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"

Виконав

студент групи ІА–33:

Китченко Д.В.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Вступ

Патерни проєктування є ключовими елементами для створення гнучких, масштабованих та легко підтримуваних програмних систем. Вони пропонують перевірені рішення для типових задач проєктування, дозволяючи розробникам уникати "винаходу велосипеда" та спілкуватися єдиною мовою архітектурних рішень.

Серед структурних патернів особливе місце займає патерн "Facade" (Фасад), який дозволяє спростити взаємодію з складними підсистемами.

У сучасних програмних системах часто виникає ситуація, коли підсистема складається з великої кількості взаємопов'язаних класів із складною логікою взаємодії. Пряма робота з такою підсистемою може бути надзвичайно складною для клієнтського коду, створюючи високий рівень зв'язаності та ускладнюючи підтримку системи.

Патерн Facade вирішує цю проблему, надаючи єдиний спрощений інтерфейс до складної підсистеми. Він виступає як "фасад" будівлі - користувач бачить лише красиву зовнішню оболонку, не заглиблюючись у складну внутрішню структуру. Це дозволяє значно знизити складність використання системи та покращити її модульність.

Мета роботи

Вивчити структуру шаблонів "Mediator", "Facade", "Bridge", "Template method" та навчитися застосовувати їх в реалізації програмної системи. Детально дослідити патерн "Facade", зрозуміти його призначення, переваги та недоліки. Реалізувати патерн Facade в контексті системи архіватора, створивши єдиний інтерфейс для роботи зі складною підсистемою, що включає різні стратегії архівування, репозиторії для збереження даних, систему логування операцій та управління закладками.

Теоретичні відомості

Шаблон "Facade"

Призначення патерну: Шаблон "Facade" (фасад) передбачає створення єдиного уніфікованого способу доступу до підсистеми без розкриття внутрішніх деталей підсистеми. Оскільки підсистема може складатися з безлічі класів, а кількість її функцій - не більше десяти, то щоб уникнути створення "спагеті-коду" (коли все тісно пов'язано між собою) виділяють один загальний інтерфейс доступу, здатний правильним чином звертатися до внутрішніх деталей.

Це також відволікає користувачів від змін в підсистемі (внутрішня реалізація може змінюватися, а наданої послуги немає), що також скоротить кількість змін в використовуваних фасад класах (без фасаду довелося б змінювати вихідні коди в безлічі точок).

Звичайно, твердої умови повного закриття внутрішніх класів підсистеми не стоїть – при необхідності можна звертатися до окремих класів безпосередньо, минаючи об'єкт фасад.

Проблема: Ви розробляєте компонент, який дозволяє відправляти запити на різні типи endpoint, а також працює з протоколами HTTP та TCP/IP.

Прототип компонента вже працює, але структура класів вийшла досить складна, а при налаштуванні на різні протоколи мають використовуватися різні класи.

Інструкція для використання також виходить досить складна та заплутана. Слід додати, так як інші системи будуть знати про внутрішню будову вашого компонента, а тому при спробі змінити внутрішню структуру в наступних версіях, вам прийдеться повідомляти про це всіх користувачів вашого компонента і для них перехід на наступну версію вашого компонента буде достатньо складним.

Рішення: Тут краще використати патерн фасад. А саме, в даному випадку, створити один клас, наприклад, InternetClient, та набір методів у нього, які будуть використовуватися для налаштування цього підключення, та його

подальшого використання. Цей клас єдиний буде позначено як public, і тільки його будуть бачити ваші клієнти. Таким чином, з точки зору зовнішнього коду вони будуть працювати з набагато простішим інтерфейсом, а значить і інструкція використання буде значно простіша. Крім того, так як внутрішня структура повністю закрита, то в наступних версіях ви можете її змінювати, як вам буде зручно, а з точки зору користувачів компонента, перехід на нову версію буде просто переключенням на нову версію бібліотеки.

Переваги та недоліки:

+ Інкапсуляція внутрішньої структури від клієнтського коду

+ Спрощується інтерфейс для роботи з модулем закритим фасадом

- Зниження гнучкості в налаштуванні та використанні програмного коду закритого фасадом

Хід роботи

У процесі розробки системи архівації файлів виникла необхідність спростити інтерфейс для клієнтського коду, зокрема для графічного інтерфейсу користувача. Система складається з багатьох компонентів: фабрика стратегій для вибору правильного обробника формату архіву, конкретні стратегії для роботи з ZIP та TAR форматами, репозиторії для збереження інформації про архіви у базі даних PostgreSQL, репозиторії для логування операцій та їхніх деталей, репозиторії для роботи із закладками користувача на файли всередині архівів.

