цей об'єкт для виконання своїх завдань.

- **Клас CommandParser**
  Цей допоміжний клас відповідає за розбір рядка, введеного користувачем. Він аналізує команду та її аргументи і на їх основі створює та повертає відповідний об'єкт-вираз (AbstractExpression). Це відокремлює логіку парсингу від решти системи.
- **Клієнт (ShellInterpreter)**
  Кінцевий клієнт, який організовує головний цикл програми "читання-виконання". Він зчитує ввід користувача, передає його парсеру для отримання об'єкта-команди, а потім викликає метод interpret() цього об'єкта, передаючи йому контекст. Клієнту не потрібно знати, яка саме команда виконується; він просто працює з абстрактним інтерфейсом AbstractExpression.

2. Опис класів програмної системи.

Клас CopyCommandExpression.java

Опис:
Клас CopyCommandExpression це конкретна реалізація інтерфейсу AbstractExpression, що представляє собою **Термінальний вираз** у структурі патерну «Інтерпретатор». Його єдина відповідальність - інкапсулювати дані для команди копіювання (copy) та ініціювати її виконання через об'єкт контексту.

Характеристики:

- Є повноцінним класом, який можна інстанціювати.
- Реалізує інтерфейс AbstractExpression, що вимагає наявності методу interpret().
- Містить private final поля src (джерело) та dst (призначення), які ініціалізуються через конструктор під час створення об'єкта парсером.
- Метод interpret() не містить власної бізнес-логіки. Він лише викликає відповідний метод copy(src, dst) у переданого об'єкта CommandContext, делегуючи йому всю роботу по взаємодії з файловою системою.

Класи MoveCommandExpression, DeleteCommandExpression, MkdirCommandExpression, ChangeDirExpression та ListCommandExpression побудовані за аналогічним принципом. Кожен з них відповідає за свою команду, зберігає необхідні для неї параметри та делегує виконання відповідному методу в CommandContext.

```
1   public class CopyCommandExpression implements AbstractExpression {  1 usage
2       private final String src;  2 usages
3       private final String dst;  2 usages
4
5       public CopyCommandExpression(String src, String dst) {  1 usage
6           this.src = src;
7           this.dst = dst;
8       }
9
10      @Override  1 usage
11      public void interpret(CommandContext context) {
12          context.copy(src, dst);
13      }
14  }
```

рис 2.1 - код класу CopyCommandExpression.java

Інтерфейс AbstractExpression.java

Опис:
Інтерфейс AbstractExpression є ядром патерну «Інтерпретатор». Він визначає загальний контракт для всіх об'єктів-виразів, тобто команд у системі. Це дозволяє клієнтському коду працювати з різними командами уніфіковано, не залежачи від їх конкретної реалізації.

Характеристики:

- Є інтерфейсом, тому не може бути інстанційований напряму.
- Оголошує єдиний метод interpret(CommandContext context). Цей метод є точкою входу для виконання будь-якої команди.
- Кожен конкретний клас команди в системі зобов'язаний реалізовувати цей інтерфейс.

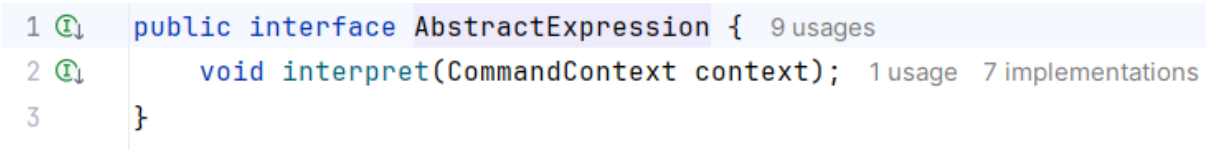


рис 2.2 - код інтерфейсу AbstractExpression.java

Клас CommandContext.java

Опис:
Клас CommandContext виконує роль **Контексту** в патерні «Інтерпретатор». Він є центральним сховищем стану оболонки (зокрема, поточної робочої директорії) та інкапсулює всю складну логіку виконання операцій над файловою системою. Цей клас відокремлює логіку виконання від представлення команд.

Характеристики:

- Зберігає стан - поле currentDirectory типу Path.
- Надає публічні методи для виконання всіх підтримуваних операцій: listDirectory(), changeDirectory(), copy(), move() тощо. Саме в цих методах реалізована вся логіка роботи з файлами та директоріями.
- Приймається як аргумент у метод interpret() кожного об'єкта-виразу, надаючи йому доступ до поточного стану та необхідних функцій.

```java
public class CommandContext {  10 usages
    private Path currentDirectory;  6 usages

    public CommandContext(Path startDirectory) { this.currentDirectory = startDirectory.toAbsolutePath().normal

    public Path getCurrentDirectory() { return currentDirectory; }

    public void setCurrentDirectory(Path currentDirectory) {  1 usage
        this.currentDirectory = currentDirectory.toAbsolutePath().normalize();
    }

    public Path resolvePath(String arg) {  7 usages
        Path p = Paths.get(arg);
        if (!p.isAbsolute()) {
            p = currentDirectory.resolve(p);
        }
        return p.normalize();
    }

    public void listDirectory() {  1 usage
        try (Stream<Path> stream = Files.list(currentDirectory)) {
            stream.forEach( Path p -> System.out.println(p.getFileName()));
        } catch (IOException e) {
            System.out.println("Помилка під час читання директорії: " + e.getMessage());
        }
    }

    public void changeDirectory(String pathArg) {  1 usage
        Path target = resolvePath(pathArg);
        if (Files.isDirectory(target)) {
            setCurrentDirectory(target);
            System.out.println("Поточна директорія: " + currentDirectory);
        } else {
            System.out.println("Директорію не знайдено: " + target);
        }
    }

    public void copy(String srcArg, String dstArg) {  1 usage
        Path src = resolvePath(srcArg);
        Path dst = resolvePath(dstArg);

        try {
            if (!Files.exists(src)) {
                System.out.println("Помилка: джерело не існує: " + src);
                return;
            }

            if (Files.isDirectory(src)) {
                copyDirectory(src, dst);
            } else {
                if (Files.exists(dst) && Files.isDirectory(dst)) {
                    dst = dst.resolve(src.getFileName());
                }
                Files.copy(src, dst, StandardCopyOption.REPLACE_EXISTING);
            }

            System.out.println("Скопійовано: " + src + " -> " + dst);

        } catch (IOException e) {
            System.out.println("Помилка копіювання: " + e.getMessage());
        } catch (RuntimeException e) {
```

```
69          } catch (RuntimeException e) {
70              System.out.println("Помилка під час рекурсивного копіювання: " + e.getMessage());
71          }
72      }
73
74      private void copyDirectory(Path src, Path dst) throws IOException {  1 usage
75          if (Files.notExists(dst)) {
76              Files.createDirectories(dst);
77          }
78
79          try (Stream<Path> stream = Files.walk(src)) {
80              stream.forEach( Path sourcePath -> {
81                  try {
82                      Path relative = src.relativize(sourcePath);
83                      Path target = dst.resolve(relative);
84
85                      if (Files.isDirectory(sourcePath)) {
86                          if (Files.notExists(target)) {
87                              Files.createDirectories(target);
88                          }
89                      } else {
90                          Files.copy(sourcePath, target, StandardCopyOption.REPLACE_EXISTING);
91                      }
92                  } catch (IOException e) {
93                      throw new RuntimeException(e);
94                  }
95              });
96          }
97      }
```

```
99      public void move(String srcArg, String dstArg) {  1 usage
100         Path src = resolvePath(srcArg);
101         Path dst = resolvePath(dstArg);
102
103         try {
104             if (!Files.exists(src)) {
105                 System.out.println("Помилка: джерело не існує: " + src);
106                 return;
107             }
108
109             if (Files.exists(dst) && Files.isDirectory(dst)) {
110                 dst = dst.resolve(src.getFileName());
111             }
112
113             Files.move(src, dst, StandardCopyOption.REPLACE_EXISTING);
114             System.out.println("Переміщено: " + src + " -> " + dst);
115
116         } catch (IOException e) {
117             System.out.println("Помилка переміщення: " + e.getMessage());
118         }
119     }
120
121     public void delete(String pathArg) {  1 usage
122         Path target = resolvePath(pathArg);
123         try {
124             Files.delete(target);
125             System.out.println("Видалено: " + target);
126         } catch (IOException e) {
127             System.out.println("Помилка видалення: " + e.getMessage());
128         }
```

```
131     public void createDirectory(String pathArg) {  1 usage
132         Path dir = resolvePath(pathArg);
133         try {
134             if (Files.exists(dir)) {
135                 System.out.println("Помилка: об'єкт із такою назвою вже існує: " + dir);
136                 return;
137             }
138             Files.createDirectory(dir);
139             System.out.println("Створено директорію: " + dir);
140         } catch (IOException e) {
141             System.out.println("Помилка створення директорії: " + e.getMessage());
142         }
143     }
144 }
```

рис 2.3 - код класу CommandContext.java

Клас CommandParser.java

Опис:
Клас CommandParser є допоміжним компонентом, що виконує роль фабрики для створення об'єктів-виразів. Він відповідає за розбір текстового рядка, введеного користувачем, і перетворення його на відповідний об'єкт команди (AbstractExpression).

Характеристики:

- Містить статичний метод parse(String line), що робить його утилітарним класом без необхідності створення екземпляра.
- Аналізує перший токен рядка для визначення назви команди.
- На основі команди створює екземпляр відповідного класу (CopyCommandExpression, MoveCommandExpression тощо), передаючи в конструктор необхідні аргументи (шляхи до файлів), витягнуті з рядка.
- Обробляє випадки некоректного вводу, наприклад, коли не вистачає аргументів для команди.

```java
public class CommandParser {  1 usage

    public static AbstractExpression parse(String line) {  1 usage
        if (line == null || line.trim().isEmpty()) {
            return  CommandContext ctx -> {};
        }

        String trimmed = line.trim();
        String[] parts = trimmed.split( regex: "\\s+");

        String cmd = parts[0].toLowerCase();

        switch (cmd) {
            case "ls":
                return new ListCommandExpression();

            case "cd":
                if (parts.length >= 2) {
                    return new ChangeDirExpression(parts[1]);
                } else {
                    System.out.println("Використання: cd <шлях>");
                    return  CommandContext ctx -> {};
                }

            case "copy":
                if (parts.length >= 3) {
                    return new CopyCommandExpression(parts[1], parts[2]);
                } else {
                    System.out.println("Використання: copy <джерело> <призначення>");
                    return  CommandContext ctx -> {};
                }
```

```
33              case "move":
34                  if (parts.length >= 3) {
35                      return new MoveCommandExpression(parts[1], parts[2]);
36                  } else {
37                      System.out.println("Використання: move <джерело> <призначення>");
38                      return CommandContext ctx -> {};
39                  }
40
41              case "rm":
42                  if (parts.length >= 2) {
43                      return new DeleteCommandExpression(parts[1]);
44                  } else {
45                      System.out.println("Використання: rm <шлях>");
46                      return CommandContext ctx -> {};
47                  }
48
49              case "mkdir":
50                  if (parts.length >= 2) {
51                      return new MkdirCommandExpression(parts[1]);
52                  } else {
53                      System.out.println("Використання: mkdir <назва_каталогу>");
54                      return CommandContext ctx -> {};
55                  }
56
57              case "exit":
58              case "quit":
59                  return CommandContext ctx -> {};
60
61              default:
62                  return new UnknownCommandExpression(line);
```

