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

Лабораторна робота № 6

Технології розроблення програмного забезпечення

«Патерни проектування»

Виконала:
студентка групи ІА-32
Чайковська С. В.

Перевірив:
Мягкий М. Ю.

Київ 2025

Зміст

Тема: Патерни проектування.

Мета: Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати їх в реалізації програмної системи.

1 Теоретичні відомості

Шаблон «Abstract Factory»

Призначення: Створює сімейства пов'язаних об'єктів без вказівки їх конкретних класів. Використовується, коли необхідно забезпечити сумісність продуктів одного сімейства.

Проблема: Наприклад, потрібно створювати меблі (крісла, дивани, столики) у різних стилях (Ар-деко, Вікторіанський, Модерн), які пасуватимуть одне до одного.

Рішення:

- Виділяються загальні інтерфейси для кожного продукту.
- Для кожного сімейства продуктів створюються свої фабрики, що реалізують інтерфейс фабрики.
- Клієнт працює лише із загальними інтерфейсами, що забезпечує гнучкість і сумісність.

Переваги:

- Забезпечує сумісність продуктів.
- Звільняє код від залежності від конкретних класів.
- Реалізує принцип відкритості/закритості.

Недоліки:

- Складність через велику кількість класів.
- Обов'язкова наявність усіх типів продуктів у кожній варіації.

Шаблон «Factory Method»

Призначення: Визначає інтерфейс для створення об'єктів певного базового типу, дозволяючи підкласам замінювати об'єкти на їх підтипи. Спрощує підтримку коду та розширення функціоналу без зміни базового коду.

Проблема: Програма для вантажних перевезень спочатку працює лише з вантажівками. Для додавання інших видів транспорту (судна, літаки) потрібно змінювати весь код, що ускладнює його підтримку.

Рішення:

- Замінити пряме створення об’єктів (new) викликом фабричного методу.
- Кожен вид транспорту (Вантажівка, Судно) реалізує спільний інтерфейс «Транспорт», а підкласи логістики повертають відповідні об’єкти через фабричний метод.

Переваги:

- Усунення прив’язки до конкретних класів продуктів.
- Спрощення підтримки коду.
- Легке додавання нових продуктів.

Недоліки:

- Може створити великі паралельні ієрархії класів.

Шаблон «Memento»

Призначення: Дозволяє зберігати та відновлювати стан об’єкта без порушення інкапсуляції. Об’єкт-знімок (Memento) зберігає стан, а початковий об’єкт (Originator) може його відновлювати. Caretaker лише зберігає або передає знімки.

Проблема: У текстовому редакторі потрібно реалізувати скасування дій. Прямий доступ до полів об’єкта порушує інкапсуляцію.

Рішення:

- Об’єкт сам створює знімки свого стану й надає обмежений доступ до них іншим об’єктам.
- Caretaker зберігає знімки та передає їх назад для відновлення стану.

Переваги:

- Не порушує інкапсуляцію.
- Спрощує вихідний об’єкт, прибираючи необхідність зберігати історію станів.

Недоліки:

- Вимагає багато пам’яті при частому створенні знімків.
- Може перевантажувати пам’ять, якщо застарілі знімки не видаляються.

Шаблон «Observer»

Призначення: Визначає залежність «один-до-багатьох», дозволяючи об’єктам-спостерігачам автоматично отримувати сповіщення про зміни стану об’єкта-видавця.

Структура предсатвлена на рис. 1:

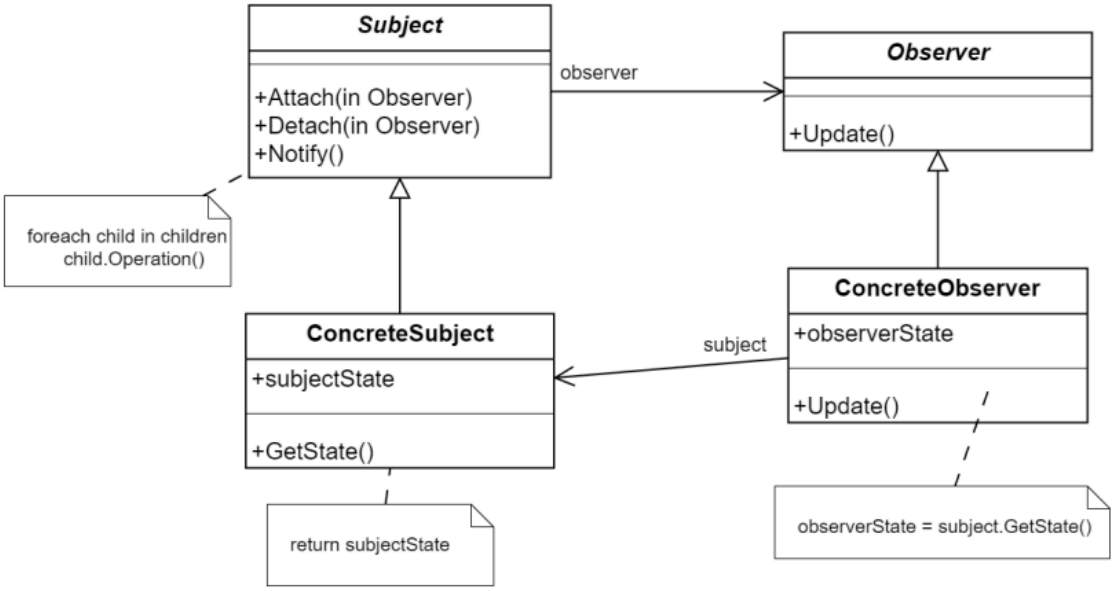


Рис. 1 – Структура патерну «Спостерігач»

Проблема: Ручне перевіряння змін (як покупець, що щодня перевіряє товар) неефективне, а масова розсилка повідомлень може викликати незадоволення користувачів.

