

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-34
Тунік О.

Перевірив:
асистент кафедри ІСТ
Мягкий М.Ю.

Зміст

Теоретичні відомості.....	3
Шаблон «Composite».....	3
Шаблон «Flyweight».....	3
Шаблон «Interpreter»	3
Шаблон «Visitor»	3
Хід роботи	4
Поставлене завдання	4
Реалізація паттерну	4
Фрагменти коду реалізації.....	6
Обґрунтування використання паттерну	8
Відповіді на контрольні запитання.....	9
Висновки	12

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

Теоретичні відомості

Шаблон «Composite»

Шаблон «Composite» це структурний патерн для створення деревоподібних ієрархій об'єктів. Всі елементи ієрархії (як контейнери, так і листя) реалізують спільний інтерфейс. Це дозволяє клієнтському коду обробляти окремі об'єкти та їх композиції єдиним способом, часто через рекурсивні операції. Патерн застосовується для представлення частинно-цілих відношень.

Шаблон «Flyweight»

Шаблон «Flyweight» використовується для оптимізації роботи з великою кількістю об'єктів. Поділяє стан об'єкта на внутрішній (незмінний, спільний) та зовнішній (унікальний, що зберігається клієнтом). Замість створення нових об'єктів повторно використовує вже існуючі спільні екземпляри. Ефективний при наявності багатьох однакових об'єктів.

Шаблон «Interpreter»

Шаблон «Interpreter» це поведінковий патерн для визначення граматики мови та інтерпретації її речень. Реалізується шляхом побудови абстрактного синтаксичного дерева, вузли якого є термінальними або нетермінальними виразами. Кожен вузол відповідає за обробку певної частини граматики. Патерн ефективний лише для нескладних граматик.

Шаблон «Visitor»

Шаблон «Visitor» відокремлює алгоритми від структури об'єктів, на яких вони працюють. Для додавання нових операцій над всією ієрархією класів створюється окремий клас «відвідувач» з методами для кожного типу елемента. Це дозволяє додавати функціональність без зміни самих класів елементів.

Хід роботи

Поставлене завдання

Flexible automation tool (strategy, command, abstract factory, facade, interpreter, SOA)

Інструмент автоматизації повинен забезпечувати найпростіші автоматичні дії для зручності користувача: завантаження нових фільмів / книг / файлів при випуску (наприклад, щоп'ятниці з'являються нові серії улюблених серіалів); встановити статуси в комунікаторах (skype – away при нульовій активності на тривалий час) і т.д. Автоматизація забезпечується шляхом введення правил (на зразок IFTTT.com сервісу), запису макросів (натискання клавіш, дії миші), планувальника завдань (о 5 ранку – початок роздачі торрент-файлів).

Реалізація паттерну

У системі автоматизації FlexibleAutomationTool реалізували поведінковий патерн проектування «Інтерпретатор» (Interpreter) з метою визначення граматики формальної мови та інтерпретації її речень для автоматичної побудови дій (Actions) з текстового макросу. Використання даного патерну дозволяє розширити архітектуру, реалізувавши можливість компіляції текстових інструкцій у виконувану послідовність команд, що забезпечує гнучкість та зручність для кінцевого користувача. Патерн «Інтерпретатор» у даній архітектурі складається з таких ключових елементів: абстрактного виразу (IExpression), термінальних виразів (RunExpression, MessageExpression), нетермінального виразу (SequenceExpression), контексту інтерпретації (InterpreterContext) та клієнта-парсера (SimpleMacroInterpreter), який будує абстрактне синтаксичне дерево. Спроектвану діаграму класів, що ілюструє використання патерну «Інтерпретатор», наведено на рис. 1.