рис 2.4 - код класу CommandParser.java

Клас ShellInterpreter.java

Опис:
Клас ShellInterpreter виступає в ролі **Клієнта** в патерні «Інтерпретатор». Це головний клас програми, що містить точку входу main(). Він організовує нескінченний цикл "читання-виконання" (REPL), керуючи взаємодією між користувачем, парсером та об'єктами-командами.

Характеристики:

- Створює та управляє єдиним екземпляром CommandContext протягом усього сеансу роботи.
- У головному циклі зчитує команди з консолі.

- Використовує CommandParser для перетворення введеного рядка на об'єкт AbstractExpression.
- Викликає метод interpret() отриманого об'єкта, передаючи йому єдиний екземпляр контексту для виконання.
- Йому не потрібно знати, яка саме команда виконується; він працює з усіма командами через єдиний абстрактний інтерфейс.

```
1   import ...
3
4   public class ShellInterpreter {
5
6       public static void main(String[] args) {
7           CommandContext context = new CommandContext(Paths.get( first: "."));
8           System.out.println("Підтримувані команди: ls, cd, copy, move, rm, mkdir, exit");
9           System.out.println("Поточна директорія: " + context.getCurrentDirectory());
10
11          try (Scanner scanner = new Scanner(System.in)) {
12              while (true) {
13                  System.out.print(context.getCurrentDirectory() + " > ");
14                  String line = scanner.nextLine();
15
16                  if (line == null) break;
17
18                  String trimmed = line.trim();
19                  if (trimmed.equalsIgnoreCase( anotherString: "exit") || trimmed.equalsIgnoreCase( anotherString: "quit")
20                      System.out.println("Вихід.");
21                      break;
22                  }
23
24                  AbstractExpression expression = CommandParser.parse(trimmed);
25                  expression.interpret(context);
26              }
27          }
28      }
29  }
```

рис 2.5 - код класу ShellInterpreter.java

Результати виконання коду

```
Підтримувані команди: ls, cd, copy, move, rm, mkdir, exit
Поточна директорія: C:\Users\arosl\IdeaProjects\TRPZ
C:\Users\arosl\IdeaProjects\TRPZ > cd C:\
Поточна директорія: C:\
C:\ > mkdir NewFolder
Створено директорію: C:\NewFolder
C:\ > cd NewFolder
Поточна директорія: C:\NewFolder
C:\NewFolder > mkdir NewSubFolder
Створено директорію: C:\NewFolder\NewSubFolder
C:\NewFolder > mkdir NewSubFolder2
Створено директорію: C:\NewFolder\NewSubFolder2
C:\NewFolder > cd NewSubFolder
Поточна директорія: C:\NewFolder\NewSubFolder
C:\NewFolder\NewSubFolder > mkdir NewSubSubFolder
Створено директорію: C:\NewFolder\NewSubFolder\NewSubSubFolder
C:\NewFolder\NewSubFolder > cd ..
Поточна директорія: C:\NewFolder
C:\NewFolder > ls
NewSubFolder
NewSubFolder2
C:\NewFolder > copy NewSubFolder NewSubFolder2
Скопійовано: C:\NewFolder\NewSubFolder -> C:\NewFolder\NewSubFolder2
C:\NewFolder > cd NewSubFolder2
Поточна директорія: C:\NewFolder\NewSubFolder2
C:\NewFolder\NewSubFolder2 > ls
NewSubSubFolder


C:\NewFolder > move NewSubFolder NewSubFolder2
Переміщено: C:\NewFolder\NewSubFolder -> C:\NewFolder\NewSubFolder2\NewSubFolder
C:\NewFolder > ls
NewSubFolder2
C:\NewFolder > cd NewSubFolder2
Поточна директорія: C:\NewFolder\NewSubFolder2
C:\NewFolder\NewSubFolder2 > ls
NewSubFolder
NewSubSubFolder
```

```
C:\NewFolder > cd C:\
Поточна директорія: C:\
C:\ > rm NewFolder
Видалено: C:\NewFolder
C:\ >
```

рис 2.6 - результати виконання коду

Надані скріншоти демонструють повний цикл тестування розробленої консольної оболонки, від створення файлової структури до її повного видалення. Користувач послідовно виконує ключові команди, що дозволяє перевірити коректність роботи кожного компонента системи, реалізованої за допомогою патерну «Інтерпретатор».