Рішення:

- Видавець зберігає список підписників.
- Підписники додають або видаляють себе зі списку за допомогою методів видавця.
- При зміні стану видавець повідомляє підписників, викликаючи їх методи.
- Загальний інтерфейс дозволяє підписникам працювати з різними видавцями.

Переваги:

- Динамічна підписка/відписка.
- Слабке зв’язування між видавцем і підписниками.
- Відповідає принципу відкритості/закритості.

Недоліки:

- Невизначена послідовність сповіщення підписників.

Шаблон «Decorator»

Призначення: Динамічно додає функціональність об’єкту під час виконання програми, зберігаючи його базову поведінку.

Проблема: Спадкування для різних комбінацій функцій (наприклад, канали повідомлень: email, SMS, Slack) створює надмірну кількість класів.

Рішення:

- Використати композицію: декоратор обгортає об’єкт і додає функції без зміни вихідного класу.

Переваги:

- Додає функціональність «на льоту».
- Гнучкість у порівнянні зі спадкуванням.
- Відповідає принципу відкритості/закритості.

Недоліки:

- Велика кількість дрібних класів.
- Складність налаштування при багатошаровій обгортці.

2 Тема

3. **Текстовий редактор** (strategy, command, observer, template method, flyweight, SOA)

Текстовий редактор повинен вміти розпізнавати текстові файли в будь-якій кодуванні, мати розширені функції редагування: макроси, сніппети, підказки, закладки, перехід на рядок / сторінку, підсвічування синтаксису (для однієї мови програмування або розмітки на розсуд студента).

3 Аргументація вибору патерну «Observer»

Патерн *Observer* було обрано для реалізації механізму автоматичного оновлення підказок у редакторі документа в реальному часі. Його застосування дозволяє відокремити логіку генерації підказок від редактора та забезпечити гнучке сповіщення всіх зацікавлених об’єктів про зміни у тексті.

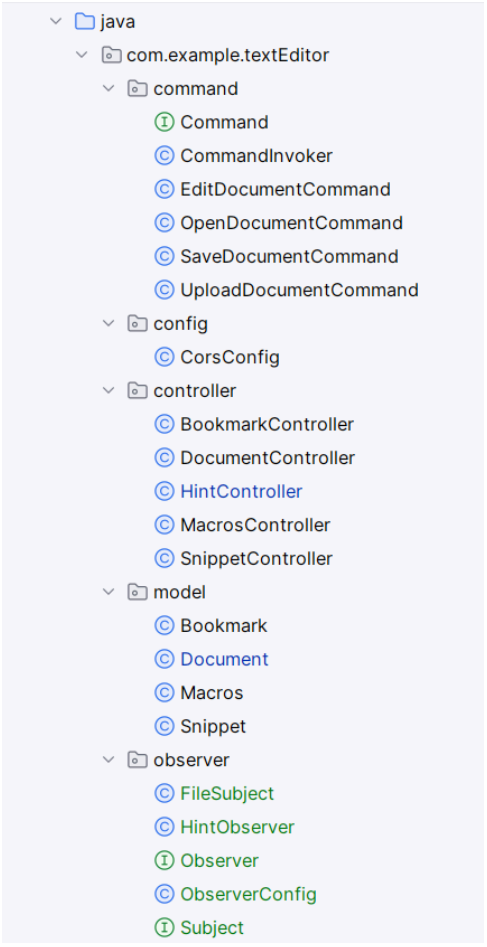
Основні аргументи вибору патерну:

- *Розділення відповідальностей.* Редактор документа не виконує безпосередньо логіку генерації підказок, а лише повідомляє про зміни. Об’єкт-підписник *(Observer)*, такий як *HintObserver*, реагує на зміни і оновлює підказки. Це забезпечує слабке зв’язування між компонентами системи.
- *Масштабованість і розширюваність.* Додавання нових підписників (наприклад, підсвічування тексту, перевірка синтаксису) можливо без зміни коду редактора або існуючих підписників — достатньо реалізувати новий *Observer* та підписати його на *Subject*.
- *Єдиний механізм сповіщення.* Всі підписники отримують повідомлення через єдиний метод *update()*, що дозволяє централізовано керувати реакцією на зміни тексту.
- *Динамічне реагування на зміни.* Патерн забезпечує автоматичне оновлення підказок у режимі реального часу, без необхідності вручну перевіряти текст після кожного введеного символу.

Таким чином, використання патерну *Observer* дозволяє побудувати гнучку та розширювану архітектуру модуля підказок у редакторі. Кожен підписник є незалежним, повторно використовуваним та централізовано керованим через *Subject*, що підвищує стійкість і підтримуваність системи.

