



Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

ЛАБОРАТОРНА РОБОТА №6

з дисципліни "Технології розроблення програмного забезпечення"

Виконав

студент групи ІА–33:

Китченко Д.В.

Перевірив:

Мягкий М.Ю.

Київ 2025

Зміст

Вступ.....	3
Мета роботи.....	4
Теоретичні відомості.....	5
Хід роботи	12
Тестування реалізації.....	15
Питання до лабораторної роботи	17
Висновки.....	22

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.

У даній лабораторній роботі розглядається породжуючий патерн "Factory Method", який надає елегантний спосіб делегування створення об'єктів підкласам, тим самим підвищуючи гнучкість системи.

Factory Method вирішує фундаментальну проблему: як створювати об'єкти, коли точний тип об'єкта має визначатися динамічно, залежно від контексту виконання програми. Замість жорсткої прив'язки до конкретних класів через оператор створення об'єктів, патерн пропонує делегувати це рішення спеціалізованим підкласам, тим самим підвищуючи гнучкість та розширюваність системи.

Мета роботи

Вивчити структуру шаблонів "Abstract Factory", "Factory Method", "Memento", "Observer", "Decorator" та навчитися застосовувати їх в реалізації програмної системи. Вивчити структуру та призначення шаблону "Factory Method" та навчитися застосовувати його на практиці для створення об'єктів у ситуаціях, коли точний тип об'єкта, що створюється, має визначатися підкласами.

6.2. Теоретичні відомості

6.2.1. Шаблон «Abstract Factory»

Призначення патерну: Шаблон «Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів [6]. Для цього вноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів. Хорошим прикладом використання абстрактної фабрики є ADO.NET: існує загальний клас DbProviderFactory, здатний створювати об'єкти типів DbConnection, DbDataReader, DbAdapter та ін.; існують реалізації цих фабрик і об'єктів – SqlProviderFactory, SqlConnection, SqlDataReader, SqlDataAdapter і так далі. Відповідно, якщо додатку необхідно працювати з різними базами даних (чи потрібна така можливість), то досить

використати базові реалізації (Db.) і підставити відповідну фабрику у момент ініціалізації фабрики (Factory = new SqlProviderFactory()).

Цей шаблон передусім структурує знання про схожі об'єкти (що називаються сімействами, як класи для доступу до БД) і створює можливість взаємозаміни різних сімейств (робота з Oracle ведеться також, як і робота з SQL Server). Проте, при використанні такої схеми у край незручно розширювати фабрику – для додавання нового методу у фабрику необхідно додати його в усіх фабриках і створити відповідні класи, що створюються цим методом.

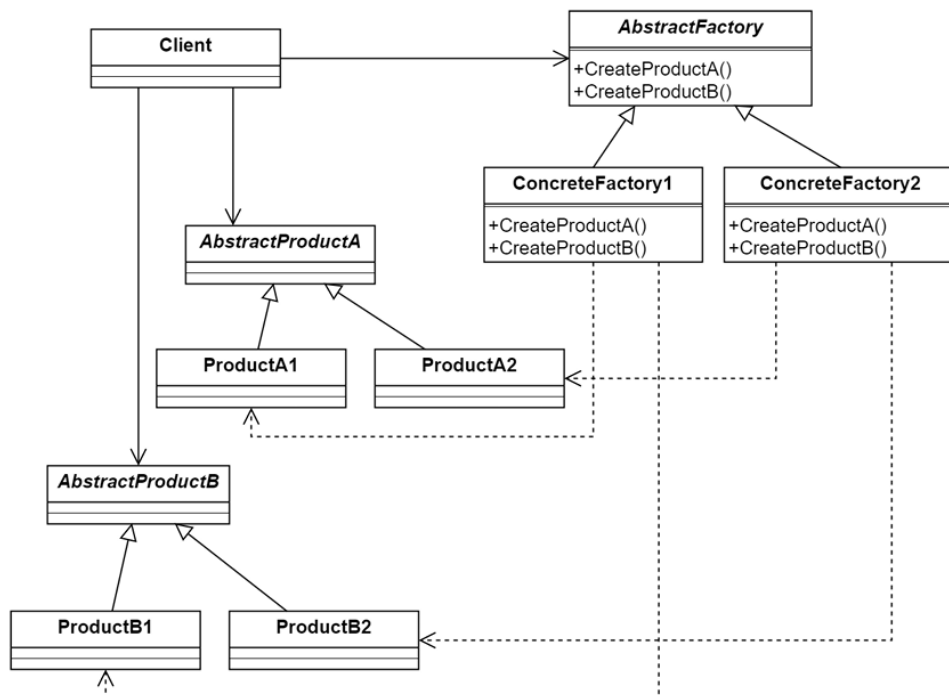


Рисунок 6.1. Структура патерну «Абстрактна фабрика»

Проблема: Ви розробляєте для Roguelike гри модуль автоматичної генерації кімнат. Модуль генерації кімнат, в процесі генерації конкретної кімнати, створює стіни, двері, вікна, підлогу та меблі в кімнаті. Модуль повинен підтримувати генерацію кімнат у різних стилях, таких як стиль хайтек, модерн,

класичний офіс. Також критично є щоб всі елементи в одній згенерованій кімнаті відносилися до одного стилю.

Рішення: Для вирішення поставленої задачі дуже добре підходить використання патерну «Абстрактна фабрика».