Без застосування патерну Facade клієнтський код повинен був би виконувати складну послідовність дій для кожної операції з архівом. Наприклад, для створення архіву потрібно: створити екземпляр фабрики стратегій, викликати метод фабрики для отримання відповідної стратегії на основі бажаного формату, викликати метод створення архіву у стратегії, створити об'єкт метаданих про архів, отримати посилання на репозиторій архівів, зберегти метадані через репозиторій, створити об'єкт запису логу операції, отримати посилання на репозиторій логів, зберегти лог через репозиторій, обробити можливі помилки на кожному етапі. Такий код є громіздким, схильним до помилок та важким для підтримки.

Для вирішення цієї проблеми було застосовано патерн Facade через створення класу ArchiverFacade, який інкапсулює всю складність взаємодії між компонентами системи. Фасад містить посилання на всі необхідні репозиторії та фабрики, ініціалізуючи їх у своєму конструкторі. Це приховує деталі створення та налаштування цих об'єктів від клієнта.

ArchiverFacade виступає центральним класом-фасадом, який координує роботу всієї підсистеми архівації. Він надає простий інтерфейс з методами для створення, відкриття, модифікації архівів, додавання закладок та перегляду історії операцій.

Підсистема стратегій архівування включає ArchiverStrategyFactory для створення відповідних стратегій залежно від формату архіву та інтерфейс

IArchiverStrategy з конкретними реалізаціями для роботи з ZIP та TAR форматами.

Підсистема збереження даних складається з чотирьох репозиторіїв: IArchiveInfoRepository для збереження інформації про архіви, IOperationLogRepository для логування операцій, IOperationDetailRepository для збереження деталей операцій та IArchiveBookmarkRepository для управління закладками користувача.

Коли клієнтський код потребує створити новий архів, він викликає простий метод фасаду. Всередині цього методу фасад виконує складну послідовність операцій: вибирає відповідну стратегію через фабрику, створює архів використовуючи обрану стратегію, зберігає метадані в базі даних через репозиторій, створює запис у журналі операцій, додає деталі для кожного файлу та обробляє можливі помилки. Клієнтський код не знає про існування цих внутрішніх компонентів і працює лише з простим інтерфейсом фасаду.

Аналогічно працюють інші методи фасаду для відкриття архівів, витягування файлів, додавання нових файлів до існуючого архіву, видалення файлів, тестування цілісності та інших операцій. Кожен метод приховує складність координації між різними підсистемами та забезпечує консистентне логування всіх дій.