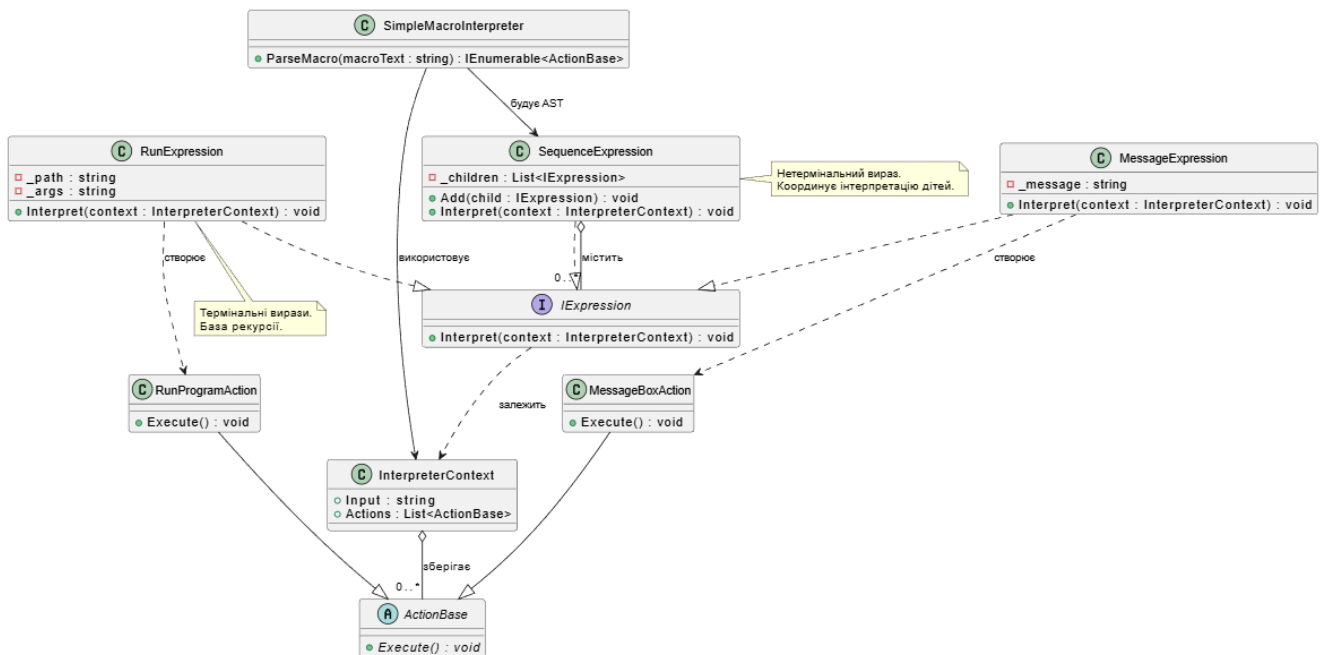


Рисунок 1. Діаграма класів паттерну Interpreter

Центральним елементом архітектури є інтерфейс `IExpression`, який визначає абстрактний контракт для всіх елементів граматики та абстрактного синтаксичного дерева. Даний інтерфейс оголошує метод `Interpret(InterpreterContext)`, який виконує інтерпретацію конкретного вузла дерева в заданому контексті. Інтерфейс забезпечує уніфікований спосіб обробки як простих (термінальних), так і складних (нетермінальних) конструкцій мови через рекурсивний механізм викликів.

Контекст інтерпретації представлений класом `InterpreterContext`, який виступає сховищем глобальної інформації, необхідної для процесу інтерпретації. Контекст зберігає вхідний текст макросу (`Input`) та накопичує результат інтерпретації – колекцію об'єктів `ActionBase` (`Actions`), які будуть виконані пізніше. Таким чином, контекст відокремлює дані, що є спільними для всіх виразів, від конкретної логіки кожного вузла дерева.

Конкретні елементи граматики реалізують інтерфейс `IExpression`. Класи `RunExpression` та `MessageExpression` є термінальними виразами, які відповідають за базові конструкції мови (запуск програми та відображення повідомлення). Вони виступають як листя абстрактного синтаксичного дерева і є основою рекурсії. У своїх методах `Interpret()` вони безпосередньо створюють відповідні об'єкти `RunProgramAction` та `MessageBoxAction` і додають їх до колекції `Actions` в контексті.