Етап 1: Підготовка тестового середовища

Спочатку користувач готує робочу область для тестування.

1. **cd C:**: Здійснюється перехід у корінь диска C: для створення чистого та передбачуваного середовища.
2. **mkdir NewFolder**: Створюється головна тестова директорія.
3. **cd NewFolder**: Виконується перехід у щойно створену директорію.
4. **mkdir NewSubFolder, mkdir NewSubFolder2, mkdir NewSubSubFolder**: Послідовно створюється вкладена структура директорій. Створення папки NewSubSubFolder всередині NewSubFolder є важливим кроком, оскільки це дозволяє перевірити, як команди copy та rm працюють з не порожніми директоріями (рекурсивно).

На цьому етапі перевіряється коректна робота команд mkdir та cd. Програма правильно створює директорії та змінює поточне робоче положення, що підтверджується шляхом, який відображається у командному рядку.

Етап 2: Тестування команди рекурсивного копіювання (copy)

Після підготовки структури починається тестування логіки копіювання.

1. **copy NewSubFolder NewSubFolder2**: Виконується команда копіювання директорії NewSubFolder (яка містить NewSubSubFolder) у директорію NewSubFolder2.
2. **Перевірка**: Користувач переходить у cd NewSubFolder2 і за допомогою команди ls переглядає її вміст. Результат показує, що директорія NewSubFolder разом з усім її вкладеним вмістом була успішно скопійована. Це підтверджує, що логіка копіювання реалізована правильно і підтримує рекурсивну роботу.

Етап 3: Тестування команди переміщення (move)

Наступним кроком є перевірка операції переміщення.

1. **move NewSubFolder NewSubFolder2**: Виконується команда переміщення оригінальної директорії NewSubFolder у NewSubFolder2.
2. **Результат**: Повідомлення "Переміщено..." показує, що оскільки NewSubFolder2 є існуючою директорією, переміщення відбулося **всередину** неї.
3. **Перевірка**: Команда ls у поточній директорії C:\NewFolder показує, що NewSubFolder зникла зі свого початкового місця. Після переходу в NewSubFolder2 команда ls підтверджує, що NewSubFolder тепер знаходиться там. Це демонструє коректну логіку роботи команди move.

Етап 4: Тестування команди видалення (rm)

1. **cd C:**: Користувач виходить із тестової структури, оскільки видалення поточної робочої директорії зазвичай блокується.
2. **rm NewFolder**: Виконується команда видалення головної директорії NewFolder.
3. **Результат**: Повідомлення "Видалено: C:\NewFolder" підтверджує, що операція пройшла успішно.

3. Діаграма класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами була побудована UML-діаграма класів. Вона наочно демонструє структуру, що відповідає шаблону проектування «Interpreter».

