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 № 7

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 «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи.

1 Теоретичні відомості

Шаблон «Mediator» (Посередник)

Призначення: Шаблон «Mediator» забезпечує взаємодію між об’єктами через окремий об’єкт-посередник, замість прямого зв’язку між компонентами. Це дозволяє знизити складність залежностей і зробити код більш гнучким та повторно використовуваним.

Проблема: При взаємодії елементів інтерфейсу (текстові поля, кнопки, чекбокси) складна логіка може ускладнити повторне використання компонентів у різних контекстах.

Рішення: Посередник координує взаємодії між елементами, зменшуючи їх залежність один від одного та дозволяючи використовувати компоненти в різних контекстах.

Переваги:

- Знижує залежності між компонентами.
- Централізує управління взаємодією.

Недоліки:

- Посередник може стати занадто великим і складним.

Шаблон «Facade» (Фасад)

Призначення: Шаблон «Facade» надає єдиний, простий інтерфейс для доступу до складної підсистеми, приховуючи її внутрішні деталі.

Проблема: При роботі зі складними бібліотеками або фреймворками бізнес-логіка може переплітатися з деталями реалізації, ускладнюючи підтримку коду.

Рішення: Фасад забезпечує спрощений доступ до підсистеми, залишаючи тільки необхідну функціональність для клієнта.

Переваги:

- Ізолює клієнта від складної підсистеми.
- Спрощує використання складних бібліотек.

Недоліки:

- Фасад може стати великим об’єктом, що взаємодіє з усіма компонентами.

Шаблон «Bridge» (Міст)

Призначення: Шаблон «Bridge» дозволяє розділити абстракцію та її реалізацію, що спрощує додавання нових абстракцій і реалізацій незалежно одна від одної.

Проблема: Комбінування кількох властивостей (наприклад, колір і форма) призводить до вибухового зростання кількості класів при використанні спадкування.

Рішення: Використовувати композицію або агрегацію: створювати окрему ієрархію для кожної «площини» (наприклад, колір) і поєднувати її з абстракцією (наприклад, форма).

Переваги:

- Дозволяє створювати платформонезалежні програми.
- Приховує зайві деталі реалізації.
- Реалізує принцип відкритості/закритості.

Недоліки:

- Ускладнює код через додавання нових класів і шарів абстракції.

Шаблон «Template Method»

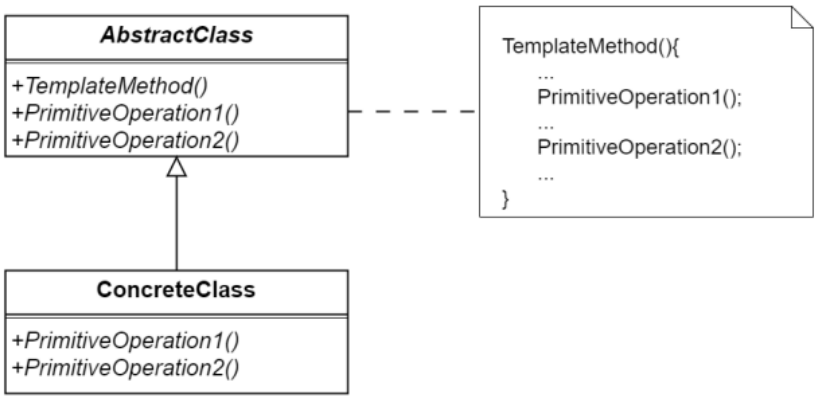


Рис. 1 – Структура патерну «Шаблонний метод»

Призначення: Шаблон «Template Method» дозволяє визначити загальний алгоритм у абстрактному класі, залишаючи конкретну реалізацію деяких кроків підкласам. Це зручно, коли потрібна однакова структура алгоритмів з можливістю змінювати окремі кроки.

Проблема: При обробці схожих даних або процесів для різних типів об’єктів виникає повторення коду та дублювання логіки.

Рішення: Визначити загальний алгоритм у базовому класі, а специфіку реалізації конкретних кроків залишити підкласам.

Приклад: Формування веб-сторінки: кроки формування заголовків, контенту, додаткових файлів і нижньої частини сторінки однакові для різних типів сторінок, але конкретна реалізація кроків (HTML, PHP, ASP.NET) може відрізнятися.

Переваги:

- Полегшує повторне використання коду.
- Спрощує підтримку: можна змінювати конкретні кроки без зміни загальної структури алгоритму.

Недоліки:

- Обмежує можливість зміни структури алгоритму через базовий клас.
- Може порушити принцип підстановки Лісков, якщо підклас змінює поведінку кроків некоректно.
- При великій кількості кроків алгоритм може стати складним для підтримки.

