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 №4
З дисципліни «Технології розроблення програмного забезпечення»
Тема: **«ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»»**
Flexible Automatical Tool

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

Перевірив:
Мягкий М. Ю.

Київ-2024

Зміст

Тема:
ШАБЛОНИ «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»
Мета:
Ознайомитися з основними шаблонами проєктування, такими як «Singleton», «Iterator», «Proxy», «State» та «Strategy», дослідити їхні принципи роботи та навчитися використовувати для створення гнучкого та масштабованого програмного забезпечення.

Завдання:

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

Хід роботи

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

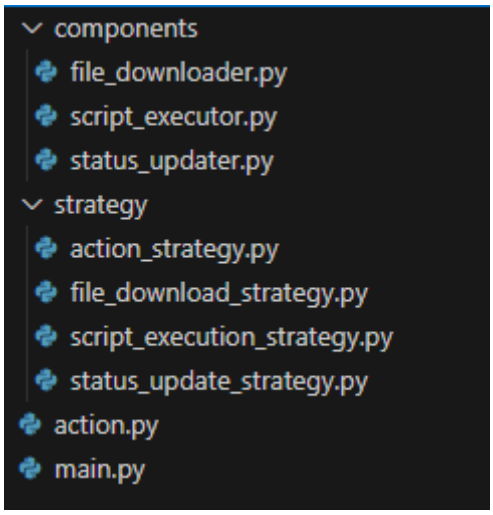


Рис. 1 — Структура проекту

Опис класів:

ActionStrategy (abstract class)

- **Опис**: Абстрактний клас для стратегії дії. Містить загальний метод execute(), який повинні реалізувати конкретні стратегії.
- **Методи**:
  - execute(): абстрактний метод, який визначає, як виконувати дію. Має бути реалізований у підкласах.

2. FileDownloadStrategy

- **Опис**: Конкретна реалізація стратегії для завантаження файлів.
- **Конструктор**:

    - __init__(file_downloader: FileDownloader): ініціалізує стратегію з об'єктом класу FileDownloader.
- **Методи**:
    - execute(): виконує завантаження файлу за допомогою об'єкта FileDownloader.

3. StatusUpdateStrategy

- **Опис**: Конкретна реалізація стратегії для оновлення статусу.
- **Конструктор**:
    - __init__(status_updater: StatusUpdater): ініціалізує стратегію з об'єктом класу StatusUpdater.
- **Методи**:
    - execute(): оновлює статус через об'єкт StatusUpdater.

4. ScriptExecutionStrategy

- **Опис**: Конкретна реалізація стратегії для виконання скриптів.
- **Конструктор**:
    - __init__(script_executor: ScriptExecutor): ініціалізує стратегію з об'єктом класу ScriptExecutor.
- **Методи**:
    - execute(): виконує скрипт за допомогою об'єкта ScriptExecutor.

5. FileDownloader

- **Опис**: Клас для завантаження файлів.
- **Методи**:
    - download_file(url: str): завантажує файл за заданою URL-адресою.

6. StatusUpdater

- **Опис**: Клас для оновлення статусу.
- **Методи**:
    - update_status(status: str): змінює статус на заданий.

7. ScriptExecutor

- **Опис**: Клас для виконання скриптів.
- **Методи**:
    - execute_script(script: str): виконує заданий скрипт.

8. Action

- **Опис**: Клас для виконання дії з використанням стратегії.
- **Конструктор**:

- __init__(strategy): ініціалізує екземпляр класу з конкретною стратегією.

- **Методи**:
  - execute_action(): викликає метод execute() стратегії для виконання відповідної дії.

9. Main

- **Опис**: Головний клас для запуску програми.
- **Методи**:
  - main(): точка входу програми, де створюються об'єкти компонентів та стратегій, а також виконується кожна дія через Action.

