

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»

Виконала:
студентка групи ІА-32
Глущенко Анастасія
Сергіївна

Перевірив:
Мягкий Михайло
Юрійович

Зміст

Вступ	3
Теоретичні відомості	4
Хід роботи.....	8
1. Шаблон проектування «Interpreter».....	8
2. Діаграма класів, яка представляє використання шаблону «Interpreter»	10
3. Фрагменти коду реалізації шаблону «Interpreter»	12
Висновки	17
Контрольні запитання.....	18

Вступ

Метою роботи є вивчення структури шаблонів «Composite», «Flyweight» (Пристосуванець), «Interpreter», «Visitor» та навчитися застосовувати «Interpreter» в реалізації програмної системи.

Було обрано тему:

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

Теоретичні відомості

1. «Composite» (Компонувальник)

Компонує об'єкти в деревоподібні структури для представлення ієархій «частина-ціле». Дозволяє клієнтам однаково трактувати як окремі об'єкти, так і їхні композиції (групи). Ключова ідея: Створення універсального інтерфейсу для "листа" (простого елемента) та "контейнера" (складного елемента). Це дозволяє будувати складні дерева об'єктів і працювати з ними рекурсивно. Застосування:

- Коли потрібно представити ієархію об'єктів типу «частина-ціле» (наприклад, файлова система з папками та файлами, графічний редактор з групами фігур).
- Коли клієнти повинні ігнорувати різницю між складеними та окремими об'єктами.
- Для спрощення коду клієнта, якому не потрібно перевіряти тип об'єкта перед викликом методу.

Структура та реалізація

- Component (Компонент): Оголошує інтерфейс для об'єктів у композиції. Впроваджує поведінку за замовчуванням для інтерфейсу, спільногого для всіх класів, якщо це доречно.
- Leaf (Лист): Представляє кінцеві об'єкти композиції (елементи без підементів). Реалізує поведінку Component.
- Composite (Контейнер/Композит): Визначає поведінку для компонентів, що мають дочірні елементи. Зберігає дочірні компоненти та реалізує операції управління ними (додавання, видалення). Делегує виконання операцій своїм нащадкам.
- Client (Клієнт): Маніпулює об'єктами композиції через інтерфейс Component.

2. «Flyweight» (Пристосуванець)

Використовує спільне використання (sharing) для ефективної підтримки великої кількості дрібних об'єктів. Основна мета: Економія оперативної пам'яті шляхом розділення спільногого стану об'єктів (внутрішній стан) від унікального для кожного екземпляра (зовнішній стан). Застосування:

- Коли програма використовує величезну кількість об'єктів (наприклад, символи в текстовому редакторі, частинки у графічній системі).
- Коли витрати на зберігання великої кількості об'єктів є критичними для пам'яті.
- Коли більшу частину стану об'єктів можна винести назовні (зробити контекстною).

Структура та реалізація

- Flyweight (Пристосуванець): Оголошує інтерфейс, через який пристосуванці можуть отримувати зовнішній стан і діяти відповідно до нього.
- ConcreteFlyweight (Конкретний Пристосуванець): Реалізує інтерфейс Flyweight і додає сховище для внутрішнього стану (intrinsic state). Цей стан має бути незалежним від контексту пристосуванця.
- UnsharedConcreteFlyweight: Не всі підкласи Flyweight обов'язково мають бути розділеними. Інтерфейс Flyweight дозволяє спільне використання, але не нав'язує його.
- FlyweightFactory (Фабрика Пристосуванців): Створює об'єкти-пристосуванці та керує ними. Забезпечує належне спільне використання: коли клієнт запитує пристосуванця, фабрика надає існуючий екземпляр або створює новий, якщо його ще немає.

3. «Interpreter» (Інтерпретатор)

Для заданої мови визначає представлення її граматики, а також інтерпретатор, який використовує це представлення для інтерпретації речень мови. Ключова ідея: Представлення кожного правила граматики

окремим класом. Речення мови будується як дерево об'єктів цих класів (абстрактне синтаксичне дерево). Застосування:

- Коли граматика мови проста та стабільна (наприклад, регулярні вирази, SQL-запити, прості математичні вирази).
- Коли ефективність не є критичним фактором (оскільки побудова та обхід великих дерев об'єктів може бути повільним).
- Для створення спеціалізованих мов налаштувань або скриптів.