Складні конструкції мови представлені класом `SequenceExpression`, який є нетермінальним виразом. Цей клас зберігає колекцію дочірніх виразів (`_children`) та реалізує ключовий принцип патерну – рекурсивну інтерпретацію. У своєму методі `Interpret()` він послідовно викликає `Interpret()` для кожного з дочірніх вузлів,

передаючи їм той самий контекст. Це дозволяє природним чином обробляти послідовності команд у макросі.

Клієнтську роль у патерні виконує клас SimpleMacroInterpreter, який відповідає за синтаксичний аналіз вхідного тексту макросу та побудову абстрактного синтаксичного дерева. Його метод ParseMacro(string macroText) аналізує рядок, створює відповідні об'єкти IExpression (такі як RunExpression, MessageExpression), організовує їх у структуру типу SequenceExpression, ініціалізує InterpreterContext та запускає процес інтерпретації, викликаючи Interpret() на кореневому вузлі дерева. Отримана після інтерпретації з контексту колекція ActionBase повертається як кінцевий результат.

Таким чином, реалізація патерну «Інтерпретатор» дозволяє чітко відокремити синтаксичний аналіз, представлення граматики та власне виконання. Ця структура робить систему легко розширюваною: для додавання нових типів команд у мову макросів достатньо створити новий клас термінального виразу, реалізувавши інтерфейс IExpression, та оновити логіку парсера в SimpleMacroInterpreter. Клієнтський код, який використовує результат інтерпретації, працює з уніфікованою колекцією дій, не залежачи від складності граматики.

Фрагменти коду реалізації

Ключовий інтерфейс IExpression, основа всіх вузлів дерева:

```
namespace FlexibleAutomationTool.Core.Interpreter
{
    public interface IExpression
    {
        void Interpret(InterpreterContext context);
    }
}
```

Клас-контекст InterpreterContext, сховище стану інтерпретації:

```
namespace FlexibleAutomationTool.Core.Interpreter
{
    public class InterpreterContext
    {
        public InterpreterContext(string input)
        {
            Input = input;
            Actions = new List<ActionBase>();
        }

        public string Input { get; }

        public List<ActionBase> Actions { get; }
    }
}
```

Конкретний термінальний вираз RunExpression. Це приклад листа дерева:

```
public class RunExpression : IExpression
{
    private readonly string _path;
```

```

private readonly string _args;

public RunExpression(string path, string args)
{
    _path = path;
    _args = args;
}

public void Interpret(InterpreterContext context)
{
    context.Actions.Add(new RunProgramAction { Path = _path, Arguments = _args });
}
}

```

Нетермінальний вираз SequenceExpression, рекурсивно обробляє дітей:

```

namespace FlexibleAutomationTool.Core.Interpreter
{
    public class SequenceExpression : IExpression
    {
        private readonly List<IExpression> _children = new();

        public SequenceExpression() { }

        public void Add(IExpression expr) => _children.Add(expr);

        public void Interpret(InterpreterContext context)
        {
            foreach (var child in _children)
            {
                child.Interpret(context);
            }
        }
    }
}

```

SimpleMacroInterpreter будує дерево та запускає інтерпретацію:

```

namespace FlexibleAutomationTool.Core.Interpreter
{
    public class SimpleMacroInterpreter : IInterpreter
    {
        public IEnumerable<ActionBase> ParseMacro(string macroText)
        {
            var context = new InterpreterContext(macroText ?? string.Empty);
            if (string.IsNullOrEmpty(macroText))
                return context.Actions;

            var root = new SequenceExpression();

            var lines = macroText.Split(new[] { '\r', '\n' },
StringSplitOptions.RemoveEmptyEntries);
            foreach (var raw in lines)
            {
                var line = raw.Trim();
                if (line.StartsWith("RUN ", StringComparison.OrdinalIgnoreCase))
                {
                    var rest = line.Substring(4).Trim();
                    var parts = rest.Split(' ', 2);
                    var path = parts[0];
                    var args = parts.Length > 1 ? parts[1] : null;
                    root.Add(new RunExpression(path, args));
                }
                else if (line.StartsWith("MSG ", StringComparison.OrdinalIgnoreCase))