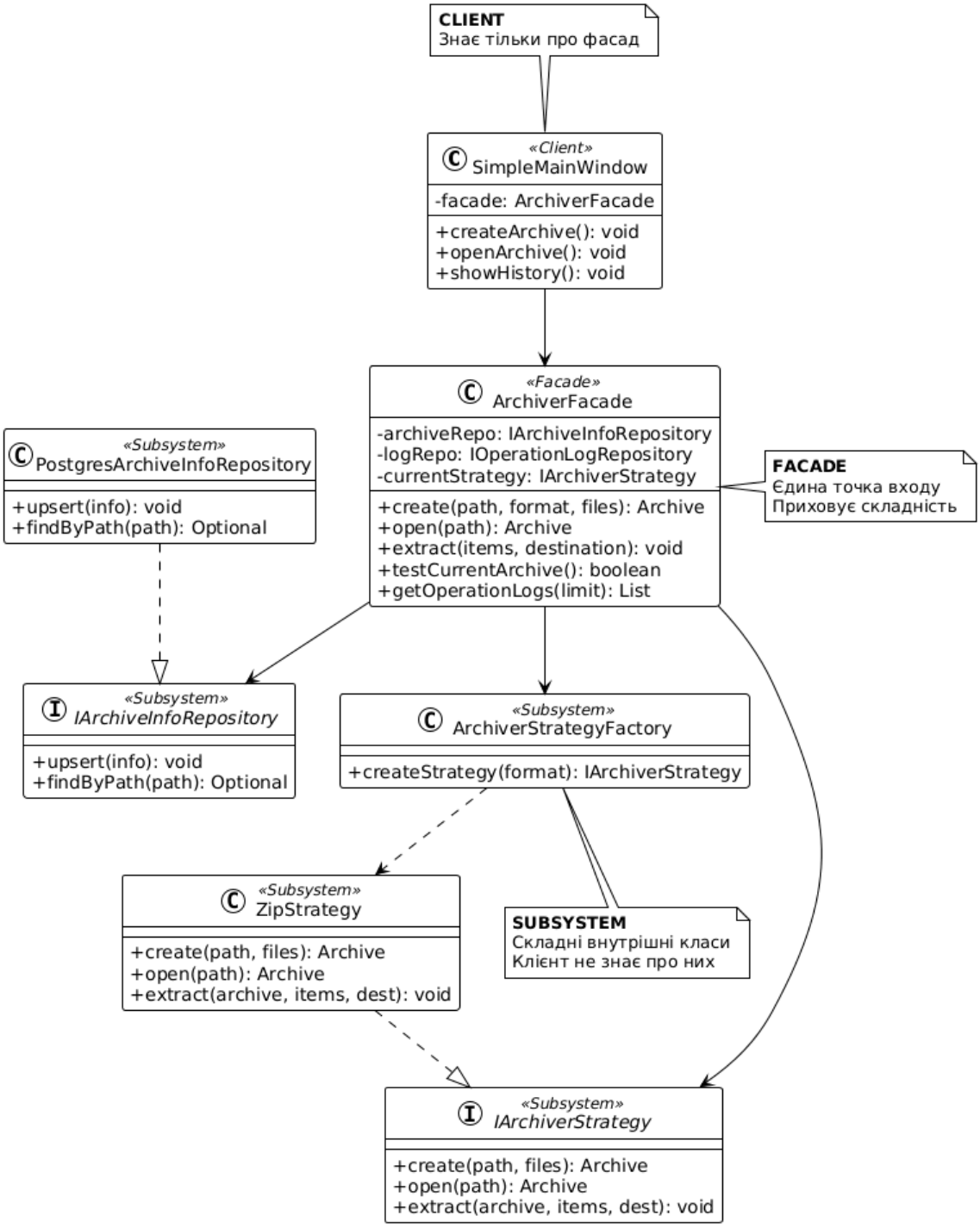


Рисунок 1. Діаграма класів для реалізації патерну Facade

Клас ArchiverFacade

Приклад (весь код класу на GitHub)

```
package facade;
public class ArchiverFacade {
    private final IArchiveInfoRepository archiveRepo;
    private final IOperationLogRepository logRepo;

    private Archive currentOpenArchive;
    private IArchiverStrategy currentStrategy;

    private final IArchiveBookmarkRepository bookmarkRepo;
```

```
private final IOperationDetailRepository detailRepo;
public ArchiverFacade(IArchiveInfoRepository archiveRepo, IOperationLogRepository logRepo,
                      IArchiveBookmarkRepository bookmarkRepo, IOperationDetailRepository detailRepo) {
    this.archiveRepo = archiveRepo;
    this.logRepo = logRepo;
    this.bookmarkRepo = bookmarkRepo;
    this.detailRepo = detailRepo;
}
public Archive create(String path, ArchiveFormat format, List<String> files) {
    OperationLog logEntry = null;
    try {
        currentStrategy = ArchiverStrategyFactory.createStrategy(format);
        currentOpenArchive = currentStrategy.create(path, files);
        ArchiveInfo newInfo = new ArchiveInfo(
                0L, path, format.name(),
                currentOpenArchive.getTotalSize(),
                LocalDateTime.now(), LocalDateTime.now());
        archiveRepo.upsert(newInfo);
        long archiveId = archiveRepo.findByPath(path)
                .map(ArchiveInfo::getId).orElse(0L);

        logEntry = new OperationLog(
                archiveId, OperationType.CREATE,
                OperationStatus.SUCCESS, "Створено архів з " + files.size() + " файлів");

        long logId = logRepo.add(logEntry);
        for (String filePath : files) {
            OperationDetail detail = new OperationDetail(
                    0,
                    logId,
                    filePath,
                    OperationType.CREATE,
                    OperationStatus.SUCCESS,
                    "Файл додано до архіву"
            );
            detailRepo.add(detail);
        }
        return currentOpenArchive;
    }catch (Exception e) {
            long logId = logRepo.add(new OperationLog(
                    0L, OperationType.CREATE,
                    OperationStatus.FAILURE, "Помилка створення архіву"
            ));
            OperationDetail detail = new OperationDetail(
                    0, logId,
                    path,
                    OperationType.CREATE,
                    OperationStatus.FAILURE,
                    "Помилка: " + e.getMessage()
            );
            detailRepo.add(detail);
        throw new RuntimeException("Помилка створення архіву", e);
    }
}
```

Приклад використання у клієнтському коді

SimpleMainWindow.java (фрагменти)

