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
Технологія розроблення програмного забезпечення

“Патерни проектування”

Виконав
студент групи ІА-31
Даценко Влад

Перевірив:
Мягкий Михайло
Юрійович

Київ 2025

Тема: Патерни проектування.

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

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

Короткі теоретичні відомості

Шаблон проєктування «**Interpreter**» (Інтерпретатор) належить до поведінкових патернів і відіграє ключову роль у ситуаціях, коли виникає потреба у розробці власної предметно-орієнтованої мови або інтерпретації специфічних наборів команд. Його основне завдання полягає у визначенні подання граматики для мови та реалізації механізму, який здатен обробляти та виконувати речення, побудовані згідно з цією граматикою. В основі функціонування цього шаблону лежить концепція абстрактного синтаксичного дерева — ієрархічної структури об’єктів, де кожен вузол представляє певне правило граматики. Ці вузли поділяються на термінальні вирази, які є кінцевими елементами й повертають конкретне значення, та нетермінальні вирази, які виступають контейнерами для інших виразів і реалізують логіку їх об’єднання. Процес виконання запиту клієнта відбувається через рекурсивний виклик методу інтерпретації для кожного вузла дерева в межах спільного контексту, який зберігає глобальну інформацію або поточний стан обчислень.

Такий підхід забезпечує виняткову гнучкість архітектури: додавання нового

граматичного правила зводиться до створення нового класу, що спрощує розширення системи без ризику пошкодити існуючий код. Це робить «Interpreter» ідеальним рішенням для таких задач, як аналіз математичних виразів, обробка пошукових запитів або реалізація простих скриптових мов. Проте варто враховувати, що цей шаблон залишається ефективним лише для мов із відносно простою граматикою. У випадку складних мовних конструкцій кількість необхідних класів починає стрімко зростати, що призводить до надмірної складності підтримки коду та зниження продуктивності, тому для масштабних граматик доцільніше використовувати повноцінні синтаксичні аналізатори або компілятори.

Хід роботи

..25 Installer generator (iterator, builder, factory method, bridge, interpreter, client-server)

Генератор інсталяційних пакетів повинен мати якийсь спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка - створити ярлик на робочому столі; ввести в текстове поле деякі дані, наприклад, ліцензійний ключ і т.д.). Генератор повинен вивести один файл .exe або .msi.

В моїй лабораторній роботі розглянуто патерн **Interpreter**.

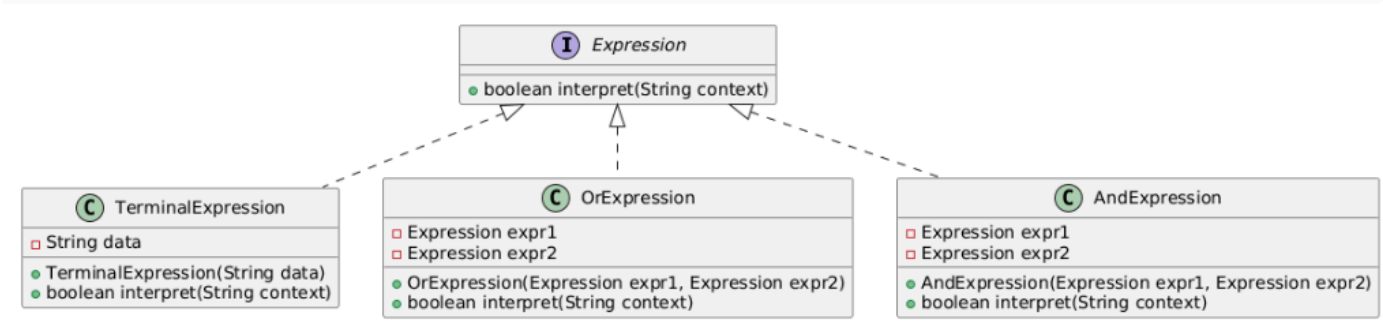

Рисунок 1. Структура патерну **Interpreter**.

Я вирішив використати патерн Interpreter для гнучкої перевірки підтримуваних форматів файлів у процесі конвертації. Це дозволяє описувати правила для визначення типів файлів у вигляді окремих класів і легко змінювати або

розширювати ці правила без зміни основної логіки програми.

