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

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Тема роботи: 3. Текстовий реадктор

Виконав
студент групи ІА–33
Супик Андрій Олександрович

Київ 2025

Тема: Патерни проектування

Мета: Вивчити структуру шаблонів «Mediator», «Facade», «Bridge», «Template method» та навчитися застосовувати їх в реалізації програмної системи

Посилання на репозиторій: https://github.com/insxlll/trpzlab

Короткі теоретичні відомості

Посередник (Mediator)

Шаблон "Посередник" забезпечує централізоване управління взаємодією між об'єктами, зменшуючи кількість прямих залежностей між ними. Використовується, коли потрібно уникнути складності, що виникає через безпосередні зв'язки між об'єктами в системі. У Java цей шаблон зазвичай реалізується шляхом створення посередника, який координує обмін даними або подіями між об'єктами, що реєструються у ньому. Це спрощує модифікацію і підтримку компонентів, зменшуючи зв'язаність між ними.

Фасад (Facade)

Шаблон "Фасад" надає єдиний інтерфейс до групи взаємопов'язаних класів або підсистем, спрощуючи їх використання. Використовується для зменшення складності системи, приховуючи її деталі реалізації та надаючи зручний API. У Java цей шаблон зазвичай реалізується через клас, який містить методи для основних операцій, що викликають функціонал внутрішніх класів або підсистем. Це дозволяє зменшити залежності між клієнтським кодом і складними підсистемами, спрощуючи розробку і підтримку.

Міст (Bridge)

Шаблон "Міст" розділяє абстракцію та її реалізацію, дозволяючи змінювати їх незалежно одна від одної. Використовується для уникнення жорсткого зв'язку між абстракцією та її конкретними реалізаціями. У Java цей шаблон реалізується через створення інтерфейсу або абстрактного класу для абстракції, а також окремих класів для реалізацій, які пов'язуються через композицію. Це забезпечує більшу гнучкість у розширенні функціональності системи.

Шаблонний метод (Template Method)

Шаблон "Шаблонний метод" визначає загальну структуру алгоритму, делегуючи реалізацію деяких його кроків підкласам. Використовується для забезпечення гнучкості та уникнення дублювання коду, коли алгоритм має спільні етапи для різних реалізацій. У Java цей шаблон реалізується через абстрактний клас із визначеним методом-шаблоном, який викликає конкретні реалізації абстрактних методів, що задаються підкласами. Це дозволяє стандартизувати алгоритм, зберігаючи варіативність його частин.

Хід роботи

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).

У цій роботі для реалізації логіки перевірки синтаксису (Syntax Checking) в текстовому редакторі було використано патерн "Шаблонний Метод" (Template Method). Цей патерн був обраний як архітектурне рішення для забезпечення стандартного, незмінного алгоритму перевірки, водночас дозволяючи підкласам (Specific Syntax Checkers) визначати конкретні кроки цього алгоритму.

Без використання патерну "Шаблонний Метод", реалізація синтаксичного аналізу для різних мов (як-от Java) вимагала б дублювання загальних, незмінних етапів (наприклад, логіки препроцесингу, постобробки помилок, або перевірки дужок) у кожному конкретному класі.

Таке дублювання логіки "розмазало" б відповідальність, порушуючи принцип Don't Repeat Yourself (DRY). Більш того, будь-яка зміна у загальній послідовності кроків (наприклад, рішення перенести постобробку помилок до початку) вимагала б модифікації всіх конкретних класів перевірки.