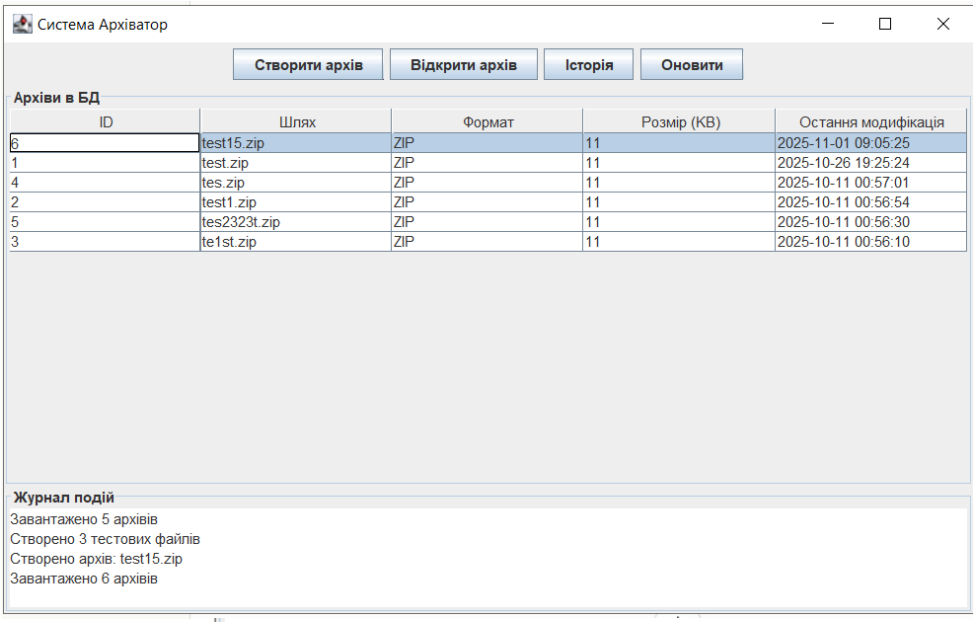
```
package ui;

public class SimpleMainWindow extends JFrame {
//  Клієнт тримає посилання ЛИШЕ на фасад
    private ArchiverFacade facade;
    private JTable table;
    private DefaultTableModel tableModel;
    private JTextArea logArea;

    public SimpleMainWindow() {

        IArchiveInfoRepository archiveRepo = new PostgresArchiveInfoRepository();
        IOperationLogRepository logRepo = new PostgresOperationLogRepository();
        IOperationDetailRepository detailRepo = new PostgresOperationDetailRepository();
        IArchiveBookmarkRepository bookmarkRepo = new PostgresArchiveBookmarkRepository();
        this.facade = new ArchiverFacade(archiveRepo, logRepo, bookmarkRepo, detailRepo);

        init();
        loadData();
    }

//  Метод створення архіву, вся складність прихована за одним методом фасаду

    private void createArchive() {

        String archiveName = JOptionPane.showInputDialog(null, "Введіть назву архіву:", "test.zip");

        if (archiveName != null && !archiveName.isEmpty()) {
            File file = new File(archiveName);
            if(file.exists()){
                int choice = JOptionPane.showConfirmDialog(null, "Файл" + archiveName + " вже існує.
Замінити?",
                        "Файл існує", JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);
                if(choice != JOptionPane.YES_OPTION){
                    logArea.append("Створення архіву скасовано\n");
                    return;
                }
                file.delete();
            }

            try {
                List<String> files = createTestFiles();
                Archive archive = facade.create(archiveName, ArchiveFormat.ZIP, files);

                logArea.append("Створено архів: " + archive.getFilePath() + "\n");
                loadData();

            } catch (Exception e) {
                JOptionPane.showMessageDialog(null, "Помилка: " + e.getMessage());
                logArea.append("Помилка: " + e.getMessage() + "\n");
            }
        }
    }
```

Тестування реалізації

Для перевірки коректності роботи реалізованого патерну було створено та запущено тестовий клас SimpleMainWindow з графічним інтерфейсом користувача. Тестування включало виконання основних операцій з архівами через методи фасаду.

Крок 1: Запуск програми та ініціалізація фасаду. При запуску SimpleMainWindow створюється екземпляр ArchiverFacade з усіма необхідними репозиторіями. Клієнтський код використовує лише посилання на фасад, не знаючи про внутрішні компоненти системи.

Крок 2: Створення нового ZIP архіву. Користувач вводить назву архіву "test15.zip" та вибирає файли (поки не дороблено) для архівації. Викликається метод facade.create(), який координує роботу ArchiverStrategyFactory для отримання ZipStrategy, створює архів, зберігає метадані в базі даних через PostgresArchiveInfoRepository та логує операцію через PostgresOperationLogRepository. Всі ці кроки приховані від клієнтського коду за єдиним викликом методу фасаду.