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 «Template Method»

Патерн *Template Method* було обрано для реалізації механізму обробки макросів у текстовому редакторі. Його застосування дозволяє відокремити загальну структуру алгоритму обробки макросу від конкретної логіки виконання окремих кроків.

Основні аргументи вибору патерну:

- *Розділення відповідальностей.* Абстрактний клас визначає послідовність кроків обробки, а конкретні класи-нащадки реалізують тільки специфічну логіку для окремих кроків. Це забезпечує слабке зв’язування між загальною структурою алгоритму та конкретними реалізаціями.
- *Масштабованість і розширюваність.* Додавання нових варіантів або реалізацій алгоритму можливе без зміни базової структури. Достатньо створити новий підклас і реалізувати необхідні методи, що дозволяє розширювати систему без порушення існуючого коду.
- *Єдиний механізм виконання алгоритму.* Шаблонний метод забезпечує централізоване керування усіма кроками алгоритму, гарантуючи правильну послідовність виконання та повторюваність дій для всіх реалізацій.
- *Гнучке налаштування поведінки окремих кроків.* Патерн дозволяє змінювати конкретну реалізацію окремих етапів алгоритму без зміни загальної структури обробки.

Таким чином, застосування патерну *Template Method* дозволяє побудувати гнучку та розширювану архітектуру, де загальний алгоритм централізовано визначається в базовому класі, а конкретні кроки можуть реалізовуватися незалежно, повторно використовуватися та легко модифікуватися без зміни основної структури алгоритму.

4 Реалізація частини функціоналу робочої програми

Структура проекту зображена на Рисунку 2:

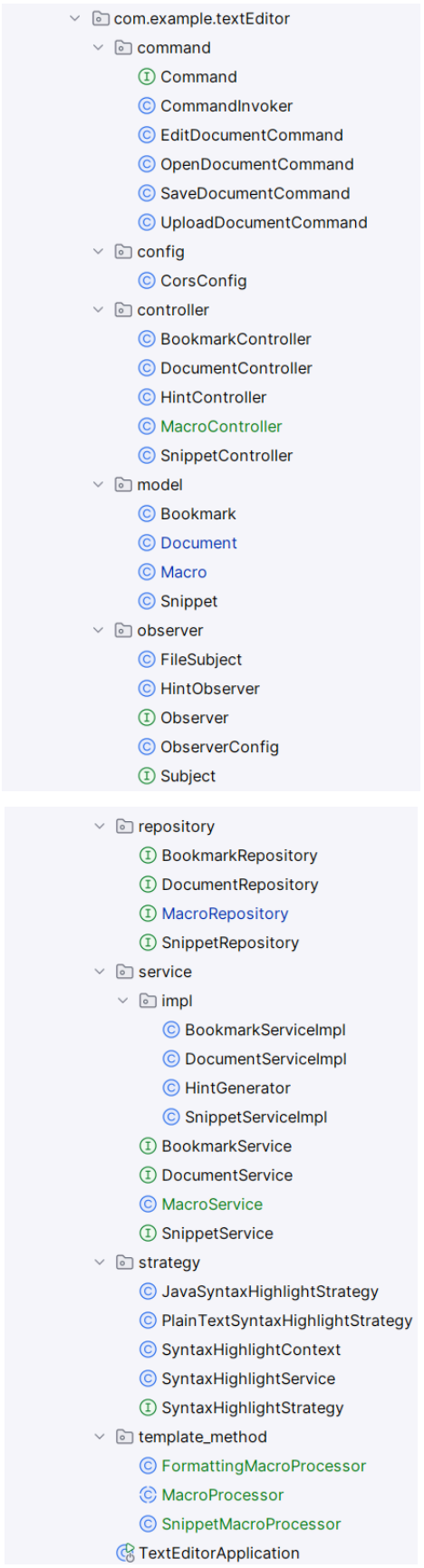


Рис. 2 – Структура проекту з використанням патерну «Template Method»

Структура проєкту TextEditor:

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

- *template_method* – реалізація патерну Template Method для стандартизації процесу обробки макросів, дозволяючи нащадкам визначати специфічну логіку кроків. Основні класи:
  - *MacroProcessor* — це абстрактний клас, який визначає контракт для всіх обробників макросів. Він встановлює уніфікований процес обробки *(process())*, який мають дотримуватися всі нащадки.
  - *SnippetMacroProcessor* — реалізує абстрактні методи (*validateInput, execute* тощо) для обробки макросів, специфічних для вставки сніпетів коду.
  - *FormattingMacroProcessor* — реалізує абстрактні методи для обробки макросів форматування тексту. Його *execute(String)* застосовує стилі тексту або змінює регістр (великий/малий).
- *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, MacroController, 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, MacroService, SnippetService.*
  Реалізації (impl): *DocumentServiceImpl, BookmarkServiceImpl, HintGenerator, 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 «Template Method»