Для кожного типу продукту створюємо свій інтерфейс продукту який об'являє функції доступні для використання з цим продуктом. Наприклад, можна виділити інтерфейси для стін, вікон, дверей, підлоги, а також окремі інтерфейси для меблів, такі як стіл, шафа, крісло та інші. Від кожного інтерфейсу наслідуюмо і реалізуємо продукти різних типів, наприклад, для стола: інтерфейс `ITable` та реалізації продуктів `HighTechTable`, `ModernTable` та інші.

Далі визначаємо інтерфейс для абстрактної фабрики, який буде містити методи для створення кожного типу продукту. Створюємо класи конкретних фабрик під кожен стиль: `HighTechFabric`, `ModernFabric` та інші. Кожна конкретна фабрика буде створювати всі типи продуктів, але всі вони будуть відноситися до одного стилю.

Далі в алгоритм генерації кімнати будемо передавати конкретну фабрику і алгоритм при створенні продуктів тільки через фабрику завжди буде отримувати продукти одного стилю і працювати з продуктами тільки через інтерфейс продукту.

Таким чином ми вирішуємо проблему узгодженості стилів всіх елементів в кімнаті, а за рахунок використання різних конкретних фабрик ми зможемо генерувати кімнати різних стилів. Якщо нам потрібно буде додати ще один стиль, то достатньо буде реалізувати нові дочірні класи для кожного елемента кімнати, а також нову конкретну фабрику під цей стиль.

Переваги та недоліки:

- + Спрощує створення об'єктів і код стає легшим для розуміння.
- + Об'єкти створені однією фабрикою добре узгоджуються один з одним і зменшується кількість помилок взаємодії між ними.
- + Відокремлення створення об'єктів від їх використання, за рахунок чого, код стає більш структурованим.

- + Додавання нових сімейств продуктів виконується без зміни існуючого коду.
- Збільшується складність коду, особливо для простих проєктів.
- Додавання нового типу продукту є складним і вимагає змін коду в багатьох місцях.

6.2.2. Шаблон «Factory Method»

Призначення: Шаблон «Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу [6]. Це зручно, коли хочеться додати можливість створення об'єктів не базового типу, а деякого дочірнього. Фабричний метод у такому разі є зачіпкою для впровадження власного конструктора об'єктів. Основна ідея полягає саме в заміні об'єктів їх підтипами, що при цьому зберігає ту ж функціональність; інша частина поведінки об'єктів не є інтерфейсною (AnOperation) і дозволяє взаємодіяти із створеними об'єктами як з об'єктами базового типу. Тому шаблон «Фабричний метод» носить ще назву «Віртуальний конструктор».

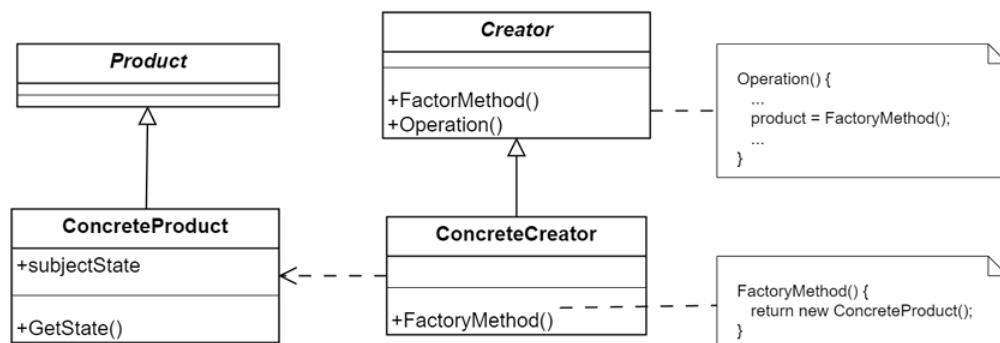


Рисунок 6.2. Структура патерну «Фабричний Метод»

Розглянемо простий приклад. Нехай наш застосунок працює з мережевими драйвер-мі і використовує клас `Packet` для зберігання даних, що передаються в мережу. Залежно від використовуваного протоколу, існує два перевантаження –

`TcpPacket`, `UdpPacket`. І відповідно два створюючі об'єкти (`TcpCreator`, `UdpCreator`) з фабричним методом (який створює відповідні реалізації).

Проте базова функціональність (передача пакету, прийом пакету, заповнення пакету даними) нічим не відрізняється один від одного, відповідно поміщається у базовий клас `PacketCreator`. Таким чином поведінка системи залишається тим же, проте з'являється можливість підстановки власних об'єктів в процес створення і роботи з пакетами.

Переваги та недоліки:

- + Позбавляє клас від прив'язки до конкретних класів продуктів.
- + Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.
- + Спрощує додавання нових продуктів до програми.
- Може призвести до створення великих паралельних ієрархій класів.

6.2.3. Шаблон «Memento»

Призначення: Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції [6]. Об'єкт «Memento» служить виключно для збереження змін над початковим об'єктом (Originator). Лише початковий об'єкт має можливість зберігати і отримувати стан об'єкту «Memento» для власних цілей, цей об'єкт є «порожнім» для кого-небудь ще. Об'єкт «Caretaker» використовується для передачі і зберігання мemento об'єктів в системі.

Таким чином вдається досягти наступних цілей:

- зберігання стану повністю відділяється від початкових об'єктів, що полегшує їх реалізацію;
- передача об'єктів «Memento» лягає на плечі Caretaker об'єктів, що дозволяє гнучкіше управляти станами об'єктів і спростити дизайн класів початкових об'єктів;
- збереження і відновлення стану реалізовані у вигляді двох простих методів і є закритими для кого-небудь ще окрім початкових об'єктів, таким чином не порушуючи інкапсуляцію.