Структура та реалізація

- **AbstractExpression** (Абстрактний Вираз): Оголошує абстрактну операцію `interpret()`, спільну для всіх вузлів абстрактного синтаксичного дерева.
- **TerminalExpression** (Термінальний Вираз): Реалізує операцію `interpret` для термінальних символів граматики (елементів, які не розкладаються далі, наприклад, числа або змінні).
- **NonterminalExpression** (Нетермінальний Вираз): По одному класу для кожного правила граматики. Здійснює інтерпретацію, викликаючи `interpret` для своїх компонентів (інших виразів).
- **Context** (Контекст): Містить інформацію, яка є глобальною для інтерпретатора (наприклад, значення змінних).
- **Client** (Клієнт): Будує (або отримує) абстрактне синтаксичне дерево, що представляє певне речення мови, та викликає операцію `interpret`.

4. «Visitor» (Відвідувач)

Дозволяє додавати нові операції до об'єктів певної структури, не змінюючи класи цих об'єктів. Ключова ідея: Техніка "подвійної диспетчеризації" (Double Dispatch). Алгоритм виносиється з класів елементів у окремий клас-відвідувач. Елемент "приймає" відвідувача і викликає відповідний метод відвідувача, передаючи себе як аргумент. Застосування:

- Коли структура об'єктів містить багато класів з різними інтерфейсами, і ви хочете виконувати над ними операції, що залежать від їхніх конкретних класів.
- Коли класи, що визначають структуру об'єктів, рідко змінюються, але ви часто хочете додавати нові операції над цією структурою (наприклад, експорт у XML, генерація звіту, підрахунок статистики).
- Щоб уникнути "забруднення" класів елементів стороною логікою.

Структура та реалізація

- **Visitor** (Відвідувач): Оголошує операцію `visit` для кожного класу `ConcreteElement` у структурі об'єктів. Ім'я та сигнатура операції ідентифікують клас, який надсилає запит відвідувачу.
- **ConcreteVisitor** (Конкретний Відвідувач): Реалізує кожну операцію, оголошенню класом `Visitor`. Кожна операція реалізує фрагмент алгоритму, визначений для відповідного класу об'єкта в структурі.
- **Element** (Елемент): Оголошує операцію `accept`, яка приймає відвідувача як аргумент.
- **ConcreteElement** (Конкретний Елемент): Реалізує операцію `accept`, викликаючи відповідний метод `visit` на об'єкті відвідувача (наприклад, `visitor.visitConcreteElementA(this)`).
- **ObjectStructure** (Структура Об'єктів): Може перераховувати свої елементи. Може надавати високорівневий інтерфейс, що дозволяє відвідувачу відвідувати його елементи.

Хід роботи

1. Шаблон проектування «Interpreter»

Призначення: Шаблон Interpreter (Інтерпретатор) використовується для визначення граматики простої мови та інтерпретації речень цієї мови. Він дозволяє побудувати структуру класів для розбору та виконання виразів, перетворюючи текстові команди або умови в об'єктно-орієнтовану ієрархію (абстрактне синтаксичне дерево), яку програма може обчислити та виконати.

Проблема, яку вирішує шаблон: У процесі розробки генератора інсталяторів виникла потреба перевіряти сумісність системи (System Requirements) перед початком генерації. Користувачі повинні мати можливість задавати гнучкі системні вимоги у вигляді текстового рядка (наприклад, os windows AND java 17). Жорстке кодування перевірок (Hardcoding) є негнучким, а простий розбір рядків не дозволяє ефективно обробляти складні логічні комбінації та розширювати набір підтримуваних команд (наприклад, додати перевірку вільного місця на диску) без втручання в основну логіку валідації.

Як шаблон вирішує проблему: Шаблон Interpreter вирішує цю проблему шляхом представлення кожного правила граматики (оператора або умови) у вигляді окремого класу. Текстовий рядок вимог перетворюється на дерево об'єктів, де:

1. **Термінальні вирази** (Terminal Expressions) відповідають за перевірку конкретних базових умов (наприклад, чи збігається версія Java).
2. **Нетермінальні вирази** (Non-terminal Expressions) відповідають за логічні операції (наприклад, AND), об'єднуючи результати інших виразів. Це дозволяє динамічно будувати та обчислювати складні логічні умови, комбінуючи прості правила.