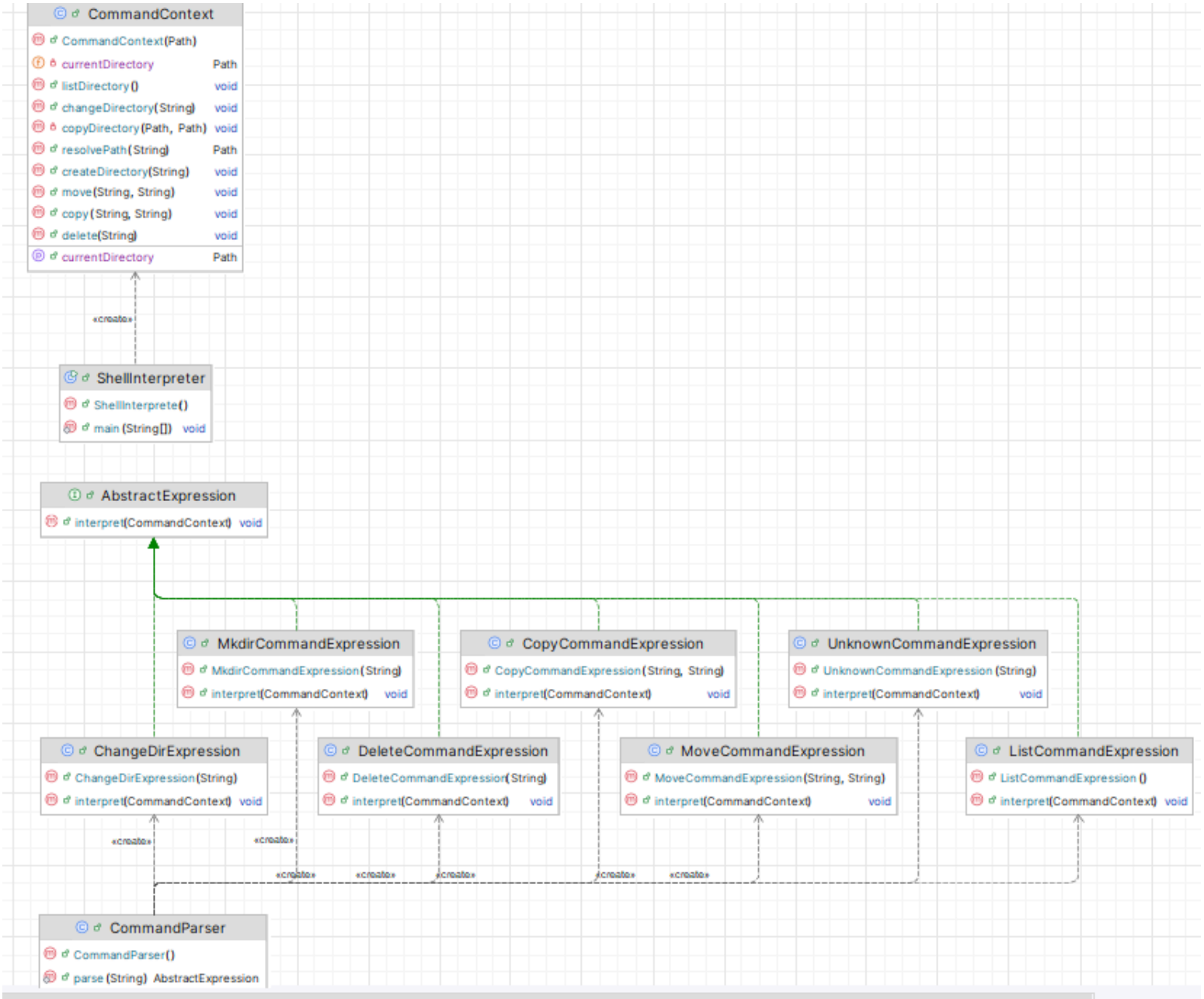


рис 3.1 - діаграма класів (повна)

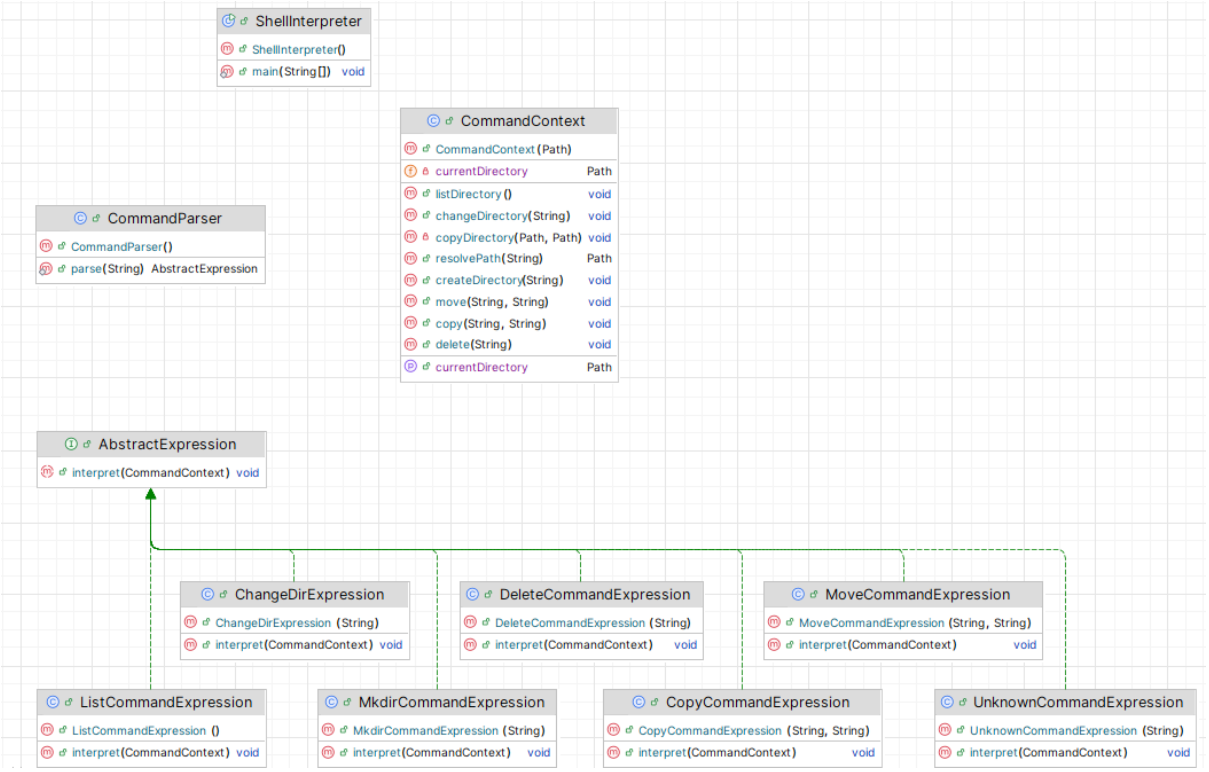


рис 3.2 - діаграма класів (спрощена)

1. Ключові компоненти патерну:

- **AbstractExpression (Інтерфейс):** Це центральний елемент патерну. Він визначає єдиний метод interpret(CommandContext), який є загальним контрактом для всіх команд. Усі конкретні класи команд (зелені лінії з трикутником) реалізують цей інтерфейс, що дозволяє клієнту працювати з ними уніфіковано.
- **Конкретні класи-вирази (MkdirCommandExpression, CopyCommandExpression, ChangeDirExpression і т.д.):** Це реалізації **Термінальних виразів**. Кожен клас відповідає за одну конкретну команду. Вони приймають необхідні параметри через конструктор (наприклад, шляхи до файлів) і реалізують метод interpret(). Важливо, що вони не виконують логіку самі, а делегують її об'єкту CommandContext.
- **CommandContext (Клас):** Це **Контекст** патерну. Цей клас інкапсулює:
    - **Стан:** Поточну робочу директорію (currentDirectory).

