

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»

Тема: «Патерни проектування.»

Виконав:  
студент групи ІА-32  
Діденко Я.О

Перевірив:  
Мягкий М.Ю

**Тема:** Патерни проектування.

**Мета:** Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» 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.

<b>Теоретичні Відомості.....</b>	<b>2</b>
Поняття шаблону проектування.....	2
Шаблон «Factory Method».....	3
<b>Хід роботи.....</b>	<b>4</b>
1. Загальний опис виконаної роботи.....	4
2. Опис класів програмної системи.....	5
Клас FileSystemItem.java.....	5
Класи FileItem.java та DirectoryItem.java.....	6
Клас FileSystemItemCreator.java.....	8
Класи FileCreator.java та DirectoryCreator.java.....	10
Клас Main.java.....	11
Опис результатів коду.....	12
3. Діаграма класів.....	16
4. Висновки.....	17
<b>Контрольні запитання.....</b>	<b>18</b>

# Теоретичні Відомості

## Поняття та структура шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator»

Шаблони проектування є універсальними рішеннями, що описують типові способи побудови об'єктно-орієнтованих систем. Вони підвищують гнучкість, повторне використання коду й розширюваність архітектури. Нижче розглянуто ключові структурні, породжувальні та поведінкові патерни, які широко застосовуються у проектуванні програмних систем.

**Abstract Factory (Абстрактна фабрика)** - породжувальний шаблон, який надає інтерфейс для створення сімейств пов'язаних об'єктів без зазначення їх конкретних класів. Це дозволяє програмі залишатися незалежною від способу створення та представлення об'єктів. Застосовується, коли потрібно легко змінювати групи взаємопов'язаних компонентів (наприклад, інтерфейси різних тем оформлення).

**Factory Method (Фабричний метод)** - породжувальний шаблон, що визначає інтерфейс для створення об'єкта, але дозволяє підкласам вирішувати, який саме клас створювати. Основна ідея - делегувати процес створення об'єктів підкласам, зберігаючи спільний алгоритм роботи. Часто використовується для розширення функціональності без зміни базового коду.

**Memento (Сувенір, або Зліпок стану)** - поведінковий шаблон, який дозволяє зберігати й відновлювати попередній стан об'єкта без порушення інкапсуляції. Він складається з трьох частин: «Originator» (об'єкт, чий стан зберігається), «Memento» (зліпок стану) і «Caretaker» (керує збереженням). Використовується, наприклад, у функціях «Скасувати дію» або «Undo».

**Observer (Спостерігач)** - поведінковий шаблон, що створює механізм підписки: один об'єкт (видавець) повідомляє інші (спостерігачів) про зміну свого стану. Це забезпечує слабе зв'язування між компонентами. Патерн часто використовується у подієвих системах, графічних інтерфейсах, потокових оновленнях.

**Decorator (Декоратор)** - структурний шаблон, який дозволяє динамічно додавати нові функції об'єкту без зміни його класу. Об'єкт «обгортається» у спеціальний клас-декоратор, який розширює його поведінку. Такий підхід є гнучкішою альтернативою успадкуванню, коли потрібно додати 2 кілька варіантів розширення функціоналу.

## Шаблон «Factory Method»

**Призначення:** Шаблон «Фабричний метод» визначає інтерфейс для створення об'єктів певного базового типу. 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».