Процес створення об'єкта виглядає так:

1. Існує спільний інтерфейс Expression, який визначає метод interpret, що приймає контекст.

2. Створюється об'єкт Context, який містить реальні дані про поточне середовище виконання (наприклад, поточна ОС, встановлена версія Java).

3. Допоміжний клас-парсер розбирає вхідний рядок вимог і будує дерево об'єктів, що реалізують інтерфейс Expression.

4. Клієнтський код викликає метод interpret на кореневому елементі дерева, передаючи йому контекст, і отримує результат перевірки (true/false).

Як реалізовано в проекті:

- **Інтерфейс Expression** визначає контракт для всіх вузлів синтаксичного дерева (метод interpret).

- **Клас Context** зберігає глобальну інформацію про систему, яка необхідна для виконання перевірок (значення os, java тощо).

- **Класи OsExpression та JavaVersionExpression** є термінальними виразами, які реалізують конкретну логіку перевірки операційної системи та версії Java.

- **Клас AndExpression** є нетермінальним виразом, який реалізує логічний оператор "I", рекурсивно викликаючи інтерпретацію для своїх підоператорів.

- **Клас RequirementsParser** виконує роль клієнта/будівельника дерева, перетворюючи текстовий рядок користувача на ланцюжок об'єктів Expression.

- **Клас InstallerBuilder** використовує отримане дерево виразів для валідації перед запуском процесу генерації.

Таким чином, шаблон Interpreter забезпечує:

- **Гнучкість:** Можливість змінювати та комбінувати правила перевірки "на льоту" шляхом зміни вхідного рядка без перекомпіляції коду.

- Розширюваність:** Додавання нових правил (наприклад, перевірка RAM) вимагає лише створення нового класу-виразу, що реалізує інтерфейс Expression.
- Ізоляція логіки:** Кожне правило перевірки інкапсульоване у своєму власному класі, що спрощує тестування та підтримку коду.

2. Діаграма класів, яка представляє використання шаблону «Interpreter»