```

```

        {
            var msg = line.Substring(4).Trim();
            root.Add(new MessageExpression(msg));
        }
    }
    root.Interpret(context);

    return context.Actions;
}
}
}

```

Обґрунтування використання паттерну

Використання патерну «Інтерпретатор» (Interpreter) у системі FlexibleAutomationTool є обґрунтованим з огляду на необхідність трансформації текстових макросів користувача у виконувані послідовності дій автоматизації. У реальних сценаріях використання системи, користувачі потребують гнучкого способу задавати дії у вигляді текстових команд або скриптів, що нагадують природну мову або простий командний рядок. Без застосування патерну «Інтерпретатор» обробка таких текстових інструкцій вимагала б жорстко закодованої логіки парсингу з високою зв'язаністю між розбором синтаксису та виконанням дій, що призвело б до складного в підтримці та важко розширюваного коду.

Патерн «Інтерпретатор» дозволяє чітко відокремити синтаксичний аналіз, представлення граматики та виконання. У даній реалізації це виражено через побудову абстрактного синтаксичного дерева (AST), де кожен вузол реалізує інтерфейс IExpression. Такий підхід дозволяє моделювати складні конструкції мови (як послідовності команд через SequenceExpression) за допомогою композиції простих елементів (як RunExpression чи MessageExpression), що повністю відповідає принципам об'єктно-орієнтованого проектування.

Застосування патерну забезпечує масштабованість системи при розширенні мови макросів. Додавання нової команди (наприклад, паузи або умовного виконання) вимагає лише створення нового класу, що реалізує IExpression, та оновлення парсера в SimpleMacroInterpreter. Логіка виконання, побудова дерева та механізм інтерпретації залишаються незмінними, що демонструє дотримання принципу відкритості/закритості.

Важливою перевагою є уніфікована обробка складних та простих конструкцій через рекурсивний механізм виклику методу Interpret(). Це дозволяє SequenceExpression обробляти довільні вкладені послідовності команд, не знаючи конкретних деталей дочірніх виразів. Така архітектура робить можливим інтерпретацію макросів зі структурою довільної складності без збільшення коду.

Порівняно з альтернативними підходами, такими як пряма трансляція тексту в код або використання шаблонних методів, патерн «Інтерпретатор» забезпечує значно кращу структуру для мов з чітко визначеною, навіть невеликою,

граматикою. Жорсткий парсинг без AST ускладнив би додавання нових конструкцій та підтримку рекурсивних структур, тоді як використання InterpreterContext централізує управління станом інтерпретації та результатами.