Діаграма класів

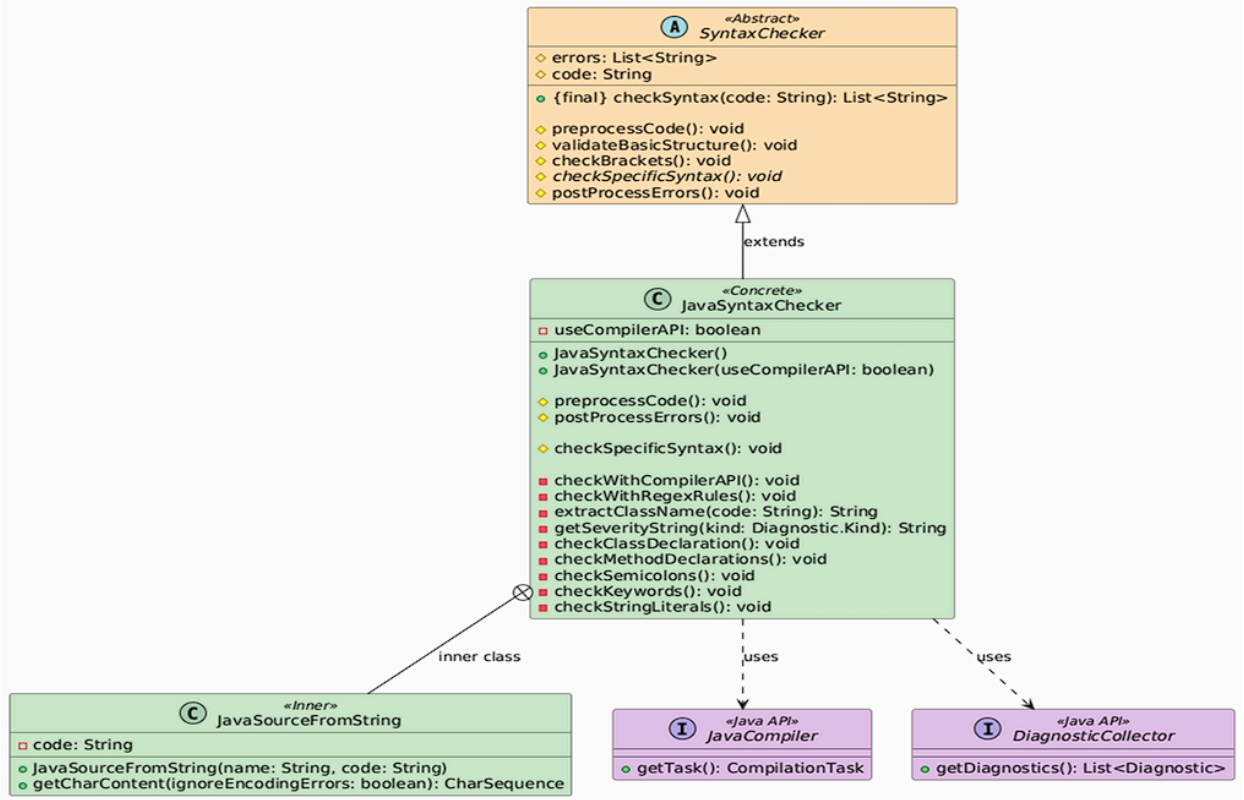


Рисунок 1 – Діаграма класів, яка представляє використання патерну Template Method

Надана діаграма класів ілюструє структуру:

- Абстрактний клас (Abstract): SyntaxChecker (Помаранчевий блок). Він визначає шаблонний метод — checkSyntax(code: List<String>) — який містить фіксований скелет алгоритму.
- Примітивні операції (Primitive Operations): Клас SyntaxChecker також містить абстрактні або конкретні методи (гачки), які будуть викликані шаблонним методом:

➢preprocessCode()

➢validateBasicStructure()

➢checkBrackets()

➢checkSpecificSyntax() (Абстрактний метод, який повинен бути реалізований підкласами).

➢postProcessErrors()

Центральною фігурою патерну є абстрактний клас SyntaxChecker (Абстрактний Клас/Шаблон), який визначає шаблонний метод:

Цей метод містить незмінний скелет (послідовність кроків) алгоритму перевірки, який гарантує, що всі перевірки, незалежно від мови, завжди виконуються в одній і тій же послідовності:

public final List<String> checkSyntax(code: String)

1. preprocessCode(): Підготовка коду (загальний гачок).
2. validateBasicStructure(): Перевірка базових вимог до структури (загальний гачок).
3. checkBrackets(): Перевірка коректності дужок (загальний гачок).
4. checkSpecificSyntax(): Головний варіативний крок (абстрактний метод).
5. postProcessErrors(): Фінальна обробка та форматування помилок (загальний гачок).

Без використання патерну "Шаблонний Метод", реалізація синтаксичного аналізу для різних мов (Java, Python) вимагала б дублювання коду для загальних етапів (таких як препроцесинг, базова перевірка структури, перевірка дужок) у кожному конкретному класі перевірки.

Наприклад, логіка завантаження коду в List<String>, зберігання та постобробки помилок, якби вона була "розмазана" по JavaSyntaxChecker, PythonSyntaxChecker тощо, порушувала б принцип Don't Repeat Yourself (DRY) і ускладнювала б підтримку: будь-яка зміна у загальній послідовності кроків вимагала б модифікації всіх класів.