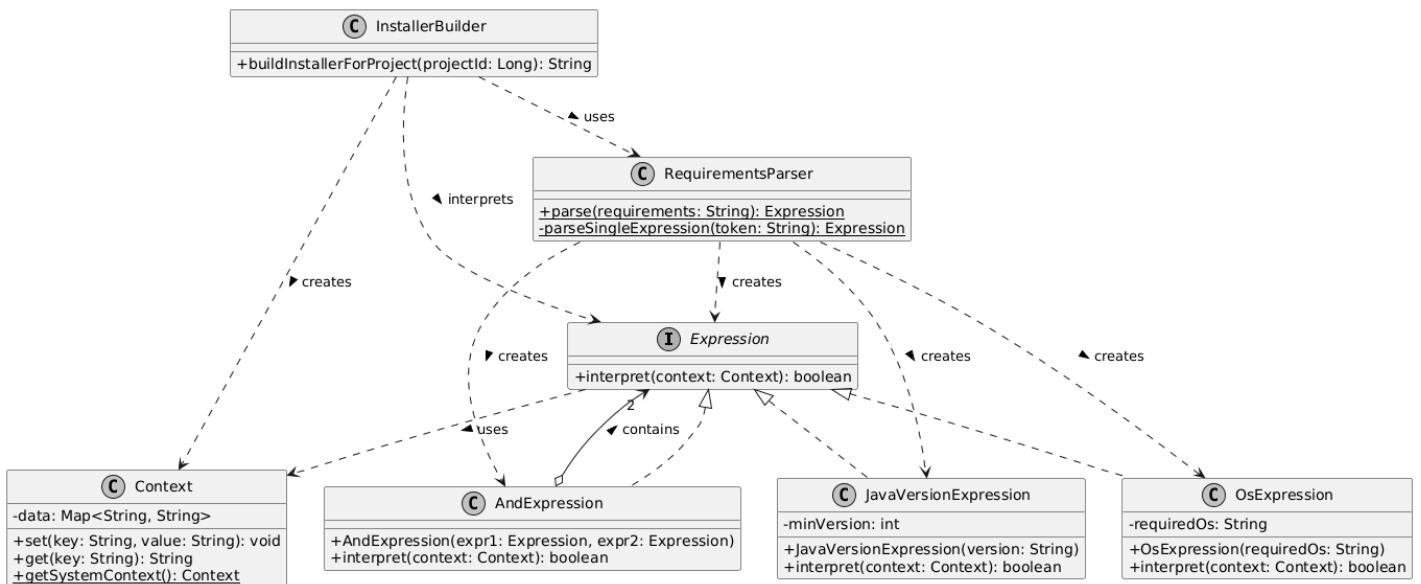


Рисунок 1 - Діаграма класів

Опис діаграми класів реалізації шаблону Interpreter

На діаграмі зображенено структуру класів, що беруть участь у реалізації шаблону проектування **Interpreter** (Інтерпретатор), який використовується для розбору та обчислення логічних виразів перевірки системних вимог (наприклад, "os windows AND java 17").

Інтерфейс Expression виступає в ролі Абстрактного Виразу (Abstract Expression). Він визначає спільний контракт для всіх вузлів синтаксичного дерева, декларуючи метод `interpret`, який приймає контекст і повертає булевий результат виконання умови. Цей інтерфейс забезпечує поліморфізм, дозволяючи клієнту працювати з будь-яким типом виразу однаково.

Класи **OsExpression** та **JavaVersionExpression** є Термінальними Виразами (Terminal Expressions). Вони імплементують інтерфейс Expression та реалізують конкретну логіку перевірки базових елементів мови (перевірка операційної системи та версії Java відповідно). Ці класи є "листками" дерева і не містять вкладених виразів.

Клас **AndExpression** є Нетермінальним Виразом (Non-terminal Expression). Він реалізує інтерфейс Expression і відповідає за логічну операцію "I" (AND). Цей клас використовує агрегацію (зв'язок з ромбиком), оскільки він містить у собі два інших об'єкти типу Expression (лівий і правий операнди) і рекурсивно викликає їхню інтерпретацію.

Клас **Context** виступає в ролі Контексту. Він зберігає глобальну інформацію про поточне середовище виконання (наприклад, реальну назву ОС та встановлену версію Java), яка необхідна виразам для виконання перевірок. Об'єкт цього класу передається як аргумент у метод `interpret`.

Клас **RequirementsParser** виконує роль парсера/будівельника. Він відповідає за розбір вхідного текстового рядка вимог і побудову відповідного абстрактного синтаксичного дерева (AST) з об'єктів Expression. Він створює (creates) екземпляри термінальних та нетермінальних виразів залежно від змісту рядка.

Клас **InstallerBuilder** виступає Клієнтом. Він ініціює процес перевірки перед генерацією інсталятора: використовує RequirementsParser для отримання дерева виразів, створює актуальний Context і викликає метод `interpret` на кореневому елементі дерева.

Ця діаграма відображає архітектуру, що забезпечує гнучкість системи валідації. Завдяки патерну Interpreter, складні логічні умови формуються динамічно з простих об'єктів, що дозволяє змінювати правила перевірки без перекомпіляції коду програми.

3. Фрагменти коду реалізації шаблону «Interpreter»

Expression

```
package com.example.installer.service.interpreter;

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

Context

```
package com.example.installer.service.interpreter;

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

public class Context {
    private final Map<String, String> data = new HashMap<>();

    public void set(String key, String value) {
        data.put(key.toLowerCase(), value.toLowerCase());
    }

    public String get(String key) {
        return data.get(key.toLowerCase());
    }

    public static Context getSystemContext() {
        Context ctx = new Context();
        String os = System.getProperty("os.name").toLowerCase();
        if (os.contains("win")) ctx.set("os", "windows");
        else if (os.contains("mac")) ctx.set("os", "macos");
        else ctx.set("os", "linux");

        ctx.set("java",
        System.getProperty("java.version").split("\\.")[0]);
        return ctx;
    }
}
```

OsExpression

