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»»

Варіант №18

Виконав:
студент групи ІА-24
Гуменюк К. Е.

Перевірив:
Мягкий Ю. М.

Київ 2025

Зміст

Contents

Тема.
Шаблони «SINGLETON», «ITERATOR», «PROXY», «STATE», «STRATEGY»

Мета.
Метою даної лабораторної роботи є ознайомлення з шаблонами проєктування, зокрема з шаблоном "State", та їх практичне застосування при розробці програмного забезпечення.

Завдання.
1 . Ознайомитися з короткими теоретичними відомостями.
2. Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.
3. Застосування одного з розглянутих шаблонів при реалізації програми

Обрана тема.
Shell (total commander) (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі - перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Короткі теоретичні відомості.

Шаблони проєктування (або патерни) — це перевірені практикою рішення загальних проблем проєктування програмного забезпечення. Вони описують стандартні підходи, які можна застосувати в різних ситуаціях, щоб вирішити певні проблеми, зберігаючи при цьому структуру системи зрозумілою та підтримуваною. Шаблони проєктування мають загальновживані назви, що дозволяє розробникам легко обговорювати та впроваджувати відповідні підходи. Приклади шаблонів включають Singleton, Factory, Observer, Command, Iterator тощо.

Основні переваги шаблонів проєктування:

- Зменшення часу та зусиль на створення архітектури.
- Гнучкість та адаптованість системи до змін.
- Полегшення підтримки та розвитку системи.
- Стійкість системи до змін та спрощення інтеграції з іншими системами.

Патерн "Стан" (State) є поведінковим патерном проектування, який дозволяє об'єкту змінювати свою поведінку залежно від свого внутрішнього стану. В цьому випадку було реалізовано консольний додаток, що моделює роботу оболонки на зразок Total Commander, і використовує патерн "Стан" для керування різними режимами роботи.

Переваги:

1) Позбавляє від безлічі великих умовних операторів машини станів.
2) Концентрує в одному місці код, пов’язаний з певним станом.
3) Спрощує код контексту.

Недоліки:

1) Може невиправдано ускладнити код, якщо станів мало, і вони рідко змінюються.

Хід роботи.

