Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №6

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Шаблони «Abstract Factory»,
«Factory Method», «Memento», «Observer»,
«Decorator»»

Варіант №18

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

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Contents

Тема.

Шаблони «Abstract Factory»,«Factory Method», «Memento», «Observer», «Decorator»

Мета.

Метою даної лабораторної роботи є ознайомлення з шаблонами проєктування, зокрема з шаблоном " Factory Method", 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.

..18 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.

Патерн "Фабричний метод" (Factory Method) є породжувальним патерном проектування, який визначає інтерфейс для створення об'єктів, але дозволяє підкласам вирішувати, який клас створювати. Таким чином, патерн "Фабричний метод" дозволяє відкласти інстанціювання до підкласів, забезпечуючи гнучкість і розширюваність системи.

Шаблон "фабричний метод" визначає інтерфейс для створення об'єктів певного базового типу. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон "фабричний метод" носить ще назву "Віртуальний конструктор".

Хід роботи.

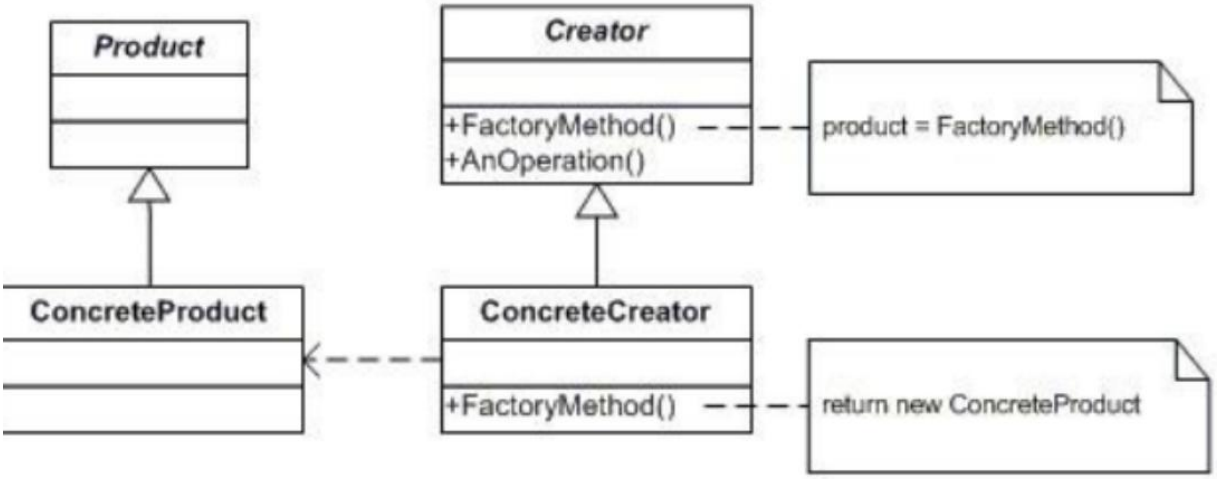

Рисунок 1.1 – UML діаграма шаблону Factory Method

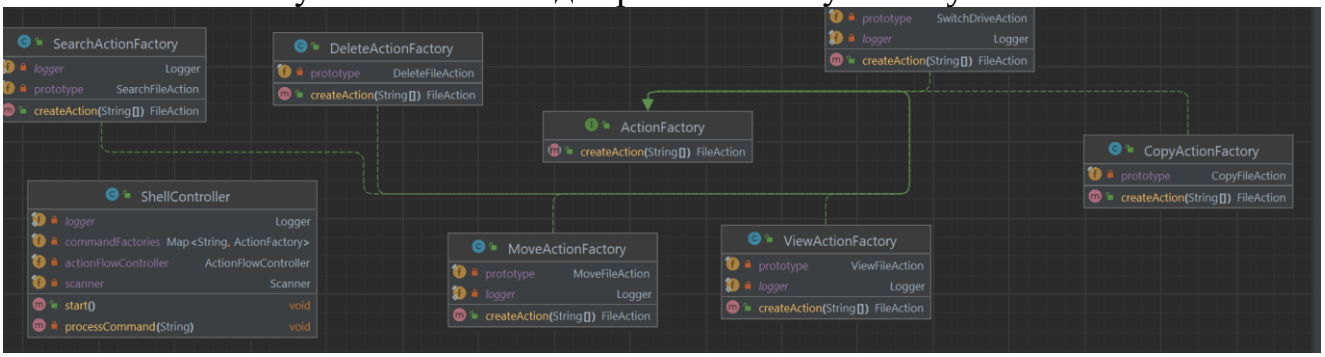

Рисунок 1.2 – Діаграма класів , згенерована IDE

. Інтерфейс ActionFactory:

- **Роль:** Базовий інтерфейс для всіх фабрик, які створюють об'єкти дій з файлами (FileAction).
- **Методи:**
  - createAction(String[] args) : FileAction - Створює об'єкт дії, використовуючи передані аргументи.

2. Клас SearchActionFactory:

- **Роль:** Фабрика для створення дій пошуку файлів (SearchFileAction).
- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - logger: Logger - Для логування дій.
  - prototype: SearchFileAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії

пошуку шляхом клонування прототипу та налаштування його з аргументів.

3. Клас DeleteActionFactory:

- **Роль:** Фабрика для створення дій видалення файлів (DeleteFileAction).
- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - prototype: DeleteFileAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії видалення шляхом клонування прототипу та налаштування його з аргументів.

4. Клас SwitchDriveActionFactory:

- **Роль:** Фабрика для створення дій перемикання диска (SwitchDriveAction).
- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - logger: Logger - Для логування дій.
  - prototype: SwitchDriveAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії перемикання диска шляхом клонування прототипу та налаштування його з аргументів.