```
package com.example.installer.service.interpreter;

public class OsExpression implements Expression {
    private final String requiredOs;

    public OsExpression(String requiredOs) {
        this.requiredOs = requiredOs.toLowerCase();
    }

    @Override
    public boolean interpret(Context context) {
        String currentOs = context.get("os");
        return currentOs != null && currentOs.equals(requiredOs);
    }
}
```

JavaVersionExpression

```
package com.example.installer.service.interpreter;

public class JavaVersionExpression implements Expression {
    private final int minVersion;

    public JavaVersionExpression(String version) {
        this.minVersion = Integer.parseInt(version);
    }

    @Override
    public boolean interpret(Context context) {
        String currentJavaStr = context.get("java");
        if (currentJavaStr == null) return false;

        try {
            int currentJava = Integer.parseInt(currentJavaStr);
            return currentJava >= minVersion;
        } catch (NumberFormatException e) {
            return false;
        }
    }
}
```

AndExpression

```
package com.example.installer.service.interpreter;

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(Context context) {
        return expr1.interpret(context) && expr2.interpret(context);
    }
}
```

RequirementsParser

```
package com.example.installer.service.interpreter;

public class RequirementsParser {

    public static Expression parse(String requirements) {
        if (requirements == null || requirements.isEmpty()) {
            return context -> true;
        }

        String[] parts = requirements.split(" AND ");
        Expression finalExpression = null;

        for (String part : parts) {
            Expression currentExpr = parseSingleExpression(part.trim());
        }
    }
}
```

```

        if (finalExpression == null) {
            finalExpression = currentExpr;
        } else {
            finalExpression = new AndExpression(finalExpression,
currentExpr);
        }
    }

    return finalExpression;
}

private static Expression parseSingleExpression(String token) {
    String[] segments = token.split(" ");
    if (segments.length != 2) {
        throw new IllegalArgumentException("Invalid expression format:
" + token);
    }

    String key = segments[0];
    String value = segments[1];

    if (key.equalsIgnoreCase("os")) {
        return new OsExpression(value);
    } else if (key.equalsIgnoreCase("java")) {
        return new JavaVersionExpression(value);
    } else {
        throw new IllegalArgumentException("Unknown requirement key: "
+ key);
    }
}
}

```

InstallerBuilder

```

import com.example.installer.service.interpreter.Context;
import com.example.installer.service.interpreter.Expression;
import com.example.installer.service.interpreter.RequirementsParser;

@Service
public class InstallerBuilder {

    @Transactional
    public String buildInstallerForProject(Long projectId) throws
Exception {
        Project project = projectRepository.findById(projectId)
            .orElseThrow(() -> new
IllegalArgumentException("Project not found"));

        String reqs = project.getInstallerSettings().getRequirements();

        if (reqs != null && !reqs.trim().isEmpty()) {
            System.out.println("Interpreter: Checking system
requirements -> " + reqs);

            Expression expressionTree = RequirementsParser.parse(reqs);

            Context systemContext = Context.getSystemContext();

            boolean isCompatible =
expressionTree.interpret(systemContext);
        }
    }
}

```

```

        if (!isCompatible) {
            throw new RuntimeException("System requirements check
failed! " +
                "Required: [" + reqs + "], but system is
incompatible.");
        }
        System.out.println("Interpreter: System check passed.");
    }

return factory.generateInstaller(jarFile, project);
}
}

```

InstallerSettings

```

@Entity
public class InstallerSettings {

    private String requirements;

    public String getRequirements() {
        return requirements;
    }

    public void setRequirements(String requirements) {
        this.requirements = requirements;
    }
}

```

ProjectController

```

@PostMapping("/projects/{id}/setInstallerSettings")
public String setInstallerSettings(@PathVariable Long id,
                                    @RequestParam String installPath,
                                    @RequestParam(required = false,
defaultValue = "false") boolean createDesktopShortcut,
                                    @RequestParam String language,
                                    @RequestParam String mainClass,
                                    @RequestParam(required = false) String
requirements,
                                    RedirectAttributes redirectAttributes) {
    Optional<Project> projectOpt = projectRepository.findById(id);
    if (projectOpt.isEmpty()) {
        redirectAttributes.addFlashAttribute("message", "Project not
found");
        return "redirect:/projects";
    }

