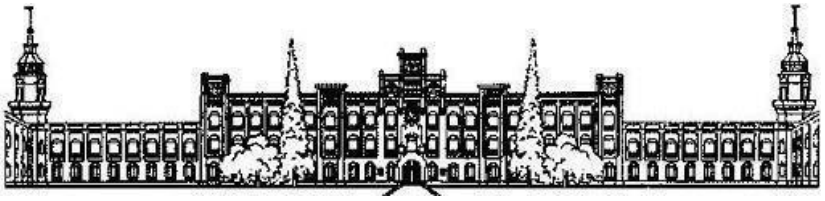


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №7

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування.»

Виконав:
студент групи ІА-32
Діденко Я.О

Перевірив:
Мягкий М.Ю

Київ 2025

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

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Теоретичні Відомості

Поняття та структура шаблонів «Mediator», «Facade», «Bridge», «Template Method»

Шаблони проєктування - це узагальнені архітектурні рішення, що допомагають створювати структуровані та гнучкі програмні системи. Вони описують типові способи взаємодії між об’єктами, забезпечуючи зниження зв’язності, підвищення повторного використання коду й спрощення підтримки системи. Нижче розглянуто чотири важливі шаблони різних типів.

Mediator (Посередник) - поведінковий шаблон, який інкапсулює взаємодію між кількома об’єктами в окремому посереднику. Об’єкти не спілкуються безпосередньо, а лише через посередника, що координує їхню роботу. Це зменшує кількість прямих зв’язків між компонентами і полегшує зміну логіки взаємодії.

Facade (Фасад) - структурний шаблон, що надає спрощений єдиний інтерфейс до складної підсистеми. Фасад приховує внутрішні деталі реалізації та об’єднує виклики кількох компонентів у зручні методи, завдяки чому клієнтський код стає чистішим і зрозумілішим. Використовується для спрощення доступу до великих або заплутаних API.

Bridge (Міст) - структурний шаблон, який розділяє абстракцію та реалізацію, дозволяючи змінювати їх незалежно. Абстракція визначає високорівневий інтерфейс, а реалізація відповідає за конкретну поведінку. Це забезпечує розширюваність системи, коли потрібно підтримувати кілька варіантів реалізацій без множинного успадкування.

Template Method (Шаблонний метод) - поведінковий шаблон, що задає загальний алгоритм у базовому класі, залишаючи реалізацію окремих кроків підкласам. Такий підхід дозволяє визначити структуру операцій один раз, а конкретну поведінку — змінювати у спадкоємцях. Патерн часто використовується для стандартизації процесів із варіативними етапами.

Шаблон «Template Method»

Призначення: Шаблон «Template Method» (шаблонний метод) дозволяє реалізувати покроково алгоритм в абстрактному класі, але залишити специфіку реалізації підкласам. Можна привести в приклад формування веб сторінки: необхідно додати заголовки, вміст сторінки, файли, що додаються, і нижню частину сторінки. Код для додавання вмісту сторінки може бути абстрактним і реалізовуватися в різних класах – AspNetCompiler, HtmlCompiler, PhpCompiler і т.п. Додавання всіх інших елементів виконується за допомогою вихідного абстрактного класу з алгоритмом.
Даний шаблон дещо нагадує шаблон «Фабричний метод», однак область його використання абсолютно інша – для покрокового визначення конкретного алгоритму; більш того, даний шаблон не обов'язково створює нові об'єкти – лише визначає послідовність дій.

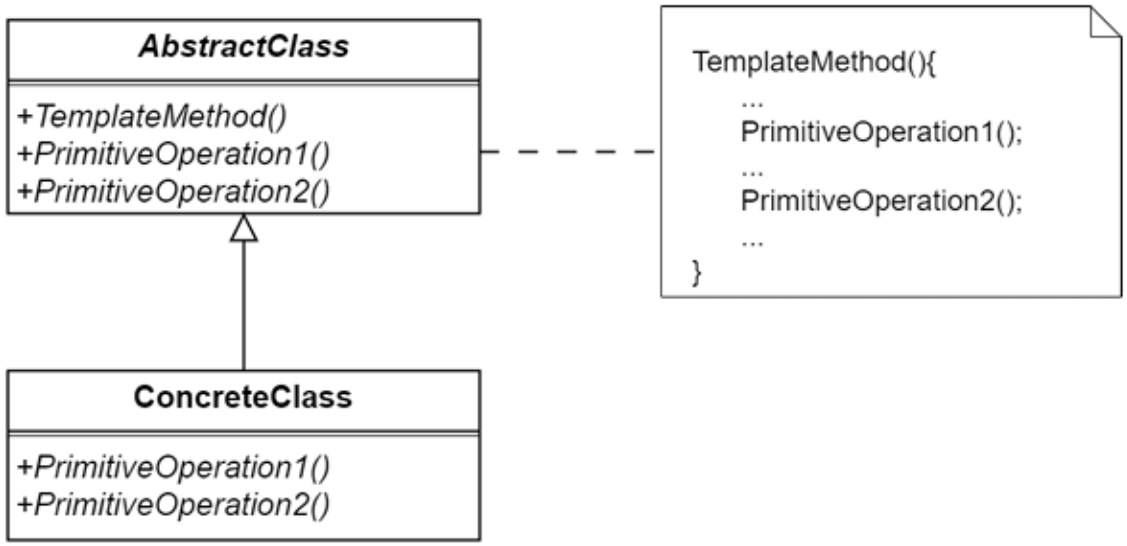


рис 1. Структура патерну Template Method

Проблема: Ви працюєте в команді, що займається розробкою застосунку для редагування відео-файлів. Застосунок вже працює з форматом відео MPEG 4, а саме дозволяє читати такі файли, виконувати попередню обробку даних для відображення в відео-редакторі.
Ви отримуєте нову задачу на реалізацію можливості роботи з більш старим форматом MPEG-2. Ви бачите два варіанта: зробити копію існуючого класу, що працює з MPEG-4, або вносити зміни в уже існуючий клас. Щоб прийняти рішення ви більш детально розбираєтеся з існуючим

алгоритмом і бачите, що близько 70 відсотків коду має бути таким самим. Тому ви вирішуєте змінити вже існуючий клас для роботи з MPEG-4 добавляючи в місцях де це потрібно умови з перевіркою, що якщо формат MPEG-2 то відпрацьовувати новий код, який ви добавили. Через деякий час, на запити від користувачів, вам на реалізацію приходить задача добавити підтримку ще більш старого формату MPEG-1. Ви вносите зміни так само в існуючий клас, тільки умови стали більш складними, тому що розгалуження логіки йде на три гілки.
Ще через деякий час приходить аналогічна задача на додавання читання даних з файлів формату H.262. Ви починаєте працювати над задачею і бачите, що код, який до цього був ще більш-менш зрозумілим стає зовсім важким для читання та внесення змін.