4 Реалізація частини функціоналу робочої програми

Структура проекту зображена на Рисунку 1:



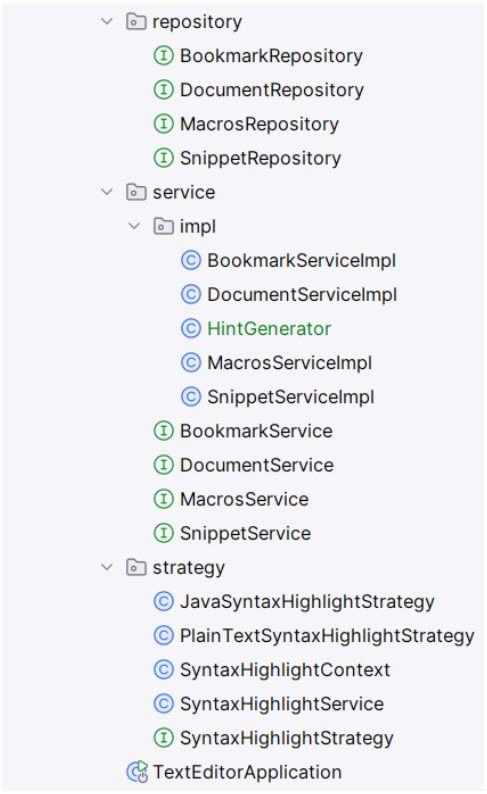


Рис. 2 – Структура проекту з використанням патерну «Command»

Структура проєкту TextEditor:

Архітектура побудована за принципом багаторівневого розподілу, що забезпечує чітке розділення відповідальностей:

- *observer* – реалізація патерну Observer (спостерігач), який дозволяє автоматично оновлювати підказки при зміні тексту документа. Основні класи:
  - *Subject* – інтерфейс, що визначає методи *attach(), detach(), notifyObservers()*.
  - *Observer* – інтерфейс, що має метод *update(String content)*.
  - *FileSubject* – конкретний суб'єкт *(ConcreteSubject)*, який зберігає поточний стан тексту *(content)* і повідомляє всіх підписаних спостерігачів про зміни.
  - *HintObserver* – конкретний спостерігач *(ConcreteObserver)*, який реагує на зміни вмісту документа, генеруючи нові підказки для користувача.
  - *ObserverConfig* – клас конфігурації Spring, який автоматично реєструє спостерігача *(HintObserver)* у суб'єкта *(FileSubject)* під час запуску застосунку.
- *command* – реалізація патерну «Command» для організації дій із документами через інкапсуляцію запитів у вигляді об'єктів. Забезпечує гнучке керування командами, підтримку історії дій. Основні класи:
  - *Command* – інтерфейс, що визначає метод *execute(),* який реалізується конкретними командами.
  - *OpenDocumentCommand, SaveDocumentCommand, UploadDocumentCommand, EditDocumentCommand* – конкретні команди, які інкапсулюють дії з відкриття, збереження, завантаження та редагування документів відповідно.

    - *CommandInvoker* – клас, що виконує команди, зберігає їх в історії та може реалізовувати скасування дій.
    - *DocumentServiceImpl* виступає отримувачем (receiver), який безпосередньо реалізує бізнес-логіку команд.
    - *DocumentController* є клієнтом, який створює конкретні команди та передає їх інвокеру для виконання.
- *controller* – містить контролери, які приймають запити від користувача, викликають бізнес-логіку та повертають відповіді на UI: *DocumentController, BookmarkController, HintController, MacrosController, SnippetController* – керують операціями з відповідними сутностями (документи, закладки, підказки, макроси, фрагменти коду).
- *model* – включає моделі даних (сутності), що відображають таблиці бази даних: *Document, Bookmark, Hint, Macros, Snippet*. Кожна з моделей описує структуру даних, які зберігаються у БД.
- *repository* – шар доступу до даних. Містить інтерфейси для роботи з БД: *DocumentRepository, BookmarkRepository, HintRepository, MacrosRepository, SnippetRepository*. Завдяки Spring Data JPA забезпечується збереження, пошук та вибірка даних без написання SQL-запитів вручну.
- *service* – шар бізнес-логіки. Інтерфейси: *DocumentService, BookmarkService, MacrosService, SnippetService.*

  Реалізації (impl): *DocumentServiceImpl, BookmarkServiceImpl, HintGenerator, MacrosServiceImpl, SnippetServiceImpl.* У цьому шарі виконується логіка обробки даних, яка відділяє контролери від репозиторіїв.
- *strategy* – реалізація патерну «Strategy» для підсвічування синтаксису. Містить інтерфейс, контекст і конкретні стратегії:
    - *SyntaxHighlightStrategy* – інтерфейс, що визначає метод *highlight(String text)* для різних алгоритмів підсвічування.

