Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційні систем та технологій

Лабораторна робота №5

із дисципліни «Технології розроблення програмного забезпечення»
Тема: «Шаблони «Adapter», «Builder», «Command», «Chain of responsibility», «prototype»»

Виконав:
Студент групи ІА-24
Боднар А. Д.

Перевірив:
Мягкий М.Ю.

Київ-2024

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

Теоретичні відомості

1. Adapter (Адаптер)

- **Призначення**: Шаблон **Adapter** використовується для перетворення інтерфейсу одного класу в інший, який очікує клієнт. Він дозволяє несумісним класам працювати разом, адаптуючи один інтерфейс до іншого.
- **Приклад використання**: Якщо у вас є стара система, яка працює з одним форматом даних, і новий модуль, який очікує інший формат, адаптер можна використати для перетворення даних між цими форматами.
- **Структура**:
    - **Клієнт** працює з **Adapter**, який інкапсулює логіку перетворення між **Target** (очікуваним інтерфейсом) і **Adaptee** (існуючим класом).

2. Builder (Будівельник)

- **Призначення**: Шаблон **Builder** використовується для створення складних об'єктів поетапно. Він дозволяє створювати різні представлення об'єкта без зміни його основної логіки.
- **Приклад використання**: Побудова складного документа з багатьма секціями або створення об'єктів з великою кількістю параметрів, наприклад, налаштувань.
- **Структура**:
    - **Director** керує процесом створення об'єкта.
    - **Builder** визначає поетапний процес створення.
    - **ConcreteBuilder** реалізує конкретні кроки будівництва.
    - **Product** — кінцевий об'єкт.

3. Command (Команда)

- **Призначення**: Шаблон **Command** інкапсулює запит як об'єкт, дозволяючи відкладати його виконання, записувати історію виконаних команд або забезпечувати скасування дій.
- **Приклад використання**: Управління діями в текстовому редакторі (запам'ятовування виконаних команд для відкату).
- **Структура**:
    - **Invoker** викликає команду.
    - **Command** визначає інтерфейс для виконання запиту.
    - **Receiver** — кінцевий виконавець дії.
    - **ConcreteCommand** реалізує конкретну команду, яка викликає методи у **Receiver**.

4. Chain of Responsibility (Ланцюг відповідальностей)

- **Призначення**: Шаблон **Chain of Responsibility** дозволяє передавати запит через ланцюг обробників, доки один з них не виконає дію. Це дозволяє динамічно змінювати обробку запитів без прив'язки до конкретного обробника.
- **Приклад використання**: Система обробки подій у додатку або система фільтрації даних, де кожен фільтр перевіряє запит і вирішує, обробляти його чи передати далі.
- **Структура**:
    - **Handler** оголошує метод для обробки запиту і зберігає посилання на наступний обробник.
    - **ConcreteHandler** реалізує метод обробки і вирішує, обробити запит чи передати далі.

5. Prototype (Прототип)

- **Призначення**: Шаблон **Prototype** дозволяє створювати нові об'єкти шляхом копіювання існуючих екземплярів, замість створення їх з нуля.
- **Приклад використання**: Копіювання складних об'єктів, які містять багато вкладених структур, або коли створення нового об'єкта є дорогим за часом чи ресурсами.
- **Структура**:
    - **Prototype** визначає інтерфейс для клонування об'єктів.
    - **ConcretePrototype** реалізує метод клонування.
    - Клієнт викликає метод клонування, отримуючи новий об'єкт.

Хід роботи

Обрано шаблон Command

```java
package snippents.command;

public interface Command {  3 usages  3 implementations
    void execute();  3 implementations
    Object getResult();  no usages  3 implementations
}
```

Рис.1 - Інтерфейс команд

Інтерфейс, який задає базову структуру команд, визначає основний метод, що повинен бути реалізований кожною командою. Основна мета цього інтерфейсу забезпечити уніфікований спосіб виконання команд, незалежно від їхньої внутрішньої реалізації.

```java
package snippents.command;

import snippents.strategy.SnippetService;

public class CreateSnippetCommand implements Command {  no usages
    private SnippetService service;  2 usages
    private String trigger;  2 usages
    private String content;  2 usages
    private Object result;  2 usages

    public CreateSnippetCommand(SnippetService service, String trigger, String content) {
        this.service = service;
        this.trigger = trigger;
        this.content = content;
    }

    @Override
    public void execute() {
        // Викликається метод сервісу для обробки створення сніпета
        result = service.processCommand(trigger,  command: "create", content);
    }

    @Override  no usages
    public Object getResult() {
        return result; // Повертає результат (trigger або повідомлення про помилку)
    }
}
```