Рішення: Патерн «Шаблонний метод» (Template Method) пропонує загальний алгоритм винести в базовий клас, а частини алгоритма, які для різних задач виконуються по різному, виділити в окремі методи. Ці методи будуть викликатися в алгоритмі, що реалізований в базовому класі. В дочірніх класах ці виділені методи будуть перевизначатися. Таким чином загальна логіка залишається в базовому класі, а специфічна частина реалізується в дочірніх класах. Якщо подивитися на задачу з відео-редактором, то застосування«Шаблонного методу» наведе лад в коді і спростить його зміни.
Як це зробити: По перше, в алгоритмі всі блоки коду де є вибір гілки на основі типу формату виділяються в окремі методи. У випадку з відео редактором, це скоріш за все будуть блоки коду пов'язані з читанням даних та розпакування їх в кадри, а також читання звукових доріжок. Далі створюється загальний базовий клас в який переноситься загальний алгоритм, а також об'являються віртуальні методи (фактично беремо сигнатуру тих методів, що виділили на попередньому кроці). Далі створюємо дочірні класи під кожен формат файлу і перевизначаємо віртуальні методи. Фактично при цьому в кожному такому методі в дочірньому класі із реалізації цих методів, що була виділена на першому кроці, залишається код гілки який відповідав вибраному формату
Після всіх цих змін ми маємо реалізацію патерна «Шаблоний метод»: в базовому класі реалізовано базовий алгоритм (по суті більша частина алгоритму) і в дочірніх класах перевизначені методи зі специфічною логікою.

Після таких змін, добавити підтримку нового формату стає легше, тому що достатньо буде добавити лише новий дочірній клас і перевизначити в ньому необхідні методи.
Слід зауважити, що якщо у вас алгоритми співпадають більше ніж на 50 відсотків, то застосування шаблонного методу буде доцільним, але якщо у вас алгоритми співпадають лише відсотків на 10 або 20, то скоріш за все, краще буде використати патерн «Стратегія».

Перевага:
+ Полегшує повторне використання коду.

Недоліки:
- Ви жорстко обмежені скелетом існуючого алгоритму.
- Ви можете порушити принцип підстановки Барбари Лісков, змінюючи базову поведінку одного з кроків алгоритму через підклас.
- З ростом складності загального алгоритму шаблонний метод стає занадто складно підтримувати, особливо, коли є багато віртуальних методів для перевизначення в підкласах.

Хід роботи

1. Загальний опис виконаної роботи

Основна мета полягала в тому, щоб визначити скелет алгоритму в абстрактному базовому класі, дозволяючи дочірнім класам перевизначати (або реалізовувати) певні кроки цього алгоритму, не змінюючи його загальну структуру. В контексті теми «Shell», було створено єдиний, надійний процес для виконання всіх файлових операцій (копіювання, переміщення, видалення).

Такий підхід дозволяє інкапсулювати загальну логіку - таку як логування, вимірювання часу, обробку помилок та загальну послідовність дій - в одному місці, базовому класі, водночас надаючи повну гнучкість для реалізації специфічних кроків у конкретних класах-операціях.

Програмна система складається з наступних компонентів:

- **Абстрактний клас (FileOperation)**
    - Це базовий клас, що визначає **шаблонний метод** execute().

    - Метод execute() є final, що забороняє його перевизначення, і чітко диктує незмінний скелет алгоритму для всіх операцій:
        1. before() (хук, опціональний крок)
        2. checkPreconditions() (абстрактний крок)
        3. performOperation() (абстрактний крок)
        4. verify() (абстрактний крок)
        5. after() (хук, опціональний крок)
        6. onError() (хук для обробки винятків)
    - Клас також оголошує protected abstract методи (checkPreconditions, performOperation, verify, getName), які є змінними частинами алгоритму і *повинні* бути реалізовані всіма спадкоємцями.
- **Конкретні класи (CopyOperation, MoveOperation, DeleteOperation)**
    - Ці класи розширюють FileOperation і надають конкретну, унікальну реалізацію для кожного абстрактного кроку, визначеного в базовому класі.
    - Наприклад, checkPreconditions() в CopyOperation перевіряє, чи існує джерело (source) і чи *не* існує ціль (target), тоді як checkPreconditions() в DeleteOperation перевіряє лише існування джерела.
    - Так само performOperation() в CopyOperation викликає Files.copy(), а в DeleteOperation - Files.delete().
- **Клієнт (Demo)**
    - Кінцевий клієнт, який ініціює виконання операцій.
    - Важливо, що клієнт працює з усіма об'єктами операцій через єдиний інтерфейс, викликаючи один і той самий публічний метод execute().
    - Клієнту не потрібно знати про внутрішню реалізацію чи відмінності між копіюванням, переміщенням чи видаленням; він просто запускає загальний алгоритм, довіряючи шаблону виконання правильних кроків.

2. Опис класів програмної системи.

Клас FileOperation.java

Опис:

Клас FileOperation є абстрактним базовим класом у структурі шаблону «Template Method». Він визначає загальний скелет алгоритму для виконання будь-якої файлової операції (копіювання, переміщення, видалення), водночас залишаючи реалізацію специфічних кроків своїм нащадкам.

Характеристики:

- Його неможливо інстанціювати напряму. Він служить "шаблоном" для конкретних операцій.
- Містить поля source (джерело) та target (ціль), які є спільними для більшості файлових операцій.
- Визначає public final метод execute() - це **шаблонний метод**. Він жорстко задає послідовність виконання кроків (перевірка, виконання, верифікація, обробка помилок) і не може бути перевизначений.
- Оголошує protected abstract методи (checkPreconditions, performOperation, verify, getName), які *повинні* бути реалізовані в кожному конкретному класі-нащадку.
- Містить protected "хуки" (before, after, onError) - це опціональні методи з реалізацією за замовчуванням, які нащадки *можуть* перевизначити для додавання специфічної поведінки.

```java
package ua.kpi.shell.template;

import java.nio.file.Path;

public abstract class FileOperation {  3 usages  3 inheritors
    protected final Path source;  26 usages
    protected final Path target;  20 usages

    protected FileOperation(Path source, Path target) {  3 usages
        this.source = source;
        this.target = target;
    }

    public final void execute() {  5 usages
        long started = System.currentTimeMillis();
        System.out.println("=== " + getName() + " ===");
        try {
            before();
            checkPreconditions();
            performOperation();
            verify();
            after();
            System.out.println(getName() + " -> Виконано (" +
                    (System.currentTimeMillis() - started) + " ms)");
        } catch (Exception e) {
            onError(e);
        }
        System.out.println();
    }

    protected abstract String getName();  3 usages  3 implementations
    protected abstract void checkPreconditions() throws Exception;  1 usage  3 implementations
    protected abstract void performOperation() throws Exception;  1 usage  3 implementations
    protected abstract void verify() throws Exception;  1 usage  3 implementations

    protected void before() throws Exception {}  1 usage  3 overrides
    protected void after() throws Exception {}  1 usage
    protected void onError(Exception e) {  1 usage
        System.err.println(getName() + " -> Помилка: " + e.getMessage());
    }
}
```

рис 2.1 - код класу FileOperation.java

Класи CopyOperation.java

Опис:

Клас CopyOperation є конкретним класом-нащадком, що успадковує FileOperation. Він реалізує специфічну логіку для операції копіювання файлу або директорії, дотримуючись загального алгоритму, визначеного в базовому класі.