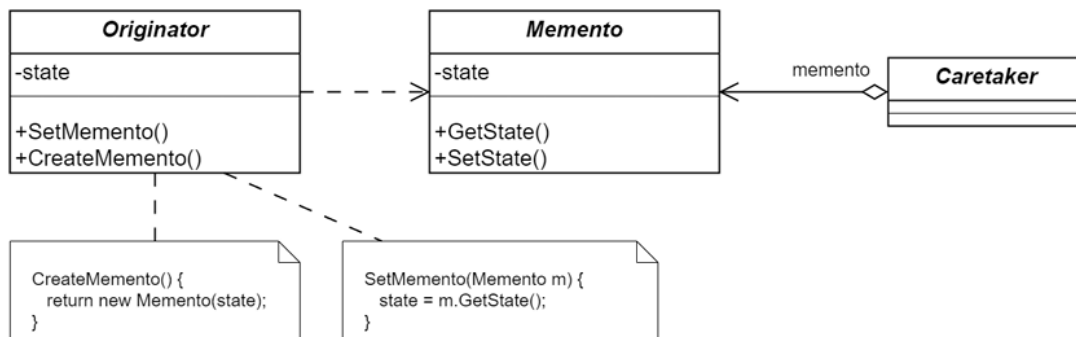


Рисунок 6.3. Структура шаблону «Знімок»

Шаблон «Мemento» дуже зручно використати разом з шаблоном «Команда» для реалізації «скасовних» дій – дані про дію зберігаються в мemento, а команда має можливість вважати і відновити початкове положення відповідних об'єктів.

Переваги та недоліки:

- + Не порушує інкапсуляцію вихідного об'єкта.
- + Спрощує структуру вихідного об'єкта. Не потрібно зберігати історію версій свого стану.
- Вимагає багато пам'яті, якщо клієнти дуже часто створюють знімки.
- Може спричинити додаткові витрати пам'яті, якщо об'єкти, що зберігають історію, не звільняють ресурси, зайняті застарілими знімками.

6.2.4. Шаблон «Observer»

Призначення: Шаблон визначає залежність «один-до-багатьох» таким чином, що коли один об'єкт змінює власний стан, усі інші об'єкти отримують про це сповіщення і мають можливість змінити власний стан також [6].

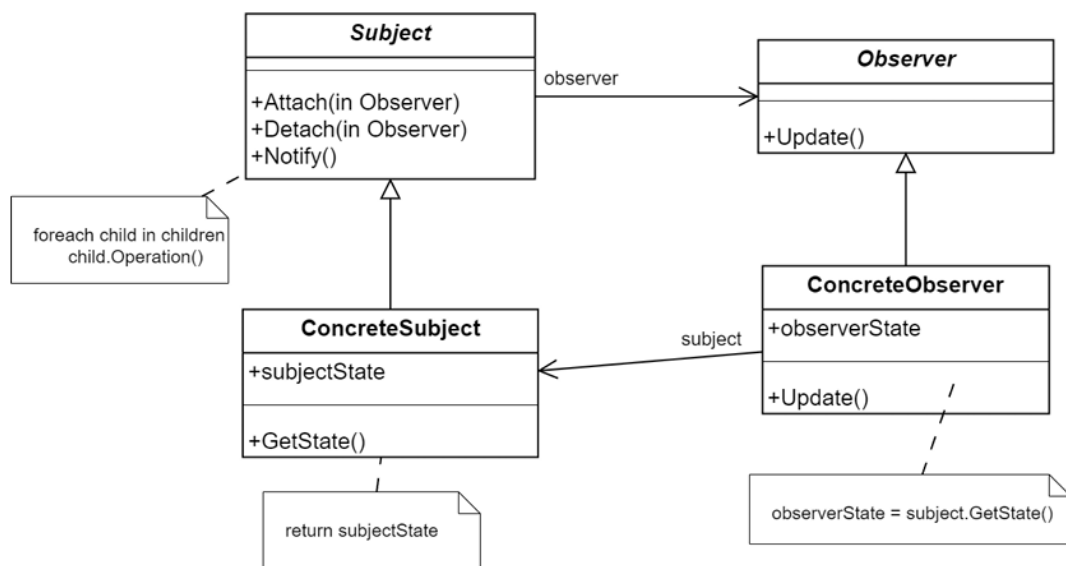


Рисунок 6.4. Структура патерна «Спостерігач»

Розглянемо цей шаблон на прикладі. Припустимо, є деяка банківська система і декілька користувачів переглядають баланс на рахунок пана І. У цей момент пан І. кладе на свій рахунок деяку суму, яка міняє загальний баланс. Кожен з користувачів, що переглядали баланс, отримує про це звістку (для користувачів ця звістка може бути прозорою – просто зміна цифр, або попередження про те, що баланс змінився). Раніше неможливі дії для користувачів (переведення до іншої категорії клієнтів і тому подібне) стають доступними.

Цей шаблон дуже широко поширений в шаблоні MVVM і механізмі «прив'язок» (bindings) в WPF і частково в WinForms. Інша назва шаблону – підписка/розсилка. Кожен з оглядачів власноручно підписується на зміни конкретного об'єкту, а об'єкти зобов'язані сповіщати своїх передплатників про усі свої зміни (на даний момент конкретних механізмів автоматичного сповіщення про зміну стану в .NET мовах не існує).