1. ***Абстрактний клас MacroProcessor*** представлений на Рисунку 3:

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

public abstract class MacroProcessor {  5 usages  2 inheritors  new *
    protected String inputText;  1 usage

    public final String process(String inputText) {  2 usages  new *
        this.inputText = inputText;
        if (!validateInput(inputText)) {
            throw new IllegalArgumentException("Невалідний макрос");
        }
        String prepared = prepare(inputText);
        String executed = execute(prepared);
        saveResult(executed);
        return executed;
    }

    protected abstract boolean validateInput(String inputText);
    protected abstract String prepare(String inputText);
    protected abstract String execute(String preparedText);
    protected abstract void saveResult(String resultText);
}
```

Рис. 3 – Абстрактний клас MacroProcessor

Абстрактний клас *MacroProcessor*, який наведено на рис. 3, визначає спільний шаблон обробки макросів у текстовому редакторі. Його основне призначення — забезпечити єдину послідовність дій при обробці макросів. Кожен клас-нащадок, що реалізує цей абстрактний клас, визначає власну логіку для перевірки коректності макросу, підготовки тексту, виконання макросу та збереження результату.

Метод *process* фіксує загальну послідовність дій:

1. Збереження вхідного тексту у полі класу.
2. Перевірка валідності макросу через *validateInput()*.
3. Підготовка тексту до виконання через *prepare()*.
4. Виконання макросу через *execute()*.
5. Збереження результату через *saveResult()*.
6. Повернення обробленого тексту.

Абстрактний клас *MacroProcessor* забезпечує слабке зв'язування між загальною структурою алгоритму та конкретними реалізаціями кроків, дозволяючи змінювати поведінку макросів без зміни основного алгоритму обробки.

2. ***Клас FormattingMacroProcessor*** представлений на Рисунку 4:

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

public class FormattingMacroProcessor extends MacroProcessor {  2 usages  new *

    private String command;  2 usages

    @Override  1 usage  new *
    protected boolean validateInput(String inputText) {
        return inputText.startsWith("@bold(") ||
                inputText.startsWith("@italic(") ||
                inputText.startsWith("@uppercase(") ||
                inputText.startsWith("@lowercase(");
    }

    @Override  1 usage  new *
    protected String prepare(String inputText) {
        int start = 1;
        int end = inputText.indexOf("(");
        command = inputText.substring(start, end);
        return inputText.substring(end + 1, inputText.lastIndexOf( str: ")"));
    }

    @Override  1 usage  new *
    protected String execute(String text) {
        switch (command) {
            case "bold": return "<b>" + text + "</b>";
            case "italic": return "<i>" + text + "</i>";
            case "uppercase": return text.toUpperCase();
            case "lowercase": return text.toLowerCase();
            default: return text;
        }
    }

    @Override  1 usage  new *
    protected void saveResult(String resultText) {
        System.out.println("Відформатовано: " + resultText);
    }
}
```

Рис. 4 – Клас FormattingMacroProcessor

Клас *FormattingMacroProcessor*, який наведено на рис. 4, є конкретною реалізацією абстрактного класу *MacroProcessor* і визначає логіку обробки форматуючих макросів у текстовому редакторі. Його основне призначення —

забезпечити виконання конкретних команд форматування тексту, таких як жирний, курсив, верхній регістр або нижній регістр.

Кожен метод класу реалізує певний крок алгоритму шаблону:

- *validateInput* перевіряє, чи відповідає введений макрос підтримуваним командам.
- *prepare* виділяє саму команду форматування та текст для обробки.
- *execute* застосовує відповідне форматування до тексту залежно від команди.
- *saveResult* забезпечує збереження або виведення результату обробки.

Клас *FormattingMacroProcessor* реалізує слабке зв'язування між загальною структурою алгоритму та конкретними діями форматування, дозволяючи виконувати різні макроси без зміни базового алгоритму обробки тексту в абстрактному класі *MacroProcessor*.