Деталі операції ID: 58
==========
Файл: D:\IdeaProjects\untitled11\test_files\test_file1.txt
Дія: CREATE
Статус: SUCCESS
Повідомлення: Файл додано до архіву

Файл: D:\IdeaProjects\untitled11\test_files\test_file2.txt
Дія: CREATE
Статус: SUCCESS
Повідомлення: Файл додано до архіву

Файл: D:\IdeaProjects\untitled11\test_files\test_file3.txt
Дія: CREATE
Статус: SUCCESS
Повідомлення: Файл додано до архіву

Рисунок 2. Вікно програми після створення

Крок 3: Відкриття існуючого архіву. Викликається метод facade.open(), який визначає формат архіву, отримує відповідну стратегію, відкриває архів та оновлює інформацію про останній доступ у базі даних.

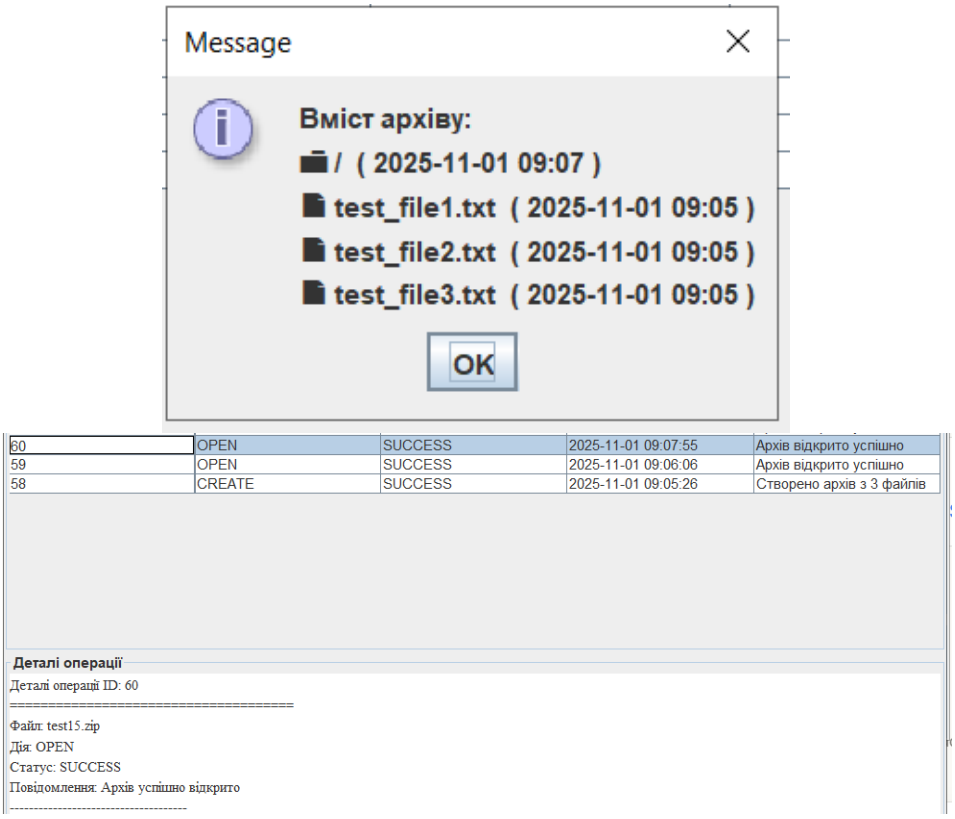


Рисунок 2. Вікно програми після відкриття архіву

Крок 4: Перегляд списку недавніх архівів. Метод facade.listRecentArchives() повертає список останніх архівів з бази даних, який відображається у таблиці графічного інтерфейсу.

(скріншот на першому кроці показує це)

Результати тестування:

Створення архіву test15.zip виконано успішно:

- Фасад автоматично вибрав ZipStrategy через фабрику стратегій
- Архів створено з 3 тестових файлів
- Метадані збережено в таблиці archive_info бази даних
- Створено запис у таблиці operation_log з типом CREATE та статусом SUCCESS
- Додано 3 записи в таблицю operationdetail для кожного файлу

Реалізація патерну Facade успішно пройшла тестування. Клієнтський код використовує лише прості методи фасаду, не знаючи про складну координацію між фабриками, стратегіями та репозиторіями. Всі операції виконуються коректно, метадані зберігаються в базі даних, логування працює належним