Патерн "Шаблонний Метод" вирішує це, ізолюючи загальний алгоритм у базовому класі (SyntaxChecker) і перекладаючи відповідальність за специфічні деталі на підкласи:

1. Базовий Клас (SyntaxChecker): Фіксує потік керування та забезпечує послідовність усіх операцій. Він містить загальну логіку, яка повторюється для всіх мов.

2. Конкретний Клас (JavaSyntaxChecker): Цей клас розширює (extends) абстрактний SyntaxChecker і надає унікальну реалізацію для абстрактного методу checkSpecificSyntax(). Саме тут зосереджена вся складна, специфічна для Java логіка, включно з викликами Фасаду (checkWithCompilerAPI()) та внутрішніми низькорівневими перевірками (checkMethodDeclarations(), checkKeywords()).

Завдяки цьому, основна бізнес-логіка редактора працює лише з абстрактним типом SyntaxChecker, викликаючи шаблонний метод.

Код класів:

```
public class SyntaxCheckController {

@PostMapping("/check")
public ResponseEntity<SyntaxCheckResponse> checkSyntax(@RequestBody
SyntaxCheckRequest request) {
try {
String code = request.getCode();
String language = request.getLanguage();

if (code == null || code.trim().isEmpty()) {
return ResponseEntity.badRequest()
.body(new SyntaxCheckResponse(
List.of("Код порожній"),
false,
"Неможливо перевірити порожній код"
));
}

SyntaxChecker checker = getSyntaxChecker(language);
List<String> errors = checker.checkSyntax(code);

boolean hasRealErrors = errors.stream()
.anyMatch(e -> e.contains("ПОМИЛКА"));
boolean success = !hasRealErrors;

String message = errors.isEmpty() || success
? "Синтаксис коректний"
: "Знайдено проблеми у коді";

return ResponseEntity.ok(new SyntaxCheckResponse(errors, success,
message));
```

```
} catch (Exception e) {
e.printStackTrace();
return ResponseEntity.internalServerError()
.body(new SyntaxCheckResponse(
List.of("Помилка сервера: " + e.getMessage()),
false,
"Внутрішня помилка сервера"
));
}
}

private SyntaxChecker getSyntaxChecker(String language) {
if (language == null) {
language = "java";
}
switch (language.toLowerCase()) {
case "java":
return new JavaSyntaxChecker();
default:
throw new IllegalArgumentException("Непідтримувана мова: " + language);
}
}
```

Код класу SyntaxCheckController

```
public static class SyntaxCheckRequest {
private String code;
private String language;

public String getCode() {
return code;
}

public void setCode(String code) {
this.code = code;
}

public String getLanguage() {
return language;
}

public void setLanguage(String language) {
this.language = language;
}
}

public static class SyntaxCheckResponse {
private List<String> errors;
private boolean success;
```

```
private String message;

public SyntaxCheckResponse(List<String> errors, boolean success, String
message) {
this.errors = errors;
this.success = success;
this.message = message;
}

public List<String> getErrors() {
return errors;
}

public void setErrors(List<String> errors) {
this.errors = errors;
}

public boolean isSuccess() {
return success;
}

public void setSuccess(boolean success) {
this.success = success;
}

public String getMessage() {
return message;
}

public void setMessage(String message) {
this.message = message;
}
}
}
```