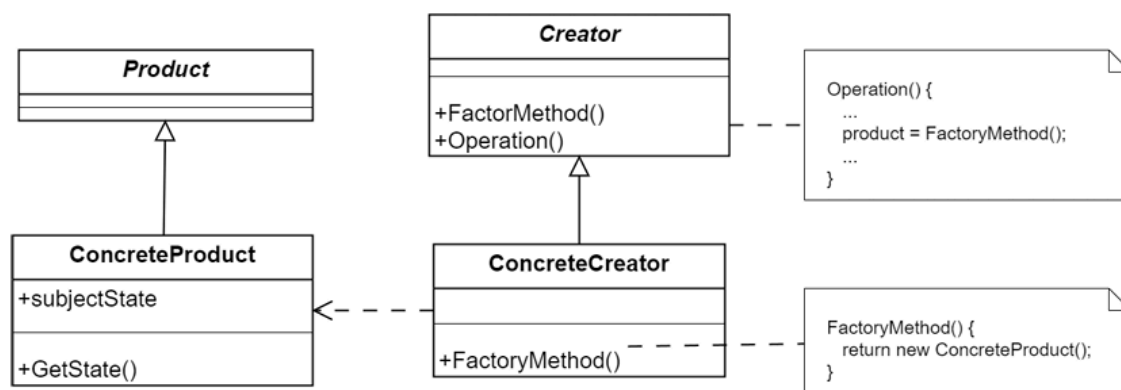


рис 1. Структура патерну Factory Method

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:

- + 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.

### Недолік:

- Може призвести до створення великих паралельних ієрархій класів.

## Хід роботи

### 1. Загальний опис виконаної роботи

В рамках даної лабораторної роботи було реалізовано шаблон проектування «**Фабричний метод**» (**Factory Method**).

Основна мета полягала в тому, щоб визначити єдиний інтерфейс для створення об'єктів, дозволяючи дочірнім класам (фабрикам) самостійно вирішувати, екземпляри яких саме класів створювати. Такий підхід дозволяє делегувати логіку створення об'єктів, відокремлюючи клієнтський код від конкретних класів продуктів, що робить систему більш гнучкою та розширюваною.

Програмна система складається з наступних ключових компонентів:

- **Абстрактний продукт (FileSystemItem):** Базовий абстрактний клас, що визначає загальний інтерфейс для всіх об'єктів файлової системи (продуктів), які будуть створюватися фабриками. Він містить методи, такі як `getInfo()`, `delete()` та `move()`.
- **Конкретні продукти (FileItem, DirectoryItem):** Класи, що розширюють `FileSystemItem` і надають конкретну реалізацію для файлів та папок. Важливою частиною їх реалізації є валідація у конструкторі, яка перевіряє, чи відповідає наданий шлях типу об'єкта, що гарантує створення лише коректних екземплярів.
- **Абстрактний творець (FileSystemItemCreator):** Клас, що оголошує «фабричний метод» `createFileSystemItem()`, який повертає абстрактний продукт `FileSystemItem`. Він також містить загальну бізнес-логіку (`performOperation`), яка працює з продуктами, не знаючи їх конкретного типу, що демонструє ключову перевагу шаблону.
- **Конкретні творці (FileCreator, DirectoryCreator):** Класи, що успадковують `FileSystemItemCreator` та перевизначають фабричний метод. Кожен конкретний творець відповідає за створення об'єкта

певного типу: FileCreator створює FileItem, а DirectoryCreator — DirectoryItem.

- **Клієнт (Main):** Кінцевий клієнт, який використовує систему. Він ініціює створення об'єктів, звертаючись до конкретної фабрики (FileCreator або DirectoryCreator), але робить це для запуску загальної логіки, визначеної в абстрактному творці.

Процес створення об'єктів повністю інкапсульований у конкретних творцях. Клієнтський код не викликає конструктори new FileItem() чи new DirectoryItem() напряду. Замість цього, він просить відповідну фабрику створити об'єкт, що робить систему гнучкою, легко розширюваною та відповідною принципу відкритості/закритості.

## 2. Опис класів програмної системи.

### Клас FileSystemItem.java

#### Опис:

Клас FileSystemItem є **абстрактним продуктом** у структурі шаблону "Фабричний метод". Він служить базовим класом для всіх об'єктів файлової системи та визначає їх загальний інтерфейс і поведінку.

#### Ключові характеристики:

- Його неможливо інстанціювати напряду. Він лише визначає контракт для своїх нащадків.
- Містить поля name та path, які є спільними для будь-якого елемента файлової системи.
- Оголошує методи getInfo(), delete() та move(), які повинні бути реалізовані в кожному конкретному класі-нащадку (FileItem, DirectoryItem), оскільки логіка цих операцій залежить від типу об'єкта.