    - *JavaSyntaxHighlightStrategy* – конкретна стратегія для підсвічування синтаксису мови Java за допомогою регулярних виразів.
    - *PlainTextSyntaxHighlightStrategy* – стратегія, яка залишає текст без змін (використовується для випадків без підсвічування).
    - *SyntaxHighlightContext* – контекст, який зберігає поточну стратегію та викликає її методи.
    - *SyntaxHighlightService* – сервіс, який визначає, яку саме стратегію застосувати до тексту залежно від розширення файлу.
- *TextEditorApplication* – головний клас, що запускає Spring Boot застосунок.

Така структура проєкту дає змогу легко підтримувати й розширювати систему, оскільки контролери, бізнес-логіка та доступ до даних розділені по різних шарах.

5 Реалізація патерну «Observer»

1. ***Інтерфейс Observer*** представлений на Рисунку 3:

```
1   package com.example.textEditor.observer;
2
3   public interface Observer {
4       void update(String text);
5   }
```

Рис. 3 – Інтерфейс Observer

Інтерфейс *Observer*, який наведено на рис. 3, визначає спільний метод *update(String text)*, який реалізовується всіма конкретними спостерігачами. Його основне призначення — забезпечити єдиний механізм отримання повідомлень про зміни в системі. Кожен клас, що реалізує цей інтерфейс, визначає власну реакцію на оновлення стану суб'єкта (у нашому випадку автоматичне оновлення підказок у текстовому редакторі при зміні вмісту документа). Інтерфейс *Observer* забезпечує слабке зв'язування між об'єктами, дозволяючи суб'єкту повідомляти декілька спостерігачів про зміни без прямої залежності від їх конкретної реалізації.

2. ***Інтерфейс Subject*** представлений на Рисунку 4:

```
1   package com.example.textEditor.observer;
2
3   public interface Subject {
4       void attach(Observer observer);
5       void detach(Observer observer);
6       void notifyObservers();
7   }
```

Рис. 4 – Інтерфейс Subject

Інтерфейс *Subject*, який наведено на рис. 4, визначає методи *attach(Observer observer)*, *detach(Observer observer)* та *notifyObservers()*, які реалізуються класами-конкретними суб'єктами.

- *attach(Observer observer)* – додає нового спостерігача до списку підписників.
- *detach(Observer observer)* – видаляє спостерігача зі списку, припиняючи отримання повідомлень.

- *notifyObservers()* – повідомляє всіх підписаних спостерігачів про зміну стану суб’єкта, викликаючи їх метод *update(String text)*.

Завдяки цьому інтерфейсу *Subject* забезпечує централізоване керування підписниками та їх повідомлення про зміни, що дозволяє реалізувати патерн *"Observer"* із слабким зв’язуванням між суб’єктом та спостерігачами.

3. ***Клас FileSubject*** представлений на Рисунку 5:

```
@Component
public class FileSubject implements Subject {

    private static final Logger logger = LoggerFactory.getLogger(FileSubject.class);

    private final List<Observer> observers = new ArrayList<>();
    private String

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
        logger.info("Attached observer: {}", observer.getClass().getSimpleName());
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
        logger.info("Detached observer: {}", observer.getClass().getSimpleName());
    }


    @Override
    public void notifyObservers() {
        logger.info("Notifying {} observers...", observers.size());
        for (Observer observer : observers) {
            observer.update(content);
        }
    }
}
```

Рис. 5 – Клас FileSubject

Клас *FileSubject*, який наведено на рис. 5, є конкретною реалізацією інтерфейсу *Subject* та відповідає за зберігання стану документа і сповіщення всіх зареєстрованих спостерігачів про його зміни.

- Список *observers* зберігає усіх підписаних спостерігачів.
- Метод *attach(Observer observer)* додає спостерігача до списку і лог-інформацію про підписку.

- Метод *detach(Observer observer)* видаляє спостерігача зі списку та веде лог про видалення.
- Метод *notifyObservers()* проходить по списку спостерігачів і викликає у кожного метод *update(content)*, передаючи поточний стан документа *(content)*.

Завдяки цьому класу *FileSubject* реалізує патерн *"Observer"*, забезпечуючи централізоване повідомлення всіх спостерігачів про зміни вмісту файлу без жорсткого зв'язку з конкретними класами спостерігачів.

4. ***Клас HintObserver*** представлений на Рисунку 6:

```java
package com.example.textEditor.observer;

import com.example.textEditor.service.impl.HintGenerator;
import org.springframework.stereotype.Service;

import java.util.List;

@Service  new *
public class HintObserver implements Observer {

    private final HintGenerator hintGenerator;  2 usages

    public HintObserver() {  new *
        this.hintGenerator = new HintGenerator();
    }

    @Override  new *
    public void update(String text) {
        List<String> hints = hintGenerator.generateHints(text);
        System.out.println("HintObserver: Підказки для тексту '" + text + "': " + hints);
    }
}
```

Рис. 6 – Клас HintObserver

Клас *HintObserver*, який наведено на рис. 6, є конкретною реалізацією інтерфейсу Observer і відповідає за генерацію підказок для тексту документа під час його редагування.

- Поле *hintGenerator* зберігає об'єкт класу *HintGenerator*, який містить словник ключових слів та алгоритм генерації підказок.
- Конструктор *HintObserver()* ініціалізує *hintGenerator*.
- Метод *update(String text)* викликається при сповіщенні від суб'єкта *(FileSubject)*. Він отримує поточний текст документа, генерує відповідні підказки через *hintGenerator.generateHints(text)* та виводить їх у консоль для перевірки.