Код класу SyntaxCheckRequest

```
class JavaSourceFromString extends SimpleJavaFileObject {
private final String code;

public JavaSourceFromString(String name, String code) {
super(
URI.create("string:///" + name.replace('.', '/') +
Kind.SOURCE.extension),
Kind.SOURCE
);
this.code = code;
}
```

```
@Override
public CharSequence getCharContent(boolean ignoreEncodingErrors) {
return code;
}
}

class JavaSourceFromString extends SimpleJavaFileObject {
private final String code;

public JavaSourceFromString(String name, String code) {
super(
URI.create("string:///" + name.replace('.', '/') +
Kind.SOURCE.extension),
Kind.SOURCE
);
this.code = code;
}

@Override
public CharSequence getCharContent(boolean ignoreEncodingErrors) {
return code;
}
}

public abstract class SyntaxChecker {
protected List<String> errors = new ArrayList<>();
protected String code;


public final List<String> checkSyntax(String code) {
this.code = code;
this.errors.clear();
preprocessCode();
validateBasicStructure();
checkBrackets();
checkSpecificSyntax();
postProcessErrors();
return new ArrayList<>(errors);
}


protected void preprocessCode() {
}


protected void validateBasicStructure() {
if (code == null || code.trim().isEmpty()) {
errors.add("ПОМИЛКА: Код порожній");
}
```

```
}


protected void checkBrackets() {
int round = 0, square = 0, curly = 0;
for (int i = 0; i < code.length(); i++) {
char c = code.charAt(i);
switch (c) {
case '(': round++; break;
case ')': round--; break;
case '[': square++; break;
case ']': square--; break;
case '{': curly++; break;
case '}': curly--; break;
}
if (round < 0) errors.add("ПОМИЛКА: Зайва закриваюча кругла дужка на
позиції " + i);
if (square < 0) errors.add("ПОМИЛКА: Зайва закриваюча квадратна дужка на
позиції " + i);
if (curly < 0) errors.add("ПОМИЛКА: Зайва закриваюча фігурна дужка на
позиції " + i);
}
if (round > 0) errors.add("ПОМИЛКА: Незакриті круглі дужки: " + round);
if (square > 0) errors.add("ПОМИЛКА: Незакриті квадратні дужки: " +
square);
if (curly > 0) errors.add("ПОМИЛКА: Незакриті фігурні дужки: " + curly);
}


protected abstract void checkSpecificSyntax();


protected void postProcessErrors() {
}
}
```

Код класу JavaSourceFromString

Висновок: Виконуючи цю лабораторну роботу, я ознайомився з принципами та структурою ключових патернів проєктування, зокрема «Facade», «Template Method», «Factory Method» та «Composite» (у контексті структури компілятора).

Особливу увагу я приділив реалізації шаблону "Шаблонний Метод" (Template Method), детально описавши його основну логіку та функціональність у контексті підсистеми синтаксичного аналізу (Syntax Checking) текстового редактора. Цей патерн було обрано як архітектурне рішення для забезпечення стандартизованого та незмінного алгоритму перевірки коду, водночас дозволяючи підкласам вносити специфічну, мовно-залежну логіку.

Під час реалізації шаблону "Шаблонний Метод" я зрозумів, наскільки цей патерн елегантно вирішує проблему дублювання коду та порушення принципу відкритості/закритості. Базовий клас SyntaxChecker фіксує послідовність кроків (таких як preprocessCode(), checkBrackets(), postProcessErrors()), а конкретний клас JavaSyntaxChecker реалізує лише варіативну частину (checkSpecificSyntax()). Це дозволяє "Клієнту" (основній логіці редактора) викликати єдиний метод checkSyntax(), не турбуючись про те, як саме вбудовані унікальні перевірки для Java.

Цей досвід роботи з патерном "Шаблонний Метод" дозволив мені краще зрозуміти його переваги у контексті побудови фреймворків та бібліотек. Я усвідомив його ключову роль у зниженні зв'язаності між загальним алгоритмом і його конкретними реалізаціями, а також у підтримці гнучкої та розширюваної архітектури програмної системи, де додавання підтримки нової мови вимагатиме лише мінімальних змін.