Цей клас дозволяє клієнтському коду працювати з різними об'єктами файлової системи (файлами, папками) через єдиний, уніфікований інтерфейс.

```

1 import java.io.File;
2
3 @ public abstract class FileSystemItem { 6 usages 2 inheritors
4     protected String name;
5     protected String path; 6 usages
6
7     public FileSystemItem(String path) { 2 usages
8         if (path == null || path.trim().isEmpty()) {
9             throw new IllegalArgumentException("Path cannot be null or empty.");
10        }
11        this.path = path;
12        this.name = new File(path).getName();
13    }
14
15    public String getName() { 2 usages
16        return name;
17    }
18
19    public String getPath() { no usages
20        return path;
21    }
22
23 @ public abstract void getInfo(); 1 usage 2 implementations
24
25 @ public abstract void delete(); 1 usage 2 implementations
26
27 @ public abstract void move(String newPath); 1 usage 2 implementations
28 }

```

рис 2.1 - код класу FileSystemItem.java

## Класи FileItem.java та DirectoryItem.java

Класи FileItem та DirectoryItem є **конкретними продуктами**, що реалізують абстрактний клас FileSystemItem. Вони надають специфічну імплементацію для файлів та папок відповідно.

### Ключові характеристики:

- Обидва класи розширюють FileSystemItem та реалізують його абстрактні методи (getInfo, delete, move), додаючи логіку, унікальну для файлів та папок. Наприклад, FileItem додатково зберігає розмір файлу (size).
- Конструктор кожного класу перевіряє, чи шлях, що передається, дійсно вказує на об'єкт відповідного типу (файл або папка) і чи цей

об'єкт існує. Якщо перевірка не проходить, викидається виняток `IllegalArgumentException`, що запобігає створенню некоректних об'єктів.

- Кожен клас інкапсулює логіку, специфічну для свого типу.  
Наприклад, метод `getInfo()` у `FileItem` виводить розмір файлу, тоді як у `DirectoryItem` - ні.

Ці класи є кінцевими об'єктами, які створюються фабриками та з якими безпосередньо працює бізнес-логіка програми.

```
1  import java.io.File;
2
3  public class FileItem extends FileSystemItem { 1 usage
4      private long size; 2 usages
5
6      public FileItem(String path) { 1 usage
7          super(path);
8          File file = new File(path);
9
10         if (!file.exists() || !file.isFile()) {
11             throw new IllegalArgumentException("Path does not point to a valid file: " + path);
12         }
13
14         this.size = file.length();
15     }
16
17     @Override 1 usage
18     public void getInfo() {
19         System.out.println("File: " + name + ", Size: " + size + " bytes, Path: " + path);
20     }
21
22     @Override 1 usage
23     public void delete() {
24         System.out.println("-> Deleting file: " + name);
25     }
26
27     @Override 1 usage
28     public void move(String newPath) {
29         System.out.println("-> Moving file from '" + path + "' to '" + newPath + "'");
30     }
31 }
```

рис 2.2.1 - код класу `FileItem.java`



```

1  import java.io.File;
2
3  public class DirectoryItem extends FileSystemItem { 1 usage
4
5      public DirectoryItem(String path) { 1 usage
6          super(path);
7          File file = new File(path);
8
9          if (!file.exists() || !file.isDirectory()) {
10             throw new IllegalArgumentException("Path does not point to a valid directory: " + path);
11          }
12      }
13
14      @Override 1 usage
15      public void getInfo() {
16          System.out.println("Directory: " + name + ", Path: " + path);
17      }
18
19      @Override 1 usage
20      public void delete() {
21          System.out.println("-> Deleting directory: " + name);
22      }
23
24      @Override 1 usage
25      public void move(String newPath) {
26          System.out.println("-> Moving directory from '" + path + "' to '" + newPath + "'");
27      }
28  }

```

рис 2.2.2 - код класу DirectoryItem.java

## Клас FileSystemItemCreator.java

### Опис:

Клас FileSystemItemCreator є **абстрактним творцем** (Creator) і центральним елементом шаблону "Фабричний метод" у даній системі. Він визначає стандартний процес роботи з об'єктами файлової системи, делегуючи створення цих об'єктів своїм нащадкам.