Завдяки цьому класу *HintObserver* реагує на зміни тексту у документі без необхідності безпосереднього зв’язку з контролером чи сервісом документа.

5. ***Клас ObserverConfig*** представлений на Рисунку 7:

```
1   package com.example.textEditor.observer;
2
3   import org.springframework.boot.CommandLineRunner;
4   import org.springframework.context.annotation.Bean;
5   import org.springframework.context.annotation.Configuration;
6
7   @Configuration  new *
8   public class ObserverConfig {
9
10      @Bean  new *
11      public CommandLineRunner registerObservers(FileSubject fileSubject, HintObserver hintObserver) {
12          return String[] args -> {
13              fileSubject.attach(hintObserver);
14          };
15      }
16  }
```

Рис. 7 – Сервіс ObserverConfig

Клас *ObserverConfig*, який наведено на рис. 7, відповідає за конфігурацію та реєстрацію спостерігачів у системі.

- Аннотація *@Configuration* визначає клас як конфігураційний компонент Spring.
- Метод r*egisterObservers(FileSubject fileSubject, HintObserver hintObserver)* з анотацією *@Bean* створює *CommandLineRunner*, який під час запуску застосунку підписує конкретного спостерігача *(hintObserver)* на суб’єкт *(fileSubject)*. Завдяки цьому спостерігач починає отримувати сповіщення про зміни стану *FileSubject* ще під час старту програми, забезпечуючи автоматичну інтеграцію механізму *Observer* у систему.

Цей підхід дозволяє централізовано управляти підписками спостерігачів і легко додавати нові спостереження без змін у бізнес-логіці або контролерах.

6 Додаткові класи для реалізації патерну «Observer»

1. ***Клас HintGenerator*** представлений на Рисунку 8:

```java
package com.example.textEditor.service.impl;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.springframework.stereotype.Service;

@Service  6 usages  new *
public class HintGenerator {

    private final List<String> dictionary = new ArrayList<>(Arrays.asList(  1 usage
            "print", "println", "printf", "for", "while", "if", "else",
            "int", "float", "double", "String", "class", "public", "private", "static"
    ));

    public List<String> generateHints(String input) {  2 usages  new *
        List<String> hints = new ArrayList<>();
        if (input == null || input.isEmpty()) return hints;
        String lowerInput = input.toLowerCase();
        for (String word : dictionary) {
            if (word.startsWith(lowerInput)) {
                hints.add(word);
            }
        }
        return hints;
    }
```

Рис. 8 – Клас HintGenerator

Клас *HintGenerator* відповідає за генерацію підказок для тексту в редакторі.

- Поле *dictionary* містить базовий набір ключових слів і конструкцій мови програмування, які використовуються для підказок.
- Метод *generateHints(String input)* приймає поточний фрагмент тексту та повертає список слів із dictionary, що починаються з введеного тексту.
    - Якщо *input* порожній або *null*, метод повертає порожній список.
    - Для кожного слова з *dictionary* перевіряється, чи починається воно з введеного фрагмента *(startsWith)*, і у випадку збігу додається до результату.

Клас не реалізує сам патерн *Observer*, а лише забезпечує функціонал підказок, який використовується спостерігачем *(HintObserver)* для оновлення

підказок у реальному часі під час редагування документа. Цей сервіс відокремлює логіку генерації підказок від UI та інших частин системи, дозволяючи легко розширювати словник і адаптувати підказки для різних мов програмування.

2. ***Контролер HintController*** представлений на Рисунку 9:

```
8   @RestController  Chaikovska Sofiia *
9   @RequestMapping("/file")
10  public class HintController {
11
12      private final HintGenerator hintGenerator;  2 usages
13
14      public HintController(HintGenerator hintGenerator) {  new *
15          this.hintGenerator = hintGenerator;
16      }
17
18      @PostMapping("/hints")  new *
19      public HintsResponse getHints(@RequestBody TextRequest request) {
20          List<String> hints = hintGenerator.generateHints(request.getText());
21          return new HintsResponse(hints);
22      }
23
24      public static class TextRequest {  1 usage  Chaikovska Sofiia *
25          private String text;  2 usages
26          public String getText() { return text; }  new *
27          public void setText(String text) { this.text = text; }  new *
28      }
29
30      public static class HintsResponse {  2 usages  Chaikovska Sofiia *
31          private List<String> hints;  2 usages
32          public HintsResponse(List<String> hints) { this.hints = hints; }
33          public List<String> getHints() { return hints; }
34      }
35  }
```

Рис. 9 – Контролер HintController

Клас *HintController* є контролером Spring Boot, який відповідає за надання підказок для тексту в редакторі через HTTP API.

- Аннотація *@RestController* визначає клас як REST-контролер, а *@RequestMapping("/file")* встановлює базовий шлях для запитів цього контролера.
- Поле *hintGenerator* інжектується через конструктор і використовується для генерації підказок на основі введеного тексту.
- Метод *getHints(@RequestBody TextRequest request)* обробляє POST-запити на */file/hints*:

- Приймає об’єкт *TextRequest*, що містить текст із редактора.
- Викликає *hintGenerator.generateHints(...)* для отримання списку підказок. Повертає об’єкт HintsResponse, який містить отримані підказки у вигляді JSON.