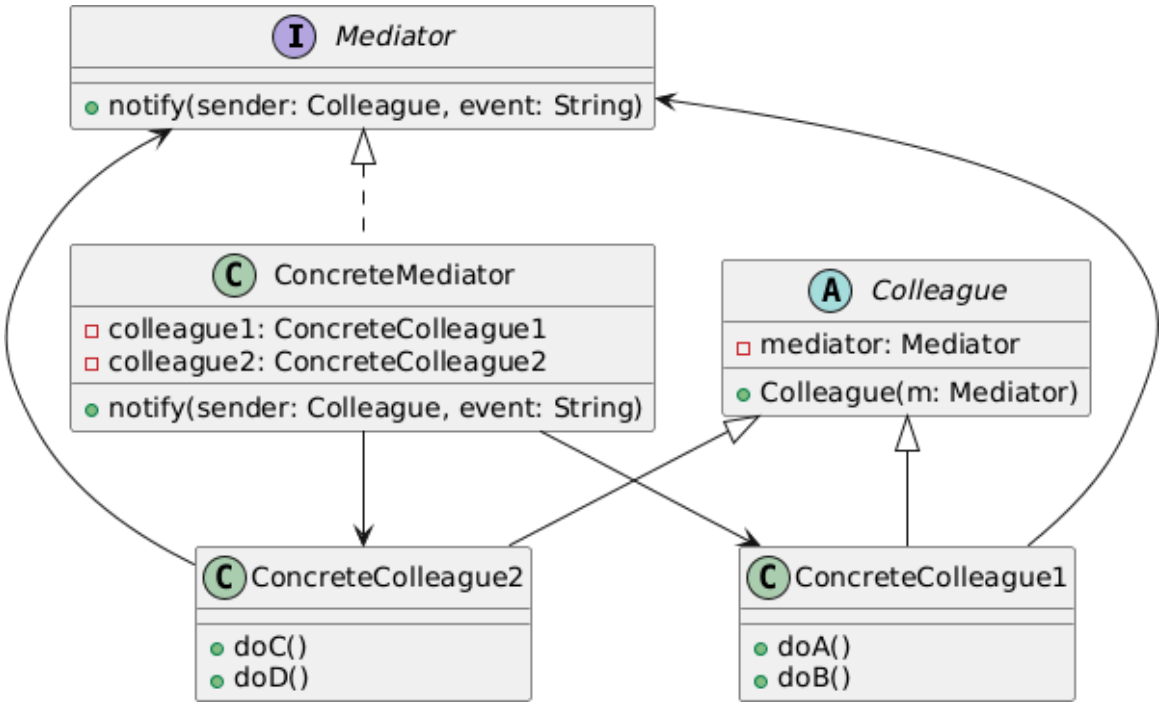
Контрольні запитання

1. Яке призначення шаблону «Посередник»?

Шаблон «Посередник» (Mediator) призначений для спрощення взаємодії між об’єктами у системі. Замість того, щоб об’єкти напряму обмінювалися повідомленнями та створювали заплутані зв’язки, усі вони спілкуються через єдиний об’єкт-посередник.

Посередник контролює обмін даними між компонентами, координує їхню роботу та зменшує кількість залежностей у програмі. Завдяки цьому система стає більш гнучкою та легкою в супроводі, оскільки зміни в одному компоненті не потребують змін в інших.

2. Нарисуйте структуру шаблону «Посередник».



3. Які класи входять в шаблон «Посередник», та яка між ними взаємодія? До шаблону «Посередник» (Mediator) входять такі основні класи:

- Mediator (Посередник) — це інтерфейс або абстрактний клас, який визначає методи для обміну інформацією між об’єктами.
- ConcreteMediator (Конкретний посередник) — реалізує інтерфейс посередника та координує взаємодію між конкретними об’єктами (колегами). Він знає, які об’єкти беруть участь у взаємодії, і спрямовує повідомлення між ними.
- Colleague (Колега) — це базовий клас або інтерфейс для об’єктів, які взаємодіють через посередника.
- ConcreteColleague (Конкретний колега) — об’єкти, які виконують певні дії, але не спілкуються напряму між собою. Вони надсилають повідомлення посереднику, а той уже вирішує, кому їх передати.

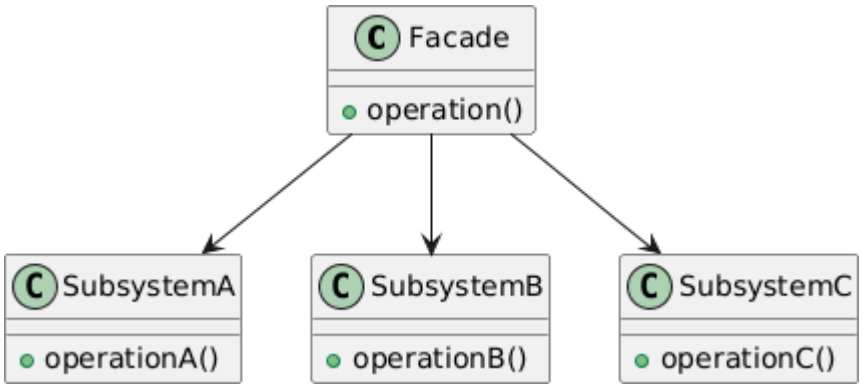
Коли один із колег хоче повідомити інший об’єкт, він не викликає його метод безпосередньо, а звертається до посередника. Посередник приймає повідомлення та вирішує, який інший колега повинен отримати цю інформацію. Таким чином, усі зв’язки проходять через посередника, що робить систему більш керованою й менш залежною.