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

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

public class SnippetMacroProcessor extends MacroProcessor {  2 usages  new *

    private String snippetName;  2 usages

    @Override  1 usage  new *
    protected boolean validateInput(String inputText) {
        return inputText.startsWith("@snippet(") && inputText.endsWith(")");
    }

    @Override  1 usage  new *
    protected String prepare(String inputText) {
        snippetName = inputText.substring(inputText.indexOf("(") + 1, inputText.lastIndexOf( str: ")"));
        return snippetName;
    }

    @Override  1 usage  new *
    protected String execute(String snippetName) {
        switch (snippetName) {
            case "forloop":
                return "for (int i = 0; i < N; i++) {\n    // код\n}";
            case "class":
                return "class MyClass {\n    private String name;\n    public MyClass(String name) { this.name = name;
            case "function":
                return "public void myFunction(int param) {\n    // код\n}";
            default:
                return "// невідомий сніпет: " + snippetName;
        }
    }
```

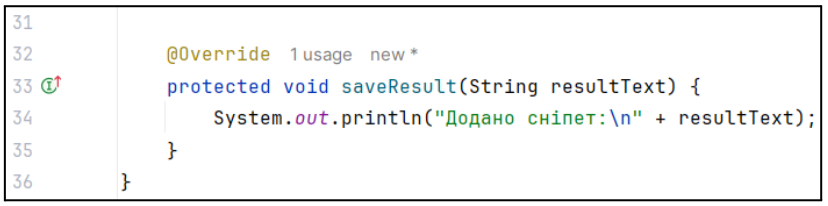


Рис. 5 – Клас SnippetMacroProcessor

Клас *SnippetMacroProcessor* є конкретною реалізацією абстрактного класу *MacroProcessor* і визначає логіку обробки макросів для вставки сніпетів коду у текстовому редакторі. Його основне призначення — забезпечити автоматичне додавання готових блоків коду (сніпетів), таких як цикли, класи або функції.

Кожен метод класу реалізує певний крок алгоритму шаблону:

- *validateInput* перевіряє, чи введений макрос відповідає формату сніпета *(@snippet(...))*.
- *prepare* виділяє назву сніпета для подальшої обробки.
- *execute* повертає відповідний код сніпета залежно від його назви або повідомляє про невідомий сніпет.
- *saveResult* забезпечує виведення результату вставки сніпета у консоль.

Клас *SnippetMacroProcessor* реалізує слабке зв’язування між загальною структурою алгоритму та конкретними діями додавання сніпетів, дозволяючи розширювати набір підтримуваних макросів без зміни базового алгоритму обробки макросів у абстрактному класі *MacroProcessor*.

6 Додаткові класи для реалізації патерну «Template Method»

1. ***Клас MacroController*** представлений на Рисунку 6:

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

import com.example.textEditor.service.MacroService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController  new *
@RequestMapping("/macro")
public class MacroController {

    private final MacroService macroService;  3 usages

    public MacroController(MacroService macroService) {  new *
        this.macroService = macroService;
    }

    @PostMapping("/format")  new *
    public ResponseEntity<String> formatMacro(
            @RequestParam String text,
            @RequestParam String type
    ) {
        String result = macroService.formatText(text, type);
        return ResponseEntity.ok(result);
    }

    @PostMapping("/snippet")  new *
    public ResponseEntity<String> snippetMacro(
            @RequestParam String type
    ) {
        String result = macroService.insertSnippet(type);
        return ResponseEntity.ok(result);
    }
}
```

Рис. 6 – Клас MacroController

Клас *MacroController* є частиною контролерного шару у архітектурі Spring і забезпечує точку взаємодії користувача з макросами у текстовому редакторі. Його основне призначення — приймати HTTP-запити на обробку макросів і делегувати їх виконання сервісному шару, який реалізує патерн Template Method.

Кожен метод класу відповідає за конкретний тип макросу:

- *formatMacro* обробляє запити на форматування тексту, приймаючи вхідний текст та тип форматування, і повертає відформатований результат.

- *snippetMacro* обробляє запити на вставку сніпетів коду, приймаючи назву сніпета і повертаючи готовий блок коду.

Клас *MacroController* реалізує слабке зв'язування між зовнішнім інтерфейсом і логікою обробки макросів, делегуючи всю конкретну роботу класам сервісного шару (*MacroService*) та класам-нащадкам *MacroProcessor*. Таким чином, контролер виконує роль додаткового класу для реалізації патерну Template Method, забезпечуючи інтеграцію користувацьких запитів з алгоритмом обробки макросів без зміни його структури.