Приклад з життя: Коли ви підписуєтесь на канал на YouTube і натискаєте на «дзвіночок» ви фактично оформлюєте підписку на отримання повідомлень про вихід нових відео. Вам не потрібно заходити на канал і перевіряти чи не вийшло нове відео. Ви будете отримувати повідомлення про вихід нових відео на каналах

Фактично на сервісі YouTube є список підписників на канал, хто має отримувати повідомлення про вихід нового відео.

В якості прикладу також можна згадати підписку «Повідомити про появу товару» на сторінці товару в магазині «Розетка».

Переваги та недоліки:

- + Можливість паралельної та асинхронної обробки повідомлень про оновлення.
- + Спостерігачів можна добавляти та видаляти в будь-який момент часу.
- + Спостерігач і суб'єкт можуть працювати в різних потоках.
- + Реалізує принцип слабкого зв'язку між об'єктами.
- Послідовність розсилки повідомлень підписникам не підтримується.

6.2.5. Шаблон «Decorator»

Призначення: Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми [6]. Декоратор деяким чином «обертає» (за рахунок агрегації) початковий об'єкт зі збереженням його функцій, проте дозволяє додати додаткові дії. Такий шаблон надає гнучкіший спосіб зміни поведінки об'єкту чим просте спадкоємство, оскільки початкова функціональність зберігається в повному об'ємі. Більше того, таку поведінку можна застосовувати до окремих об'єктів, а не до усієї системи в цілому.

Простим прикладом є накладення смуги прокрутки до усіх візуальних елементів. Кожен об'єкт, який може прокручуватися, обертається в «прокручуваному» елементі, і при необхідності з'являється полоса прокрутки. Початкові функції елементу (наприклад, рядки статусу) залишаються незмінними.

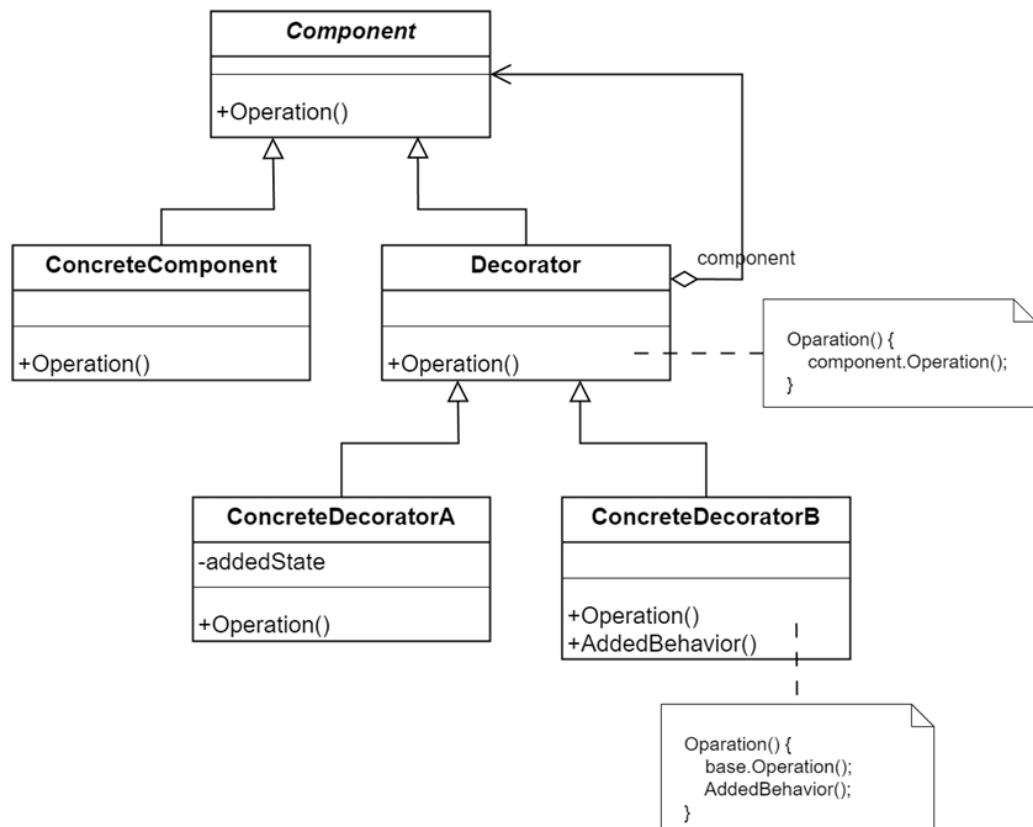


Рисунок 6.5. Структура патерну «Декоратор»

Переваги та недоліки:

- + Дозволяє мати кілька дрібних об'єктів, замість одного об'єкта «на всі випадки життя».
- + Дозволяє додавати обов'язки «на льоту».
- + Більша гнучкість, ніж у спадкування.
- Велика кількість крихітних класів.
- Важко конфігурувати об'єкти, які загорнуто в декілька обгортки одночасно.

Хід роботи

В рамках розробки програми "Архіватор", яка повинна підтримувати роботу з різними форматами архівів (ZIP, RAR, 7Z тощо), виникла потреба в гнучкому механізмі створення відповідних об'єктів-стратегій (IArchiverStrategy) для кожного формату. Існуюча реалізація через статичну фабрику (ArchiverStrategyFactory) є простою, але менш гнучкою та не відповідає класичному патерну "Фабричний метод".

