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

```python
from abc import ABC, abstractmethod

# Загальний інтерфейс стратегій
class Strategy(ABC):
    @abstractmethod
    def execute(self, task_details):
        pass

# Конкретні стратегії для завдань автоматизації
class ConcreteStrategyDownload(Strategy):
    def execute(self, task_details):
        return f"Завантаження файлів за планом: {task_details['url']}"

class ConcreteStrategySetStatus(Strategy):
    def execute(self, task_details):
        return f"Встановлення статусу: {task_details['status']} в {task_details['messenger']}"

class ConcreteStrategySchedule(Strategy):
    def execute(self, task_details):
        return f"Запуск завдання за розкладом: {task_details['time']}"

# Контекст для виконання завдань
class TaskContext:
    def __init__(self, strategy: Strategy = None):
        self.strategy = strategy

    def set_strategy(self, strategy: Strategy):
        self.strategy = strategy

    def execute_strategy(self, task_details):
        if not self.strategy:
            raise ValueError("Стратегія не встановлена")
        return self.strategy.execute(task_details)
```

```python
# Приклад роботи програми
class AutomationApp:
    def main(self):
        # 1. Створити контекст для завдання
        context = TaskContext(None)

        # 2. Вибір стратегії та передача деталей завдання
        task_details = {
            'url': 'https://example.com/video.mp4',
            'status': 'Офлайн',
            'messenger': 'Skype',
            'time': '05:00'
        }

        # Вибір стратегії
        action = input("Виберіть тип завдання (download/status/schedule): ").lower()

        if action == 'download':
            context.set_strategy(ConcreteStrategyDownload())
        elif action == 'status':
            context.set_strategy(ConcreteStrategySetStatus())
        elif action == 'schedule':
            context.set_strategy(ConcreteStrategySchedule())
        else:
            print("Невідома операція.")
            return

        # 3. Виконати завдання за допомогою стратегії
        return context.execute_strategy(task_details)
```

Опис класів:

1. Strategy (Абстрактний клас)

- **Опис**: Це абстрактний клас, що визначає загальний інтерфейс для всіх стратегій. Кожна стратегія повинна реалізувати метод execute, який приймає деталі завдання та виконує необхідну дію, залежно від реалізації конкретної стратегії. Клас **Strategy** є основою для створення конкретних стратегій для автоматизації різних завдань.
- **Методи**:
    - execute(task_details): абстрактний метод, що приймає параметр task_details (деталі завдання) та виконує відповідну операцію, яку реалізує конкретна стратегія. У кожному класі-нащадку цей метод буде мати свою логіку.

2. ConcreteStrategyDownload

- **Опис**: Конкретна стратегія для завантаження файлів. Вона реалізує метод execute, що здійснює завантаження файлів за наданою URL-адресою. Цей клас відповідає за виконання завдання, пов'язаного із завантаженням файлів (наприклад, завантаження відео чи інших ресурсів).
- **Методи**:
    - execute(task_details): Приймає task_details (наприклад, URL для завантаження) і виконує завантаження файлів за зазначеною адресою.

3. ConcreteStrategySetStatus

- **Опис**: Конкретна стратегія для встановлення статусів у месенджерах (наприклад, Skype). Вона реалізує метод execute, що змінює статус користувача в залежності від параметрів, переданих через task_details (наприклад, статус і месенджер). Цей клас дозволяє автоматизувати процес зміни статусу в комунікаторах.
- **Методи**:
  - execute(task_details): Приймає task_details, які містять інформацію про статус і месенджер, і змінює статус користувача.

4. ConcreteStrategySchedule

- **Опис**: Конкретна стратегія для виконання завдань за розкладом. Вона реалізує метод execute, який забезпечує запуск завдання в зазначений час. Цей клас використовується для автоматизації процесу запуску завдань у заданий час.
- **Методи**:
  - execute(task_details): Приймає task_details, які містять інформацію про час запуску завдання, і запускає завдання в зазначений час.

5. TaskContext

- **Опис**: Клас **TaskContext** зберігає поточну стратегію і забезпечує її виконання. Це клас, який працює з різними стратегіями для виконання завдань. Він дозволяє змінювати стратегію в процесі виконання програми, а також виконувати конкретне завдання через вибрану стратегію.
- **Атрибути**:
  - strategy: Об'єкт класу, що реалізує інтерфейс **Strategy**. Це конкретна стратегія, яку обирає користувач або система для виконання завдання.
- **Методи**:
  - set_strategy(strategy: Strategy): Задає стратегію для виконання завдання. Цей метод дозволяє змінювати стратегію на іншу під час роботи програми.
  - execute_strategy(task_details): Виконує задачу через поточну стратегію. Приймає task_details, які містять деталі завдання, і передає їх обраній стратегії для виконання.

