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 №7

із дисципліни «Технології розроблення програмного забезпечення»
Тема**:** «Шаблони «mediator», «facade», «bridge», «template method»»

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. Шаблон "Mediator" (Посередник)

Опис: Паттерн **Mediator** є структурним патерном, який визначає об'єкт, що інкапсулює взаємодії між групою об'єктів. Це дозволяє знизити зв'язність між класами, оскільки замість того, щоб об'єкти безпосередньо спілкувалися один з одним, вони спілкуються через посередника. Посередник координує взаємодію між об'єктами, забезпечуючи таким чином слабке зв'язування між ними.

Ключові компоненти:

- **Mediator** — абстракція, яка визначає методи для комунікації з іншими компонентами.
- **ConcreteMediator** — конкретна реалізація посередника, яка керує обміном інформацією між об'єктами.
- **Colleagues** — об'єкти, які взаємодіють через посередника. Вони не мають прямих посилань один на одного.

Переваги:

- Зменшує кількість прямих зв'язків між класами.
- Спрощує систему, централізуючи логіку взаємодії.
- Спрощує зміну поведінки взаємодії, оскільки зміни потрібно вносити лише в один клас (Mediator).

Недоліки:

- Може стати єдиною точкою відмови, що ускладнює підтримку великої кількості взаємодій.
- Може призвести до надмірної складності, якщо об'єкти мають велику кількість взаємодій.

2. Шаблон "Facade" (Фасад)

Опис: Патерн **Facade** є структурним патерном, який надає спрощений інтерфейс до складної системи класів. Він приховує складність внутрішніх підсистем і дозволяє користувачам взаємодіяти з системою через один спрощений інтерфейс, що робить систему легшою у використанні.

Ключові компоненти:

- **Facade** — клас, який надає спрощений інтерфейс для клієнта.
- **Subsystems** — складні підсистеми, до яких фасад надає доступ. Вони виконують конкретні завдання, але не мають прямого взаємодії з клієнтами.

Переваги:

- Спрощує інтерфейс для складних підсистем.
- Знижує залежність між клієнтами і конкретними підсистемами.
- Покращує читабельність коду, оскільки приховує складність.

Недоліки:

- Може приховувати важливі деталі про систему, що може ускладнити подальше розширення або налаштування.
- Якщо фасад стає занадто великим, то він може перетворитися на "антипатерн", де весь код зводиться до одного класу.

3. Шаблон "Bridge" (Міст)

Опис: Патерн **Bridge** є структурним патерном, який дозволяє відокремити абстракцію від її реалізації так, щоб обидва могли змінюватися незалежно. Цей патерн корисний, коли вам потрібно варіативно змінювати як абстракцію, так і реалізацію без взаємного впливу.

Ключові компоненти:

- **Abstraction** — абстракція, яка містить посилання на реалізацію і використовує її через інтерфейс.
- **RefinedAbstraction** — розширена абстракція, яка додає додаткову функціональність.
- **Implementor** — інтерфейс для реалізації, який визначає методи для конкретної реалізації.

- **ConcreteImplementor** — конкретна реалізація, що реалізує методи інтерфейсу Implementor.

Переваги:

- Дозволяє змінювати абстракцію і реалізацію незалежно одна від одної.
- Покращує гнучкість та підтримку коду.
- Спрощує розширення системи, оскільки нові реалізації можуть бути додані без змін в абстракції.

Недоліки:

- Може збільшити кількість класів в системі.
- Вимагає ретельного проектування та планування абстракцій і реалізацій.

4. Шаблон "Template Method" (Метод шаблону)

Опис: Патерн **Template Method** є поведінковим патерном, який визначає скелет алгоритму в методі, залишаючи деякі кроки для реалізації підкласами. Це дозволяє підкласам змінювати частини алгоритму без зміни його структури. Патерн дозволяє реалізувати загальну частину алгоритму в базовому класі, а специфічні частини залишити для нащадків.

Ключові компоненти:

- **AbstractClass** — клас, який містить шаблон (метод) та основну логіку. Визначає кроки алгоритму, деякі з яких можуть бути абстрактними і реалізовані в підкласах.
- **ConcreteClass** — підкласи, які реалізують специфічні кроки алгоритму.

Переваги:

- Спрощує повторне використання коду, дозволяючи централізовано визначати основну структуру алгоритму.
- Легко дозволяє підкласам змінювати частини алгоритму без зміни загальної структури.
- Покращує підтримку та розширюваність.