    Project project = projectOpt.get();

    if (project.getInstallerSettings() == null) {
        InstallerSettings newInstallerSettings = new
        InstallerSettings(installPath, createDesktopShortcut, language);
        newInstallerSettings.setMainClass(mainClass);
        newInstallerSettings.setRequirements(requirements);
        newInstallerSettings.setProject(project);
        project.setInstallerSettings(newInstallerSettings);
    } else {
        project.setInstallerSettings(installPath, createDesktopShortcut,
language);
        project.getInstallerSettings().setMainClass(mainClass);
        project.getInstallerSettings().setRequirements(requirements);
    }
}

```

```
projectRepository.save(project);
redirectAttributes.addFlashAttribute("message", "Installer settings
updated.");
return "redirect:/projects/" + id;
}
```

У проекті реалізовано поведінковий шаблон Interpreter, який дозволив створити гнучку систему валідації системних вимог на основі текстових виразів:

- **Об'єктне подання граматики:** Клас RequirementsParser перетворює текстові команди користувача (наприклад, "os windows") у дерево об'єктів, що імплементують інтерфейс Expression, дозволяючи програмі автоматично розбирати та виконувати логічні умови перевірки.
- **Комбінування правил:** Завдяки використанню нетермінальних виразів (клас AndExpression), прості перевірки (OsExpression, JavaVersionExpression) можуть динамічно об'єднуватися у складні логічні ланцюжки, що дозволяє змінювати вимоги до інсталятора без втручання в програмний код.
- **Контекстно-залежне обчислення:** Використання класу Context забезпечує ізоляцію даних про поточне середовище від логіки самих виразів, що дозволяє клієнту (InstallerBuilder) коректно інтерпретувати дерево вимог і блокувати генерацію у разі несумісності системи.

Висновки