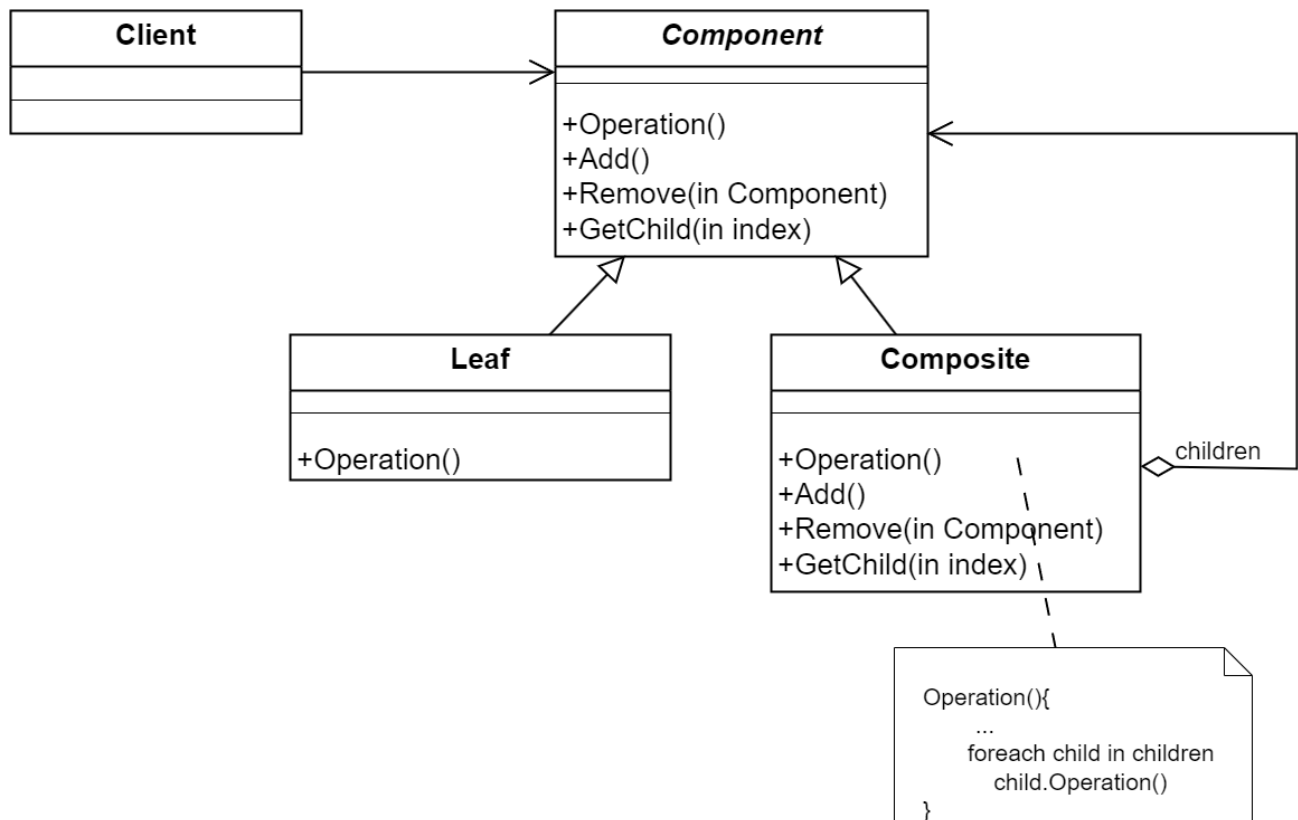
Таким чином, патерн «Інтерпретатор» не лише вирішує поточну задачу трансформації текстових макросів у дії, але й закладає архітектурну основу для майбутнього розширення мови автоматизації, зберігаючи систему чистою, модульною та легко підтримуваною.

Відповіді на контрольні запитання

1. Яке призначення шаблону «Композит»?

Композит створює деревоподібну структуру об'єктів за принципом "частина-ціле" для їх уніфікованої обробки.

2. Нарисуйте структуру шаблону «Композит».