2. ***Клас MacroService*** представлений на Рисунку 7:

```
1   package com.example.textEditor.service;
2
3   import com.example.textEditor.template_method.FormattingMacroProcessor;
4   import com.example.textEditor.template_method.MacroProcessor;
5   import com.example.textEditor.template_method.SnippetMacroProcessor;
6   import org.springframework.stereotype.Service;
7
8   @Service  3 usages  new *
9   public class MacroService {
10
11      public String formatText(String text, String formatType) {  1 usage  new *
12          MacroProcessor processor = new FormattingMacroProcessor();
13          return processor.process( inputText: "@" + formatType + "(" + text + ")");
14      }
15
16      public String insertSnippet(String snippetType) {  1 usage  new *
17          MacroProcessor processor = new SnippetMacroProcessor();
18          return processor.process( inputText: "@snippet(" + snippetType + ")");
19      }
20  }
```

Рис. 7 – Клас MacroService

Клас *MacroService* є частиною сервісного шару у архітектурі Spring і відповідає за реалізацію бізнес-логіки обробки макросів у текстовому редакторі. Його основне призначення — створювати конкретні об'єкти макропроцесорів і виконувати алгоритм обробки макросів, реалізований через патерн Template Method у класі *MacroProcessor*.

Кожен метод класу виконує конкретну задачу:

- *formatText* створює об'єкт *FormattingMacroProcessor* і виконує обробку тексту залежно від типу форматування (жирний, курсив, верхній чи нижній регістр).

- *insertSnippet* створює об’єкт *SnippetMacroProcessor* і виконує вставку відповідного сніпета коду.

Клас *MacroService* забезпечує слабке зв’язування між контролером та конкретними реалізаціями макросів, делегуючи всю специфіку обробки абстрактному класу *MacroProcessor* та його нащадкам. Таким чином, сервіс виступає як додатковий клас для реалізації патерну Template Method, який інтегрує користувацькі запити з алгоритмом обробки макросів, не змінюючи його структуру.

7 Діаграма класів для патерну «Template Method»

Діаграма класів для патерну «Template Method» зображена на Рисунку 8:

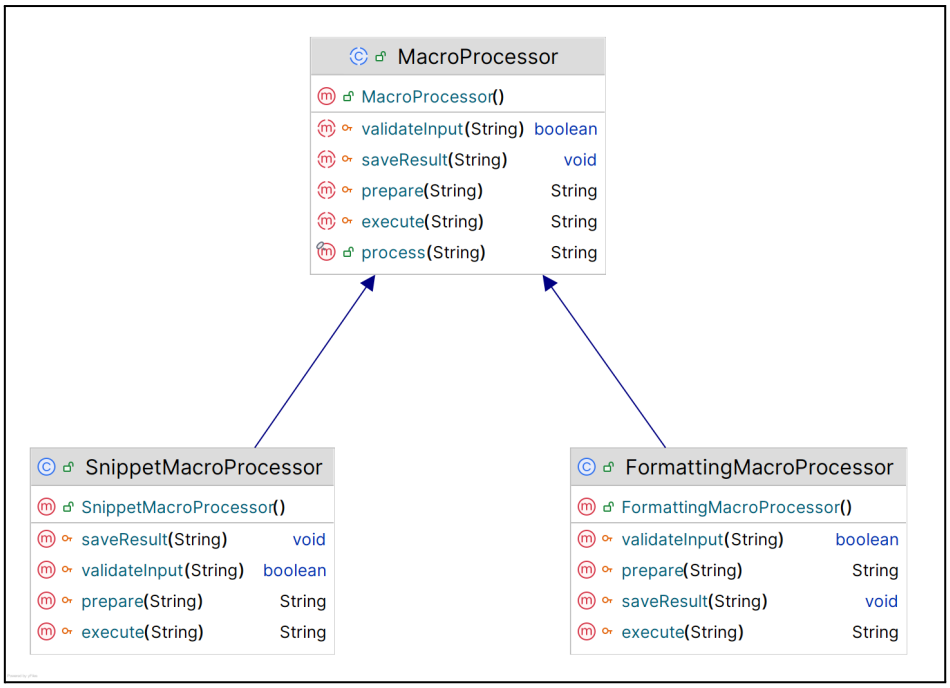


Рис. 8 – Діаграма класів для патерну «Template Method»

Патерн *Template Method* дозволяє визначити стабільну структуру алгоритму в базовому класі, залишаючи при цьому підкласам свободу реалізовувати окремі кроки за власною логікою. Завдяки цьому загальна послідовність дій залишається незмінною, а конкретна поведінка окремих етапів може змінюватися без втручання у весь алгоритм.

Абстрактний клас:

MacroProcessor — це абстрактний клас, який визначає контракт для всіх обробників макросів. Він встановлює уніфікований процес обробки, який мають дотримуватися всі нащадки.