Характеристики:

- Надає конкретну реалізацію для всіх абстрактних методів батьківського класу.
- checkPreconditions(): Перевіряє, що source існує і що target *ще не існує*, аби уникнути перезапису.
- performOperation(): Виконує копіювання. Він також враховує, чи є source файлом чи директорією (використовуючи приватний допоміжний метод copyDirectory з walkFileTree для рекурсивного копіювання).
- verify(): Перевіряє, що target був успішно створений після операції.
- getName(): Повертає рядок "CopyOperation" для ідентифікації в логах.
- Також перевизначає хук before() для виведення інформативного повідомлення перед початком операції.

```java
import java.io.IOException;
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;

public class CopyOperation extends FileOperation {  2 usages
    public CopyOperation(Path source, Path target) {  2 usages
        super(source, target);
    }

    @Override  3 usages
    protected String getName() {
        return "CopyOperation";
    }

    @Override  1 usage
    protected void checkPreconditions() throws Exception {
        if (source == null || target == null) throw new IllegalArgumentException("source/target не можуть бути null");
        if (!Files.exists(source)) throw new NoSuchFileException("Не існує: " + source);
        if (Files.exists(target)) throw new FileAlreadyExistsException("Ціль вже існує: " + target);
    }

    @Override  1 usage
    protected void before() {
        System.out.println("Підготовка до копіювання: " + source + " -> " + target);
    }

    @Override  1 usage
    protected void performOperation() throws Exception {
        if (Files.isDirectory(source)) {
            copyDirectory(source, target);
        } else {
            Files.createDirectories(target.getParent());
            Files.copy(source, target);
        }
    }

    @Override  1 usage
    protected void verify() throws Exception {
        if (!Files.exists(target)) throw new IOException("Копіювання не вдалося: " + target);
        System.out.println("Перевірка пройшла: ціль існує -> " + target);
    }

    private static void copyDirectory(Path from, Path to) throws IOException {  1 usage
        Files.walkFileTree(from, new SimpleFileVisitor<>() {
            @Override  4 usages
            public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {
                Path rel = from.relativize(dir);
                Files.createDirectories(to.resolve(rel));
                return FileVisitResult.CONTINUE;
            }
            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
                Path rel = from.relativize(file);
                Files.copy(file, to.resolve(rel));
                return FileVisitResult.CONTINUE;
            }
        });
```

рис 2.2 - код класу CopyOperation.java

Клас MoveOperation.java

Опис:

Клас MoveOperation є ще одним конкретним класом-нащадком FileOperation. Він інкапсулює логіку, специфічну для операції переміщення файлу або директорії.

Ключові характеристики:

- checkPreconditions(): Аналогічно до CopyOperation, перевіряє, що source існує, а target - ні.
- performOperation(): Намагається виконати атомарне переміщення (Files.move з StandardCopyOption.ATOMIC_MOVE). Якщо це неможливо (наприклад, при переміщенні між різними дисками), він використовує резервну стратегію: **створює екземпляри CopyOperation та DeleteOperation** і послідовно їх виконує. Це демонструє гнучкість системи.
- verify(): Виконує повну перевірку: target має з'явитися, а source - зникнути.
- getName(): Повертає рядок "MoveOperation".
- Також перевизначає хук before() для логування початку операції.

```java
package ua.kpi.shell.template;

import java.io.IOException;
import java.nio.file.*;

public class MoveOperation extends FileOperation {  1 usage
    public MoveOperation(Path source, Path target) {  1 usage
        super(source, target);
    }

    @Override  3 usages
    protected String getName() {
        return "MoveOperation";
    }

    @Override  1 usage
    protected void checkPreconditions() throws Exception {
        if (source == null || target == null) throw new IllegalArgumentException("source/target не можуть бути null");
        if (!Files.exists(source)) throw new NoSuchFileException("Не існує: " + source);
        if (Files.exists(target)) throw new FileAlreadyExistsException("Ціль вже існує: " + target);
    }

    @Override  1 usage
    protected void before() {
        System.out.println("Підготовка до переміщення: " + source + " -> " + target);
    }
```

```java
    @Override  1 usage
    protected void performOperation() throws Exception {
        try {
            Files.createDirectories(target.getParent());
            Files.move(source, target, StandardCopyOption.ATOMIC_MOVE);
        } catch (IOException e) {
            System.out.println("ATOMIC_MOVE недоступний, виконуємо copy+delete...");
            new CopyOperation(source, target).execute();
            new DeleteOperation(source).execute();
        }
    }

    @Override  1 usage
    protected void verify() throws Exception {
        if (!Files.exists(target)) throw new IOException("Переміщення не вдалося: " + target);
        if (Files.exists(source)) throw new IOException("Переміщення не завершено: " + source);
        System.out.println("Перевірка пройшла: ціль існує, джерела нема.");
    }
```

рис 2.3 - код класу MoveOperation.java

Клас DeleteOperation.java

Опис:

Клас DeleteOperation - третій конкретний клас-нащадок FileOperation, що реалізує логіку видалення файлу або директорії.

Характеристики:

- Використовує конструктор, який приймає лише source, оскільки target для видалення не потрібен (в FileOperation він буде null).
- checkPreconditions(): Перевіряє лише те, що source існує і доступний для видалення.
- performOperation(): Виконує видалення. Подібно до CopyOperation, він розрізняє файли (просте видалення) та директорії (використовуючи приватний допоміжний метод deleteRecursively з walkFileTree для рекурсивного видалення вмісту).
- verify(): Перевіряє, що source успішно зник після операції.
- getName(): Повертає рядок "DeleteOperation".
- Також перевизначає хук before() для логування.

```java
public class DeleteOperation extends FileOperation {  2 usages
    public DeleteOperation(Path source) {  2 usages
        super(source,  target: null);
    }

    @Override  3 usages
    protected String getName() {
        return "DeleteOperation";
    }

    @Override  1 usage
    protected void checkPreconditions() throws Exception {
        if (source == null) throw new IllegalArgumentException("source не може бути null");
        if (!Files.exists(source)) throw new NoSuchFileException("Не існує: " + source);
    }

    @Override  1 usage
    protected void before() {
        System.out.println("Підготовка до видалення: " + source);
    }

    @Override  1 usage
    protected void performOperation() throws Exception {
        if (Files.isDirectory(source)) deleteRecursively(source);
        else Files.delete(source);
    }

    @Override  1 usage
    protected void verify() throws Exception {
        if (Files.exists(source)) throw new IOException("Видалення не вдалося: " + source);
        System.out.println("Перевірка пройшла: об'єкт видалено.");
    }

    private static void deleteRecursively(Path root) throws IOException {  1 usage
        Files.walkFileTree(root, new SimpleFileVisitor<>() {
            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
                Files.delete(file);
                return FileVisitResult.CONTINUE;
            }
            @Override  3 usages
            public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
                Files.delete(dir);
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```

рис 2.4 - код класу DeleteOperation.java

Клас Demo.java

Опис:

Клас Demo виступає в ролі Клієнта. Його єдина мета - продемонструвати роботу шаблону «Template Method» шляхом створення та запуску різних файлових операцій.

Характеристики:

- Містить головний метод main(), який є точкою входу до програми.
- Створює екземпляри конкретних класів (CopyOperation, MoveOperation, DeleteOperation).
- Взаємодіє з усіма об'єктами операцій однаково - викликаючи їх єдиний публічний метод execute(). Клієнту не потрібно знати внутрішню структуру алгоритму (про checkPreconditions чи verify), він просто запускає "шаблон".
- Містить допоміжний private static метод prepareEnvironment(), який створює тестові файли та директорії, щоб демонстрація була чистою, самодостатньою та відтворюваною.