У проекті патерн реалізовано через внутрішні класи Expression, TerminalExpression, OrExpression, які інкапсулюють логіку перевірки типу файлу. У методі generatePackage() ці вирази комбінуються для перевірки, чи належить вхідний або вихідний файл до підтримуваних форматів.

Завдяки такому підходу, підтримка нових форматів або складніших умов перевірки реалізується додаванням нових виразів (Expression), що забезпечує масштабованість і зрозумілість коду. Патерн Interpreter дозволяє інкапсулювати логіку перевірки у вигляді об’єктів, спростити підтримку та розширення системи, а також зробити її більш гнучкою до змін вимог. Така структура особливо корисна, коли потрібно часто змінювати або комбінувати правила перевірки для різних сценаріїв обробки файлів.

Код:

```
public interface Expression {
    boolean interpret(String context);
}
```

Рисунок 2. Інтерфейс Expression

```java
public class TerminalExpression implements Expression {
    private final String data;

    public TerminalExpression(String data) {
        this.data = data;
    }

    @Override
    public boolean interpret(String context) {
        return context != null && context.contains(data);
    }
}
```

Рисунок 3.Клас TerminalExpression

```java
public class OrExpression implements Expression {
    private final Expression expr1;
    private final Expression expr2;

    public OrExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) || expr2.interpret(context);
    }
}
```

Рисунок 4.Клас OrExpression

```java
public class AndExpression implements Expression {
    private final Expression expr1;
    private final Expression expr2;

    public AndExpression(Expression expr1, Expression expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }

    @Override
    public boolean interpret(String context) {
        return expr1.interpret(context) && expr2.interpret(context);
    }
}
```

Рисунок 5.Клас AndExpression

```java
public class InterpreterDemo {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) {

        Expression isExe = new TerminalExpression(data: "exe");
        Expression isMsi = new TerminalExpression(data: "msi");
        Expression isExeOrMsi = new OrExpression(isExe, isMsi);

        String test1 = "setup.exe";
        String test2 = "installer.msi";
        String test3 = "readme.txt";

        System.out.println(test1 + " is exe or msi? " + isExeOrMsi.interpret(test1));
        System.out.println(test2 + " is exe or msi? " + isExeOrMsi.interpret(test2));
        System.out.println(test3 + " is exe or msi? " + isExeOrMsi.interpret(test3));
    }
}
```

Рисунок 6.Клас InterpreterDemo

```java
interface Expression {
  boolean interpret(String context);
}
```

Рисунок 7. Інтерфейс Expression у класі Installer

```java
public void generatePackage() {
    notifyObservers("Starting package generation...", 0);

    Expression py = new TerminalExpression(data: "PY");        You, 27 minutes
    Expression jar = new TerminalExpression(data: "JAR");
    Expression validInput = new OrExpression(py, jar);
    String inputType = file.getFileType().toString().toUpperCase();
    if (!validInput.interpret(inputType)) {
        notifyError("Unsupported input file type: " + inputType);
        return;
    }

    Expression exe = new TerminalExpression(data: "EXE");
    Expression msi = new TerminalExpression(data: "MSI");
    Expression validOutput = new OrExpression(exe, msi);
    String outputType = outputFile.getFileType().toString().toUpperCase();
    if (!validOutput.interpret(outputType)) {
        notifyError("Unsupported output file type: " + outputType);
        return;
    }
```

Рисунок 8. У методі generatePackage() ці вирази комбінуються для перевірки

Висновок: У ході виконання лабораторної роботи було реалізовано патерн Interpreter для гнучкої перевірки підтримуваних форматів файлів у процесі конвертації. Такий підхід дозволив інкапсулювати правила перевірки у вигляді окремих класів-виразів, що значно спростило розширення та модифікацію логіки без зміни основного коду програми.

Завдяки використанню Interpreter стало можливим легко додавати нові формати або комбінувати різні умови перевірки, що підвищує масштабованість і підтримуваність системи. Реалізація цього патерну зробила архітектуру більш гнучкою до змін вимог, а також покращила зрозумілість і структурованість коду. Таким чином, застосування Interpreter у даній роботі є доцільним і ефективним для вирішення задач, пов’язаних із динамічною перевіркою та обробкою різних сценаріїв.

.