2. Реалізувати один з розглянутих шаблонів за обраною темою

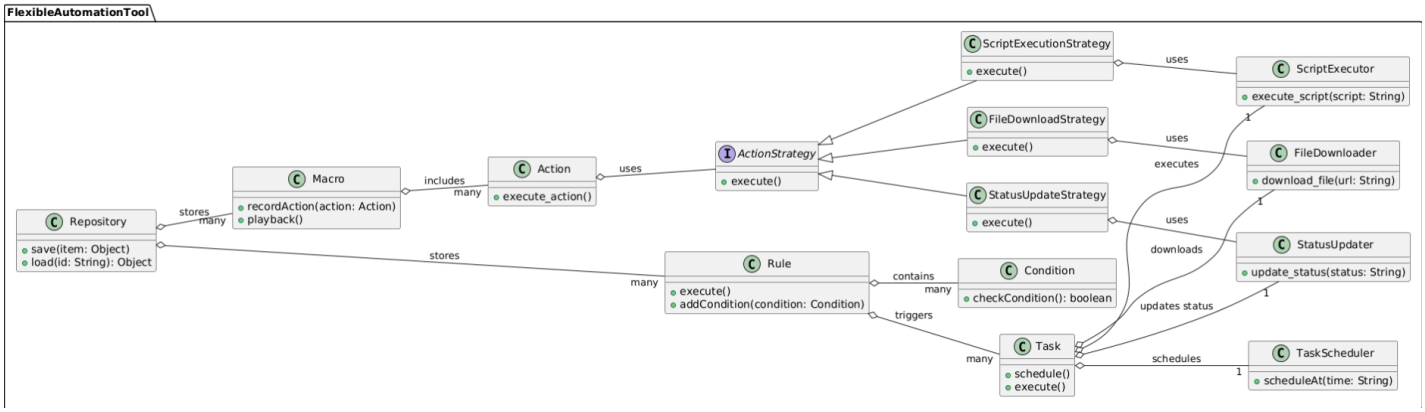


Рис. 2 — Діаграма класів

Strategy — це структурний патерн проєктування, який дозволяє змінювати поведінку об'єкта в залежності від вибраної стратегії без необхідності змінювати код цього об'єкта. Ключова ідея — створити набір взаємозамінних стратегій для виконання певної задачі та дозволити клієнту вибирати відповідну стратегію для кожного випадку.

Як це працює в проєкті:

У цьому проєкті патерн **Strategy** застосовано для визначення різних способів виконання певних дій (наприклад, завантаження файлів, оновлення статусів або виконання скриптів).

1. **ActionStrategy** — інтерфейс, що визначає загальний метод **execute()**, який реалізують конкретні стратегії.
2. Кожна стратегія (наприклад, **FileDownloadStrategy**, **StatusUpdateStrategy**, **ScriptExecutionStrategy**) містить конкретну реалізацію методу **execute()**, який визначає, як саме буде виконано завдання (завантаження файлів, оновлення статусу або виконання скрипта).
3. Клас **Action** використовує ці стратегії, викликаючи метод **execute()** обраної стратегії для виконання відповідної дії.

4. **Task** може мати один із видів дій, що залежить від обраної стратегії.
5. Стратегії взаємодіють із класами **FileDownloader**, **StatusUpdater**, **ScriptExecutor**, що відповідають за виконання конкретних операцій.

Проблеми, які вирішує патерн Strategy:

1. **Жорстка залежність від конкретних реалізацій**:
   - Без патерну **Strategy** ми б мали одну велику ієрархію класів з умовними операціями, які б вибиралися залежно від контексту. Це створює проблему підтримки та розширення системи.
   - Патерн **Strategy** дозволяє створити набір окремих стратегій, кожна з яких реалізує конкретний спосіб виконання дії. Це знижує зв'язність між класами та забезпечує гнучкість у виборі стратегії.
2. **Розширення функціональності**:
   - Якщо необхідно додати нову стратегію для виконання іншої дії (наприклад, новий спосіб завантаження файлів), то можна легко створити новий клас стратегії без необхідності змінювати існуючий код.
3. **Великий обсяг умов у класах**:
   - Без патерну довелося б додавати багато умовних операторів, що б вибирали потрібний алгоритм у залежності від ситуації (наприклад, if-else або switch). Це може призвести до важкості розуміння та тестування коду. Патерн **Strategy** дозволяє уникнути цих умов, оскільки вибір стратегії відбувається через делегування, а не через умови.
4. **Невикористання спільних інтерфейсів**:
   - Різні класи, що мають різні алгоритми, можуть мати однаковий інтерфейс, що дозволяє використовувати їх взаємозамінно.