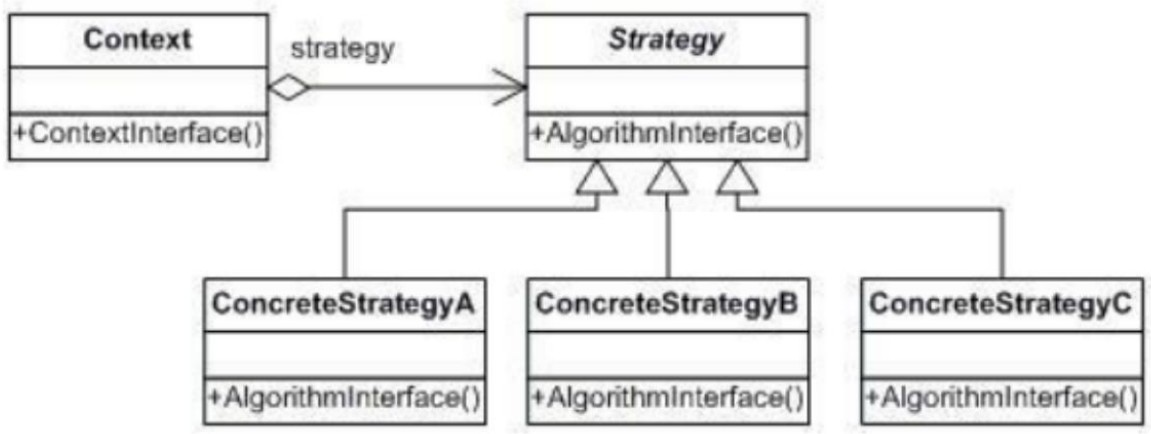

Рисунок 1.1 – UML діаграма шаблону State

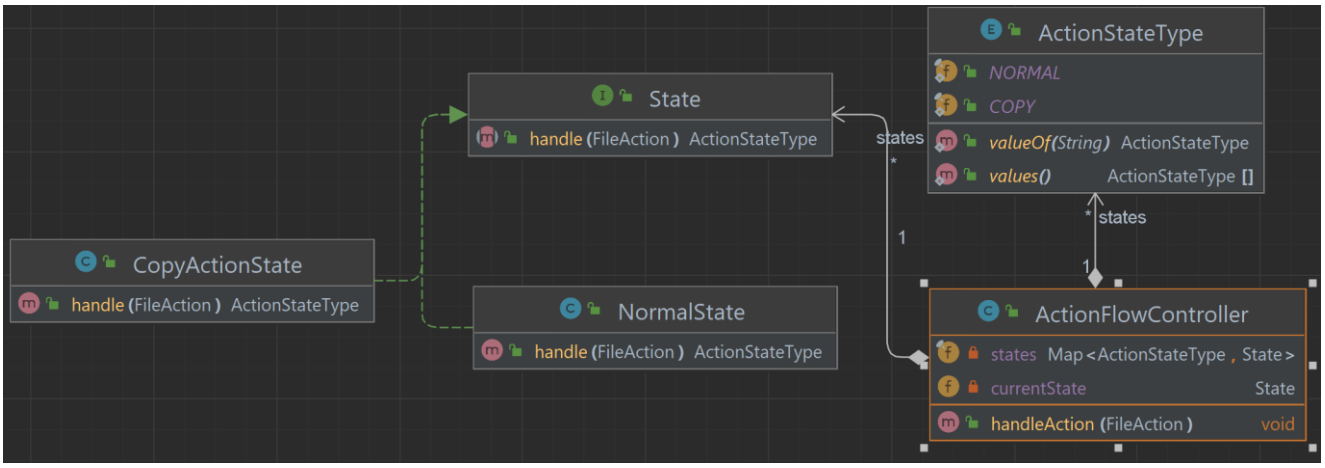

Рисунок 1.2 – Діаграма класів , згенерована IDE

Інтерфейс State:

- Визначає метод handle, який приймає команду FileAction і повертає StateType, що вказує на наступний стан.
- Це дозволяє кожному стану обробляти команди та визначати, чи потрібно перейти до іншого стану.

Перерахунок ActionStateType:

- Містить перелік можливих станів системи (NORMAL, COPY).
- Використовується для ідентифікації та переходу між станами.

Класи станів (NormalState, CopyActionState):

- NormalState: Представляє стан за замовчуванням, у якому доступні основні команди, такі як перегляд файлів, пошук, видалення тощо.
- CopyActionState: Стан копіювання, в якому обробляється команда копіювання файлів.
- Кожен стан реалізує інтерфейс State і визначає, які команди він може обробляти.

Клас ActionFlowController:

- Відповідає за керування поточним станом системи.
- Метод handleCommand передає команду поточному стану та оновлює поточний стан залежно від результату.
- Містить мапу states, яка зберігає відповідність між StateType та екземплярами станів.

Висновок.

Використання патерну **State** в темі Shell Total Commander є ефективним рішенням для організації поведінки системи залежно від її поточного стану. Завдяки цьому патерну, програма стає більш гнучкою та розширюваною: замість численних умовних конструкцій, кожен стан має свій окремий клас із відповідною поведінкою. Це покращує читабельність коду, спрощує підтримку та дозволяє легко додавати нові стани без значної модифікації існуючого коду.

ДОДАТОК

```java
@Component
public class ActionFlowController {
    4 usages
    private final Map<ActionStateType, State> states = new EnumMap<>(ActionStateType.class);
    7 usages
    private State currentState;

    @Autowired
    public ActionFlowController(NormalState normalState, CopyActionState copyActionState) {
        states.put(ActionStateType.NORMAL, normalState);
        states.put(ActionStateType.COPY, copyActionState);
        currentState = normalState;
    }

    1 usage
    public void handleCommand(FileAction FileAction) {
        ActionStateType nextActionStateType = currentState.handle(FileAction);
        State nextState = states.get(nextActionStateType);

        if (nextState != null && nextState != currentState) {
            currentState = nextState;
            nextActionStateType = currentState.handle(FileAction);
            nextState = states.get(nextActionStateType);
            if (nextState != null && nextState != currentState) {
                currentState = nextState;
            }
        }
    }
```

```java
public enum ActionStateType {
    4 usages
    NORMAL,
    3 usages
    COPY
}
```

```java
@Component
public class CopyActionState implements State {
    no usages
    private static final Logger logger = LoggerFactory.getLogger(CopyActionState.class);

    2 usages
    @Override
    public ActionStateType handle(FileAction FileAction) {
        if (FileAction instanceof CopyFileAction) {
            FileAction.run();
            return ActionStateType.NORMAL;

        } else {
            return ActionStateType.COPY;
        }
    }
}

@Component
public class NormalState implements State {

    2 usages
    @Override
    public ActionStateType handle(FileAction FileAction) {
        if (FileAction instanceof ViewFileAction || FileAction instanceof SearchFileAction ||
                FileAction instanceof DeleteFileAction || FileAction instanceof MoveFileAction ||
                FileAction instanceof SwitchDriveAction) {

            FileAction.run();
            return ActionStateType.NORMAL;

        } else if (FileAction instanceof CopyFileAction) {
            return ActionStateType.COPY;

        } else {
            return ActionStateType.NORMAL;
        }
    }
public interface State {
    2 usages  2 implementations
    ActionStateType handle(FileAction FileAction);
}
```