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 №6

із дисципліни «Технології розроблення програмного забезпечення»
Тема: «Шаблони «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»»

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. Abstract Factory

**Призначення**: Шаблон "абстрактна фабрика" використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього створюється загальний інтерфейс фабрики, а потім її реалізації для різних сімейств продуктів.

**Приклад**: Для роботи з різними базами даних, як в ADO.NET, використовуються загальні фабрики для створення різних типів підключень, адаптерів та читачів даних.

**Переваги та недоліки**:

- Звільняє клієнтський код від прив'язки до конкретних класів продуктів.
- Легко замінювати сімейства продуктів.
- Вимагає змін у всіх фабриках при додаванні нових методів .

## 2. Factory Method

**Призначення**: Шаблон "фабричний метод" визначає інтерфейс для створення об'єктів певного типу. Він делегує створення об'єктів конкретним підкласам.

**Приклад**: Менеджер найму делегує співбесіду для різних вакансій різним спеціалістам.

**Переваги та недоліки**:

- Позбавляє від прив'язки до конкретних класів продуктів.
- Спрощує додавання нових продуктів.
- Може призвести до створення великих паралельних ієрархій класів .

## 3. Memento

**Призначення**: Шаблон "Мементо" використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Він дозволяє зберігати знімки стану об'єкта в окремому об'єкті, який не порушує інкапсуляцію початкового об'єкта.

**Приклад**: У текстовому редакторі можна зберігати стани до та після редагування для можливості відкотити зміни.

**Переваги та недоліки**:

- Не порушує інкапсуляцію вихідного об'єкта.
- Вимагає багато пам'яті, якщо часто створюються знімки .

## 4. Observer

**Призначення**: Шаблон "Спостерігач" визначає залежність один-до-багатьох між об'єктами. Коли один об'єкт змінює свій стан, всі інші об'єкти, які підписалися на ці зміни, отримують сповіщення.

**Приклад**: Користувачі банківської системи отримують сповіщення про зміни балансу рахунку.

**Переваги та недоліки**:

- Підписники можуть бути додані або видалені в будь-який час.
- Не потрібно залежати від конкретних класів підписників.
- Підписники можуть сповіщатися в випадковому порядку .

## 5. Decorator

**Призначення**: Шаблон "Декоратор" дозволяє динамічно додавати функціональність до об'єкта під час виконання програми, обертаючи об'єкт у новий, який зберігає функції оригінального об'єкта та додає нові.

**Приклад**: Декоратор може бути використаний для додавання смуги прокрутки до візуальних елементів в інтерфейсі.

**Переваги та недоліки**:

- Гнучкість у зміні поведінки об'єкта.
- Може ускладнити код програми через введення великої кількості класів.

# Хід роботи

## Варіант - 3

## Текстовий редактор

```
public interface Observer {
    void update(String data);
}
```

Рис. 1 - інтерфейс спостерігачів

```
public interface Subject {  2 usages  1 implemen
    void addObserver(Observer observer);  2 u
    void removeObserver(Observer observer);
    void notifyObservers();  2 usages  1 impleme
}
```

Рис. 2 - інтерфейс джерел сповіщень

Разом ці інтерфейси реалізують шаблон **"Спостерігач"**, забезпечуючи механізм сповіщення про зміни між об'єктами.

```
public class Main implements Subject {
    private final List<Observer> observers = new ArrayList<>();
    private String text = "";  3 usages

    public static void main(String[] args) {
        new Main(); // Запускаємо редактор
    }

    public Main() {  1 usage
        // Створення основного вікна
        JFrame frame = new JFrame( title: "TextEditor");
        JTextPane textPane = new JTextPane();
        JMenuBar menuBar = new JMenuBar();
        JMenu fileMenu = new JMenu( s: "File");
        JMenuItem openItem = new JMenuItem( text: "Open");
        JMenuItem saveItem = new JMenuItem( text: "Save");
        JMenuItem exitItem = new JMenuItem( text: "Exit");

        // Додавання елементів меню
        fileMenu.add(openItem);
        fileMenu.add(saveItem);
        fileMenu.addSeparator();
        fileMenu.add(exitItem);
        menuBar.add(fileMenu);
        frame.setJMenuBar(menuBar);
```