```java
public class Demo {
    public static void main(String[] args) {
        System.out.println("DEMO Template Method: Shell Operations\n");

        Path base = Paths.get(System.getProperty("user.home"),  ...more: "TemplateMethodDemo");
        Path sourceDir = base.resolve( other: "sourceDir");
        Path targetDir = base.resolve( other: "targetDir");
        Path file = sourceDir.resolve( other: "demo.txt");
        Path copyInTarget = targetDir.resolve( other: "demo_copy.txt");
        Path movedIntoTarget = targetDir.resolve( other: "demo_moved.txt");

        try {
            prepareEnvironment(base, sourceDir, targetDir, file);

            if (Files.exists(copyInTarget)) Files.delete(copyInTarget);
            new CopyOperation(file, copyInTarget).execute();

            if (Files.exists(movedIntoTarget)) Files.delete(movedIntoTarget);
            new MoveOperation(file, movedIntoTarget).execute();

            new DeleteOperation(movedIntoTarget).execute();

            System.out.println("\nСтруктура після операцій:");
            System.out.println(" - " + sourceDir);
            System.out.println(" - " + targetDir);

            System.out.println("\nDEMO завершено");

        } catch (Exception e) {
            System.err.println("DEMO ERROR: " + e.getMessage());
        }
    }

    private static void prepareEnvironment(Path base, Path sourceDir, Path targetDir, Path file) throws IOException {  1 usage
        System.out.println("Створення робочих папок: " + base);
        Files.createDirectories(sourceDir);
        Files.createDirectories(targetDir);
        if (!Files.exists(file)) {
            Files.writeString(file,  csq: "Це тестовий файл для демонстрації Template Method.\n");
            System.out.println("Створено файл: " + file);
        } else {
            System.out.println("Файл вже існує: " + file);
        }
        System.out.println("Початкове середовище підготовлено.\n");
    }
}
```

рис 2.5 - код класу Demo.java

Опис результатів коду

Для демонстрації роботи шаблону «Template Method» було запущено клас Demo. Цей клас спочатку готує тестове середовище - створює папки та

тестовий файл, а потім послідовно ініціює три різні файлові операції, викликаючи їх єдиний публічний метод execute().

На рисунку 2.6.1 представлено консольний вивід роботи програми.

```
DEMO Template Method: Shell Operations

Створення робочих папок: C:\Users\arosl\TemplateMethodDemo
Створено файл: C:\Users\arosl\TemplateMethodDemo\sourceDir\demo.txt
Початкове середовище підготовлено.

=== CopyOperation ===
Підготовка до копіювання: C:\Users\arosl\TemplateMethodDemo\sourceDir\demo.txt -> C:\Users\arosl\TemplateMethodDemo\targetDir\demo_copy.txt
Перевірка пройшла: ціль існує -> C:\Users\arosl\TemplateMethodDemo\targetDir\demo_copy.txt
CopyOperation -> Виконано (14 ms)

=== MoveOperation ===
Підготовка до переміщення: C:\Users\arosl\TemplateMethodDemo\sourceDir\demo.txt -> C:\Users\arosl\TemplateMethodDemo\targetDir\demo_moved.txt
Перевірка пройшла: ціль існує, джерела нема.
MoveOperation -> Виконано (2 ms)

=== DeleteOperation ===
Підготовка до видалення: C:\Users\arosl\TemplateMethodDemo\targetDir\demo_moved.txt
Перевірка пройшла: об'єкт видалено.
DeleteOperation -> Виконано (0 ms)


Структура після операцій:
 - C:\Users\arosl\TemplateMethodDemo\sourceDir
 - C:\Users\arosl\TemplateMethodDemo\targetDir