5. Клас CopyActionFactory:

- **Роль:** Фабрика для створення дій копіювання файлів (CopyFileAction).
- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - prototype: CopyFileAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії копіювання шляхом клонування прототипу та налаштування його з аргументів.

6. Клас MoveActionFactory:

- **Роль:** Фабрика для створення дій переміщення файлів (MoveFileAction).

- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - logger: Logger - Для логування дій.
  - prototype: MoveFileAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії переміщення шляхом клонування прототипу та налаштування його з аргументів.

7. Клас ViewActionFactory:

- **Роль:** Фабрика для створення дій перегляду файлів (ViewFileAction).
- **Реалізує:** Інтерфейс ActionFactory.
- **Поля:**
  - logger: Logger - Для логування дій.
  - prototype: ViewFileAction - Прототип для клонування.
- **Методи:**
  - createAction(String[] args) : FileAction - Створює новий об'єкт дії перегляду шляхом клонування прототипу та налаштування його з аргументів.

8. Клас ShellController:

- **Роль:** Основний клас для обробки команд користувача.
- **Поля:**
  - logger: Logger - Для логування дій.
  - commandFactories : Map<String, ActionFactory> - Карта фабрик дій, де ключ - назва команди, а значення - відповідна фабрика.
  - actionFlowController: ActionFlowController - Контролер для управління потоком дій.
  - scanner: Scanner - Для читання введених користувачем команд.
- **Методи:**
  - start() : void - Запускає цикл обробки команд.
  - processCommand(String command) : void - Обробляє введену команду, розпізнає назву дії та передає її відповідній фабриці для створення та виконання.

Робота паттерну.

Виконаємо команду перегляду

3.808+02:00 INFO 14656 --- [shell] [restartedMain] c.s.commands.factory.ViewActionFactory   : Викликано createCommand у ViewCommandFactory з аргументами: [view]
2025-01-24T12:37:03.808+02:00 INFO 14656 --- [shell] [restartedMain] c.s.commands.factory.ViewActionFactory   : Initializing ViewFileAction with directory path
2025-01-24T12:37:03.810+02:00 INFO 14656 --- [shell] [restartedMain] coursework.shell.ShellController   : Передача команди 'view' до ActionFlowController
2025-01-24T12:37:0

Висновок.

Використання патерну "Фабрика" забезпечує гнучке та зручне створення команд у проєкті. Це особливо важливо, оскільки кількість команд може збільшуватися:

- **Масштабованість:** Система легко масштабується, оскільки додавання нових команд не потребує змін в існуючому коді.
- **Розширеність:** Кожен тип команди створюється окремою фабрикою (наприклад, CopyCommandFactory), що спрощує додавання нових команд через реалізацію інтерфейсу CommandFactory.
- **Інкапсуляція:** Процес створення команд прихований у фабриках, тому клієнтський код (ShellConsole) не залежить від деталей їх ініціалізації, що робить код чистішим.
- **Принцип відкритості/закритості:** Система закрита для модифікацій, але відкрита для розширення. Додавання нової команди не вимагає змін у ShellConsole; достатньо створити нову фабрику.

Таким чином, "Фабрика" покращує організацію коду, спрощує підтримку та дозволяє легко додавати нові функції.

ДОДАТОК

```
@Component
public class ShellController {

    private static final Logger logger = LoggerFactory.getLogger(ShellController.class);
    private final ActionFlowController actionFlowController;
    private final Map<String, ActionFactory> commandFactories;
    private final Scanner scanner;


    @Autowired
    public ShellController(ActionFlowController actionFlowController, Map<String, ActionFactory>
commandFactories) {
        this.actionFlowController = actionFlowController;
        this.commandFactories = commandFactories;
        this.scanner = new Scanner(System.in);
    }

    public void start() {
        System.out.println("Ласкаво просимо до оболонки Total Commander!");
        String commandLine;
        while (true) {
            System.out.print("Введіть команду (або 'exit' для виходу): ");
            commandLine = scanner.nextLine();
            if ("exit".equalsIgnoreCase(commandLine)) {
                System.out.println("Вихід з програми.");
                break;
            }
            processCommand(commandLine);
        }
    }

    private void processCommand(String commandLine) {
        String[] parts = commandLine.trim().split("\\s+");
        if (parts.length == 0) {
            return;
        }

        String action = parts[0].toLowerCase();

        ActionFactory factory = commandFactories.get(action);
        if (factory != null) {
            logger.info("Отримано команду '{}'", action);
            FileAction fileAction = factory.createAction(parts);
            if (fileAction != null) {
                logger.info("Передача команди '{}' до ActionFlowController", action);
                actionFlowController.handleAction(fileAction);
            }
        } else {
            System.out.println("Невідома команда. Введіть 'help' для списку доступних команд.");
        }
    }
}
```

```java
public interface ActionFactory {
    1 usage   6 implementations
    FileAction createAction(String[] args);
}
```

```java
@Component("view")
public class ViewActionFactory implements ActionFactory {
    3 usages
    private static final Logger logger = LoggerFactory.getLogger(ViewActionFactory.class);

    2 usages
    private final ViewFileAction prototype;

    public ViewActionFactory() {
        this.prototype = new ViewFileAction();
        logger.info("ViewActionFactory created with prototype ViewFileAction");
    }

    1 usage
    @Override
    public FileAction createAction(String[] args) {
        logger.info("Викликано createCommand у ViewCommandFactory з аргументами: {}", (Object) args);

        FileAction fileAction = prototype.clone();
        logger.info("Initializing ViewFileAction with directory path");

        return fileAction;
    }
}
```

}