Переваги використання паттерну Strategy:

1. **Гнучкість**:
   - Ви можете змінювати поведінку об'єкта в процесі роботи, просто змінюючи стратегію, не змінюючи сам об'єкт. Це дає гнучкість у налаштуванні різних варіантів поведінки.
2. **Розширюваність**:
   - Легко додавати нові стратегії. Якщо потрібно змінити спосіб виконання якоїсь дії (наприклад, завантаження файлів, оновлення статусу), достатньо створити нову стратегію без модифікації вже існуючого коду.
3. **Знижена зв'язність**:
   - Використання патерну дозволяє знизити зв'язність між класами. Наприклад, клас **Action** не залежить від конкретної реалізації стратегії (завантаження файлів, оновлення статусу), а лише від інтерфейсу **ActionStrategy**.
4. **Покращена підтримка та тестування**:

- Кожна стратегія реалізує один конкретний алгоритм. Тому код можна тестувати на рівні окремих стратегій, що спрощує підтримку та покращує покриття тестами.

5. **Легкість у підтримці коду**:
   - Код стає чистішим і легшим для розуміння, оскільки логіка кожної стратегії розділяється на окремі класи, що відповідають лише за свою частину задачі.
6. **Уникнення великої кількості умовних операторів**:
   - Патерн дозволяє уникнути використання великої кількості умов (if-else, switch), що часто призводить до заплутаності та важкості підтримки коду.

Перевірка патерну

```python
import unittest

class FileDownloadStrategy:
    def execute(self):
        return "Downloading file..."

class StatusUpdateStrategy:
    def execute(self):
        return "Updating status..."

class ScriptExecutionStrategy:
    def execute(self):
        return "Executing script..."


class Action:
    def __init__(self, strategy):
        self.strategy = strategy

    def execute_action(self):
        return self.strategy.execute()


class TestStrategyPattern(unittest.TestCase):

    def test_file_download_strategy(self):
        action = Action(FileDownloadStrategy())
        result = action.execute_action()
        self.assertEqual(result, "Downloading file...")

    def test_status_update_strategy(self):
        action = Action(StatusUpdateStrategy())
        result = action.execute_action()
        self.assertEqual(result, "Updating status...")

    def test_script_execution_strategy(self):
        action = Action(ScriptExecutionStrategy())
        result = action.execute_action()
        self.assertEqual(result, "Executing script...")

    def test_strategy_switch(self):
        # Перемикання стратегій
        action = Action(FileDownloadStrategy())
        result = action.execute_action()
        self.assertEqual(result, "Downloading file...")

        action.strategy = StatusUpdateStrategy()
        result = action.execute_action()
        self.assertEqual(result, "Updating status...")

        action.strategy = ScriptExecutionStrategy()
        result = action.execute_action()
        self.assertEqual(result, "Executing script...")

if __name__ == '__main__':
    unittest.main()
```

Рис. 3 — Перевірка роботи

Опис

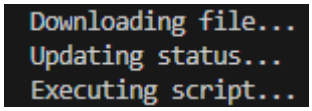
1. **Код створює об'єкт Action**, який виступає як контекст для виконання різних стратегій. Він приймає стратегію як параметр і викликає метод **execute()** для виконання конкретної операції.
2. **По черзі встановлюються різні стратегії**:
    - **FileDownloadStrategy**: для завантаження файлів.

- **StatusUpdateStrategy**: для оновлення статусу в різних додатках.
- **ScriptExecutionStrategy**: для виконання скриптів або автоматизації процесів.

3. **Для кожної стратегії викликається метод execute_action()**, який запускає відповідну стратегію через метод **execute()** та виконує завдання згідно з вибраною стратегією.

Вивід програми:

Очікуваний результат у терміналі:



Висновок:

У рамках лабораторної роботи було реалізовано паттерн **Strategy**, який дозволяє змінювати поведінку об'єктів без зміни їхнього основного коду. Для цього створено три стратегії, що відповідають за різні завдання: завантаження файлів, оновлення статусу та виконання скриптів. Клас **Action** використовує ці стратегії для виконання відповідних дій, що дозволяє легко змінювати поведінку програми в залежності від вибраної стратегії.

У процесі виконання роботи було проведено тестування, яке підтвердило, що кожна стратегія працює коректно, виводячи правильні повідомлення у терміналі. Завдяки використанню паттерну **Strategy**, система стала гнучкішою, що дозволяє без труднощів додавати нові стратегії без змін у основному коді.

Таким чином, лабораторна робота показала ефективність використання паттерну **Strategy** для забезпечення гнучкості та масштабованості програми при виконанні різних завдань.

Код: https://github.com/Lepseich/trpz/tree/main/lab4