DEMO завершено
```

рис 2.6.1 - консольний вивід роботи програми Demo

Аналізуючи вивід, можна побачити чіткий покроковий процес, який відповідає алгоритму, закладеному в FileOperation.execute():

1. Програма створює робочу директорію TemplateMethodDemo та тестовий файл sourceDir\demo.txt.
2. Виконується перша операція. Логи показують підготовку, успішну перевірку (Перевірка пройшла: ціль існує) та час виконання. Файл demo.txt копіюється у targetDir\demo_copy.txt.
3. Виконується друга операція. Файл sourceDir\demo.txt переміщується у targetDir\demo_moved.txt. Лог верифікації (Перевірка пройшла: ціль існує, джерела нема) підтверджує, що вихідний файл зник, а цільовий з'явився.
4. Виконується третя операція. Файл targetDir\demo_moved.txt (створений на попередньому кроці) видаляється. Лог (Перевірка пройшла: об'єкт видалено) підтверджує успішне видалення.

На рисунках 2.6.2 - 2.6.4 показано кінцевий стан файлової системи після виконання всіх операцій, який повністю збігається з логами консолі.

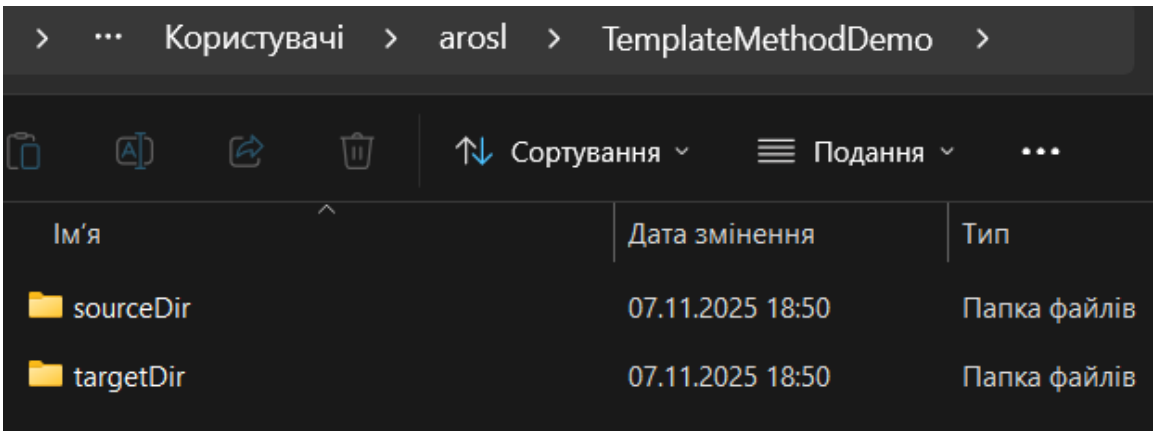


рис 2.6.2 - Головна робоча директорія TemplateMethodDemo

На рисунку 2.6.3 видно, що папка sourceDir тепер пуста. Це коректний результат, оскільки єдиний файл у ній (demo.txt) був переміщений (а не скопійований) під час MoveOperation.

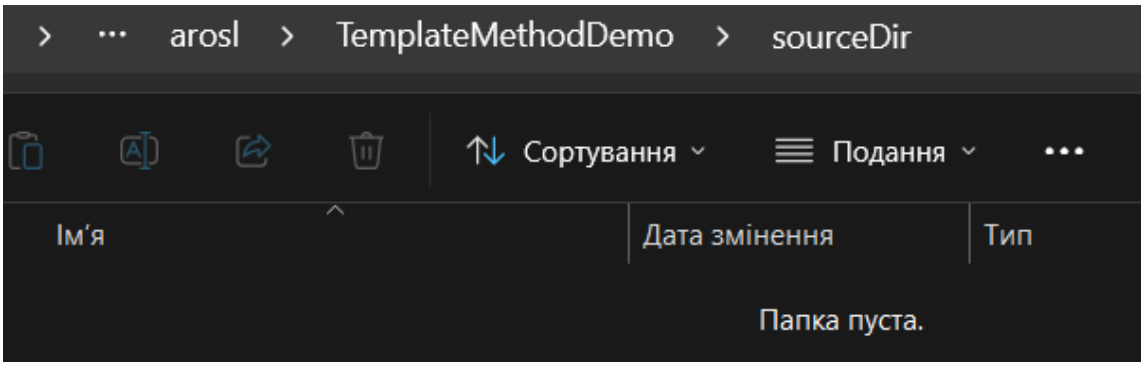


рис 2.6.3 - Вміст директорії sourceDir після виконання програми

На рисунку 2.6.4 показано вміст папки targetDir. Вона містить лише файл demo_copy.txt, який був створений операцією CopyOperation. Файл demo_moved.txt (створений MoveOperation) відсутній, оскільки він був успішно видалений операцією DeleteOperation.

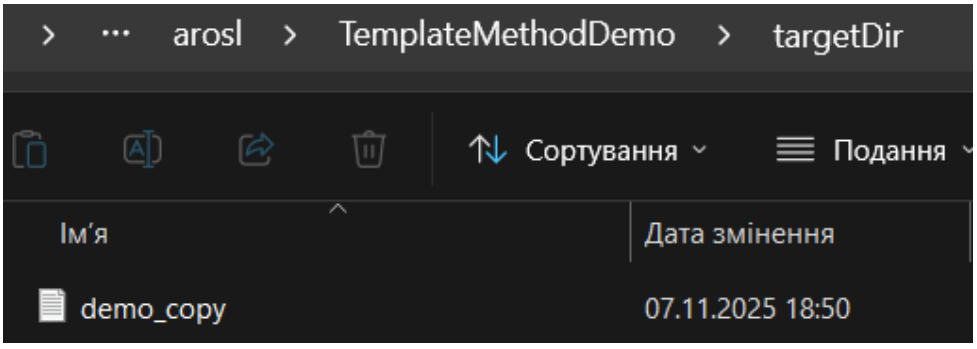


рис 2.6.4 - Вміст директорії targetDir після виконання програми

3. Діаграма класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами була побудована UML-діаграма класів. Вона наочно демонструє структуру, що відповідає шаблону проектування «Factory Method».

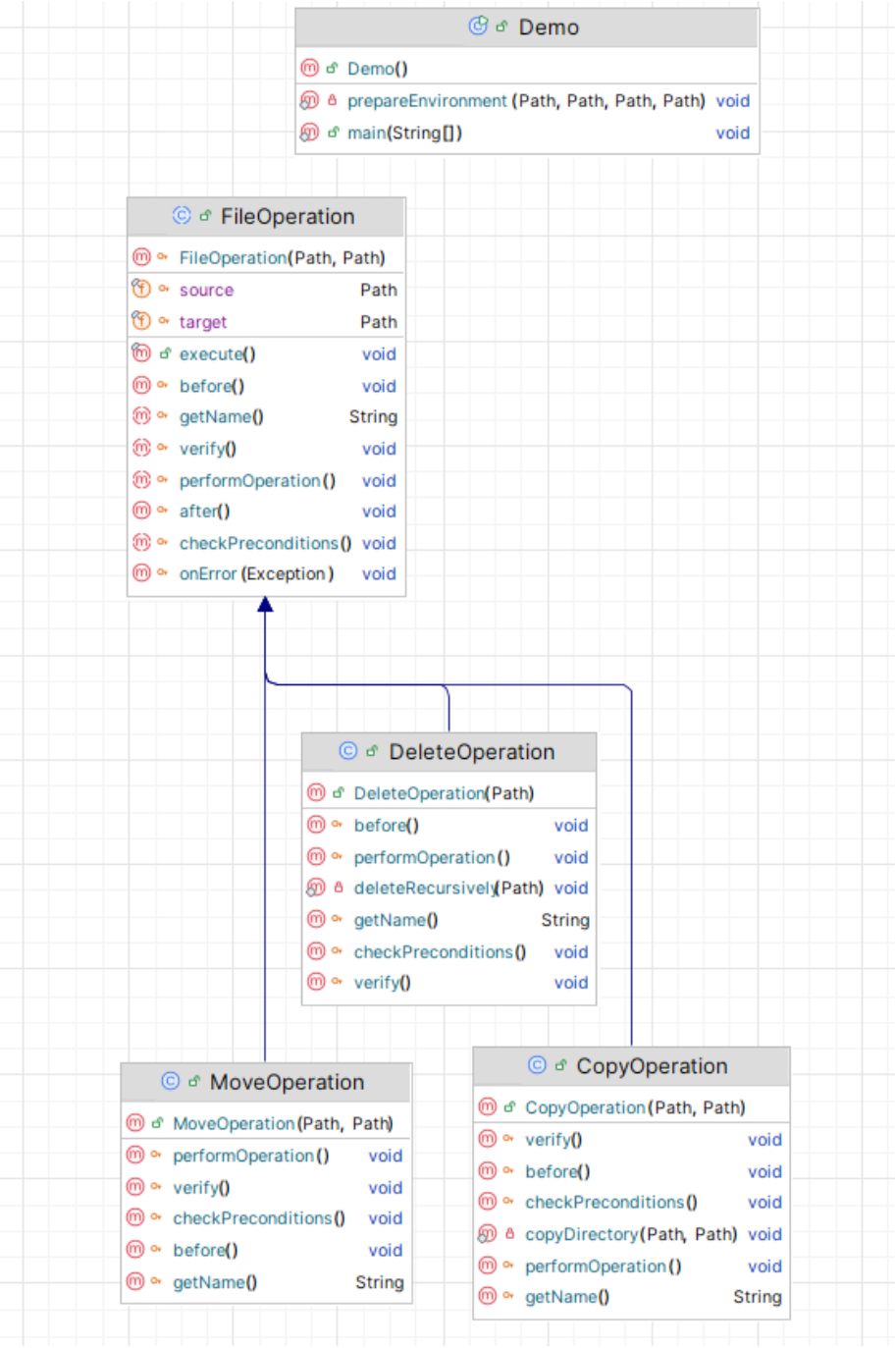


рис 3.1 - діаграма класів (спрощена)

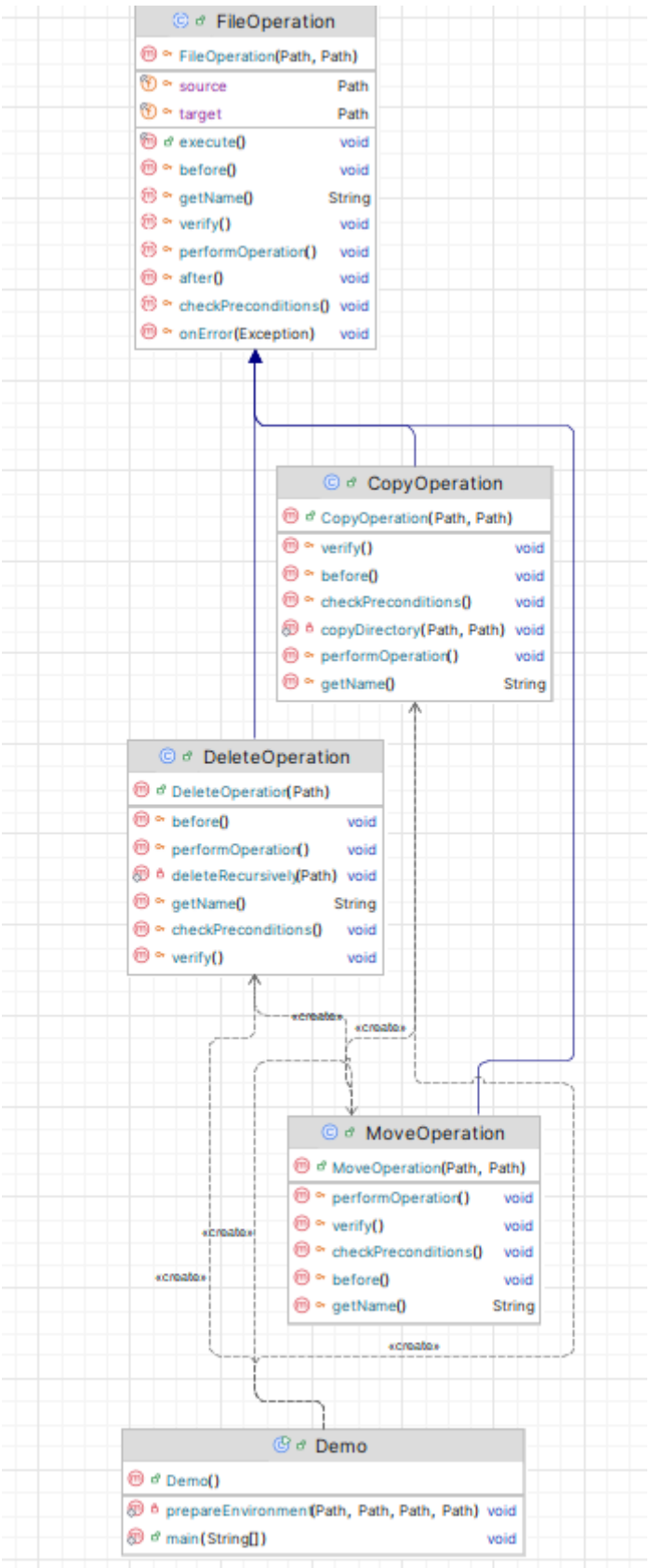


рис 3.2 - діаграма класів (усі зв’язки)

1. **Абстрактний клас (FileOperation)**
    - Цей клас є ядром шаблону і виступає в ролі AbstractClass. Він визначає загальний скелет алгоритму, який є спільним для всіх файлових операцій.
    - Ключовим елементом є public метод execute(), який є **шаблонним методом**. Це єдина точка входу для клієнтського коду, і саме він диктує послідовність виконання операції.
    - Клас також оголошує набір protected методів (на діаграмі позначені іконкою ключа), таких як checkPreconditions(), performOperation() та verify(). Це і є абстрактні кроки та хуки, реалізацію яких FileOperation делегує своїм спадкоємцям. Використання protected видимості приховує ці деталі реалізації від клієнта, але відкриває їх для наслідування.