Для вирішення задачі було застосовано патерн "Фабричний метод". Було спроектовано ієрархію класів-творців:

- [ArchiveProcessor](#) (Creator): Абстрактний клас, що містить загальну логіку обробки архівів (створення, відкриття, витягування). Він оголошує абстрактний фабричний метод `createStrategy()`, який має повертати об'єкт типу IArchiverStrategy.
- [ZipArchiveProcessor](#), [RarArchiveProcessor](#) (ConcreteCreator): Конкретні класи, що успадковують ArchiveProcessor. Кожен з них перевизначає фабричний метод `createStrategy()`, повертаючи відповідну конкретну стратегію (ZipStrategy, RarStrategy)

Клієнтський код (у даному випадку, тестовий клас, але в реальній системі це міг би бути ArchiverFacade або GUI) створює екземпляр конкретного ArchiveProcessor (наприклад, `new ZipArchiveProcessor()`) і викликає його загальні методи (`createArchive`, `openArchive`). Внутрішньо ці методи викликають фабричний метод `createStrategy()`, який, завдяки поліморфізму, повертає правильну стратегію для даного процесора.

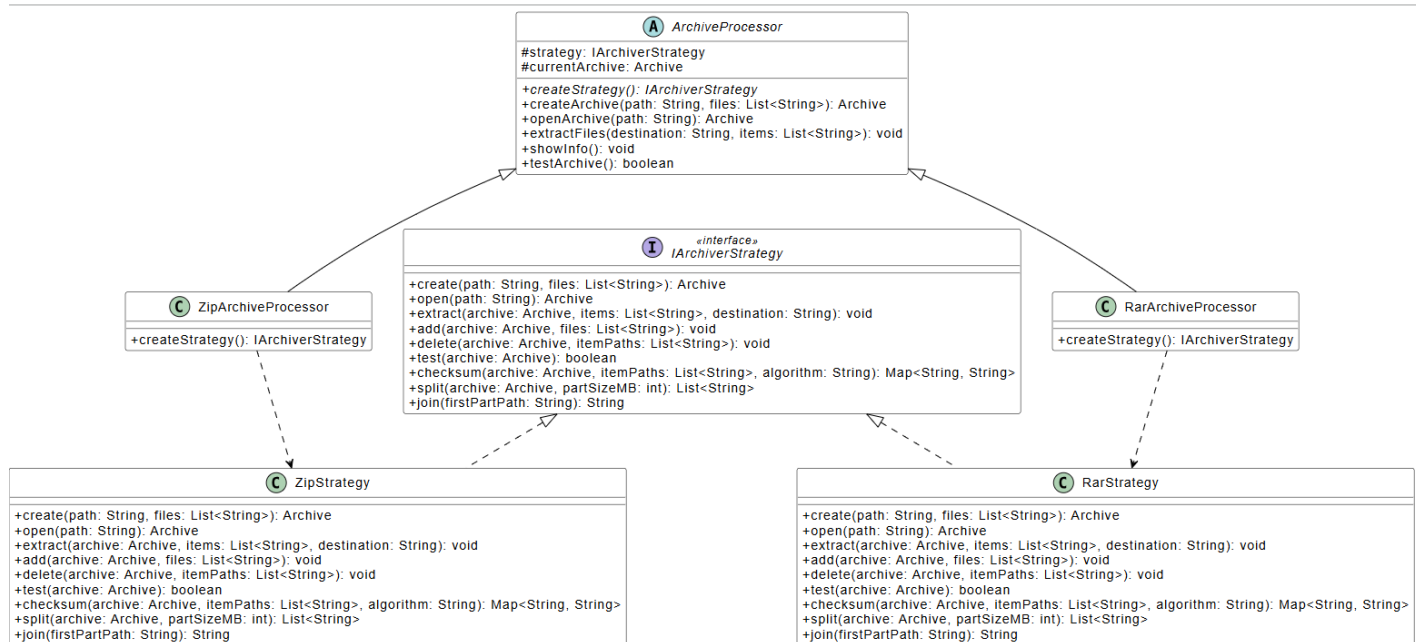


Рисунок 1. Діаграма класів для реалізації патерну Adapter

- ArchiveProcessor (Creator) визначає абстрактний фабричний метод createStrategy().
- ZipArchiveProcessor, RarArchiveProcessor (ConcreteCreator) реалізують цей метод, створюючи відповідні конкретні продукти (ZipStrategy, RarStrategy).
- IArchiverStrategy (Product) є інтерфейсом для створюваних об'єктів.
- ArchiveProcessor використовує створений продукт (IArchiverStrategy) для виконання своєї роботи.

Відповідно до завдання було реалізовано 3 нових класи, що безпосередньо відносяться до патерну: ArchiveProcessor, ZipArchiveProcessor, RarArchiveProcessor.

1. Абстрактний клас ArchiveProcessor з фабричним методом

Цей клас містить бізнес-логіку та оголошення абстрактного фабричного методу createStrategy().

```

package factory;

import model.Archive;
import strategy.IArchiverStrategy;

import java.util.List;

public abstract class ArchiveProcessor {

    protected IArchiverStrategy strategy;
    protected Archive currentArchive;
  
```

```

protected abstract IArchiverStrategy createStrategy();

public Archive createArchive(String path, List<String> files) {
    System.out.println("\n ArchiveProcessor: створення архіву " + path);

    // виклик фабричного методу
    strategy = createStrategy();

    System.out.println("Використовується: " +
strategy.getClass().getSimpleName());

    currentArchive = strategy.create(path, files);

    System.out.println("Архів створено!");
    return currentArchive;
}
//повний код на GitHub
}

```

2. Конкретний клас ZipArchiveProcessor

Цей клас успадковує ArchiveProcessor і реалізує фабричний метод, повертаючи ZipStrategy.

```

package factory;

import strategy.IArchiverStrategy;
import strategy.ZipStrategy;

public class ZipArchiveProcessor extends ArchiveProcessor {
    @Override
    protected IArchiverStrategy createStrategy() {
        System.out.println("Factory Method: створює ZipStrategy");
        return new ZipStrategy();
    }
}