6. AutomationApp

- **Опис**: Клас **AutomationApp** є головним класом програми, що ініціює виконання завдань. Він взаємодіє з користувачем, запитує вибір завдання та передає його контексту. Цей клас служить для взаємодії з користувачем, вибору стратегії та виконання завдань за допомогою **TaskContext**.
- **Методи**:

- main(): Основний метод програми. Викликається для запуску програми, запитує у користувача тип завдання, вибирає відповідну стратегію, передає її контексту та виконує завдання. Це центральний метод для роботи користувача з програмою.

2. Реалізувати один з розглянутих шаблонів за обраною темою

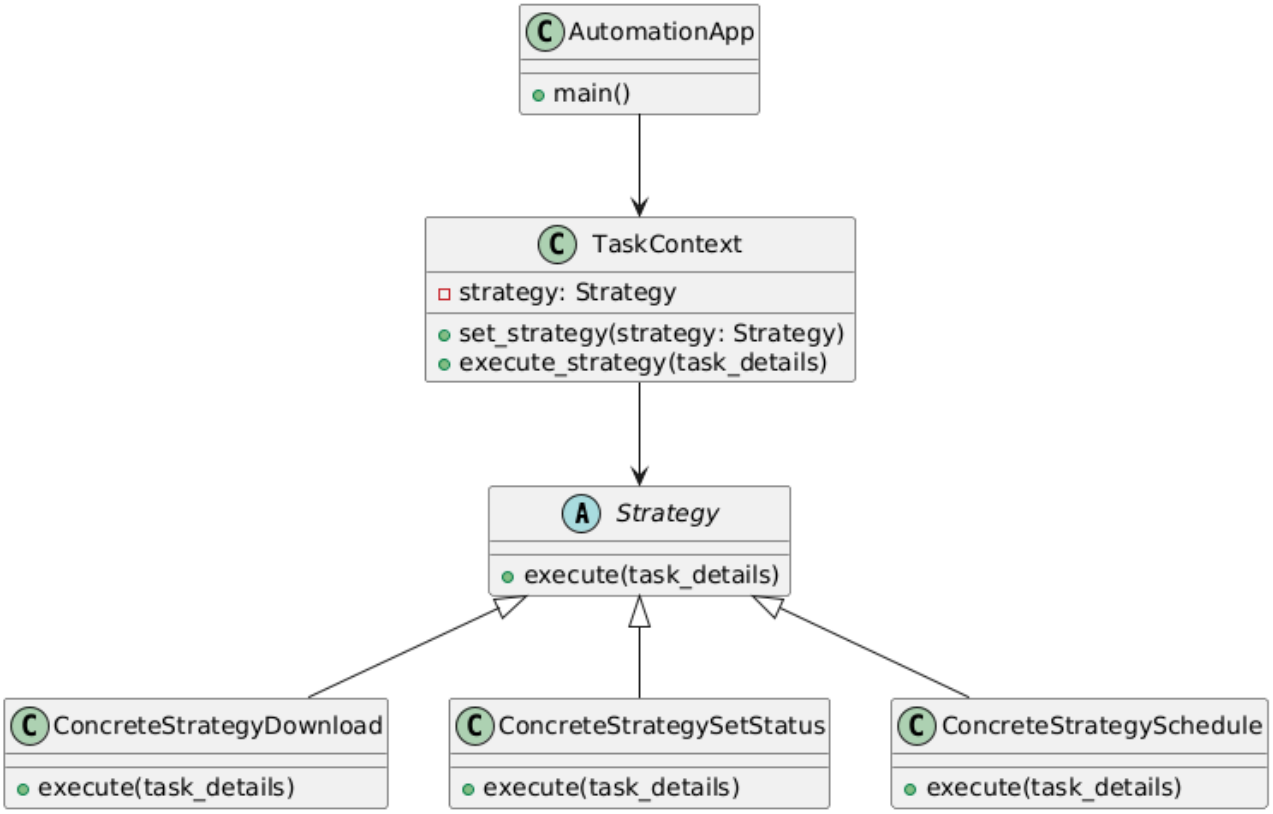

Рис. 2 — Діаграма класів

Strategy (Абстрактний клас)

- **Опис**: Це абстрактний клас, що визначає загальний інтерфейс для всіх стратегій. Він містить метод execute, який має бути реалізований у конкретних класах стратегій. Всі стратегії, які наслідують цей клас, повинні реалізовувати цей метод для виконання специфічних дій (наприклад, завантаження файлів, встановлення статусів або планування задач).
- **Методи**:
  - execute(task_details): абстрактний метод, що приймає деталі завдання і виконує відповідну операцію (завантаження, встановлення статусу, запуск за розкладом).