### Ключові характеристики:

- **Абстрактний Фабричний Метод:** Клас оголошує абстрактний метод `createFileSystemItem(String path)`. Це і є "**фабричний метод**". Він змушує всі дочірні класи реалізувати власну логіку створення об'єктів.
- Містить метод `performOperation(String path)`, який реалізує загальний алгоритм для роботи з продуктами: він викликає фабричний метод для створення об'єкта, а потім виконує над ним низку операцій (`getInfo`, `move`, `delete`). Важливо, що цей метод працює з абстрактним

типом `FileSystemItem`, не знаючи, чи це файл, чи папка. Це демонструє головну перевагу шаблону - відокремлення загальної логіки від конкретних класів.

- Метод `performOperation` обгортає створення об'єкта в блок `try-catch`, що дозволяє коректно обробляти винятки `IllegalArgumentException`, які можуть виникнути, якщо конкретний продукт не може бути створений (наприклад, файл не існує).

Цей клас встановлює "каркас" для створення та обробки об'єктів, залишаючи деталі реалізації створення дочірнім класам.

```
1  import java.io.File;
2
3  public abstract class FileSystemItemCreator { 4 usages 2 inheritors
4
5      public abstract FileSystemItem createFileSystemItem(String path); 1 usage 2 implementations
6
7      public void performOperation(String path) { 3 usages
8          try {
9              FileSystemItem item = createFileSystemItem(path);
10
11              System.out.println("--- Performing operations on: " + item.getName() + " ---");
12              item.getInfo();
13              String parentDirectory = new File(path).getParent();
14              if (parentDirectory == null) {
15                  parentDirectory = "";
16              }
17              String newPath = parentDirectory + File.separator + item.getName() + "_moved";
18              item.move(newPath);
19              item.delete();
20              System.out.println("-----\n");
21          } catch (IllegalArgumentException e) {
22              System.out.println("Operation failed: " + e.getMessage() + "\n");
23          }
24      }
25  }
```

рис 2.3 - код класу `FileSystemItemCreator.java`

## Класи FileCreator.java та DirectoryCreator.java

Класи FileCreator та DirectoryCreator є **конкретними творцями** (Concrete Creators). Вони успадковують FileSystemItemCreator та надають конкретну реалізацію його абстрактного фабричного методу.

### Ключові характеристики:

- Кожен з цих класів перевизначає метод createFileSystemItem(String path). Це єдина відповідальність цих класів.
  - FileCreator всередині цього методу створює та повертає екземпляр класу FileItem.
  - DirectoryCreator створює та повертає екземпляр класу DirectoryItem.
- Вони інкапсулюють знання про те, як і який конкретний продукт потрібно створити. Клієнтський код не знає про конструктори new FileItem() чи new DirectoryItem(), він лише обирає, яку фабрику використати.
- Вони повністю слідують контракту, визначеному в абстрактному класі FileSystemItemCreator, не змінюючи іншої логіки (наприклад, performOperation).

Саме ці класи дозволяють системі бути гнучкою. Для додавання нового типу об'єкта (наприклад, SymbolicLinkItem) достатньо було б створити новий клас SymbolicLinkCreator, не змінюючи існуючий код.

```
1      public class DirectoryCreator extends FileSystemItemCreator { 1 usage
2
3          @Override 1 usage
4      public FileSystemItem createFileSystemItem(String path) {
5          return new DirectoryItem(path);
6      }
7      }
```

рис 2.4.1 - код класу DirectoryCreator.java

```

1      public class FileCreator extends FileSystemItemCreator { 1 usage
2
3          @Override 1 usage
4      public FileSystemItem createFileSystemItem(String path) {
5          return new FileItem(path);
6      }
7  }

```

рис 2.4.2 - код класу FileCreator.java

## Клас Main.java

Клас Main виступає в ролі **Клієнта** (Client) у даній системі. Він є точкою входу в програму (public static void main) і відповідає за демонстрацію роботи шаблону "Фабричний метод".