Цей контролер забезпечує зв’язок між фронтендом та сервісом генерації підказок, дозволяючи отримувати актуальні підказки під час редагування тексту. Він не реалізує сам патерн Observer, а використовує HintGenerator як джерело підказок.

7 Діаграма класів для патерну «Observer»

Діаграма класів для патерну «Observer» зображена на Рисунку 10:

Рис. 10 – Діаграма класів для патерну «Observer»

Патерн *Observer* забезпечує механізм автоматичного сповіщення об’єктів про зміну стану іншого об’єкта.

Це створює слабке зв’язування між джерелом даних *(Subject)* і підписниками *(Observers)*:

- *FileSubject* не знає, скільки спостерігачів існує і що саме вони роблять;
- *HintObserver* просто реагує на повідомлення, не змінюючи логіку джерела.

Інтерфейси:

- *Subject* — визначає контракт для об’єктів, які можуть мати підписників (спостерігачів). Забезпечує незалежність спостерігачів від конкретної реалізації об’єкта, за яким вони стежать. Містить методи:
  - *attach(Observer)* — додає нового спостерігача до списку;
  - *detach(Observer)* — видаляє спостерігача;
  - *notifyObservers()* — повідомляє всіх підписаних спостерігачів про зміну стану.
- *Observer* — визначає інтерфейс спостерігача, який реагує на зміни в об’єкті Subject. Містить метод:

- *update(String text)* — викликається при отриманні сповіщення про зміну стану об’єкта.

Конкретні реалізації:

- *FileSubject* — реалізує інтерфейс *Subject*. Виступає джерелом сповіщень для спостерігачів. Містить:
  - список спостерігачів *(List<Observer> observers)*;
  - поле content, яке представляє поточний стан.

  Під час зміни стану об’єкт викликає метод *notifyObservers()*, який надсилає оновлення всім зареєстрованим спостерігачам. Методи *attach()* і *detach()* дозволяють динамічно додавати або прибирати спостерігачів.
- *HintObserver* — реалізує інтерфейс *Observer*.

  Отримує сповіщення від *FileSubject* через метод *update(String text)* і виконує певну дію — генерує підказки на основі зміненого тексту. Для цього використовує допоміжний клас *HintGenerator*. Після обробки тексту відображає сформовані підказки у консоль. Така поведінка дозволяє легко додати інші типи спостерігачів, не змінюючи логіку *FileSubject*.

Конфігурація:

- *ObserverConfig* — конфігураційний клас Spring, який автоматично реєструє спостерігачів.

  Через метод *registerObservers(FileSubject, HintObserver)* підписує *HintObserver* на події *FileSubject.* Таким чином, при запуску застосунку всі необхідні зв’язки встановлюються автоматично.

Посилання на GitHub: https://github.com/chaikovsska/textEditor

8 Висновки

У ході виконання лабораторної роботи було реалізовано патерн проєктування «Observer» для текстового редактора, що забезпечив автоматичне сповіщення підписників про зміни у тексті документа. Завдяки цьому патерну генерація підказок була відокремлена від основного редактора, що дозволило підвищити модульність та зменшити зв’язність між компонентами системи. Використання «Observer» дозволило підписникам реагувати на зміни в

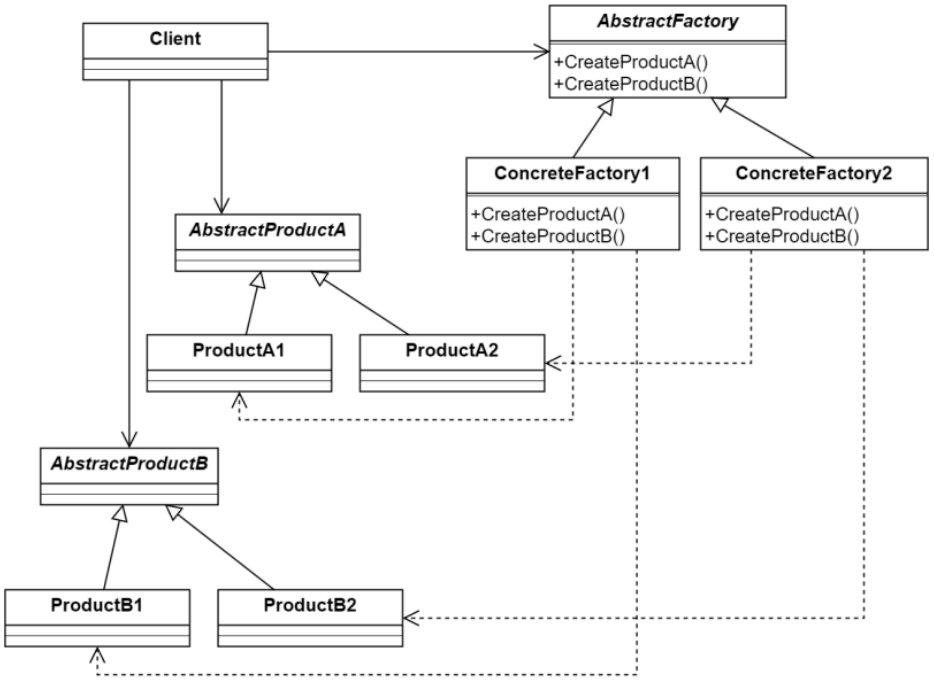
реальному часі, забезпечивши динамічне оновлення підказок і спрощуючи можливість додавання нових функцій, таких як перевірка синтаксису або підсвітка помилок, без зміни коду редактора. Реалізація цього патерну зробила архітектуру застосунку більш гнучкою, підтримуваною та придатною до розширення в майбутньому.