4. Яке призначення шаблону «Фасад»?

Шаблон «Фасад» (Facade) призначений для спрощення роботи зі складною системою. Він надає єдиний узагальнений інтерфейс для взаємодії з групою класів або підсистем, приховуючи їхню внутрішню складність.

Завдяки фасаду клієнт може виконувати складні операції за допомогою одного простого виклику методу, не знаючи, як саме усе реалізовано всередині. Це робить код зрозумілішим, зручнішим у використанні та легшим у супроводі.

5. Нарисуйте структуру шаблону «Фасад».



6. Які класи входять в шаблон «Фасад», та яка між ними взаємодія? До шаблону «Фасад» (Facade) входять такі основні класи:

- Facade (Фасад) — головний клас, який надає спрощений інтерфейс для клієнта. Він містить посилання на класи підсистеми та викликає їхні методи у потрібному порядку.
- Subsystem classes (Класи підсистеми) — це реальні компоненти системи, які виконують основну роботу. Вони не знають про фасад і можуть використовуватися незалежно від нього.
- Client (Клієнт) — це код, який використовує фасад, щоб отримати доступ до функціональності підсистеми через простий інтерфейс.

Клієнт звертається лише до фасаду, не взаємодіючи безпосередньо з класами підсистеми. Фасад приймає запит, викликає необхідні методи підсистеми та повертає результат. Таким чином, фасад приховує складність внутрішньої структури системи й спрощує роботу з нею.

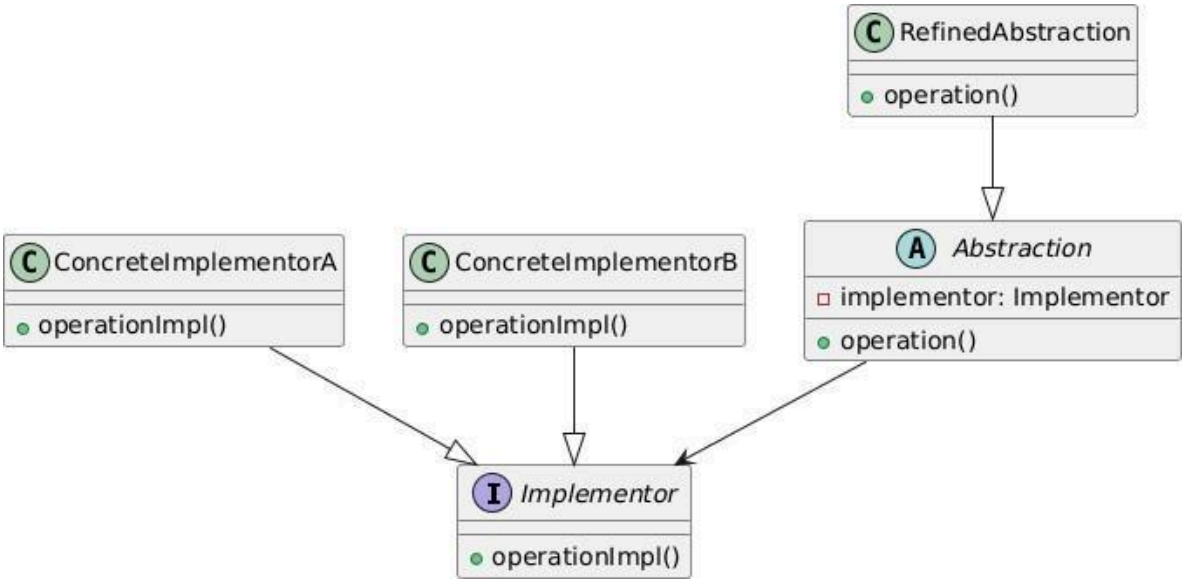
7. Яке призначення шаблону «Міст»?

Шаблон «Міст» (Bridge) призначений для відокремлення абстракції від її реалізації, щоб вони могли розвиватися незалежно одна від одної.

Замість того, щоб жорстко пов’язувати абстрактний клас із конкретною реалізацією, «Міст» створює між ними окремий зв’язок через інтерфейс. Це дозволяє легко змінювати або розширювати як абстракцію, так і реалізацію, не впливаючи одна на одну.