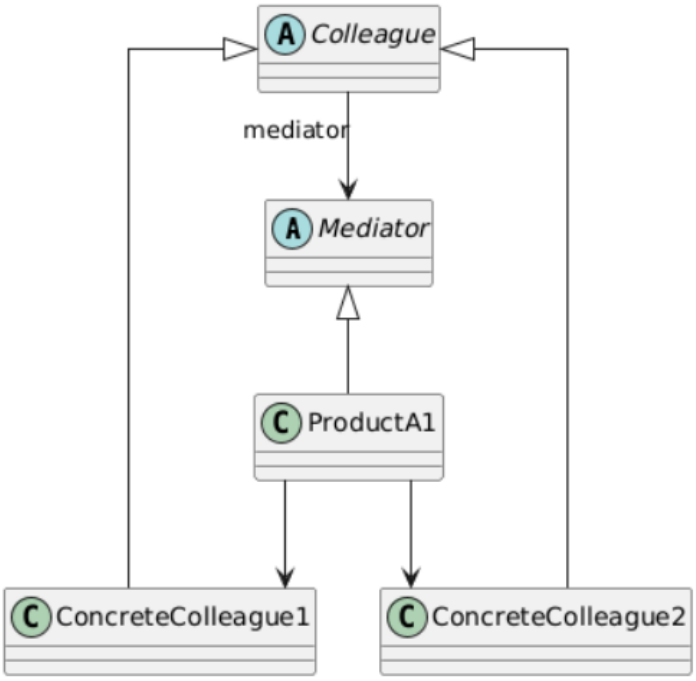
чином. Патерн Facade досягає своєї мети - спрощує інтерфейс для роботи зі складною підсистемою архівації.

Питання до лабораторної роботи

1. Яке призначення шаблону "Посередник"?

"Посередник" (Mediator) - це як "староста" у групі або "диригент" в оркестрі. Його завдання - розплутати хаос, коли купа об'єктів (наприклад, кнопок, галочок і текстових полів на одному вікні) починають спілкуватися одне з одним напряму. Замість того, щоб кожна кнопка знала про всі інші елементи, вони всі знають тільки про одного "старосту"-посередника. Коли щось відбувається (наприклад, натиснули галочку), компонент просто каже про це посереднику, а вже посередник вирішує, кому ще про це треба знати (наприклад, які кнопки зробити неактивними).

2. Нарисуйте структуру шаблону "Посередник".



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

Mediator: Це інтерфейс (або абстрактний клас), який визначає, як "Колеги" будуть спілкуватися з Посередником.

ProductA1: Це конкретна реалізація Посередника (ConcreteMediator). Він успадковує від Mediator.

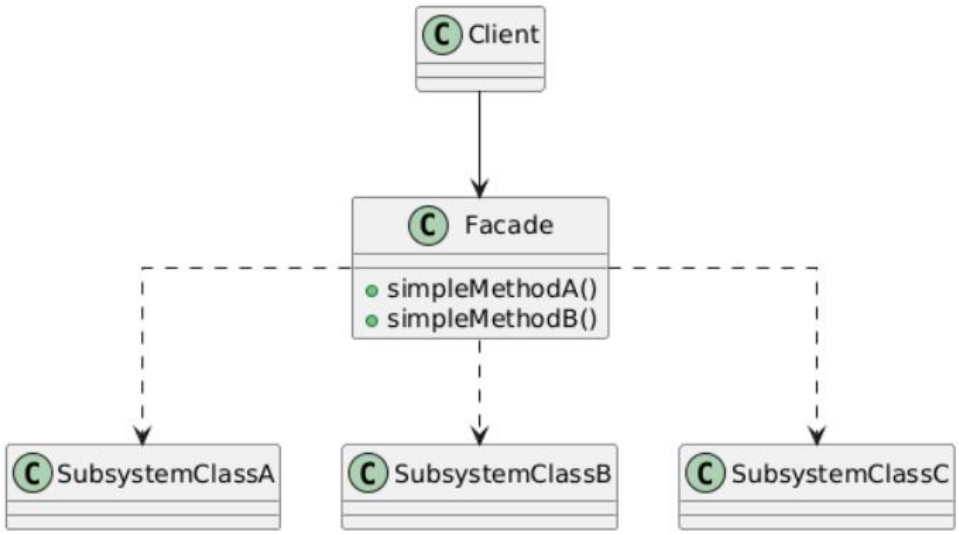
Colleague: Це інтерфейс (або абстрактний клас) для "Колег" (компонентів системи). Він містить посилання на об'єкт Mediator.

ConcreteColleague1 та ConcreteColleague2: Це конкретні компоненти, які успадковують від Colleague.

4. Яке призначення шаблону "Фасад"?