2. **Конкретні класи (CopyOperation, MoveOperation, DeleteOperation)**
    - Ці класи є ConcreteClass у структурі шаблону. На діаграмі вони пов'язані з FileOperation через стрілку **узагальнення (спадкування)**.
    - Їхня єдина відповідальність - надати конкретну реалізацію для protected абстрактних методів, успадкованих від FileOperation. Кожен клас визначає *що* саме робити на кожному кроці (наприклад, performOperation для CopyOperation реалізує логіку копіювання), але не контролює *коли* цей крок буде викликаний. Цей контроль залишається за шаблонним методом execute() в базовому класі.
3. **Клієнтський клас (Demo)**
    - Клас Demo виконує роль "клієнта" - коду, який використовує шаблон.
    - На діаграмі це показано через зв'язок **залежності (dependency)**, уточнений стереотипом «create». Це означає, що Demo створює екземпляри *конкретних* операцій.
    - Однак, після створення, Demo взаємодіє з усіма об'єктами однаково, викликаючи лише їхній public метод execute(). Це демонструє ключову перевагу шаблону: клієнт працює з єдиним, стабільним алгоритмом, не заглиблюючись у специфічні деталі його кроків.

4. Висновки

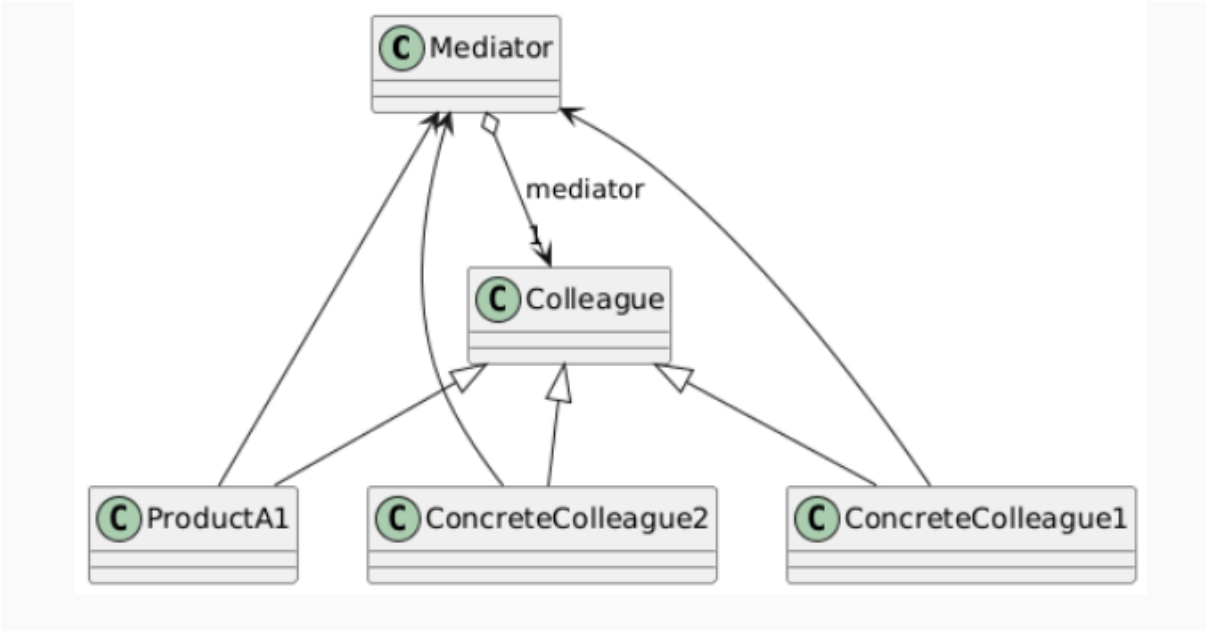
Висновки: В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано патерн проектування «Шаблонний метод» (Template Method) для вирішення задачі визначення єдиного алгоритму виконання файлових операцій в програмі-оболонці. Була створена чітка ієрархія класів: абстрактний клас FileOperation та його конкретні реалізації (CopyOperation, MoveOperation, DeleteOperation). Особливу увагу було приділено реалізації шаблонного методу execute() в абстрактному класі. Впровадження абстрактного класу FileOperation у ролі Абстрактного класу дозволило централізувати всю загальну логіку, логування та обробку винятків в одному місці. Водночас, відповідальність за реалізацію специфічних деталей кожного кроку (наприклад, performOperation або verify) була успішно делегована конкретним класам-нащадкам. Практична реалізація продемонструвала ключові переваги патерну: високий рівень повторного використання коду, оскільки загальний алгоритм не дублюється; чітке розділення відповідальності між загальною структурою алгоритму та його конкретними кроками; та гнучкість системи, що дозволяє легко додавати нові типи операцій без зміни існуючого коду, що повністю відповідає принципу відкритості/закритості.

Контрольні запитання

1. Яке призначення шаблону «Посередник»?