Містить методи:

- *process(String)* — шаблонний метод, який керує виконанням всього алгоритму.
- *validateInput(String): boolean, prepare(String): String, execute(String): String, saveResult(String): void* — абстрактні примітивні операції, які повинні реалізовувати конкретні підкласи. Вони виступають як «гачки», що дозволяють нащадкам надати власну специфічну логіку для окремих кроків алгоритму.

Конкретні реалізації:

Конкретні класи-нащадки реалізують абстрактні кроки, надаючи специфічні варіації загального алгоритму. Вони реагують на виклик шаблонного методу *process()*, надаючи деталі виконання без зміни його структури.

- *SnippetMacroProcessor* — реалізує абстрактні методи для обробки макросів, специфічних для вставки сніпетів коду.
- *FormattingMacroProcessor* — реалізує абстрактні методи для обробки макросів форматування тексту. Його *execute(String)* застосовує стилі тексту або змінює регістр (великий/малий).

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

8 Висновки

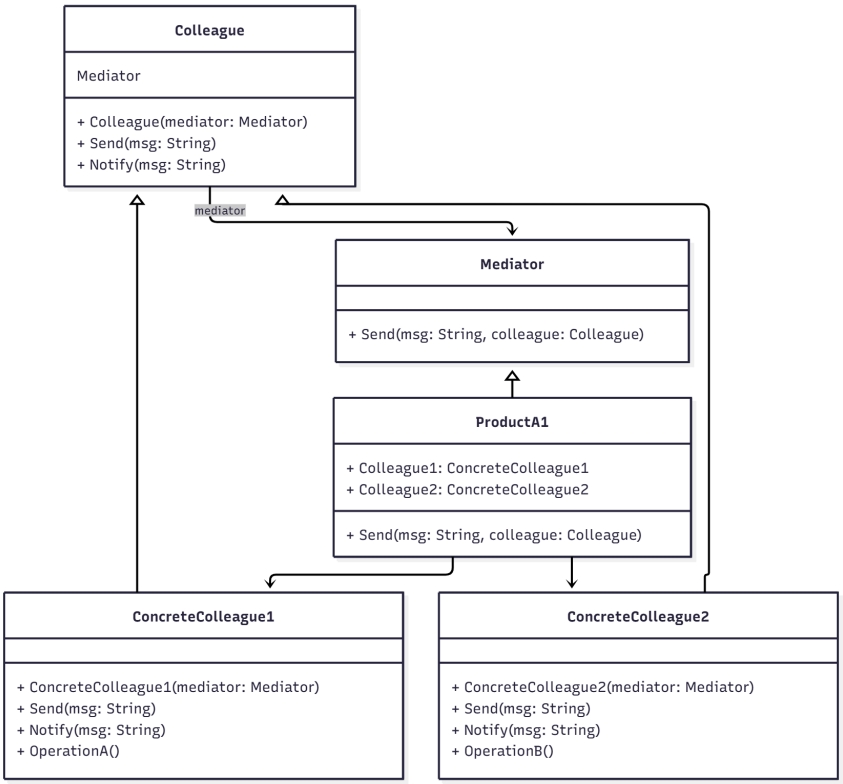
У ході виконання лабораторної роботи було реалізовано патерн проєктування «Template Method» для обробки макросів у текстовому редакторі. Завдяки цьому патерну загальний алгоритм обробки макросу був визначений у базовому класі, а конкретні кроки реалізовані у підкласах, що дозволило відокремити загальну логіку від специфічної реалізації. Використання Template Method підвищило модульність та зменшило зв'язність між компонентами системи, забезпечивши повторне використання загальних частин алгоритму. Це дозволяє легко додавати нове або змінювати окремі кроки обробки без модифікації базового класу, роблячи архітектуру застосунку більш гнучкою, підтримуваною та придатною до розширення в майбутньому.

9 Відповіді на питання

1. Яке призначення шаблону «Посередник»?

Шаблон «Посередник» використовується для організації взаємодії між об'єктами через окремий об'єкт-посередник замість прямої комунікації між ними. Це дозволяє зменшити зв'язність між компонентами системи, спростити логіку взаємодії та підвищити її масштабованість. Посередник координує об'єкти, схоже як диригент управляє оркестром: він стежить за тим, щоб кожен об'єкт виконував свою роль у потрібний час, без потреби знати про внутрішні деталі інших об'єктів.

2. Нарисуйте структуру шаблону «Посередник».