Недоліки:

- Може призвести до надмірної залежності від базового класу.

- Якщо алгоритм складний і має багато кроків, це може призвести до того, що базовий клас стане дуже великим і складним.

Хід роботи

Варіант - 3

Текстовий редактор

Використання шаблону “Template Method”

```
// Додаємо обробку гарячих клавіш
setupKeyListener(textPane);
```

Рис. 1 - Обробка гарячих клавіш

```
private void setupKeyListener(JTextPane textPane) {  1 usage
    textPane.addKeyListener(new KeyAdapter() {
        @Override
        public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_CONTROL) {
                System.out.println("(1)Ctrl pressed"); // Логування натискання Ctrl
                String trigger = getTriggerFromText(textPane); // Отримуємо тригер
                System.out.println("(2)Trigger to process: " + trigger); // Логування тригера
                SnippetProcessor snippetProcessor = new SnippetProcessorImpl();
                snippetProcessor.processSnippet(trigger, textPane); // Передаємо тригер на обробку
            }
        }
    });
}

private String getTriggerFromText(JTextPane textPane) {  1 usage
    try {
        // Отримуємо поточну позицію курсора в JTextPane
        int caretPosition = textPane.getCaretPosition();
        // Отримуємо текст з початку до позиції курсора
        String textBeforeCaret = textPane.getText( offs: 0, caretPosition);
        // Розбиваємо текст на слова, використовуючи пробіли та інші роздільники
        String[] words = textBeforeCaret.split( regex: "\\s+");
        // Якщо є слова перед курсором, останнє слово буде тригером
        String trigger = words.length > 0 ? words[words.length - 1] : "";
        System.out.println("(3)Тригер : " + trigger); // Логування тригера
        return trigger;
    } catch (BadLocationException e) {
        e.printStackTrace();
        return ""; // Якщо сталася помилка, повертаємо порожній тригер
```

Рис. 2 - Обробка гарячих клавіш

Перша частина: Виклик setupKeyListener(textPane);

Цей рядок ініціалізує процес налаштування обробки натискання клавіші. Метод setupKeyListener встановлює слухача для клавіатурних подій, який реагуватиме на натискання клавіші в компоненті JTextPane.

Друга частина: Метод setupKeyListener(JTextPane textPane)

1. **Додавання слухача клавіші**: Створюється обробник події для натискання клавіші в компоненті JTextPane.
2. Перевірка натискання клавіші **Ctrl**: Коли натискається клавіша **Ctrl**, виконується логування цього натискання для налагодження.
3. **Отримання тригера**: Викликається метод, який визначає, яке саме слово користувач ввів перед курсором в JTextPane. Це слово стане тригером для подальшої обробки.
4. **Створення обробника сніпетів**: Ініціалізується об'єкт класу, який буде обробляти сніпети (як точні, так і схожі).
5. **Обробка сніпетів**: Передається тригер на обробку в метод, який шукає точні або схожі сніпети, і виконує відповідні дії в JTextPane.

Метод getTriggerFromText(JTextPane textPane)

1. **Отримання поточної позиції курсора**: Визначається, де саме знаходиться курсор у тексті.
2. **Отримання тексту до курсора**: Береться текст з початку документа до поточної позиції курсора.
3. **Розбиття тексту на слова**: Текст розбивається на окремі слова для того, щоб визначити тригер.
4. **Визначення тригера**: Останнє слово перед курсором стає тригером.
5. **Повернення тригера**: Повертається отримане слово (тригер), яке далі використовується для пошуку сніпетів.

```java
public class SnippetProcessorImpl extends SnippetProcessor {
    public SnippetProcessorImpl() {  1 usage
        super(
                new DefaultExactMatchHandler(),
                new DefaultSimilarMatchesHandler(),
                new DefaultNoMatchHandler()
        );
    }
}
```