9 Відповіді на питання

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика (Abstract Factory)» використовується для створення сімейств взаємопов'язаних об'єктів, не вказуючи їх конкретних класів. Він дозволяє створювати об'єкти певного стилю або типу, гарантуючи узгодженість між ними (наприклад, усі елементи кімнати в одному стилі).

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

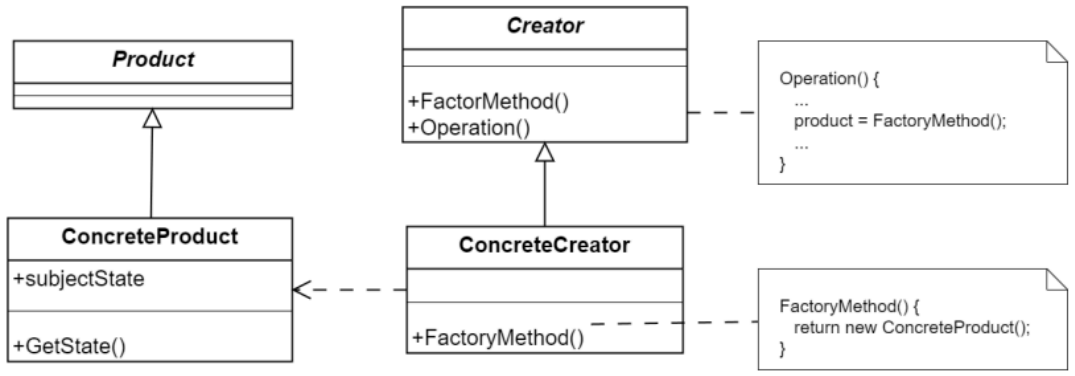
- AbstractFactory — оголошує інтерфейси створення об'єктів різних типів.
- ConcreteFactory — реалізує методи створення об'єктів конкретного стилю або сімейства.
- AbstractProduct — описує інтерфейс продукту.

- ConcreteProduct — конкретна реалізація продукту певного стилю.
- Client — використовує фабрику для створення продуктів, не знаючи їх конкретних класів.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод (Factory Method)» визначає інтерфейс для створення об’єктів, але дозволяє підкласам вирішувати, який саме клас створювати. Використовується для підміни створюваних об’єктів їх підтипами, не змінюючи код клієнта.

5. Нарисуйте структуру шаблону «Фабричний метод».



6. Які класи входять в шаблон «Фабричний метод», та яка між ними взаємодія?

- Creator (абстрактний клас) — визначає фабричний метод для створення об’єктів.
- ConcreteCreator — перевизначає фабричний метод, створюючи конкретний тип об’єкта.
- Product — загальний інтерфейс створюваних об’єктів.
- ConcreteProduct — конкретна реалізація продукту.

Клієнт працює через Creator, не знаючи, який саме об’єкт створюється.

7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

Відмінність між шаблонами «Абстрактна фабрика» та «Фабричний метод» полягає у рівні абстракції та масштабі створюваних об’єктів.

Шаблон «Фабричний метод» визначає інтерфейс для створення одного типу об’єкта, але дозволяє підкласам вирішувати, який саме клас буде створено. Тобто він відповідає за окремий продукт і використовується, коли потрібно делегувати процес створення об’єктів підкласам.

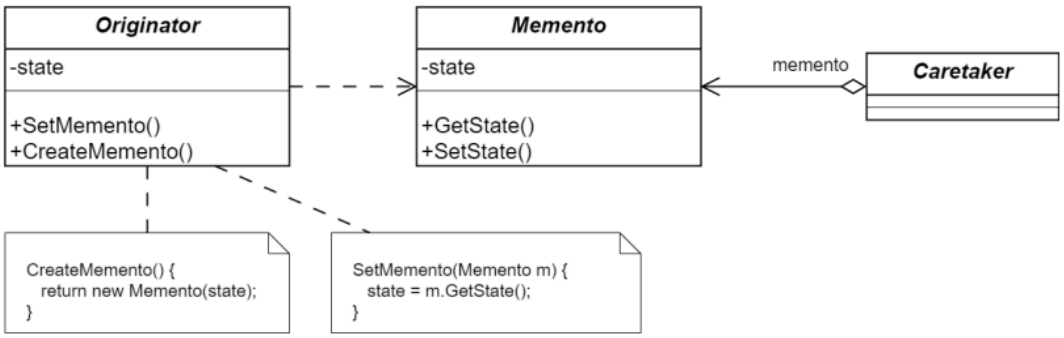
Натомість «Абстрактна фабрика» створює цілі сімейства взаємопов’язаних об’єктів, не вказуючи їх конкретних класів. Вона використовує всередині себе кілька фабричних методів і дозволяє клієнту отримувати узгоджені об’єкти, що належать до одного стилю або категорії.