    - **Логіку:** Усі методи для реальної роботи з файловою системою (listDirectory, copy, move, delete тощо).
      Він передається в метод interpret() кожної команди, надаючи їй доступ до поточного стану та необхідних функцій.

2. Допоміжні компоненти системи:

- **CommandParser (Клас):** Цей клас відіграє роль **фабрики** або **парсера**. Його статичний метод parse() приймає рядок, введений користувачем, аналізує його і створює екземпляр відповідного конкретного класу-виразу. Як показано пунктирними стрілками зі стереотипом <<create>>, CommandParser є єдиним компонентом, відповідальним за створення об'єктів команд.
- **ShellInterpreter (Клас):** Це **Клієнт**, тобто головний клас програми. Він містить метод main(), який організовує життєвий цикл застосунку:
    1. Створює екземпляр CommandContext.
    2. У циклі зчитує ввід користувача.
    3. Передає ввід до CommandParser для отримання об'єкта AbstractExpression.
    4. Викликає метод interpret() отриманого об'єкта, передаючи йому контекст.

Загальний потік роботи:

Коли користувач вводить команду (наприклад, copy file.txt /backup), ShellInterpreter зчитує цей рядок. Потім CommandParser розбирає його, розуміє, що це команда copy, і створює об'єкт new CopyCommandExpression("file.txt", "/backup"). ShellInterpreter отримує цей об'єкт і викликає його метод interpret(context). У свою чергу, CopyCommandExpression викликає context.copy("file.txt", "/backup"), де і відбувається вся реальна робота з файлами.