Рис. 3 - Клас SnippetProcessorImpl

Клас **SnippetProcessorImpl** є конкретною реалізацією абстрактного класу SnippetProcessor і виконує роль налаштування всього механізму обробки сніпетів. У контексті шаблону **Template Method**, цей клас відповідає за "фіксацію" деталей того, які саме обробники (handlers) будуть використовуватися під час роботи.

```java
public abstract class SnippetProcessor {  1 inheritor  3 usages
    private final ExactMatchHandler exactMatchHandler;  2 usages
    private final SimilarMatchesHandler similarMatchesHandler;  2 usages
    private final NoMatchHandler noMatchHandler;  3 usages

    public SnippetProcessor(ExactMatchHandler exactMatchHandler,  1 usage
                            SimilarMatchesHandler similarMatchesHandler,
                            NoMatchHandler noMatchHandler) {
        this.exactMatchHandler = exactMatchHandler;
        this.similarMatchesHandler = similarMatchesHandler;
        this.noMatchHandler = noMatchHandler;
    }

    public void processSnippet(String trigger, JTextPane textPane) {  1 usage
        if (trigger.isEmpty()) {
            noMatchHandler.handleNoMatch("Text is empty. Please provide a valid trigger.");
        } else {
            int idSetting = CurrentSettingRepository.getCurrentSettingId(); //отримуємо CurrentSettingId
            int idSnippedList = SettingRepository.getSnippetListId(idSetting); //отримуємо SnippedListId
            String exactMatch = SnippetRepository.getSnippetByTrigger(trigger, idSnippedList );// тут вже
            if (exactMatch != null) {
                exactMatchHandler.handleExactMatch(exactMatch, textPane, trigger);
            } else {
                List<Snippet> similarMatches = SnippetRepository.getSimilarSnippets(trigger, idSetting);

                if (!similarMatches.isEmpty()) {
                    similarMatchesHandler.handleSimilarMatches(similarMatches);
                } else {
                    noMatchHandler.handleNoMatch("No matching or similar snippets found.");
                }
            }
        }
```

Рис. 4 - Клас SnippetProcessor

Клас **SnippetProcessor** є базовим (абстрактним) класом, що визначає загальний алгоритм обробки тригера для пошуку відповідних сніпетів. У цьому класі реалізується основна структура процесу, що виконується незалежно від конкретної реалізації обробників.

Основна мета класу:

1. **Шаблон Template Method**:
    - Клас визначає загальний алгоритм обробки тригера через метод processSnippet. Цей метод не змінюється в підкласах, але його поведінка залежить від конкретних реалізацій обробників, які передаються через конструктор.

2. **Делегування відповідальностей**:
    - Логіка обробки розподіляється між трьома основними обробниками:
        - **ExactMatchHandler**: Обробляє випадок точного збігу тригера з існуючим сніпетом.
        - **SimilarMatchesHandler**: Обробляє ситуацію, коли знайдено схожі сніпети.
        - **NoMatchHandler**: Виконує дії у випадку, коли збіги (точні чи схожі) відсутні.

```
public interface ExactMatchHandler {  3 usages  1 implementation
    void handleExactMatch(String snippetContent, JTextPane textPane, String trigger);
}
```

Рис. 5 - Інтерфейс ExactMatchHandler

```
public interface NoMatchHandler {  3 usag
    void handleNoMatch(String message);
}
```

Рис. 6 - Інтерфейс NoMatchHandler

```
public interface SimilarMatchesHandler {  3 usages  1 implementat
    void handleSimilarMatches(List<Snippet> similarMatches)
}
```

Рис. 7 - Інтерфейс SimilarMatchesHandler

Ці три інтерфейси — **ExactMatchHandler**, **NoMatchHandler** і **SimilarMatchesHandler** — визначають контракти для обробки різних ситуацій, що виникають під час роботи зі сніпетами. Вони є важливими частинами системи, яка використовує шаблон **Template Method**, оскільки надають можливість підключати різні реалізації обробки без змін базового алгоритму.

```java
public class DefaultExactMatchHandler implements ExactMatchHandler {  1 usage
    @Override  1 usage
    public void handleExactMatch(String snippetContent, JTextPane textPane, String trigger) {
        try {
            // Зберігаємо поточну позицію курсора
            int caretPosition = textPane.getCaretPosition();

            // Отримуємо поточний текст з текстового поля
            String text = textPane.getText();

            // Знайдемо індекс тригера в тексті
            int triggerIndex = text.indexOf(trigger);

            if (triggerIndex != -1) {
                // Якщо тригер знайдено, створимо новий текст з заміною
                StringBuilder newText = new StringBuilder(text);

                // Видаляємо тригер з тексту (по довжині тригера)
                newText.replace(triggerIndex,  end: triggerIndex + trigger.length(), snippetContent);

                // Оновлюємо текст в JTextPane
                textPane.setText(newText.toString());

                // Розбиваємо текст на рядки
                String[] lines = textPane.getText().split( regex: "\n");
```