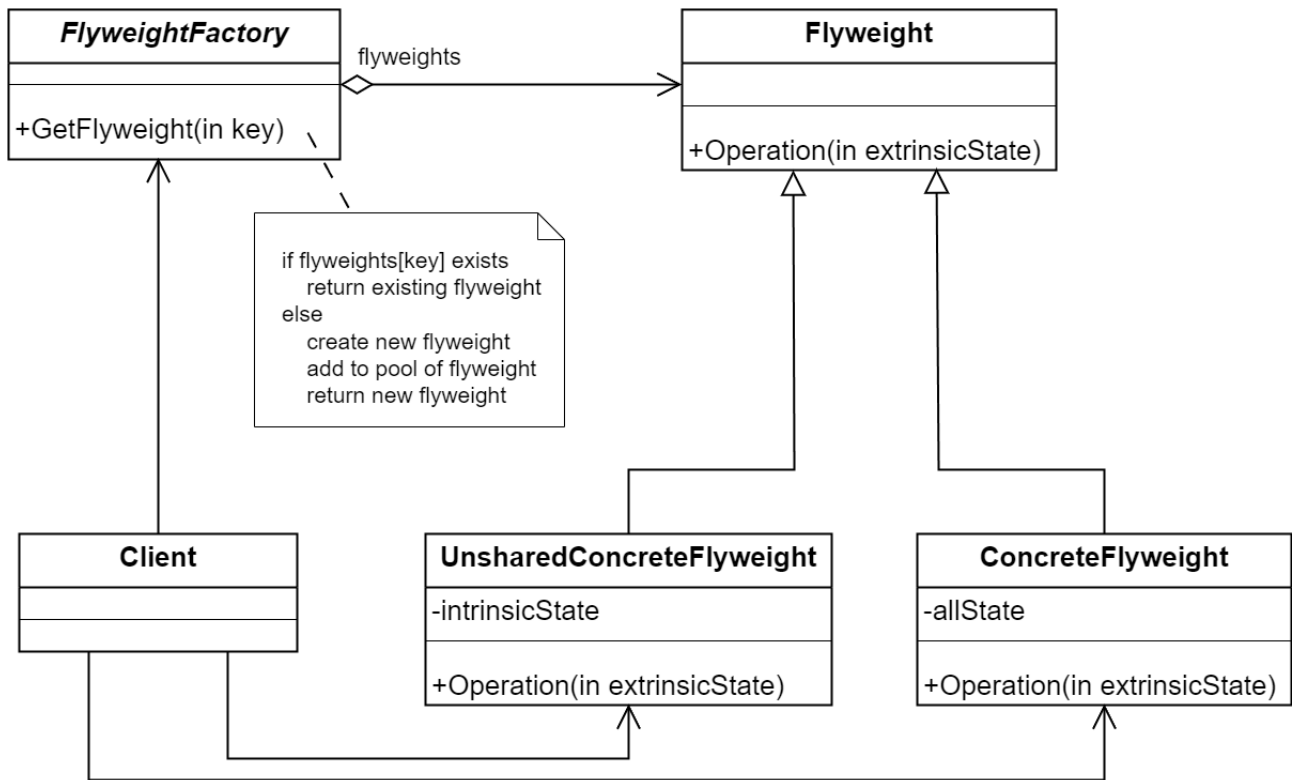
3. Які класи входять в шаблон «Композит», та яка між ними взаємодія?

У Композит входять: **Component** (інтерфейс), **Leaf** (кінцевий об'єкт) та **Composite** (контейнер). **Composite** містить колекцію **Component** і делегує операції дітям.

4. Яке призначення шаблону «Легковаговик»?

Легковаговик зменшує кількість об'єктів шляхом розділення спільного стану між багатьма екземплярами.

5. Нарисуйте структуру шаблону «Легковаговик».



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

У Легковаговику є: **Flyweight** (спільний стан), **ConcreteFlyweight**, **FlyweightFactory** (створює/повертає спільні об'єкти) та **Client** (зберігає зовнішній стан).