Рис. 3 - частина коду класса Main для створення вікна застосунку

```
// Обробка змін у текстовій області
textPane.addKeyListener(new KeyAdapter() {
    @Override
    public void keyReleased(KeyEvent e) {
        text = textPane.getText();
        notifyObservers(); // Сповіщаємо всіх підписників при зміні тексту
    }
});
```

Рис. 4 - частина коду класса Main, котра відповідає за сповіщення

```
// Додаємо підписників
addObserver(new SyntaxHighlighterService(textPane));
addObserver(new AutoSave());
```

Рис. 5 - частина коду класса Main, котра відповідає за додавання підписників

```
public class SyntaxHighlighterService implements Observer {  2 usages
    private final JTextPane textPane;  4 usages
    private final StyledDocument doc;  10 usages
    private final SyntaxHighlighterWordRepository wordRepository;  2 usages

    public SyntaxHighlighterService(JTextPane textPane) {  1 usage
        this.textPane = textPane;
        this.doc = textPane.getStyledDocument();
        this.wordRepository = new SyntaxHighlighterWordRepository(DatabaseConnection.connect());

        System.out.println("Connected to word repository.");

        Style style = doc.getStyle( nm: "Keyword");
        if (style == null) {
            style = doc.addStyle( nm: "Keyword",  parent: null);
            StyleConstants.setForeground(style, Color.BLUE); // Встановлюємо колір для ключових слів
        }

    }
```

Рис. 6 - частина коду класса SyntaxHighlighterService(підписника Main)

Ця частина коду представляє конструктор класу **SyntaxHighlighterService**, який є реалізацією інтерфейсу **Observer**. Його основна мета — налаштувати базові параметри для підсвічування синтаксису в текстовій панелі. Опис його роботи можна сформулювати так:

1. **Ініціалізація полів класу:**
    - **textPane** — отримує текстову панель, у якій буде виконуватись підсвічування.
    - **doc** — отримує об'єкт **StyledDocument** з текстової панелі для роботи зі стилями тексту.
    - **wordRepository** — створює екземпляр репозиторію **SyntaxHighlighterWordRepository**, який дозволяє взаємодіяти з базою даних через з'єднання **DatabaseConnection.connect()**.
2. **Логування успішного підключення до репозиторію:**
    - Виводить повідомлення **"Connected to word repository."** у консоль.
3. **Налаштування стилю "Keyword" для ключових слів:**
    - Перевіряє, чи існує стиль з назвою **"Keyword"** у документі:
        - Якщо стиль відсутній, створює його за допомогою **doc.addStyle("Keyword", null)**.
        - Встановлює базовий колір для стилю (чорний) за допомогою **StyleConstants.setForeground**.

```java
@Override  1 usage
public void update(String data) {
    SwingUtilities.invokeLater(this::applyHighlighting);
}


private void applyHighlighting() {  1 usage
    try {
        // Зберігаємо поточну позицію курсора
        int cursorPosition = textPane.getCaretPosition();

        // Зчитуємо весь текст
        String text = textPane.getText();

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

        // Позиція для кожного символа в тексті
        int offset = 0;

        // Перевірка стилю "Default"
        Style defaultStyle = doc.getStyle( nm: "Default");
        if (defaultStyle == null) {
            defaultStyle = doc.addStyle( nm: "Default",  parent: null);
            StyleConstants.setForeground(defaultStyle, Color.BLACK);
        }

        // Скидаємо стилі для всього тексту
        doc.setCharacterAttributes( offset: 0, text.length(), defaultStyle,  replace: true);
```

Рис. 7 - частина коду класса SyntaxHighlighterService

Ця частина коду забезпечує оновлення підсвічування синтаксису в текстовій панелі:

1. **Оновлення викликається через update:**
    - Метод **update** викликає **applyHighlighting** у потоці інтерфейсу Swing для уникнення проблем багатопотоковості.
2. **Метод applyHighlighting:**
    - Зберігає поточну позицію курсора.
    - Зчитує текст із панелі та розбиває його на рядки.
    - Перевіряє існування стилю "Default" (чорний текст), створює його за потреби.
    - Скидає всі стилі до "Default", видаляючи попереднє підсвічування.

```java
// Отримуємо всі слова для підсвітки з бази
List<SyntaxHighlighterWord> words = wordRepository.findByHighlighterId(1);

// Обробляємо кожен рядок
for (String line : lines) {
    // Скидаємо стилі для поточного рядка
    doc.setCharacterAttributes(offset, line.length(), defaultStyle,  replace: true);

    // Підсвічуємо кожне знайдене слово
    for (SyntaxHighlighterWord word : words) {
        String searchWord = word.getWord();
        String color = word.getColor();

        if (searchWord != null && color != null) {
            int index = 0;
            // Використовуємо регулярний вираз для пошуку повних слів
            String regex = "\\b" + Pattern.quote(searchWord) + "\\b";
            Pattern pattern = Pattern.compile(regex);
            Matcher matcher = pattern.matcher(line);

            while (matcher.find()) {
                try {
                    // Перевіряємо і застосовуємо колір
                    Color highlightColor = parseColor(color);
                    if (highlightColor != null) {
                        Style style = doc.getStyle(searchWord);
                        if (style == null) {
                            style = doc.addStyle(searchWord,  parent: null);
                            StyleConstants.setForeground(style, highlightColor);
```

Рис. 8 - частина коду класса SyntaxHighlighterService

Ця частина коду відповідає за підсвічування слів у тексті згідно з даними з бази:

1. **Отримання слів для підсвітки:**
   - З бази даних витягується список слів, які потрібно підсвітити, з використанням методу `findByHighlighterId(1)`.
2. **Обробка кожного рядка тексту:**
   - Скидаються стилі для поточного рядка, щоб прибрати попереднє підсвічування.

3. **Пошук та підсвічування слів:**
    - Для кожного слова зі списку формується регулярний вираз, який дозволяє знайти слово лише як окрему сутність (враховуючи межі слова \b).
    - Кожен збіг (матч) перевіряється, і на його основі застосовується стиль.
4. **Застосування кольору:**
    - Якщо стиль для слова ще не існує, він створюється з використанням кольору, отриманого з бази.
    - Знайдені збіги підсвічуються відповідним стилем, додаючи колір до тексту.

```
                        doc.setCharacterAttributes( offset: offset + matcher.start(), searchWord.length(), style,
                    }
                } catch (Exception e) {
                    System.err.println("Invalid color format for word: " + searchWord);
                }
            }
        }
    }

    // Оновлюємо зміщення для наступного рядка
    offset += line.length(); // Оновлюємо зміщення без додавання 1 для символа нового рядка
}

// Відновлюємо позицію курсора
textPane.setCaretPosition(cursorPosition);

} catch (Exception e) {
    e.printStackTrace();
}
}
```

# Рис. 9 - частина коду класса SyntaxHighlighterService

Ця частина коду завершує процес підсвічування слів у тексті:

1. **Обробка помилок кольору:**
    - Якщо колір слова має неправильний формат (наприклад, не є дійсним кольором або невірно зчитаний з бази даних), генерується виняток. У такому випадку виводиться повідомлення про помилку для відповідного слова, але процес обробки триває для інших слів.
2. **Оновлення зміщення:**
    - Для кожного рядка тексту обчислюється зміщення на основі довжини обробленого рядка, щоб перейти до наступного рядка. Символ нового рядка \n враховується автоматично.
3. **Відновлення позиції курсора:**
    - Після завершення підсвічування курсор текстової панелі повертається на те місце, де він був перед початком обробки.
4. **Обробка загальних винятків:**
    - Якщо під час виконання процесу виникає будь-яка інша помилка, вона виводиться в консоль для діагностики.

```
private Color parseColor(String color) {  1 usage
    try {
        if (color.startsWith("#")) {
            return Color.decode(color);
        } else {
            Field field = Color.class.getField(color.toLowerCase());
            return (Color) field.get(null);
        }
    } catch (Exception e) {
        System.err.println("Invalid color format: " + color);
        return null;
    }
}
```