На даному етапі реалізації проекту «Installer Generator» було успішно впроваджено шаблон проектування «Interpreter» (Інтерпретатор), який дозволив реалізувати механізм гнучкої валідації системних вимог на основі текстових виразів, забезпечивши "розумну" перевірку середовища перед генерацією інсталятора.

Було створено спільний інтерфейс Expression та клас Context, що забезпечують основу для обчислення умов. Розроблено термінальні вирази OsExpression та JavaVersionExpression для перевірки конкретних параметрів системи, нетермінальний вираз AndExpression для комбінування умов, а також клас RequirementsParser, який відповідає за побудову абстрактного синтаксичного дерева з вхідного рядка користувача.

Таким чином, на поточному етапі було:

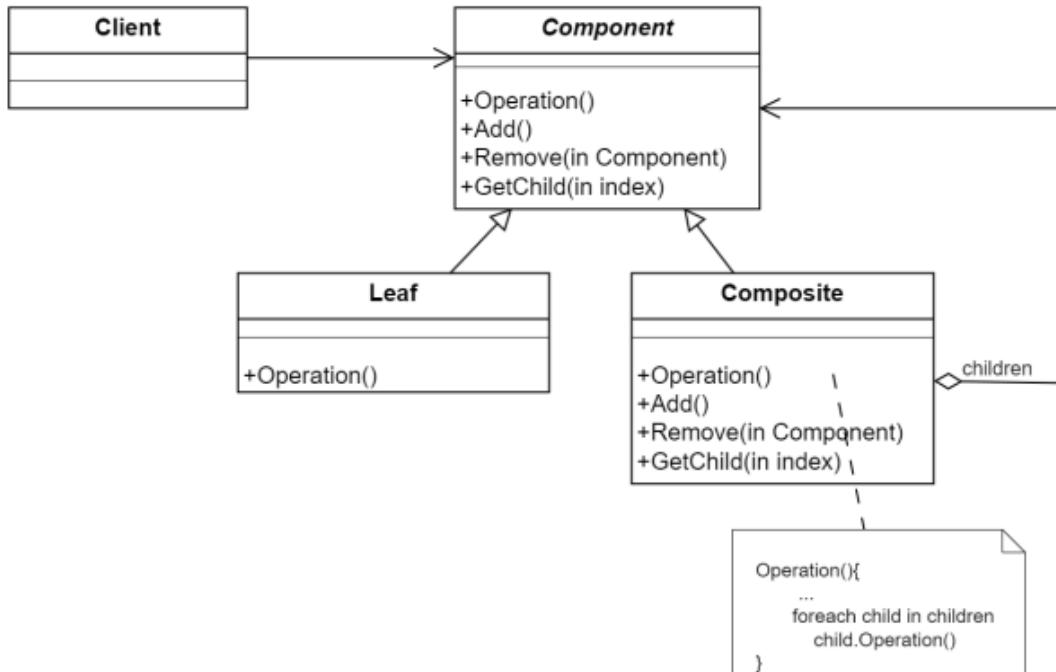
- реалізовано шаблон проектування Interpreter для автоматичного розбору та виконання логічних правил перевірки сумісності системи;
- забезпечено можливість динамічної зміни системних вимог (через редагування текстового рядка конфігурації) без необхідності втручання в програмний код;
- ізольовано логіку валідації окремих параметрів у незалежні класи, що спрощує додавання нових типів перевірок у майбутньому;
- інтегровано механізм попередньої перевірки в процес побудови інсталятора, що дозволяє запобігти генерації файлів для несумісних операційних систем або версій Java.

Контрольні запитання

1. Яке призначення шаблону «Композит»?

Призначення - об'єднувати об'єкти в деревоподібні структури для представлення ієрархій «частина-ціле». Це дозволяє клієнтам однаково працювати як з окремими об'єктами, так і з їхніми групами (композиціями).

2. Нарисуйте структуру шаблону «Композит».



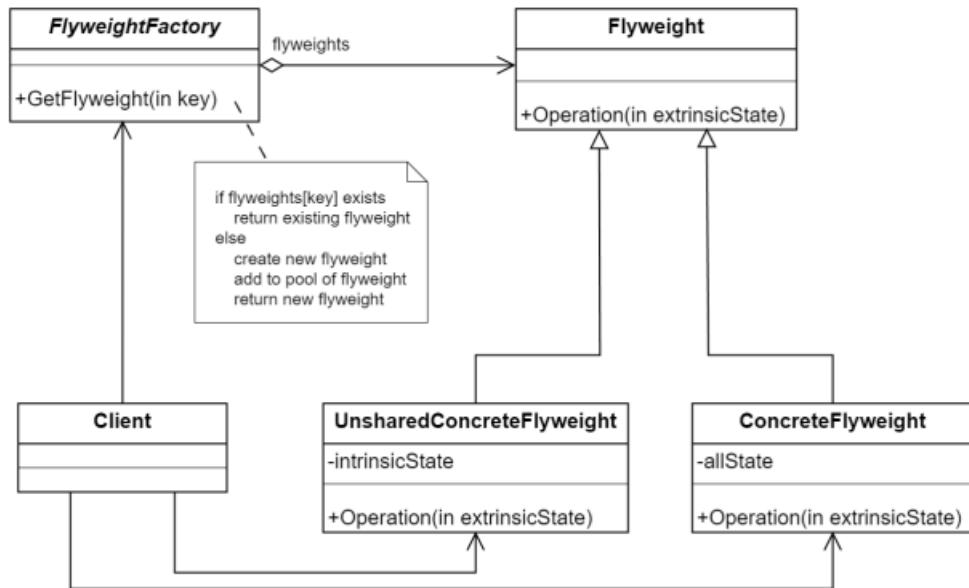
3. Які класи входять в шаблон «Композит», та яка між ними взаємодія?