```

3. Приклад використання у TestFactoryMethod

Демонстрація поліморфізму: клієнтський код працює з різними процесорами через базовий тип ArchiveProcessor

```

package factory;

import java.nio.file.Files;
import java.io.File;
import java.util.*;

public class TestFactoryMethod {
    public static void main(String[] args) {
        System.out.println("FACTORY METHOD ПАТЕРН");
        try {
            createTestFiles();

            List<String> files = List.of(
                "test_files2/document.txt",
                "test_files2/data.txt"
            );

```

```

        System.out.println("СТВОРЕННЯ АРХІВІВ через Factory Method:\n");
        ArchiveProcessor[] processors = {
            new ZipArchiveProcessor(),
            new RarArchiveProcessor()
        };

        String[] paths = {"demo.zip", "demo.rar"};
        String[] names = {"ZIP", "RAR"};

        for (int i = 0; i < processors.length; i++) {
            System.out.println("—" + names[i]);
            processors[i].createArchive(paths[i], files);
            processors[i].showInfo();
            processors[i].testArchive();
            System.out.println();
        }

        System.out.println("ВИТЯГУВАННЯ");
        new File("extracted").mkdirs();
        processors[0].extractFiles("extracted", new ArrayList<>());

        System.out.println("FACTORY METHOD ПРАЦЮЄ");

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

```

Тестування реалізації

Для перевірки коректності роботи реалізованого патерну було створено та запущено тестовий клас [TestFactoryMethod](#). Цей клас створює екземпляри різних ConcreteCreator (ZipArchiveProcessor, RarArchiveProcessor), викликає їхні методи для створення та обробки архівів і виводить результати в консоль.

FACTORY METHOD ПАТЕРН

Тестові файли створено

СТВОРЕННЯ АРХІВІВ через Factory Method:

—ZIP

ArchiveProcessor: створення архіву demo.zip

Factory Method: створює ZipStrategy

Використовується: ZipStrategy

Створення ZIP архіву: demo.zip

Архів створено!

ІНФОРМАЦІЯ:

Шлях: demo.zip

Формат: ZIP

Розмір: 64 байт

Strategy: ZipStrategy

ArchiveProcessor: тестування...

Тестування архіву demo.zip

Тест ОК

—RAR

ArchiveProcessor: створення архіву demo.rar

Factory Method: створює RarStrategy

Використовується: RarStrategy

RarStrategy: створення RAR архіву demo.rar

обрано RAR

Архів створено!

ІНФОРМАЦІЯ:

Шлях: demo.rar

Формат: RAR

Розмір: 74 байт

Strategy: RarStrategy

ArchiveProcessor: тестування...

Тест ОК

ВИТЯГУВАННЯ

ArchiveProcessor: витягування до extracted

Витягування файлів з demo.zip

Розпакування в extracted завершено

Файли витягнуто

FACTORY METHOD ПРАЦЮЄ

Process finished with exit code 0

Результати виконання тесту:

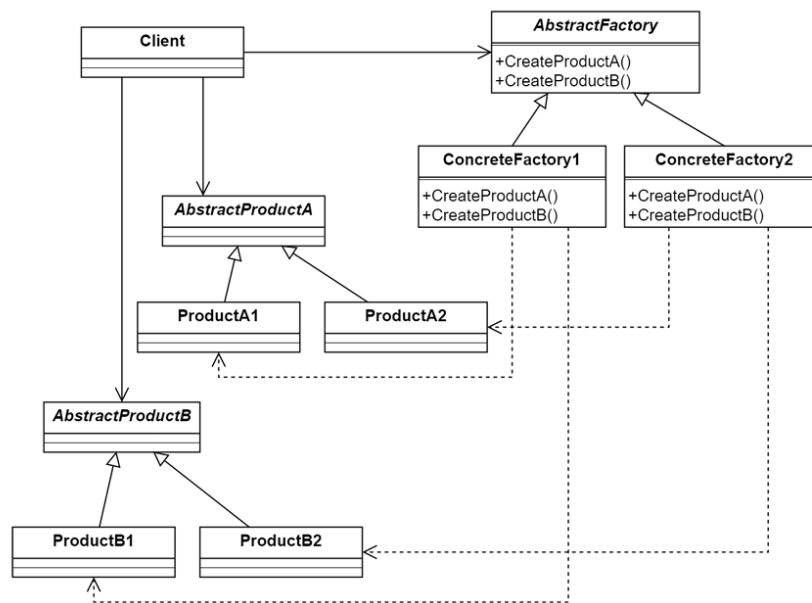
Виконання тестового класу демонструє успішне створення архівів різних форматів за допомогою відповідних ConcreteCreator-ів. У консоль виводиться лог роботи фабричного методу та інформація про створені архіви, підтверджуючи коректність реалізації патерну.

Питання до лабораторної роботи

1. Яке призначення шаблону «Абстрактна фабрика»?

Шаблон «Абстрактна фабрика» (Abstract Factory) призначений для створення сімейств споріднених або взаємозалежних об'єктів без вказівки їхніх конкретних класів. Цей патерн гарантує, що всі створені фабрикою об'єкти будуть сумісними між собою, оскільки належатимуть до одного сімейства або стилю. Наприклад, фабрика може створювати набір UI-елементів (кнопки, текстові поля, вікна) в єдиному стилі (Windows, MacOS, Linux), забезпечуючи їхню візуальну та функціональну сумісність.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

AbstractFactory: Інтерфейс або абстрактний клас, що оголошує методи для створення кожного типу продукту в сімействі.