2. ConcreteStrategyDownload

- **Опис**: Конкретна стратегія для автоматизованого завантаження файлів. Вона реалізує метод execute, який виконує завдання завантаження в залежності від деталей, переданих через task_details.
- **Методи**:

- execute(task_details): здійснює завантаження файлів з наданої URL-адреси.

3. ConcreteStrategySetStatus

- **Опис**: Конкретна стратегія для встановлення статусів у комунікаторах (наприклад, Skype). Вона реалізує метод execute, який змінює статус в залежності від параметрів у task_details.
- **Методи**:
  - execute(task_details): змінює статус користувача в певному месенджері (Skype, Telegram і т.д.).

4. ConcreteStrategySchedule

- **Опис**: Конкретна стратегія для автоматичного запуску завдань за розкладом. Вона реалізує метод execute, який забезпечує запуск задачі в зазначений час.
- **Методи**:
  - execute(task_details): здійснює запуск завдання за розкладом.

5. TaskContext

- **Опис**: Клас, який працює з різними стратегіями. Він зберігає поточну стратегію і може змінювати її за допомогою методу set_strategy. Контекст виконує завдання через метод execute_strategy, передаючи деталі завдання конкретній стратегії.
- **Атрибути**:
  - strategy: об'єкт класу, що реалізує інтерфейс Strategy. Це конкретна стратегія, яка виконується.
- **Методи**:
  - set_strategy(strategy: Strategy): дозволяє змінити стратегію на нову.
  - execute_strategy(task_details): виконує задачу через поточну стратегію.

6. AutomationApp

- **Опис**: Головний клас програми, що ініціює виконання завдань. Він взаємодіє з користувачем для вибору типу завдання та визначає відповідну стратегію для виконання.
- **Методи**:
  - main(): основний метод програми, що запитує у користувача тип завдання та виконує його за допомогою відповідної стратегії.

Як це працює в проєкті:

1. **Контекст завдання**: Клас **TaskContext** зберігає поточну стратегію та виконує операцію через метод **execute_strategy**. Це дозволяє замінювати стратегії в процесі виконання програми без зміни основного коду.
2. **Зміна стратегії**: У **AutomationApp** програма запитує у користувача тип завдання (завантаження файлів, встановлення статусу або запуск завдання за розкладом), після чого встановлює відповідну стратегію через метод **set_strategy**.
3. **Виконання завдання**: Після вибору стратегії виконується відповідне завдання, наприклад, завантаження файлів через **ConcreteStrategyDownload** або встановлення статусу через **ConcreteStrategySetStatus**.

Проблеми, які вирішує патерн Strategy:

1. **Залежність від конкретної реалізації**: Використання стратегії дозволяє змінювати реалізацію певної операції (наприклад, завантаження файлів або налаштування статусу) без зміни коду, що використовує ці операції. Це знижує залежність між компонентами.
2. **Розширення функціональності**: Легко додавати нові стратегії для виконання різних завдань без необхідності модифікації існуючого коду. Нові стратегії можна додавати, просто реалізувавши новий клас, що успадковує від Strategy.
3. **Зменшення складності**: Оскільки кожна стратегія реалізує свою окрему логіку, це дозволяє уникнути великої кількості умовних операторів у коді і спрощує його.

Переваги використання патерну Strategy:

1. **Гнучкість**: Патерн дозволяє змінювати поведінку об'єкта без зміни його коду. Можна змінити стратегію в будь-який час під час виконання програми, що дає високу гнучкість.
2. **Масштабованість**: Легко додавати нові стратегії. Якщо з'явиться новий тип завдання, можна просто додати нову стратегію без необхідності змінювати існуючі класи.
3. **Чистота коду**: Розбиття коду на окремі стратегії дозволяє зберігати код чистим і легко зрозумілим. Логіка виконання кожної стратегії ізольована в окремих класах, що полегшує підтримку і тестування.
4. **Покращення тестування**: Кожна стратегія є окремим класом, що полегшує тестування. Можна протестувати кожну стратегію незалежно від інших частин програми.