Рис.2 - Команда для створення сніппета

```java
package snippents.command;

import snippents.strategy.SnippetService;

public class DeleteSnippetCommand implements Command {  no usages
    private SnippetService service;  2 usages
    private String trigger;  2 usages
    private Object result;  2 usages

    public DeleteSnippetCommand(SnippetService service, String trigger) {  no usages
        this.service = service;
        this.trigger = trigger;
    }

    @Override
    public void execute() {
        // Викликається метод сервісу для обробки видалення сніпета
        result = service.processCommand(trigger,  command: "delete",  content: null);
    }

    @Override  no usages
    public Object getResult() {
        return result; // Повертає результат (ID сніпета або "NaN")
    }
}
```

Рис.3 - Команда для видалення сніппета

```java
package snippents.command;


import snippents.strategy.SnippetService;

public class UseSnippetCommand implements Command {  no usages
    private SnippetService service;  2 usages
    private String trigger;  2 usages
    private Object result;  2 usages

    public UseSnippetCommand(SnippetService service, String trigger) {  no usages
        this.service = service;
        this.trigger = trigger;
    }

    @Override
    public void execute() {
        // Викликається метод сервісу для обробки використання сніпета
        result = service.processCommand(trigger,  command: "use",  content: null);
    }

    @Override  no usages
    public Object getResult() { return result; // Повертає результат (вміст сніпета або пропозиції) }
}
```

Рис.4 - Команда для використання сніппета

```java
package snippents.strategy;

import models.Snippet;
import repository.SnippetRepository;

public class SnippetService {  9 usages
    private SnippetRepository repository;  8 usages

    public SnippetService(SnippetRepository repository) {  no usages
        this.repository = repository;
    }

    // Основний метод, що обробляє всі команди
    public Object processCommand(String trigger, String command, String content) {  3 usages
        switch (command) {
            case "create":
                return processCreateSnippet(trigger, content);  // Створення сніпета
            case "delete":
                return processDeleteSnippet(trigger);  // Видалення сніпета
            case "use":
                return processUseSnippet(trigger);  // Використання сніпета
            default:
                throw new IllegalArgumentException("Unknown command: " + command);  // Якщо команда невідома
        }
    }

```

```java
// Логіка вибору стратегії для створення сніпета
private Object processCreateSnippet(String trigger, String content) {  1 usage
    SnippetStrategy strategy;

    // Вибір стратегії
    if (repository.findByTrigger(trigger) == null) {
        strategy = new CreateSnippetStrategy(); // Якщо сніпет не існує
    } else {
        strategy = new SnippetAlreadyExistsStrategy(); // Якщо сніпет вже існує
    }

    // Виконання обраної стратегії
    return strategy.execute(trigger, content, repository);
}
```

```java
    // Логіка видалення сніпета
    private Object processDeleteSnippet(String trigger) {  1 usage
        SnippetStrategy strategy;

        // Вибір стратегії
        if (repository.findByTrigger(trigger) == null) {
            strategy = new NotFoundSnippetStrategy(); // Сніпет не знайдено
        } else {
            strategy = new DeleteSnippetStrategy(); // Сніпет знайдено
        }

        // Виконання обраної стратегії
        return strategy.execute(trigger,  content: null, repository);
    }
```

```
61      // Логіка використання сніпета
62      private Object processUseSnippet(String trigger) {  1 usage
63          SnippetStrategy strategy;
64
65          // Вибір стратегії
66          if (repository.findByTrigger(trigger) != null) {
67              strategy = new ExactMatchSnippetStrategy(); // Точне співпадіння
68          } else if (!repository.findSuggestions(trigger).isEmpty()) {
69              strategy = new SuggestSnippetStrategy(); // Схожі варіанти
70          } else {
71              strategy = new NotFoundSnippetStrategy(); // Нічого не знайдено
72          }
73
74          // Виконання обраної стратегії
75          return strategy.execute(trigger,  content: null, repository);
76      }
77
78  }
```

Рис.5-8 - Клас котрому передають дані команди

Висновки

У цій лабораторній роботі було реалізовано шаблон проектування **Command** для управління операціями над сніпетами в текстовому редакторі. В рамках роботи було виконано наступне:

1. **Розробка команд для основних дій**:
    - Реалізовано команди для створення, видалення, та використання сніпетів.
    - Кожна команда інкапсулює запит і передає його до сервісу для виконання.
2. **Інтеграція з шаблоном Стратегія**:
    - Команди використовуються для передачі даних до сервісу, який вибирає відповідну стратегію для виконання операції.
    - Стратегії відповідають за логіку обробки запиту, наприклад, створення нового сніпета або видалення існуючого.
3. **Гнучкість та розширюваність**:

- Завдяки використанню шаблону **Command** вдалося розділити логіку передачі даних (команди) і виконання операцій (стратегії).
- Цей підхід дозволяє легко додавати нові типи команд або розширювати функціональність сервісу.

4. **Можливість логування та відкату дій**:
   - Команди можуть зберігати інформацію про виконані операції, що забезпечує можливість реалізації механізмів відкату або повторного виконання дій.