ConcreteFactory: Класи, що реалізують **AbstractFactory**. Кожна фабрика створює продукти одного конкретного сімейства.

AbstractProduct: Інтерфейси або абстрактні класи для кожного типу продукту.

ConcreteProduct: Класи, що реалізують інтерфейси **AbstractProduct**. Вони створюються конкретними фабриками.

Client: Використовує тільки інтерфейси **AbstractFactory** та **AbstractProduct**.

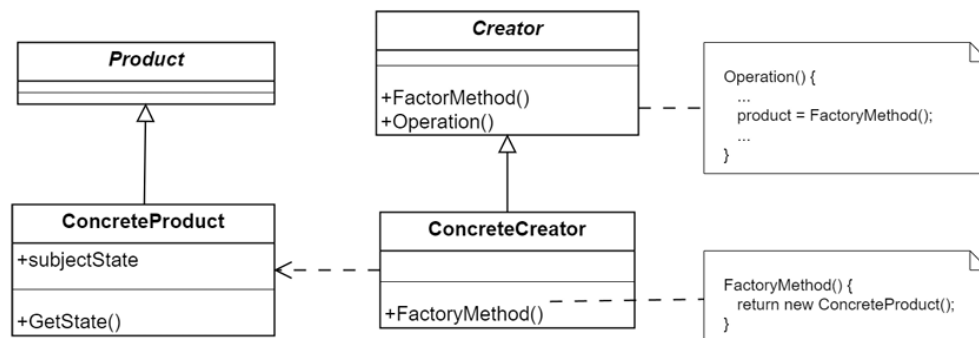
Взаємодія відбувається так: клієнт отримує посилання на **ConcreteFactory** (через абстрактний тип **AbstractFactory**), викликає на ній методи створення

продуктів, отримує продукти через абстрактні типи і працює з ними, не знаючи конкретних класів.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» визначає інтерфейс для створення об'єкта, але дозволяє підкласам вирішувати, екземпляр якого саме класу створювати. Він делегує створення об'єкта підкласам.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

Product: Інтерфейс або абстрактний клас для об'єктів, що створюються.

ConcreteProduct: Класи, що реалізують інтерфейс Product.

Creator: Абстрактний клас, що оголошує фабричний метод factoryMethod(): Product. Може містити й іншу логіку, що працює з Product.

ConcreteCreator: Клас, що успадковує Creator і перевизначає factoryMethod(), повертаючи екземпляр ConcreteProduct.

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

Фабричний метод визначає інтерфейс для створення одного об'єкта, але дозволяє підкласам вирішувати, екземпляр якого конкретного класу створювати. Це як "віртуальний конструктор".

Абстрактна фабрика використовується для створення сімейств пов'язаних об'єктів без вказівки їх конкретних класів. Вона гарантує, що всі створені нею об'єкти будуть сумісні між собою (належатимуть до одного стилю або групи). Часто Абстрактна фабрика може використовувати кілька Фабричних методів для створення об'єктів свого сімейства.

Просто:

- Factory Method = "Створи мені кнопку" (одна річ)
- Abstract Factory = "Створи мені весь набір UI в стилі Windows" (багато речей одного стилю)

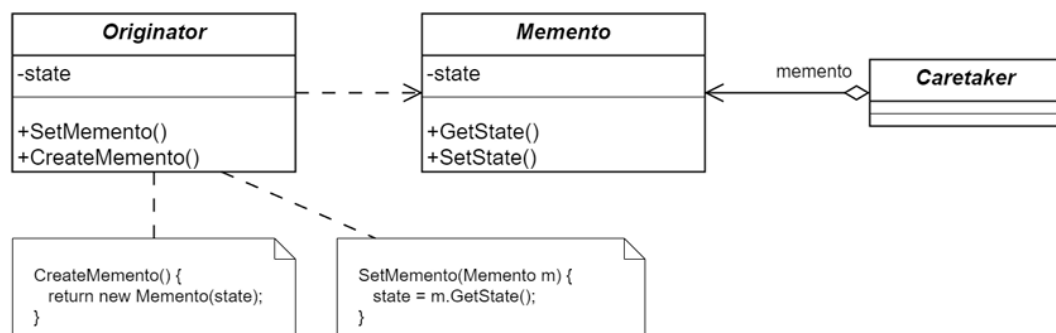
8. Яке призначення шаблону «Знімок»?

Шаблон «Знімок» (Memento) використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. Це дозволяє, наприклад, реалізувати механізм скасування дій (Undo).

Приклад:

Текстовий редактор - збереження тексту перед кожною зміною, щоб можна було натиснути Ctrl+Z.

9. Нарисуйте структуру шаблону «Знімок».



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

Originator (Творець/Оригінатор): Об'єкт, стан якого потрібно зберегти. Він вміє створювати Memento зі своїм поточним станом і відновлювати свій стан з Memento.

Memento (Знімок): Об'єкт, що зберігає внутрішній стан Originator. Важливо, що він надає повний доступ до свого стану тільки Originator-у, а для інших об'єктів (наприклад, Caretaker) він може мати обмежений або порожній інтерфейс.

Caretaker (Охоронець): Відповідає за зберігання об'єктів Memento. Він запитує знімок у Originator-а, зберігає його, але не знає і не може змінити його внутрішній вміст. Коли потрібно, він повертає Memento назад Originator-у для відновлення стану.

11. Яке призначення шаблону «Декоратор»?