Рис. 8 - Перша частина класу DefaultExactMatchHandler

```java
                // Знаходимо номер рядка, куди вставлено сніппет
                int lineNumber = 0;
                int charCount = 0;
                for (int i = 0; i < lines.length; i++) {
                    charCount += lines[i].length() + 1;  // +1 для нового рядка
                    if (triggerIndex < charCount) {
                        lineNumber = i;
                        break;
                    }
                }

                // Встановлюємо нову позицію курсора
                int position = newText.indexOf(snippetContent) + snippetContent.length();
                textPane.setCaretPosition(position);
                // Можна додатково застосувати log чи консоль для перевірки правильності
                System.out.println("Новий рядок: " + (lineNumber + 1)); // Рядок, куди вставлено
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Рис. 9 - друга частина класу DefaultExactMatchHandler

```java
public class DefaultSimilarMatchesHandler implements SimilarMatchesHandler {  1 usage
    @Override  1 usage
    public void handleSimilarMatches(List<Snippet> similarMatches) {
        if (similarMatches.isEmpty()) {
            JOptionPane.showMessageDialog( parentComponent: null,  message: "Немає схожих сніппетів.",
                    title: "Інформація", JOptionPane.INFORMATION_MESSAGE);
        } else {
            StringBuilder message = new StringBuilder("Знайдені схожі сніппети:\n");
            for (Snippet snippet : similarMatches) {
                message.append("Тригер: ").append(snippet.getTrigger()).append("\n");
                message.append("Контент: ").append(snippet.getContent()).append("\n\n");
            }
            JOptionPane.showMessageDialog( parentComponent: null, message.toString(),
                    title: "Схожі сніппети", JOptionPane.INFORMATION_MESSAGE);
        }
    }
}
```

Рис. 10 - Клас DefaultSimilarMatchesHandler

```java
public class DefaultNoMatchHandler implements NoMatchHandler {  1 usage
    @Override  2 usages
    public void handleNoMatch(String message) {
        JOptionPane.showMessageDialog( parentComponent: null, message,  title: "Помилка", JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Рис. 11 - Клас DefaultSimilarMatchesHandler

Ці три класи — **DefaultExactMatchHandler**, **DefaultNoMatchHandler** і **DefaultSimilarMatchesHandler** — реалізують інтерфейси для обробки відповідних ситуацій (точний збіг, відсутність збігу, схожі збіги). Вони працюють разом із шаблоном **Template Method**, надаючи конкретну реалізацію для кожної з поведінок.

Висновки

У ході виконання лабораторної роботи було успішно реалізовано шаблон проектирования **Template Method**, що дозволило створити гнучку і легко розширювану архітектуру для роботи зі сніппетами. Реалізація цього шаблону надала наступні переваги:

1. **Уніфікація алгоритму обробки**
   Основна логіка обробки тригерів для сніппетів була винесена в базовий клас SnippetProcessor, що забезпечило чітку структуру коду. Спільні частини алгоритму стали централізованими, а специфічні аспекти делегувалися до окремих класів-обробників (DefaultExactMatchHandler, DefaultSimilarMatchesHandler, DefaultNoMatchHandler).
2. **Можливість розширення**
   Завдяки використанню інтерфейсів та окремих класів для хендлерів, у майбутньому можна легко додати нові способи обробки тригерів, не змінюючи основної структури коду. Це відповідає принципу **Open-Closed Principle** із SOLID.
3. **Чітка відповідальність компонентів**
   Усі класи мають чітко визначену відповідальність. Наприклад, ExactMatchHandler займається точними збігами, SimilarMatchesHandler — схожими збігами, а NoMatchHandler — відсутністю збігів. Це підвищує читабельність і підтримуваність проекту.
4. **Гнучкість і повторне використання**
   Завдяки застосуванню шаблону **Template Method**, логіка обробки сніппетів стала універсальною. Базовий клас можна повторно використовувати з різними реалізаціями обробників, що робить систему більш гнучкою.
5. **Мінімізація дублювання коду**
   Винесення спільних кроків алгоритму в базовий клас дозволило уникнути дублювання коду в різних частинах програми.
6. **Практичне застосування шаблону**
   Реалізація **Template Method** у реальному прикладі з обробкою тригерів і сніппетів продемонструвала, як шаблони проектування можуть спрощувати розробку, підвищувати якість коду та забезпечувати легкість його модифікації.

У підсумку, виконання цієї лабораторної роботи допомогло на практиці закріпити знання про шаблон **Template Method**, зрозуміти його ключові аспекти та навчитися правильно застосовувати його у реальних завданнях.