Входять: **Component** (спільний інтерфейс), **Leaf** (простий елемент, "листок"), **Composite** (складний елемент, що містить дочірні компоненти), **Client**. Взаємодія: Клієнт взаємодіє через інтерфейс **Component**. **Composite** переадресовує запити своїм підементам (**Leaf** або іншим **Composite**) і може виконувати додаткові операції до або після делегування.

4. Яке призначення шаблону «Легковаговик»?

Призначення - ефективно підтримувати велику кількість дрібних об'єктів шляхом спільного використання (sharing) їхнього внутрішнього стану. Це дозволяє суттєво економити оперативну пам'ять.

5. Нарисуйте структуру шаблону «Легковаговик».



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

Входять: **Flyweight** (інтерфейс), **ConcreteFlyweight** (об'єкт, що розділяється), **FlyweightFactory** (фабрика, що керує пулом об'єктів), **Client**. Взаємодія: Клієнт запитує об'єкт у Фабрики. Фабрика повертає існуючий екземпляр або створює новий. Клієнт передає зовнішній стан (контекст) у методи пристосування під час виклику.

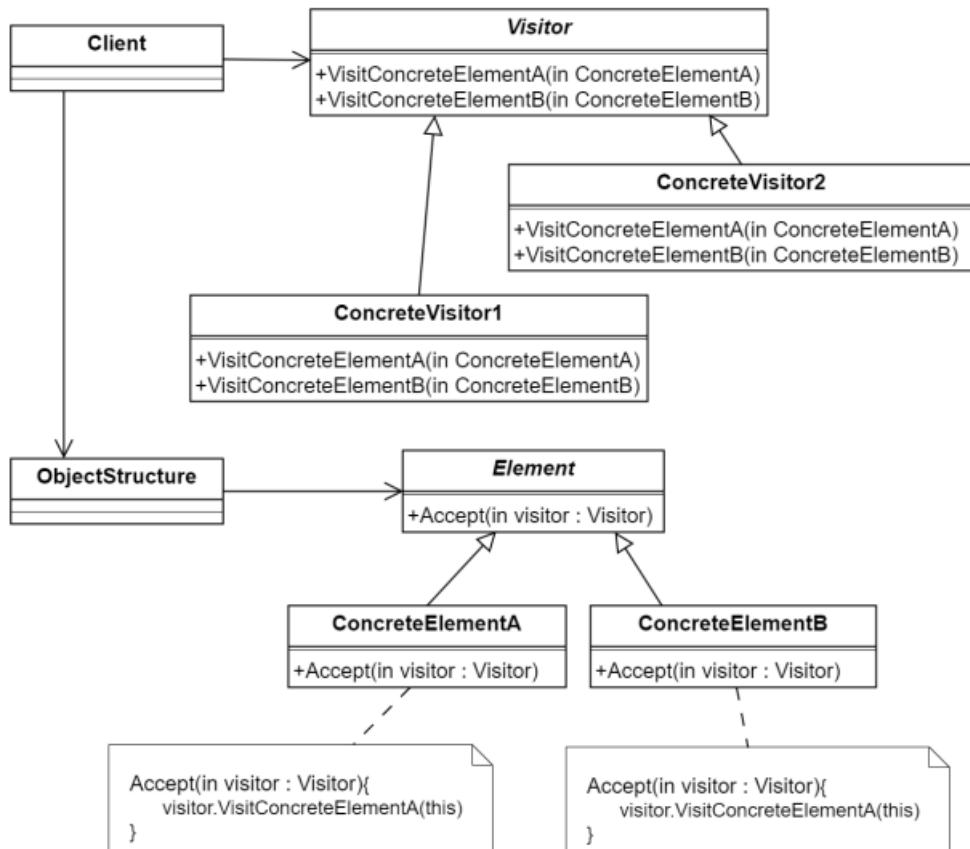
7. Яке призначення шаблону «Інтерпретатор»?

Призначення - для заданої мови визначити представлення її граматики, а також інтерпретатор, який використовує це представлення для інтерпретації речень (виразів) цієї мови.

8. Яке призначення шаблону «Відвідувач»?

Призначення - дозволити додавати нові операції до об'єктів певної структури (ієрархії класів), не змінюючи код самих цих класів.

9. Нарисуйте структуру шаблону «Відвідувач».



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

Входять: **Visitor** (інтерфейс відвідувача), **ConcreteVisitor** (реалізує конкретну операцію), **Element** (інтерфейс елемента структури), **ConcreteElement** (конкретний елемент). Взаємодія: Використовується механізм подвійної диспетчеризації: Клієнт викликає метод `accept(visitor)` на Елементі, а Елемент всередині цього методу викликає `visitor.visit(this)`, передаючи себе як аргумент.