«Фабричний метод» — це частковий випадок «Абстрактної фабрики»: перший створює один продукт, а друга — групу сумісних продуктів.

8. Яке призначення шаблону «Знімок»?

Шаблон «Memento» зберігає і відновлює стан об’єкта без порушення інкапсуляції. Дозволяє повернути об’єкт до попереднього стану (наприклад, «Undo» у редакторі тексту). Тобто патерн дозволяє зробити «знімок» внутрішнього стану об’єкта (його полів у певний момент часу) і зберегти його окремо. Пізніше цей знімок можна використати, щоб повернути об’єкт до попереднього стану.

9. Нарисуйте структуру шаблону «Знімок».



10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Originator

- Це об’єкт, стан якого потрібно зберігати.
- Він створює об’єкти типу Memento, які містять копію його внутрішнього стану.
- Має методи:

- save(): створює новий знімок (memento);
- restore(Memento): відновлює стан із знімка.

Memento

- Зберігає стан об’єкта Originator.
- Надає доступ лише самому Originator для відновлення стану.
- Інші класи не можуть змінювати або переглядати його внутрішній вміст (інкапсуляція збережена).

Caretaker (Опікун / Керівник)

- Відповідає за зберігання знімків.
- Не має доступу до вмісту Memento, а лише зберігає їх у стек, список або іншу структуру даних.
- Використовується для управління історією станів (наприклад, список undo/redo).

11. Яке призначення шаблону «Декоратор»?

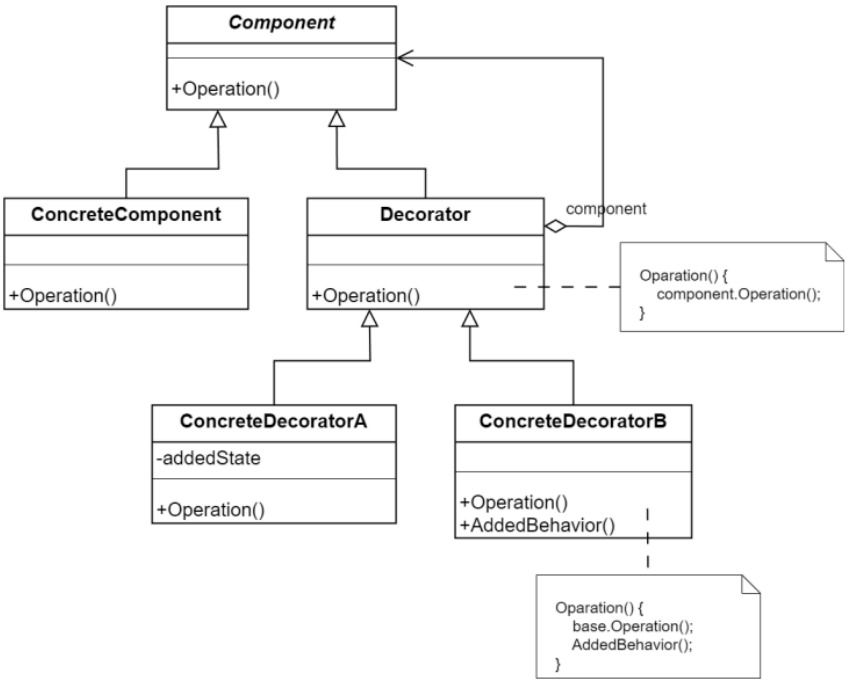
Шаблон «Декоратор (Decorator)» дозволяє динамічно додавати об’єкту нову поведінку без зміни його класу. Він «обгортає» базовий об’єкт додатковим функціоналом під час виконання програми.

Замість того щоб створювати безліч підкласів для кожної можливої комбінації функцій, ми створюємо ланцюжок декораторів, кожен з яких додає певну поведінку до базового об’єкта.

Це дозволяє:

- додавати або прибирати функціонал під час виконання програми;
- комбінувати декоратори в довільному порядку;
- уникати громіздкої ієрархії класів.

12. Нарисуйте структуру шаблону «Декоратор».



13. Які класи входять в шаблон «Декоратор», та яка між ними взаємодія?

Component (Компонент)

- Абстрактний інтерфейс, який описує базову поведінку, спільну для всіх об’єктів.

ConcreteComponent

- Клас, що реалізує базову поведінку об’єкта.

Decorator

- Абстрактний клас, який реалізує той самий інтерфейс, що й компонент, і має посилання на об’єкт типу Component.
- Делегує базовий виклик об’єкту, який декорується, і може додавати власну логіку.

ConcreteDecorator

- Наслідують клас Decorator.
- Додають нову поведінку (до або після виклику базового методу).

14. Які є обмеження використання шаблону «декоратор»?

- Ускладнює структуру програми — може створювати велику кількість дрібних класів.

- Важко розібратися, який саме набір декораторів застосований до об’єкта.
- Може бути складно керувати послідовністю викликів при кількох обгортках.