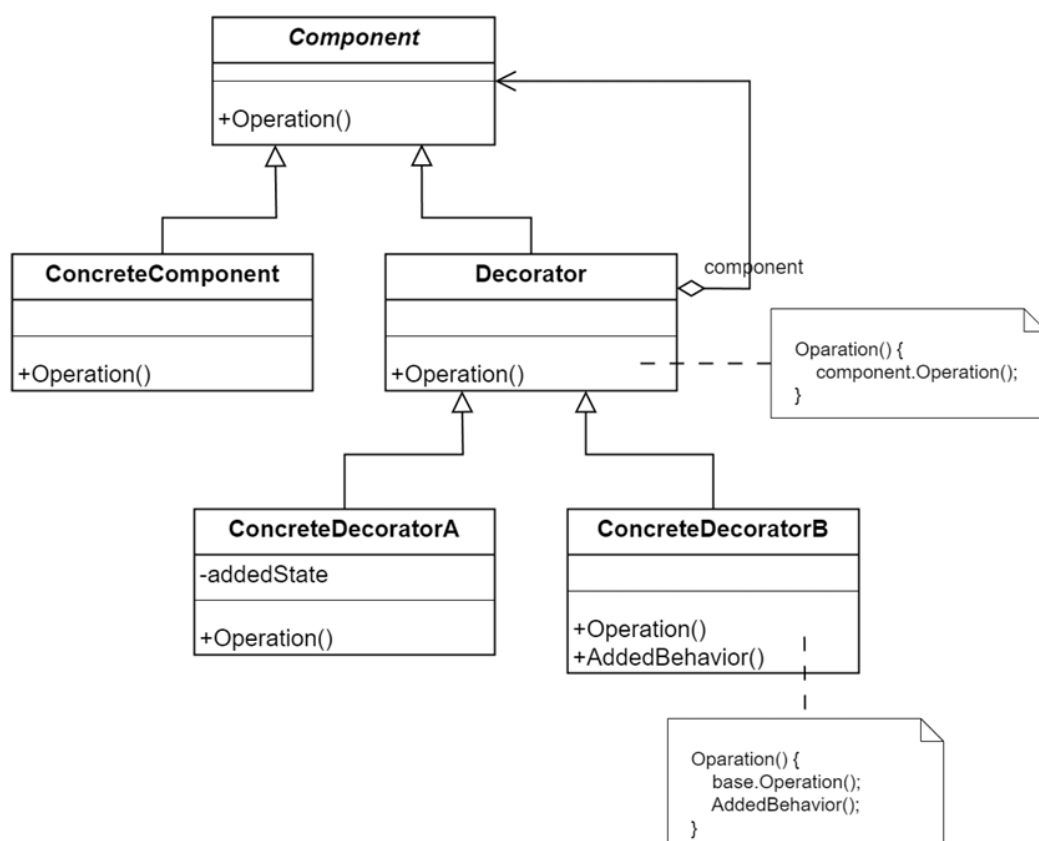
Шаблон «Декоратор» (Decorator) призначений для динамічного додавання нових обов'язків або функціональних можливостей об'єкту під час роботи програми, не змінюючи його вихідний код. Це гнучка альтернатива успадкуванню для розширення функціональності.

Приклад: Є каву. Можна додати молоко → каву з молоком. Потім додати цукор → каву з молоком і цукром. Кожна добавка "обгортає" попередній об'єкт, додаючи функціонал.

Без Decorator: Треба класи Coffee, CoffeeWithMilk, CoffeeWithSugar, CoffeeWithMilkAndSugar, CoffeeWithMilkAndSugarAndCream... (комбінаторний вибух!)

З Decorator: Coffee → обгортаємо в MilkDecorator → обгортаємо в SugarDecorator (гнучкі комбінації!)

12. Нарисуйте структуру шаблону «Декоратор».



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

Component (Компонент): Інтерфейс або абстрактний клас, що визначає загальний інтерфейс для об'єктів, які можна декорувати.

ConcreteComponent (Конкретний компонент): Базовий клас з початковою функціональністю, яку потрібно розширити.

Decorator (Декоратор): Абстрактний клас, що реалізує інтерфейс Component і містить посилання (агрегацію) на інший об'єкт типу Component (той, що обгортається). Він зазвичай просто делегує всі виклики обгорнутому об'єкту.

ConcreteDecorator (Конкретний декоратор): Класи, що успадковують Decorator. Вони додають новий функціонал до або після виклику методу обгорнутого об'єкта.

14. Які є обмеження використання шаблону «декоратор»?

Може призвести до створення великої кількості крихтих класів-обгорт, що ускладнює розуміння системи.

Важко конфігурувати об'єкти, які загорнуто в декілька обгорт одночасно.

Також буває складно видалити конкретну обгортку зі стеку декораторів.

Висновки

В ході виконання даної лабораторної роботи було детально вивчено та успішно застосовано на практиці породжуючий патерн проєктування «Factory Method».

Було реалізовано ієрархію класів-творців (ArchiveProcessor, ZipArchiveProcessor, RarArchiveProcessor), яка дозволяє гнучко створювати об'єкти-стратегії (IArchiverStrategy) для роботи з різними форматами архівів. Абстрактний клас ArchiveProcessor містить загальну логіку обробки архівів і делегує створення конкретної стратегії своїм підкласам через абстрактний фабричний метод createStrategy(). Кожен підклас (ZipArchiveProcessor, RarArchiveProcessor) реалізує цей метод, повертаючи відповідну стратегію (ZipStrategy, RarStrategy).