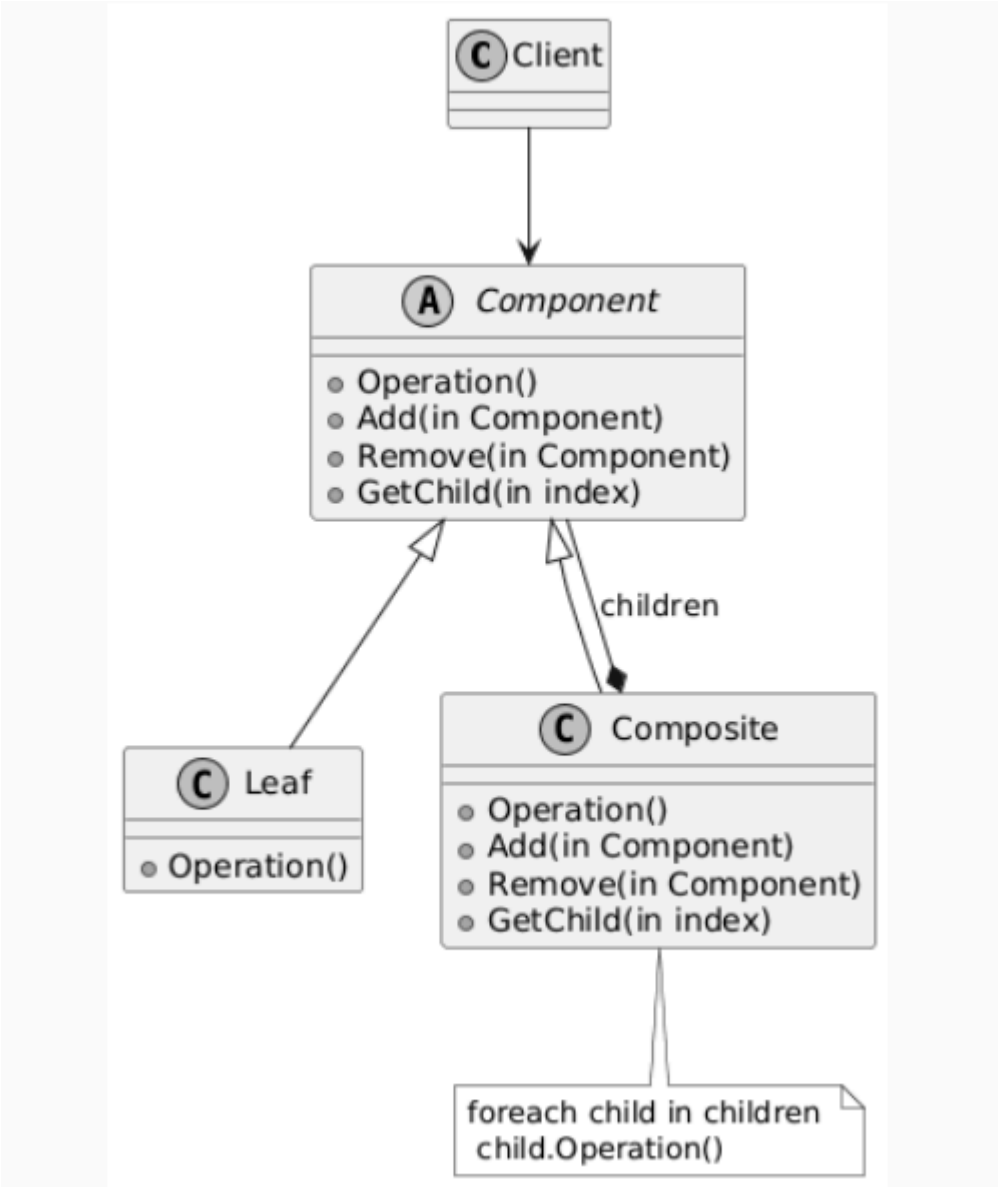
4. Висновки

Висновки: В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано патерн проектування «Інтерпретатор» (Interpreter) для вирішення задачі розробки оболонки командного рядка. Була створена чітка архітектура, що складається з інтерфейсу AbstractExpression, набору конкретних класів-виразів (таких як CopyCommandExpression, MoveCommandExpression), центрального класу CommandContext та допоміжного парсера CommandParser. Особливу увагу було приділено розподілу відповідальності: впровадження класу CommandContext у ролі Контексту дозволило централізувати всю складну логіку взаємодії з файловою системою в одному місці, тоді як відповідальність за представлення кожної команди та її параметрів була успішно делегована простим класам-виразам. Практична реалізація продемонструвала ключові переваги патерну: чітке розділення відповідальності між логікою розбору, представлення та виконання команд; високу гнучкість системи, що дозволяє легко додавати нові типи команд без зміни існуючого коду, що повністю відповідає принципу відкритості/закритості; та простоту супроводу коду завдяки єдиній відповідальності кожного класу.

Контрольні запитання

1. Яке призначення шаблону «Композит»?
Шаблон «Композит» (Composite) використовується для об'єднання об'єктів у деревоподібну структуру для представлення ієрархій «частина-ціле». Головне призначення - дозволити клієнтському коду працювати з окремими об'єктами (листками) та групами об'єктів (композитами) однаковим чином через спільний інтерфейс.

2. Нарисуйте структуру шаблону «Композит».



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

- **Component (Компонент):** Це спільний інтерфейс або абстрактний клас для всіх об'єктів в композиції (як для листків, так і для композитів). Він оголошує загальні операції.
- **Leaf (Листок):** Клас, що представляє кінцевий об'єкт в ієрархії (не має дочірніх елементів). Він реалізує операції, визначені в Component.
- **Composite (Композит/Вузол):** Клас, що представляє складний об'єкт, який може мати дочірні елементи (типу Component). Він реалізує загальні операції, зазвичай делегуючи їх виконання своїм

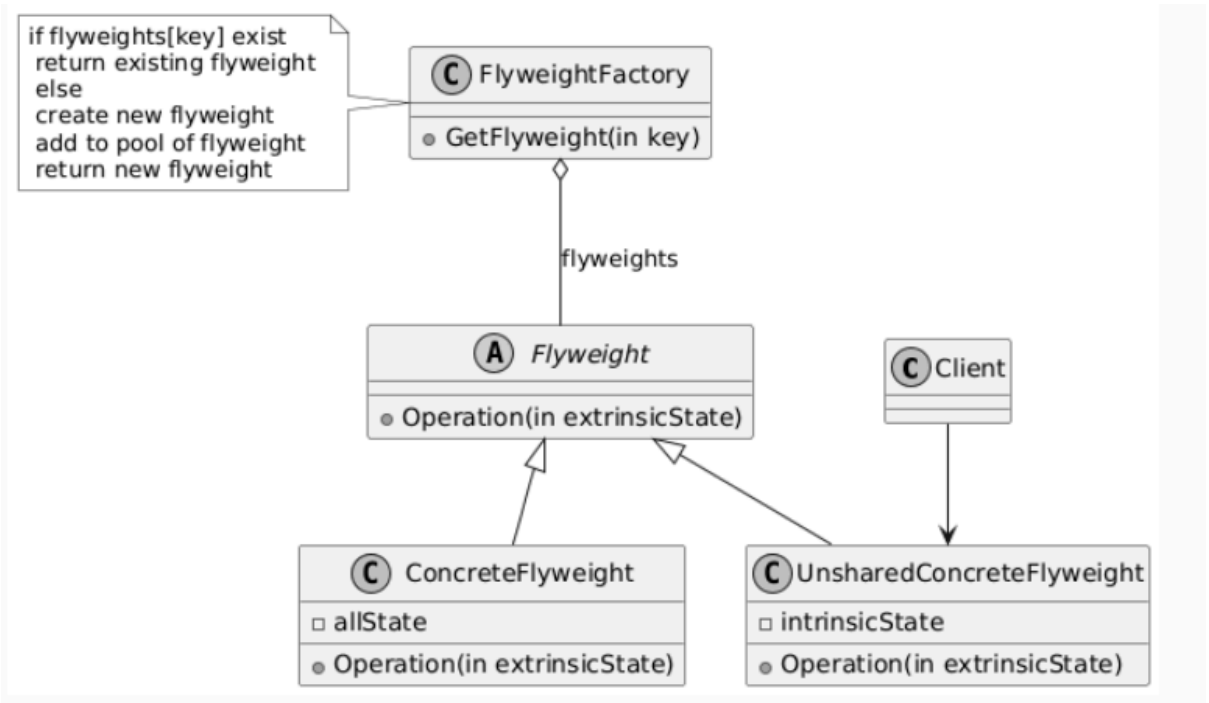
дочірнім елементам, а також надає методи для управління дочірніми елементами (додавання, видалення).

Клієнт працює з усіма об'єктами через інтерфейс Component. Коли клієнт викликає операцію на об'єкті-композиті, той рекурсивно викликає ту саму операцію для всіх своїх дочірніх елементів.