Шаблон «Посередник» (Mediator) призначений для зменшення кількості прямих зв'язків між множиною об'єктів (Колег). Він інкапсулює логіку взаємодії між цими об'єктами в одному окремому класі (Посереднику). Завдяки цьому, об'єкти більше не спілкуються один з одним напряму, а роблять це виключно через Посередника, що спрощує їхню взаємодію та підтримку коду.

2. Нарисуйте структуру шаблону «Посередник».



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

- **Mediator (Посередник):** Абстрактний інтерфейс, що визначає метод для спілкування Колег з Посередником.
- **ConcreteMediator (Конкретний посередник):** Реалізує логіку взаємодії. Він знає про всіх конкретних Колег і служить центральним вузлом комунікації.
- **Colleague (Колега):** Абстрактний клас або інтерфейс, що визначає метод для отримання повідомлень від Посередника та має посилання на свого Посередника.
- **ConcreteColleague (Конкретний колега):** Реалізує бізнес-логіку.

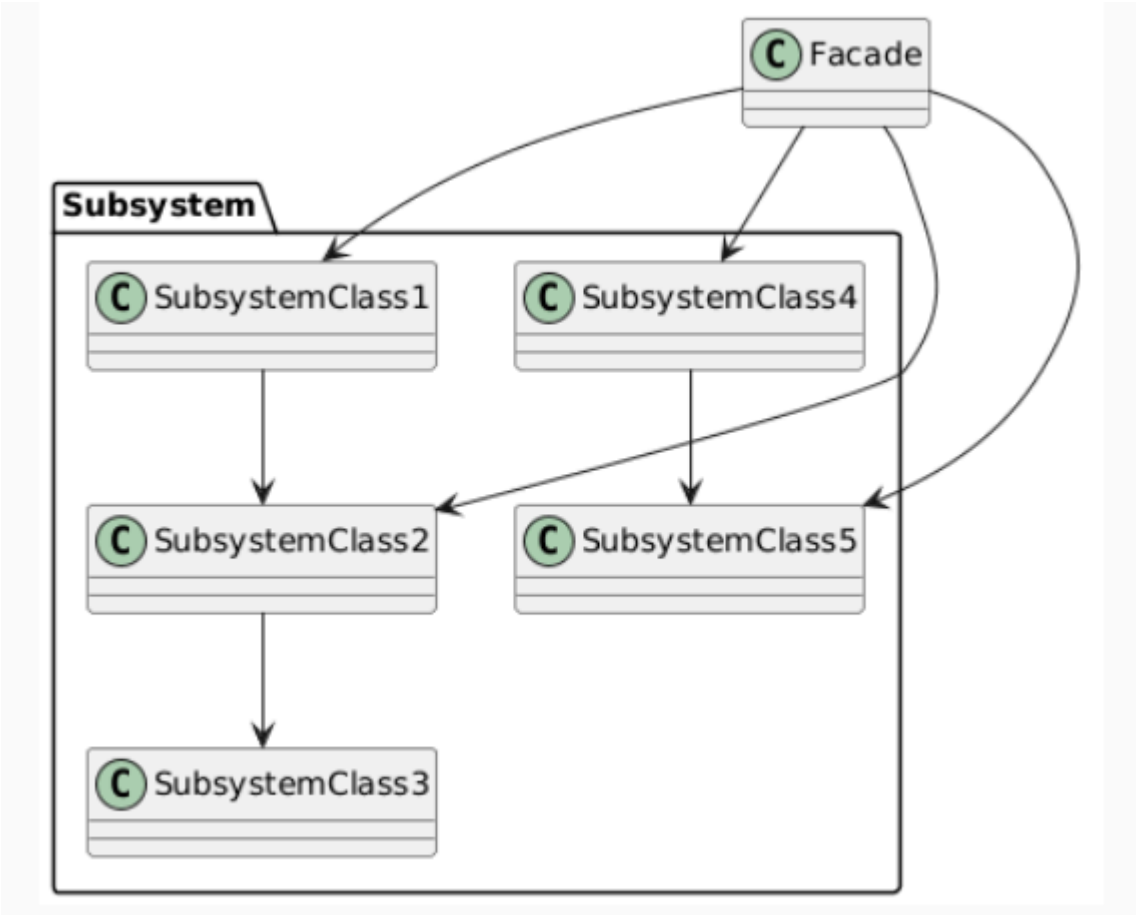
Колеги не посилаються один на одного. Коли ConcreteColleague1 виконує дію, він повідомляє про це ConcreteMediator. ConcreteMediator обробляє це повідомлення і, за необхідності, повідомляє інших Колег (наприклад, ConcreteColleague2) про цю подію.

4. Яке призначення шаблону «Фасад»?

Шаблон «Фасад» (Facade) призначений для надання єдиного, спрощеного інтерфейсу до складної підсистеми, що складається з багатьох класів,

бібліотек або фреймворків. Він приховує внутрішню складність підсистеми, значно полегшуючи її використання клієнтським кодом.

5. Нарисуйте структуру шаблону «Фасад».



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

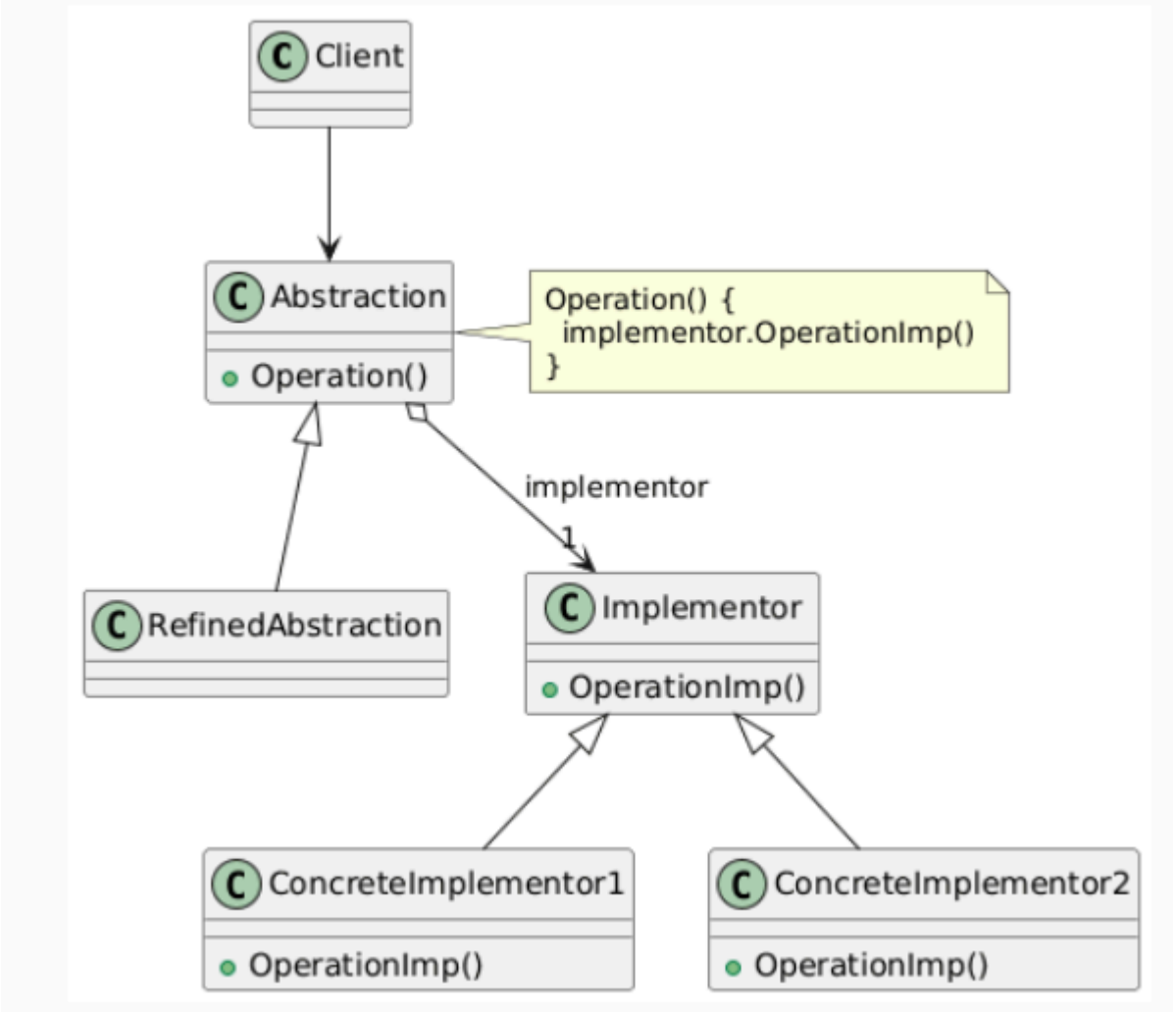
- **Facade (Фасад):** Клас, що надає простий, високорівневий інтерфейс. Він знає, яким класам підсистеми потрібно делегувати запит.
- **Subsystem Classes (Класи підсистеми):** Набір класів, які реалізують складну функціональність. Вони не знають про існування Фасаду і не посилаються на нього.

Клієнтський код спілкується лише з об'єктом Facade. Коли клієнт викликає метод Фасаду (наприклад, facade.startSystem()), Фасад всередині себе викликає необхідні методи одного або декількох Subsystem Classes у правильному порядку для виконання запиту.

7. Яке призначення шаблону «Міст»?

Шаблон «Міст» (Bridge) призначений для розділення абстракції та її реалізації так, щоб вони могли змінюватися і розвиватися незалежно одна від одної. Це дозволяє уникнути жорсткої прив'язки та "розростання" ієрархії класів, коли існує декілька варіантів абстракції і декілька варіантів реалізації.

8. Нарисуйте структуру шаблону «Міст».



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

- **Abstraction (Абстракція):** Визначає високорівневий інтерфейс керування (напр., RemoteControl). Містить посилання на об'єкт Implementation.
