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
З дисципліни «Технології розроблення програмного забезпечення»
Тема: **«ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»»**
Flexible Automatical Tool

Виконав:
Студент групи ІА-22
Сидорін Д.О.

Перевірив:
Мягкий М. Ю.

Київ-2024

Зміст

Тема:
ШАБЛОНИ «COMPOSITE», «FLYWEIGHT», «INTERPRETER», «VISITOR»

Мета:
Ознайомитися з основними шаблонами проєктування, такими як «Composite», «Flyweight», «Interpreter», «Visitor», дослідити їхні принципи роботи та навчитися використовувати їх для створення гнучкого та масштабованого програмного забезпечення.

Завдання:

Інструмент автоматизації (стратегія, прототип, абстрактна фабрика, міст, композит, SOA)
Десктопний додаток для автоматизації повсякденних завдань із можливістю створення правил (аналогічно сервісу IFTTT), запису макросів (натискання клавіш, дії миші) та використання планувальника завдань. Додаток забезпечує функції, як-от автоматичне завантаження нових серій серіалів, книг чи інших файлів у визначений час, зміну статусів у месенджерах (наприклад, встановлення статусу "відсутній" у Skype при довгій неактивності), а також виконання завдань за розкладом (наприклад, запуск роздачі торрентів о 5 ранку).

Хід роботи

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

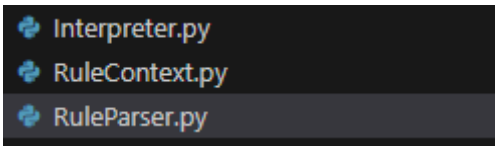

Рис. 1 — Структура проекту

У ході роботи було розроблено наступні класи:

 Interpreter.py (RuleInterpreter) Клас, що є основним інтерпретатором правил. Він відповідає за обробку користувацьких правил і їх виконання. Метод interpret() розбирає текстові правила, перевіряючи умови та параметри, і викликає відповідні дії. Клас також має метод execute_action(), який виконує конкретні дії на основі інтерпретованого правила, наприклад, ініціює автоматичне завантаження файлів чи змінює статус у месенджері.

 RuleContext.py (RuleContext) Клас для збереження та управління контекстом під час виконання правил. Він відповідає за зберігання змінних і умов, які можуть використовуватись в процесі інтерпретації правил. Метод set_variable() дозволяє задавати значення змінних, а метод get_variable() надає доступ до них для подальшого використання під час виконання правил.

 RuleParser.py (RuleParser) Клас для парсингу (розбору) введених правил. Він відповідає за перетворення текстових правил у структури, які можна інтерпретувати. Метод parse() аналізує правило, розбиваючи його на складові частини, в той час як parse_condition() дозволяє обробляти умови, що можуть бути присутніми в правилах, наприклад, для виконання дії при досягненні певного часу чи події.