"Фасад" (Facade) - це проста "ширма" або "обгортка" для дуже складної системи. Уявіть, що є 10-20 класів, які роблять щось складне (як у архіваторі: стратегії, репозиторії, логи). Замість того, щоб змушувати програміста розбиратися в них усіх, створюється один клас ArchiverFacade з кількома простими методами: createArchive(), extractArchive(). Він приховує всю внутрішню "кухню", і програміст (клієнт) працює тільки з цим простим інтерфейсом.

5. Нарисуйте структуру шаблону "Фасад".



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

Facade (Фасад): Це той єдиний клас (як InternetClient у прикладі), який надає прості методи для зовнішнього світу.

Subsystem (Підсистема): Це купа складних класів "всередині", які роблять всю брудну роботу. Вони не знають про існування фасаду.

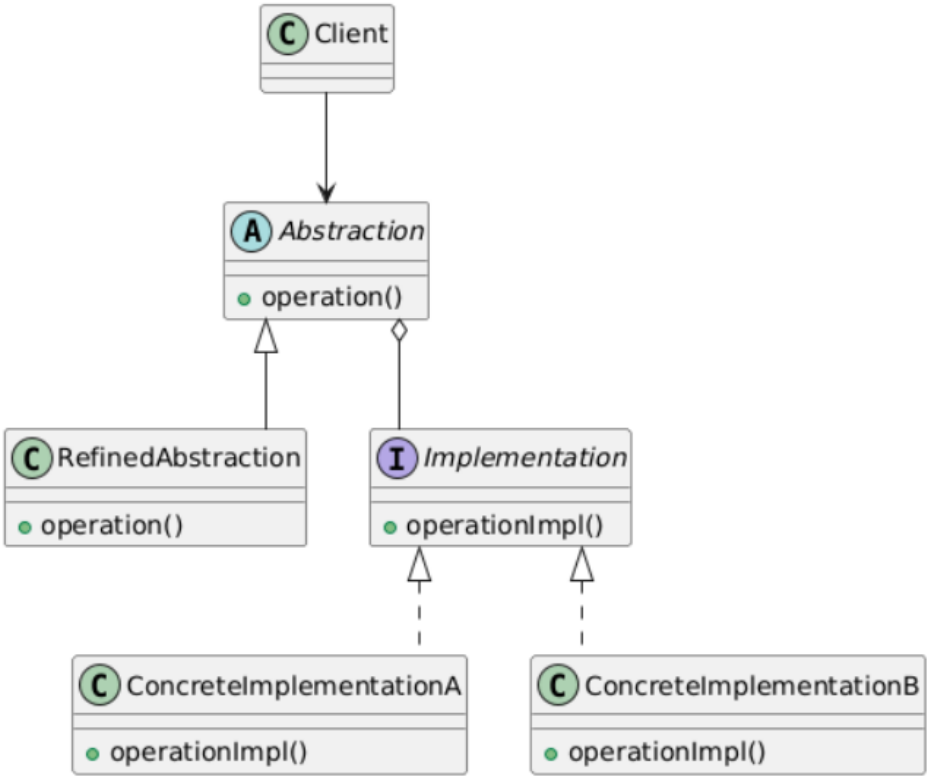
Client (Клієнт): Це код (наприклад, UI), який спілкується тільки з Фасадом і не лізе у складну підсистему.

7. Яке призначення шаблону "Міст"?

"Міст" (Bridge) потрібен, щоб "роз'єднати" дві різні ієрархії (два набори класів), які зазвичай переплітаються. Він розділяє "що ми робимо" (абстракція) від "як ми це робимо" (реалізація).

Приклад з методички шикарний: є Фігури (коло, лінія) і є Способи Малювання (на екрані, на принтері). Замість того, щоб створювати 4 класи (CirclePrint, CircleDraw, LinePrint, LineDraw), ми створюємо дві окремі ієрархії: (Circle, Line) і (DrawOnScreen, DrawOnPrinter). А "Міст" дозволяє їх комбінувати як завгодно.

8. Нарисуйте структуру шаблону "Міст".



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

Abstraction (Абстракція): Це "що ми робимо" (наприклад, базовий клас Shape). Вона має посилання на Implementation.

RefinedAbstraction (Уточнена Абстракція): Це конкретні "що" (наприклад, Circle або Line).

Implementation (Реалізація): Це інтерфейс того, "як ми це робимо" (наприклад, DrawApi).