- **RefinedAbstraction (Уточнена абстракція):** Розширює Abstraction, додаючи нову логіку (напр., AdvancedRemoteControl).

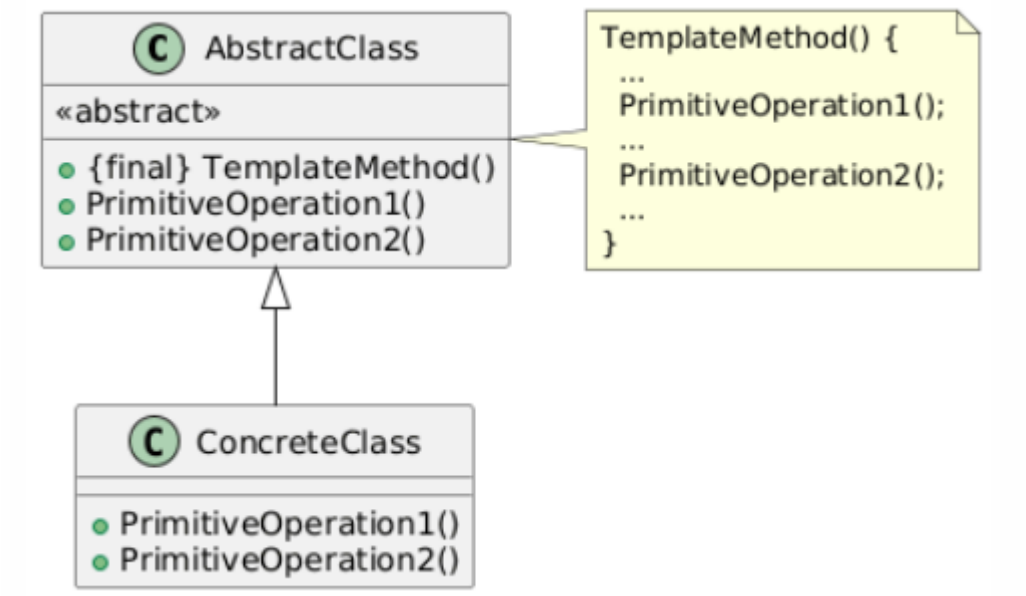
- **Implementation (Реалізатор/Інтерфейс реалізації):** Визначає інтерфейс для низькорівневих, платформо-залежних операцій (напр., Device).
- **ConcreteImplementation (Конкретний реалізатор):** Надає конкретну реалізацію Implementation (напр., TV, Radio).

Клієнт працює з об'єктом Abstraction. Коли клієнт викликає метод Abstraction, цей метод всередині себе делегує виконання низькорівневої операції об'єкту Implementation, на який він посилається через "міст".

10. Яке призначення шаблону «Шаблоний метод»?

Шаблон «Шаблонний метод» (Template Method) призначений для визначення скелету (каркасу) алгоритму в базовому (абстрактному) класі. При цьому він дозволяє дочірнім класам перевизначати або реалізовувати певні кроки цього алгоритму, не змінюючи його загальну структуру.

11. Нарисуйте структуру шаблону «Шаблоний метод».



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

- **AbstractClass (Абстрактний клас):** Містить **шаблонний метод** (templateMethod()), який, як правило, є final. Цей метод визначає послідовність виклику кроків. Також оголошує абстрактні (abstract) або опціональні (protected, хуки) методи-кроки.

- **ConcreteClass (Конкретний клас):** Наслідує AbstractClass та надає конкретну реалізацію для абстрактних кроків, успадкованих від батьківського класу.

Клієнтський код викликає templateMethod() у об'єкта ConcreteClass (через інтерфейс AbstractClass). Цей метод, визначений у AbstractClass, послідовно виконує кроки алгоритму, викликаючи як власні реалізації, так і реалізації, надані ConcreteClass.

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

- «Шаблонний метод» визначає скелет алгоритму. «Фабричний метод» визначає інтерфейс для створення об'єктів (творення).
- «Шаблонному методі» базовий клас диктує загальну послідовність дій, а дочірні класи реалізують *конкретні кроки* цієї послідовності. У «Фабричному методі» базовий клас має код, який працює з абстрактним "продуктом", а дочірні класи реалізують *метод, що створює* цей продукт.
- «Шаблонний метод» фіксує алгоритм, але дозволяє змінювати його кроки. «Фабричний метод» дозволяє дочірнім класам вирішувати, який саме об'єкт буде створено.

14. Яку функціональність додає шаблон «Міст»?

Шаблон «Міст» додає можливість незалежно змінювати та розширювати дві окремі ієрархії класів - ієрархію «Абстракцій» (високорівнева логіка) та ієрархію «Реалізацій» (низькорівнева, платформенна логіка). Це дозволяє уникнути "вибуху" кількості класів і дає змогу гнучко комбінувати будь-яку Абстракцію з будь-якою Реалізацією під час виконання програми (run-time).