Рис. 10 - остання частина коду класса SyntaxHighlighterService

Метод parseColor перетворює текстове представлення кольору на об'єкт Color.

1. **Перевірка формату кольору:**
   - Якщо рядок починається з #, вважається, що це колір у форматі HEX (наприклад, #FF5733), і використовується Color.decode для створення кольору.
   - В іншому випадку метод вважає, що це назва кольору (наприклад, red, blue). За допомогою рефлексії шукається відповідне поле в класі Color.
2. **Обробка помилок:**
   - Якщо формат кольору невірний (HEX з помилками або недійсна назва), генерується виняток, і метод повертає null.

```java
public class AutoSave implements Observer {  2 usages
    private static final String FILE_NAME = "autosave.txt";  2 usages

    @Override  1 usage
    public void update(String data) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(FILE_NAME))) {
            writer.write(data);
            System.out.println("Text auto-saved to " + FILE_NAME);
        } catch (IOException e) {
            System.err.println("Error saving text: " + e.getMessage());
        }
    }
}
```

Рис. 11 - Код класса AutoSave(ще одного підписника Main), котрий відповідає за автозбереження

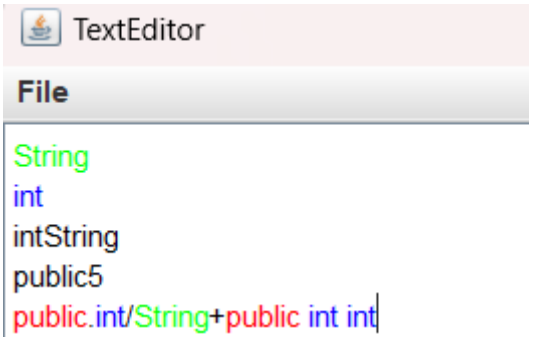


Рис. 12 - Перевірка роботи підсвітки

# Висновки

У ході виконання лабораторної роботи ми реалізували **шаблон проектування Observer**, який дозволяє забезпечити сповіщення підписників (об'єктів) про зміни стану суб’єкта (об'єкта-джерела). Цей шаблон продемонстрував свою ефективність у побудові динамічної системи, де текстовий редактор взаємодіє з кількома підписниками, такими як система автозбереження та сервіс підсвічування синтаксису.

Основні досягнення:

1. **Реалізація шаблону Observer:**
    - Було створено інтерфейси Observer (спостерігач) та Subject (об’єкт спостереження), які визначають взаємодію між компонентами.
    - Відповідно до шаблону, клас Main виконує роль об'єкта-джерела, а класи SyntaxHighlighterService і AutoSave реалізують інтерфейс Observer, реагуючи на зміну тексту.
2. **Динамічне сповіщення підписників:**
    - Завдяки шаблону, при кожній зміні тексту або завантаженні нового документа система сповіщає підписників, забезпечуючи їхню актуальність та синхронізацію зі станом редактора.
3. **Розширюваність системи:**
    - Архітектура, побудована на основі шаблону Observer, дозволяє легко додавати нові функціональні модулі (наприклад, модуль перевірки орфографії або аналізу коду), які будуть автоматично інтегровані в систему через механізм підписки.
    -
4. **Реалізація функціоналу:**
    - Сервіс автозбереження (AutoSave) успішно зберігає текст при кожній зміні, демонструючи переваги автоматизації.
    - Сервіс підсвічування синтаксису (SyntaxHighlighterService) коректно обробляє ключові слова, забезпечуючи точне і візуально зрозуміле виділення.