Перевірка патерну

test_strategy.py

test_strategy.py > TestStrategyPattern > setUp

```python
import unittest
from strategy import ConcreteStrategyDownload, ConcreteStrategySetStatus, ConcreteStrategySchedule, TaskContext

class TestStrategyPattern(unittest.TestCase):

    def setUp(self):
        """Ініціалізація перед кожним тестом"""
        self.download_strategy = ConcreteStrategyDownload()
        self.set_status_strategy = ConcreteStrategySetStatus()
        self.schedule_strategy = ConcreteStrategySchedule()

        self.task_details = {
            'url': 'https://example.com/video.mp4',
            'status': 'Офлайн',
            'messenger': 'Skype',
            'time': '05:00'
        }

        # Створюємо контекст
        self.context = TaskContext()

    def test_download_strategy(self):
        """Тестуємо стратегію завантаження"""
        self.context.set_strategy(self.download_strategy)
        result = self.context.execute_strategy(self.task_details)
        self.assertEqual(result, "Завантаження файлів за планом: https://example.com/video.mp4")

    def test_set_status_strategy(self):
        """Тестуємо стратегію встановлення статусу"""
        self.context.set_strategy(self.set_status_strategy)
        result = self.context.execute_strategy(self.task_details)
        self.assertEqual(result, "Встановлення статусу: Офлайн в Skype")

    def test_schedule_strategy(self):
        """Тестуємо стратегію запуску завдання за розкладом"""
        self.context.set_strategy(self.schedule_strategy)
        result = self.context.execute_strategy(self.task_details)
        self.assertEqual(result, "Запуск завдання за розкладом: 05:00")

    def test_strategy_change(self):
        """Тестуємо зміну стратегії"""
        self.context.set_strategy(self.download_strategy)
        download_result = self.context.execute_strategy(self.task_details)

        self.context.set_strategy(self.set_status_strategy)
        status_result = self.context.execute_strategy(self.task_details)

        self.assertNotEqual(download_result, status_result)
        self.assertEqual(download_result, "Завантаження файлів за планом: https://example.com/video.mp4")
        self.assertEqual(status_result, "Встановлення статусу: Офлайн в Skype")

    def test_no_strategy_set(self):
        """Тестуємо, що станеться, якщо не встановлено стратегію"""
        with self.assertRaises(ValueError):
            self.context.execute_strategy(self.task_details)

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

Рис. 3 — Перевірка роботи

Опис:

Основний процес:

1. **Ініціалізація об'єктів**: У кожному тесті створюються потрібні об'єкти стратегій і контексту.

2. **Вибір стратегії**: Вибрана стратегія передається в контекст через метод **set_strategy()**.
3. **Виконання стратегії**: Метод **execute_strategy()** в контексті виконує відповідну стратегію.
4. **Перевірка результатів**: За допомогою **assertEqual()** перевіряється, чи правильно виконано завдання.

Вивід програми:

Очікуваний результат у терміналі:

```
PS C:\trpz\lab4\files new> python -m unittest test_strategy.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Код перевірки тестує роботу патерну Strategy: він перевіряє, чи кожна стратегія правильно виконує свою задачу (завантаження файлів, встановлення статусів, запуск за розкладом), чи може контекст змінювати стратегію під час виконання, та чи генерується помилка, коли стратегія не задана. Тести допомагають забезпечити коректність і надійність роботи

Висновок:

У ході виконання лабораторної роботи було реалізовано та перевірено патерн **Strategy** в контексті автоматизації завдань. Патерн **Strategy** дозволяє динамічно змінювати алгоритми виконання завдань без необхідності змінювати код контексту, що забезпечує високу гнучкість і розширюваність програми.

У результаті виконання роботи:

1. Було створено кілька стратегій для різних типів завдань: завантаження файлів, встановлення статусів у месенджерах, запуск завдань за розкладом.
2. Реалізовано контекст, який взаємодіє з конкретними стратегіями та дозволяє їх змінювати в процесі виконання.
3. Створено юніт-тести, які перевіряють коректність роботи кожної стратегії, забезпечують правильність виконання завдань та перевіряють зміну стратегії в контексті.

Результати тестування показали, що патерн **Strategy** працює коректно, дозволяючи змінювати стратегії в залежності від типу завдання, що робить програму більш гнучкою та легкою в підтримці. Таким чином, використання цього патерну дозволяє спростити структуру коду та зменшити кількість дублювання логіки, що є важливим аспектом у розробці масштабованих і підтримуваних програм.

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