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

- Mediator – визначає загальний інтерфейс для взаємодії між об’єктами. Не містить конкретної логіки, лише визначає методи для обміну повідомленнями між колегами.
- ProductA1 (ConcreteMediator) – конкретна реалізація медіатора, яка містить правила взаємодії між колегами. Вирішує, які об’єкти і в який спосіб слід оновити після отримання повідомлення від одного з колег.
- Colleague – базовий клас або інтерфейс для об’єктів, що взаємодіють через медіатор. Кожен колега зберігає посилання на Mediator і використовує його для обміну повідомленнями з іншими об’єктами.
- ConcreteColleague – конкретна реалізація колеги. Об’єкти цього класу змінюють свій стан, надсилають повідомлення медіатору та отримують оновлення через нього. ProductA1 може бути прикладом конкретного колеги з власною специфічною логікою.

Взаємодія:

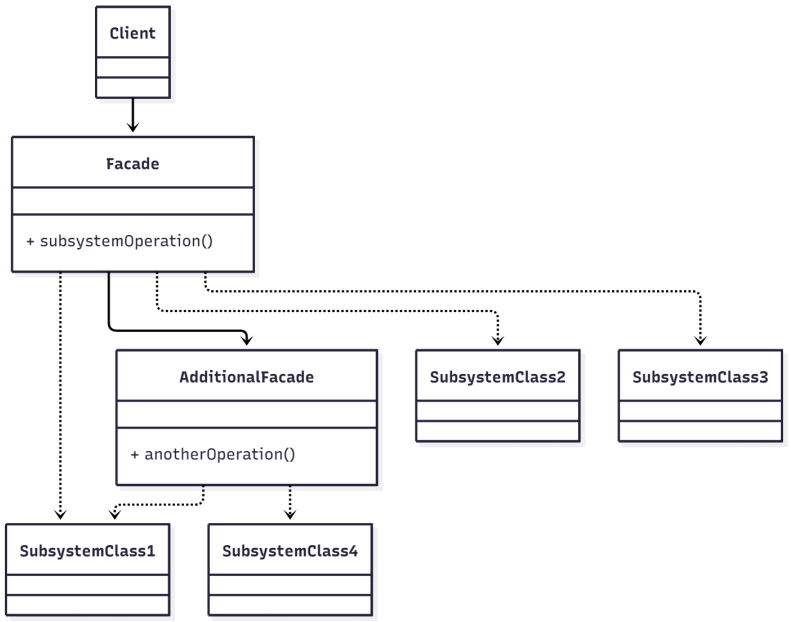
- Коли ConcreteColleague змінює стан або виконує дію, він повідомляє Mediator.
- ProductA1 (ConcreteMediator) обробляє це повідомлення та визначає, які інші колеги потрібно оновити.
- Медіатор повідомляє відповідні ConcreteColleague, і вони змінюють свій стан або реагують на подію.

Таким чином, усі колеги взаємодіють через посередника, а не напряму між собою, що спрощує керування логікою, дозволяє легко додавати нові об’єкти та зменшує зв’язність системи.

4. Яке призначення шаблону «Фасад»?

Шаблон «Фасад» створює єдиний, спрощений інтерфейс для роботи зі складною підсистемою, приховуючи внутрішню структуру класів. Це дозволяє користувачам системи працювати з простим і зрозумілим інтерфейсом, не турбуючись про складні деталі реалізації. Фасад також забезпечує ізоляцію від змін внутрішньої структури підсистеми: можна модифікувати підсистему, не змінюючи код клієнтів.

5. Нарисуйте структуру шаблону «Фасад».



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

- Facade – забезпечує спрощений інтерфейс до складної системи.
- Subsystem classes – виконують конкретні функції підсистеми.

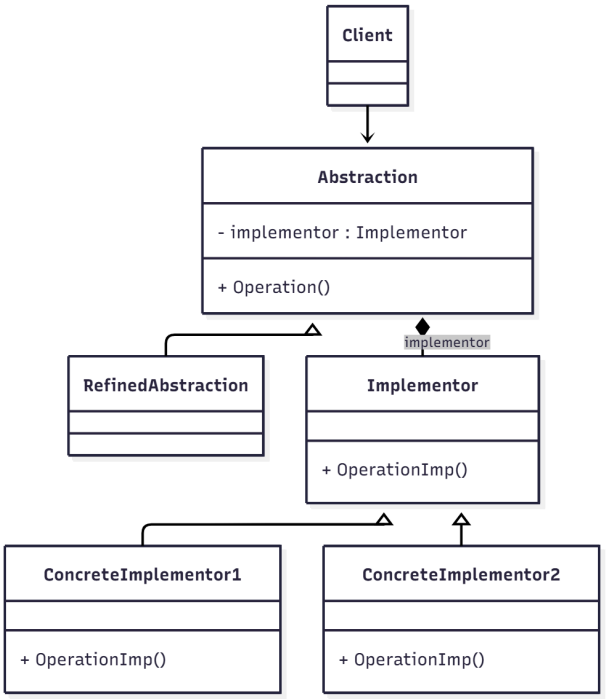
- Client – звертається до Facade, а не до підсистем напряму.