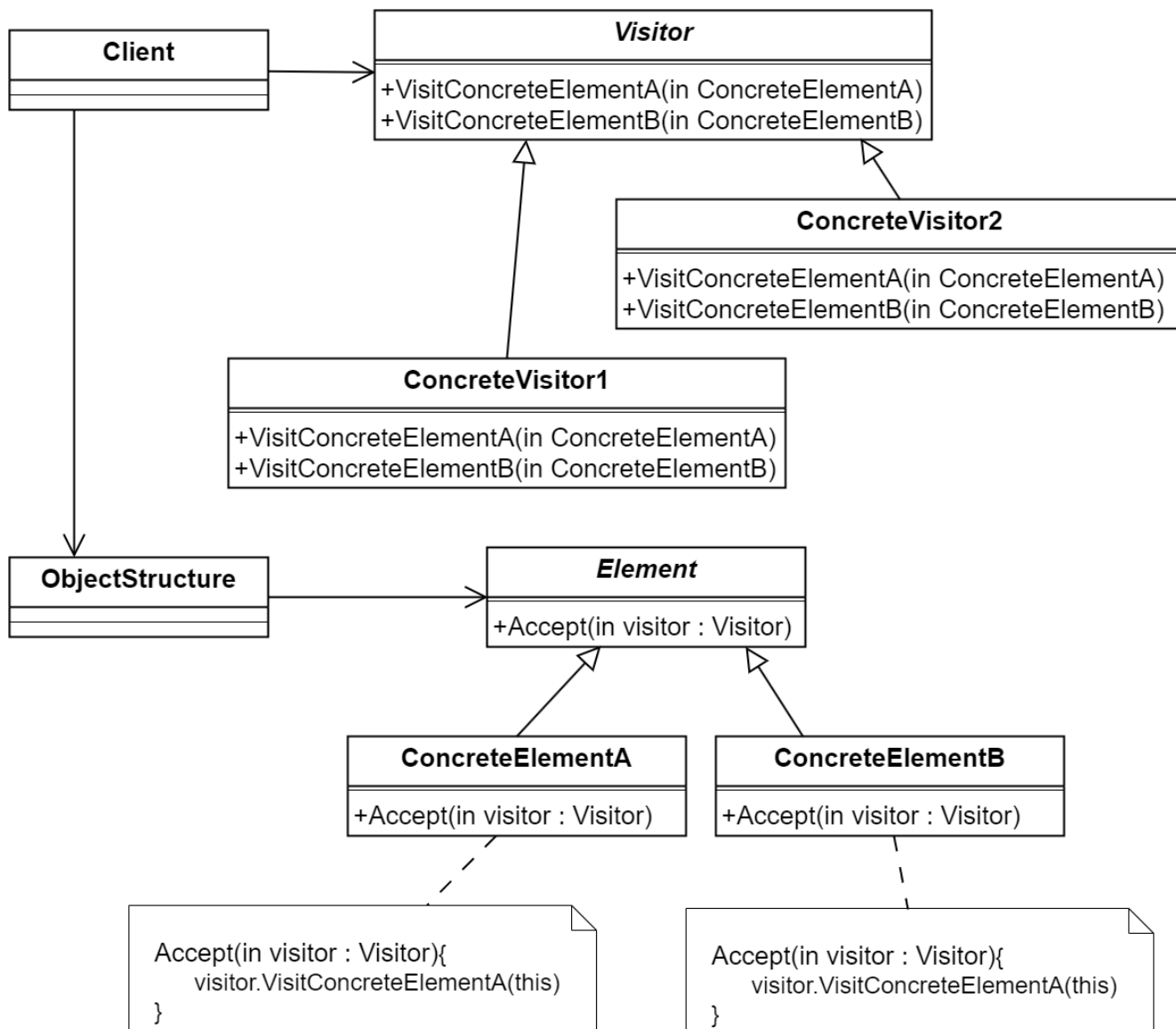
7. Яке призначення шаблону «Інтерпретатор»?

Інтерпретатор визначає граматику мови та інтерпретує її речення, представляючи їх у вигляді абстрактного синтаксичного дерева.

8. Яке призначення шаблону «Відвідувач»?

Відвідувач відокремлює операції від структури об'єктів, дозволяючи додавати нові операції без зміни класів елементів.

9. Нарисуйте структуру шаблону «Відвідувач».



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

У Відвідувачі є: **Element** (приймає **Visitor**), **ConcreteElement**, **Visitor** (інтерфейс з методами для кожного **Element**), **ConcreteVisitor** та **ObjectStructure** (колекція елементів).

Висновки

У ході виконання лабораторної роботи розглянули та реалізували поведінковий патерн проєктування «Інтерпретатор» (Interpreter) у контексті системи автоматизації FlexibleAutomationTool. Застосування даного патерну дозволило визначити формальну граматику мови макросів та реалізувати механізм їх трансформації у виконувані послідовності дій, що забезпечило гнучкий та зручний для користувача спосіб опису автоматизації.

Реалізована архітектура з базовим інтерфейсом IExpression, термінальними виразами (RunExpression, MessageExpression), нетермінальним виразом SequenceExpression, контекстом інтерпретації InterpreterContext та клієнтом-парсером SimpleMacroInterpreter демонструє чітке розділення відповідальностей між компонентами та ефективне використання рекурсивного механізму для побудови абстрактного синтаксичного дерева.

Результати використання підтверджують доцільність використання патерну «Інтерпретатор» у системах, що потребують обробки текстових інструкцій з чіткою структурою. Реалізований підхід забезпечує високу розширюваність: для додавання нових команд у мову макросів достатньо створити новий клас термінального виразу без зміни існуючої логіки інтерпретації.