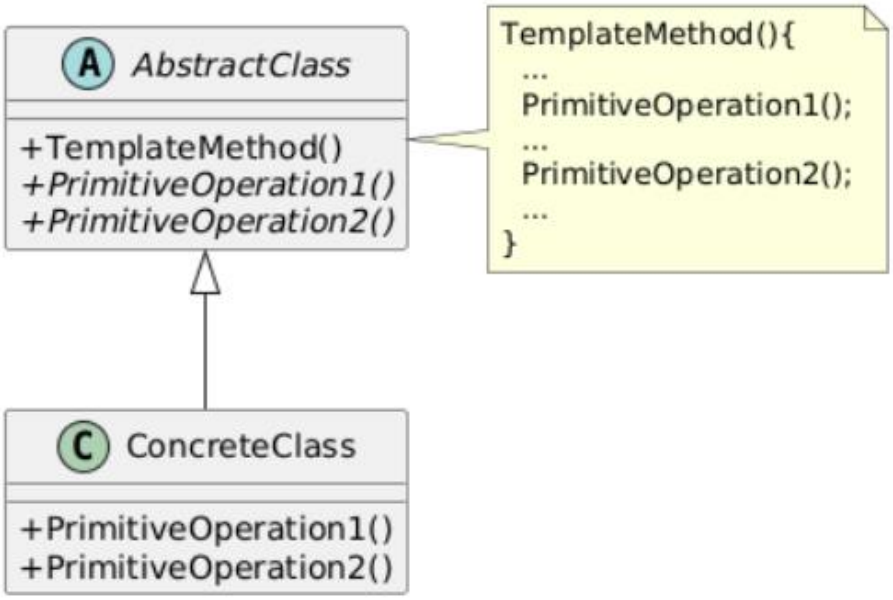
ConcreteImplementation (Конкретна Реалізація): Це конкретні "як" (наприклад, WindowDrawApi або PrinterDrawApi).

Взаємодія: Клієнт викликає метод у Abstraction (наприклад, circle.draw()). А Abstraction (фігура) делегує (передає) цю роботу об'єкту Implementation (драйверу малювання).

10. Яке призначення шаблону "Шаблоний метод"?

"Шаблонний метод" (Template Method) - це коли ви створюєте "скелет" алгоритму в базовому класі, але дозволяєте дочірнім класам перевизначити деякі конкретні кроки. Як у прикладі з відеоредактором: базовий клас каже: "Алгоритм такий: 1. Відкрити файл. 2. Прочитати кадри. 3. Прочитати звук". Крок 1 і 3 однакові для всіх, а крок 2 (читання кадрів) буде різним для MPEG-2, MPEG-4 і H.262, тому його реалізують дочірні класи

11. Нарисуйте структуру шаблону "Шаблоний метод".



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

AbstractClass (Абстрактний клас):

- Містить головний метод TemplateMethod(). Це і є "шаблон" або "скелет" алгоритму.
- Всередині TemplateMethod() відбувається виклик інших, примітивних операцій (PrimitiveOperation1(), PrimitiveOperation2()) у заданій послідовності.
- Оголошує ці примітивні операції (зазвичай як абстрактні).

ConcreteClass (Конкретний клас):

- Успадковує від AbstractClass.
- Надає конкретну реалізацію для кроків PrimitiveOperation1() та PrimitiveOperation2(), які були абстрактними у батьківському класі.

Взаємодія: Клієнтський код викликає метод TemplateMethod() у об'єкта ConcreteClass. Виконується "скелет" алгоритму з AbstractClass, але в потрібні моменти він викликає реалізації кроків (PrimitiveOperation1(), PrimitiveOperation2()) з ConcreteClass.

13. Чим відрізняється шаблон "Шаблонний метод" від "Фабричного методу"?

Вони схожі тим, що обидва використовують успадкування, але мета у них зовсім різна.

Шаблонний метод: Визначає послідовність дій (скелет алгоритму).

Фабричний метод: Використовується, щоб створити новий об'єкт, але дозволяє підкласам вирішувати, який саме клас об'єкта створювати.

14. Яку функціональність додає шаблон "Міст"?

Він додає гнучкість. Він дозволяє вам додавати нові "абстракції" (наприклад, нові фігури) і нові "реалізації" (наприклад, нові способи малювання) незалежно одне від одного. Вам не доведеться переписувати 10 класів малювання, якщо ви додав одну нову фігуру.

Висновки

В ході виконання даної лабораторної роботи було вивчено патерн проектування "Facade" та успішно реалізовано його в системі "Архіватор". Створення класу ArchiverFacade дозволило інкапсулювати складну взаємодію між підсистемами (стратегіями, репозиторіями, логуванням) та надати клієнтському коду простий і уніфікований інтерфейс, що значно знизило загальну складність системи.