Взаємодія: Client викликає метод Facade, який усередині звертається до кількох класів підсистеми, приховуючи складність.

7. Яке призначення шаблону «Міст»?

Шаблон «Міст» дозволяє розділити абстракцію і реалізацію, щоб їх можна було змінювати незалежно. Це особливо корисно, коли існує багато варіантів абстракцій і одночасно багато способів реалізації. Наприклад, у графічному редакторі одна абстракція – фігури (лінія, коло), а реалізація – методи відображення на екрані або принтері.

8. Нарисуйте структуру шаблону «Міст».



9. Які класи входять в шаблон «Міст», та яка між ними взаємодія?

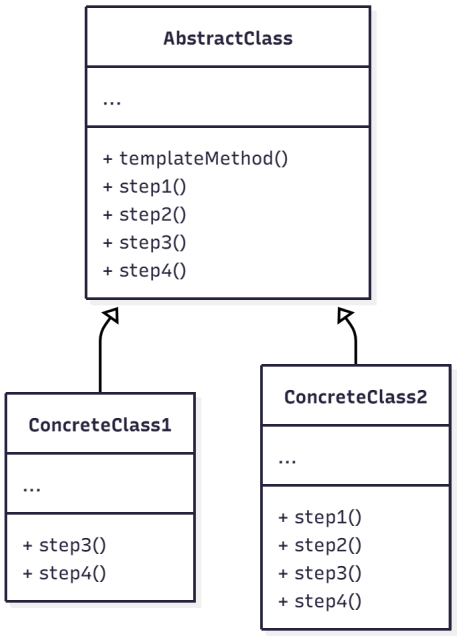
- Abstraction – інтерфейс або базовий клас абстракції, делегує роботу об’єкту реалізації.
- RefinedAbstraction – розширена абстракція, додає додаткові операції.
- Implementor – інтерфейс реалізації.
- ConcreteImplementor – конкретна реалізація.

Взаємодія: абстракція використовує об’єкт реалізації для виконання операцій. Таким чином, можна додавати нові абстракції чи реалізації без модифікації існуючих класів.

10. Яке призначення шаблону «Шаблоний метод»?

Шаблон «Шаблонний метод» визначає загальний скелет алгоритму у базовому класі, залишаючи конкретні кроки для реалізації підкласами. Це дозволяє повторно використовувати загальний код, зберігаючи при цьому можливість модифікувати окремі кроки алгоритму.

11. Нарисуйте структуру шаблону «Шаблоний метод».



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

- AbstractClass – містить загальний алгоритм (шаблонний метод) і визначає абстрактні кроки.
- ConcreteClass – реалізує конкретні кроки алгоритму.

Взаємодія: шаблонний метод викликає кроки алгоритму, частина з яких реалізується у підкласах. Це дозволяє легко додавати нові реалізації алгоритму, створюючи нові підкласи, без зміни базового коду.

13. Чим відрізняється шаблон «Шаблонний метод» від «Фабричного методу»?

Призначення патернів:

- Шаблонний метод використовується для визначення загальної структури алгоритму в базовому класі, залишаючи деякі кроки для реалізації в підкласах. Він дозволяє забезпечити повторне використання коду та уніфіковану послідовність дій, при цьому конкретна реалізація окремих кроків може змінюватися підкласами.
- Фабричний метод призначений для створення об’єктів без прив’язки клієнтського коду до конкретних класів цих об’єктів. Він делегує відповідальність за створення об’єктів підкласам, що дозволяє легко додавати нові типи продуктів без зміни існуючого коду.

Область використання:

- Шаблонний метод застосовується, коли важливо визначити покроковий алгоритм, який може змінюватися частково, залежно від конкретних потреб підкласів.
- Фабричний метод застосовується, коли потрібно створювати об’єкти певного типу, але при цьому не знати заздалегідь їх конкретну реалізацію.

14. Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» забезпечує незалежність абстракцій від реалізацій. Це дозволяє:

- уникнути комбінаційної експоненціальної кількості підкласів;
- додавати нові абстракції або реалізації без зміни існуючих класів;
- підвищити гнучкість і підтримуваність системи.