У результаті код стає більш гнучким і масштабованим, особливо коли потрібно підтримувати різні варіанти реалізацій або платформ.

8. Нарисуйте структуру шаблону «Міст».



9. Які класи входять в шаблон «Міст», та яка між ними взаємодія? До шаблону «Міст» (Bridge) входять такі основні класи:

- Abstraction (Абстракція) — визначає базовий інтерфейс для користувачів і містить посилання на об’єкт реалізації. Вона делегує частину роботи об’єкту реалізації, замість того щоб виконувати її самостійно.
- RefinedAbstraction (Уточнена абстракція) — розширює функціональність базової абстракції, але все одно використовує реалізацію через міст.
- Implementor (Реалізатор) — це інтерфейс, який визначає методи, що мають бути реалізовані в конкретних реалізаторах.
- ConcreteImplementor (Конкретний реалізатор) — надає специфічну реалізацію методів, оголошених в інтерфейсі Implementor.

Абстракція зберігає посилання на об’єкт типу Implementor і викликає його методи для виконання конкретних дій. Клієнт працює лише з абстракцією, не знаючи про деталі реалізації. Таким чином, абстракція та реалізація можуть змінюватися незалежно, що забезпечує гнучкість і розширюваність системи.

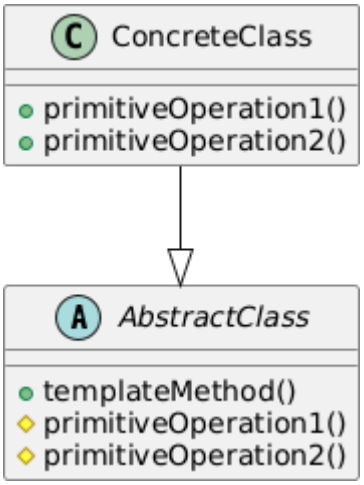
10. Яке призначення шаблону «Шаблоний метод»?

Шаблон «Шаблонний метод» (Template Method) призначений для визначення загальної послідовності дій алгоритму в базовому класі, залишаючи реалізацію окремих кроків підкласам.

Це дозволяє створити спільний “каркас” алгоритму, який не потрібно переписувати в кожному класі, а лише перевизначати окремі частини, що відрізняються.

Завдяки цьому шаблон забезпечує повторне використання коду, підтримує єдину структуру процесу та дозволяє легко змінювати поведінку алгоритму через наслідування.

11. Нарисуйте структуру шаблону «Шаблонний метод».



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

До шаблону «Шаблонний метод» (Template Method) входять такі основні класи:

- AbstractClass (Абстрактний клас) — визначає шаблонний метод, який описує загальну послідовність виконання алгоритму. Деякі кроки алгоритму реалізовані тут, а деякі оголошені як абстрактні методи, які мають бути реалізовані підкласами.
- ConcreteClass (Конкретний клас) — наслідує абстрактний клас і реалізує абстрактні методи, тобто визначає конкретну поведінку окремих кроків алгоритму.

Клієнт викликає шаблонний метод абстрактного класу. Шаблонний метод послідовно викликає внутрішні методи: частина з них реалізована в абстрактному класі, частина — у підкласах. Підклас відповідає лише за деталі окремих кроків, не змінюючи загальну структуру алгоритму.

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

Шаблон «Шаблонний метод» використовується для визначення загальної структури алгоритму в базовому класі, залишаючи підкласам реалізацію окремих кроків. Він дозволяє повторно використовувати код і змінювати лише деталі алгоритму без зміни його загальної структури.

Шаблон «Фабричний метод» застосовується для створення об’єктів без жорсткого зв’язку з конкретними класами. Підкласи вирішують, який конкретний об’єкт створювати, а клієнт користується лише базовим інтерфейсом або класом.

Основна відмінність полягає в тому, що «Шаблонний метод» визначає послідовність дій, а «Фабричний метод» — спосіб створення об’єктів.

14. Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» (Bridge) додає функціональність для незалежного розділення абстракції та її реалізації. Він дозволяє змінювати або розширювати абстракцію і реалізацію окремо, не впливаючи одна на одну. Завдяки цьому система стає гнучкішою, масштабованішою та легшою для підтримки, особливо коли потрібно підтримувати кілька варіантів реалізації або платформ.