4. Яке призначення шаблону «Легковаговик»?

Шаблон «Легковаговик» (Flyweight) призначений для ефективної підтримки великої кількості дрібних об'єктів шляхом зменшення споживання пам'яті. Це досягається шляхом винесення спільної частини стану об'єктів (внутрішній стан) в один спільний екземпляр, який використовується повторно в різних контекстах. Унікальна для кожного об'єкта частина стану (зовнішній стан) передається ззовні під час виклику методів.

5. Нарисуйте структуру шаблону «Легковаговик».



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

- **Flyweight (Пристосуванець):** Інтерфейс, що визначає метод, якому передається зовнішній стан.

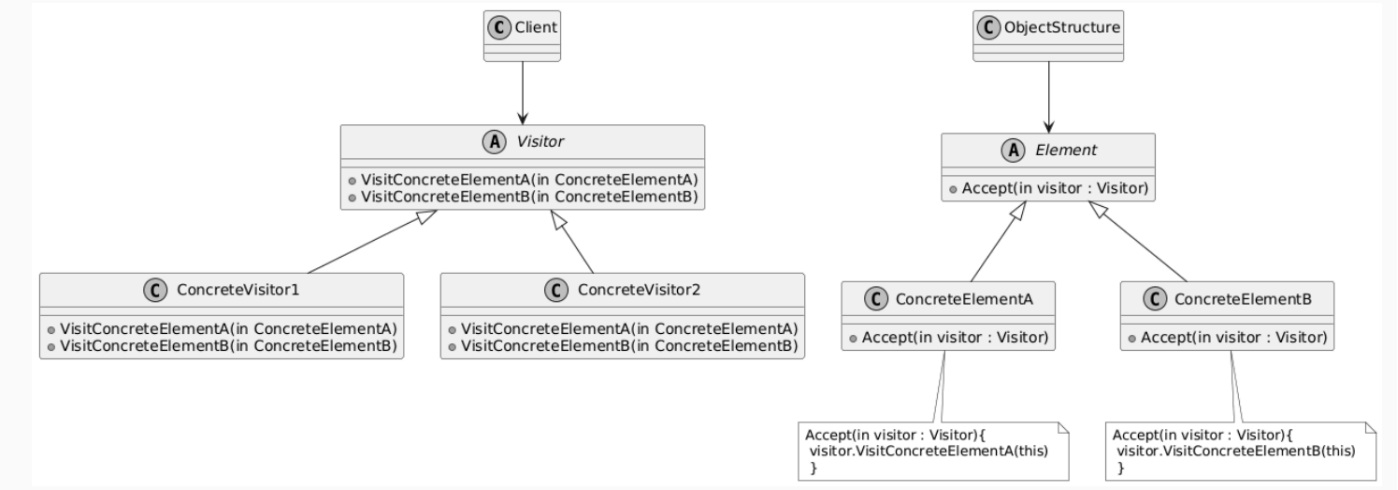
- **ConcreteFlyweight (Конкретний пристосуванець):** Реалізує інтерфейс і зберігає внутрішній (спільний) стан. Об'єкти цього класу є поділюваними.
- **FlyweightFactory (Фабрика пристосуванців):** Створює та керує пулом об'єктів-пристосуванців. Коли клієнт запитує пристосуванець, фабрика повертає існуючий екземпляр або створює новий, якщо його ще немає.
- **Client (Клієнт):** Зберігає або обчислює зовнішній стан і передає його в методи пристосуванців, яких він отримує від фабрики.

Клієнт запитує об'єкт у Фабрики. Фабрика повертає йому спільний екземпляр. Клієнт викликає метод цього екземпляра, передаючи унікальний зовнішній стан як аргумент.

7. Яке призначення шаблону «Інтерпретатор»?
Шаблон «Інтерпретатор» (Interpreter) використовується для визначення представлення граматики для певної мови та створення інтерпретатора, який використовує це представлення для розбору (інтерпретації) речень цієї мови. Він дозволяє легко реалізовувати прості мови запитів або скриптові мови.

8. Яке призначення шаблону «Відвідувач»?
Шаблон «Відвідувач» (Visitor) дозволяє додавати нові операції до існуючої структури об'єктів, не змінюючи класи цих об'єктів. Основне призначення - відокремити алгоритм від структури об'єктів, над якими він оперує, що дозволяє легко додавати нові операції.

9. Нарисуйте структуру шаблону «Відвідувач».



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

- **Visitor (Відвідувач):** Інтерфейс, що оголошує набір методів visit() для кожного типу конкретного елемента (ConcreteElement).
- **ConcreteVisitor (Конкретний відвідувач):** Реалізує інтерфейс Visitor і містить конкретну логіку операції для кожного типу елемента.
- **Element (Елемент):** Інтерфейс, що оголошує метод accept(Visitor), який приймає відвідувача як аргумент.
- **ConcreteElement (Конкретний елемент):** Реалізує інтерфейс Element. Метод accept() в ньому зазвичай викликає visitor.visit(this).

Клієнт створює об'єкт ConcreteVisitor і передає його в метод accept() об'єкта ConcreteElement. Елемент, у свою чергу, викликає відповідний метод visit() у відвідувача, передаючи себе (this) як аргумент. Це дозволяє відвідувачу виконати операцію над елементом.