2. Реалізувати один з розглянутих шаблонів за обраною темою

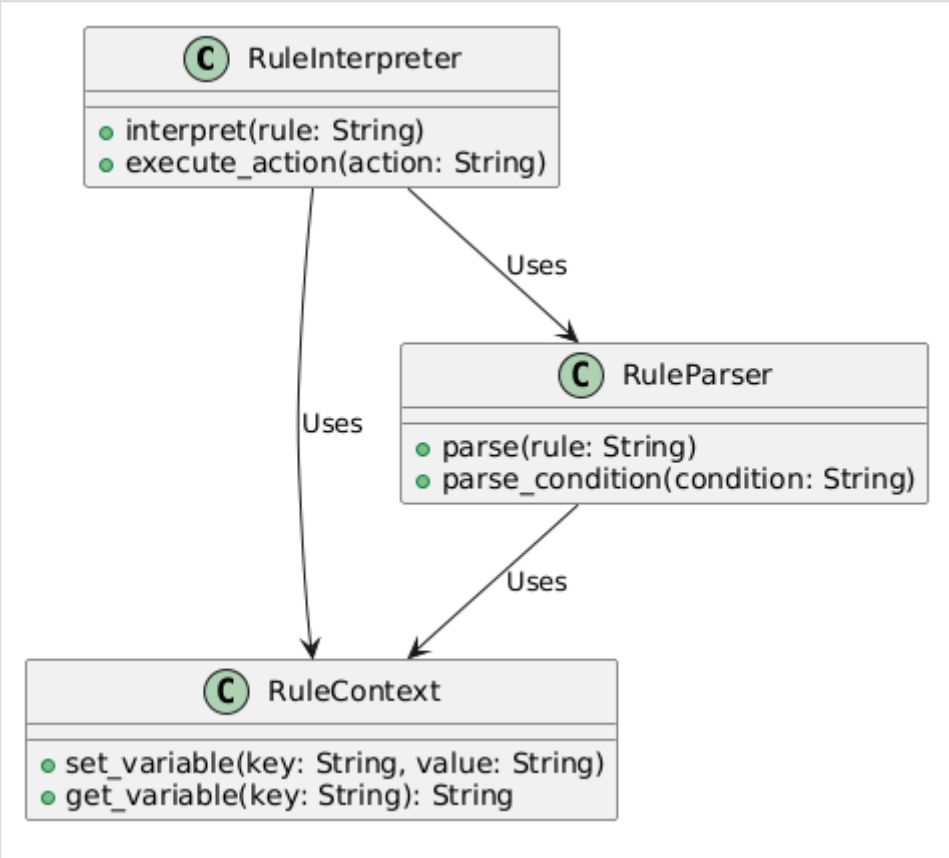

Рис. 2 — Діаграма класів

У контексті проєкту "Інструмент автоматизації" патерн **Interpreter** реалізовано для інтерпретації та виконання правил, визначених користувачем для автоматичних дій. Кожне правило може містити умови, дії або їх комбінації, і всі ці елементи обробляються через єдиний інтерфейс. Клас **RuleInterpreter** відповідає за інтерпретацію правил і виконання відповідних дій. Усі змінні і умови зберігаються в класі **RuleContext**, що дозволяє управляти станом системи під час виконання правил. Клас **RuleParser** займається парсингом введених правил та їх перетворенням у структури, що можуть бути інтерпретовані.

Проблеми, які вирішує патерн Interpreter:

1. **Обробка складних правил** Патерн дозволяє інтерпретувати та виконувати навіть складні правила з умовами і діями, що можуть бути комбіновані за допомогою різних операторів.

2. **Розширюваність та масштабованість** Легко додавати нові типи правил чи змінювати існуючі без значних змін в основній частині системи.
3. **Єдиний підхід до обробки правил** Використання єдиного інтерфейсу для інтерпретації правил дозволяє спростити обробку, навіть коли правила можуть варіюватися за складністю.

Переваги використання патерну Interpreter:

1. **Гнучкість** Користувач може визначати різноманітні правила, комбінуючи умови, дії і параметри, що дає змогу створювати унікальні сценарії автоматизації.
2. **Простота розширення** Легко додавати нові типи умов чи дій, що забезпечує модульність і дає можливість для подальшої еволюції проєкту.
3. **Можливість повторного використання** Інтерпретовані правила та дії можуть бути використані в різних частинах програми без необхідності переписування логіки.
4. **Чітка структура** Завдяки чітко визначеній ролі кожного класу (інтерпретація правил, збереження контексту, парсинг), система стає більш прозорою та легшою для розуміння і підтримки.

Перевірка патерну

```python
class RuleContext:
    def __init__(self):
        self.variables = {}

    def set_variable(self, key, value):
        self.variables[key] = value

    def get_variable(self, key):
        return self.variables.get(key, None)


class RuleParser:
    def parse(self, rule: str):
        """ Парсить правило на дію і умову (якщо є) """
        if "download" in rule:
            return "download"
        elif "status" in rule:
            return "status"
        return None


class RuleInterpreter:
    def __init__(self, context: RuleContext, parser: RuleParser):
        self.context = context
        self.parser = parser

    def interpret(self, rule: str):
        action = self.parser.parse(rule)
        if action == "download":
            self.execute_action("Downloading file...")
        elif action == "status":
            self.execute_action("Changing status...")
        else:
            print("Unknown rule")

    def execute_action(self, action: str):
        """ Виконує дію на основі інтерпретованого правила """
        print(action)



context = RuleContext()
parser = RuleParser()
interpreter = RuleInterpreter(context, parser)


context.set_variable("download_path", "/path/to/download")

# Інтерпретуємо правила
interpreter.interpret("download new movies")
interpreter.interpret("status update in skype")
```

Рис. 3 — Перевірка роботи

У методі main демонструється реалізація патерну **Interpreter** через створення контексту та інтерпретацію правил. Спершу створюється контекст, в якому зберігаються змінні, наприклад, шлях для завантаження файлів. Потім ініціалізується парсер для обробки правил. Далі використовуються інтерпретатор і парсер для інтерпретації двох простих правил: завантаження файлів та зміни статусу в Skype. Правила передаються в інтерпретатор, який виконує відповідні дії, такі як завантаження файлів або оновлення статусу. Цей процес демонструє, як можна автоматизувати завдання за допомогою патерну **Interpreter**.

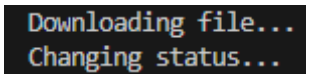

Рис. 4 — Результат роботи

Метод interpret() ітеративно обробляє правила, підтримуючи вкладені структури умов і дій. У результаті програма демонструє, як складні правила можуть бути оброблені через інтерпретацію, зберігаючи чітку організацію та виконання автоматичних дій. Патерн **Interpreter** дозволяє інтерпретувати правила та здійснювати відповідні дії за допомогою класів **RuleParser** та **RuleInterpreter**, зберігаючи при цьому гнучкість у розширенні логіки для нових типів правил.

Висновки:

У цій лабораторній роботі ми реалізували патерн **Interpreter**, щоб ефективно працювати з правилами автоматизації, що складаються з умов та дій. Це дозволяє спростити розробку системи, зробити її більш модульною, зрозумілою та зручною для подальшого розширення з новими типами правил чи дій.

Код: https://github.com/Lepseich/trpz/tree/main/lab8/files