### Ключові характеристики:

- **Відокремлення від конкретних продуктів:** Головна особливість клієнтського коду полягає в тому, що він **не створює об'єкти (FileItem, DirectoryItem) напряму** через їх конструктори.
- **Використання творців:** Замість цього, він створює екземпляри конкретних творців (FileCreator та DirectoryCreator) і делегує їм завдання створення та обробки об'єктів через виклик методу performOperation.
- **Демонстрація гнучкості:** Клієнт вирішує, яку саме фабрику використовувати залежно від задачі (робота з файлом чи з папкою). Це показує, як легко можна керувати процесом створення, не змінюючи основну логіку.
- **Тестові сценарії:** У даній реалізації клас Main також готує тестові дані — шляхи до існуючих та неіснуючих файлів/папок — для повної ілюстрації роботи системи, включаючи як успішне виконання, так і коректну обробку помилок.

Таким чином, Main демонструє, як клієнтський код може залишатися чистим і незалежним від складної логіки створення об'єктів.

```

1 > public class Main {
2 >     public static void main(String[] args) {
3         System.out.println("Starting Shell Application simulation...\n");
4
5         String filePath = "C:/temp/my_document.txt";
6         String dirPath = "C:/temp/my_folder";
7         String nonExistentPath = "C:/temp/non_existent_file.txt";
8
9         FileSystemItemCreator fileCreator = new FileCreator();
10        FileSystemItemCreator directoryCreator = new DirectoryCreator();
11
12        System.out.println("1. Processing a valid file...");
13        fileCreator.performOperation(filePath);
14
15        System.out.println("2. Processing a valid directory...");
16        directoryCreator.performOperation(dirPath);
17
18        System.out.println("3. Processing a non-existent file...");
19        fileCreator.performOperation(nonExistentPath);
20
21        System.out.println("\nShell Application simulation finished.");
22    }
23 }

```

**рис 2.5 - код класу Main.java**

## Опис результатів коду

Для демонстрації функціональності програми було проведено серію тестів, що ілюструють її поведінку в різних умовах.

Спочатку програма була запущена в умовах, коли цільовий каталог C:/temp та його вміст ще не існували. Метою цього тесту була перевірка системи обробки помилок. Результат виконання підтвердив коректність реалізованої логіки валідації: для кожної операції було виведено повідомлення Operation failed, оскільки фабрики не змогли створити об'єкти для неіснуючих шляхів.

Starting Shell Application simulation...

1. Processing a valid file...

Operation failed: Path does not point to a valid file: C:/temp/my\_document.txt

2. Processing a valid directory...

Operation failed: Path does not point to a valid directory: C:/temp/my\_folder

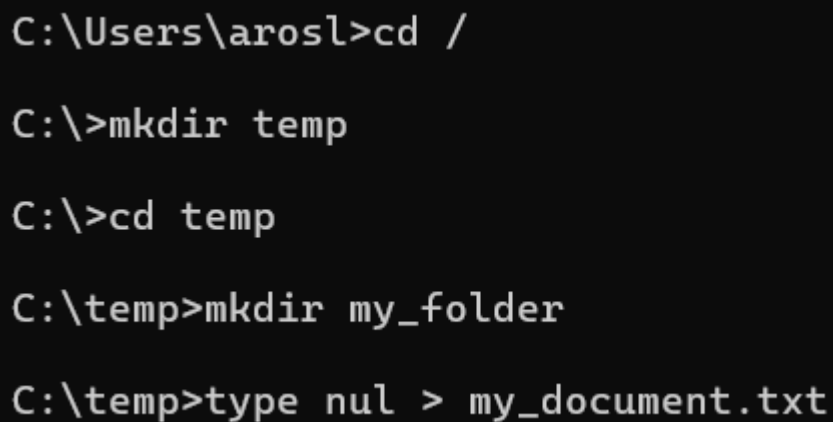
3. Processing a non-existent file...

Operation failed: Path does not point to a valid file: C:/temp/non\_existent\_file.txt

Shell Application simulation finished.

### рис 2.6.1 - початковий результат

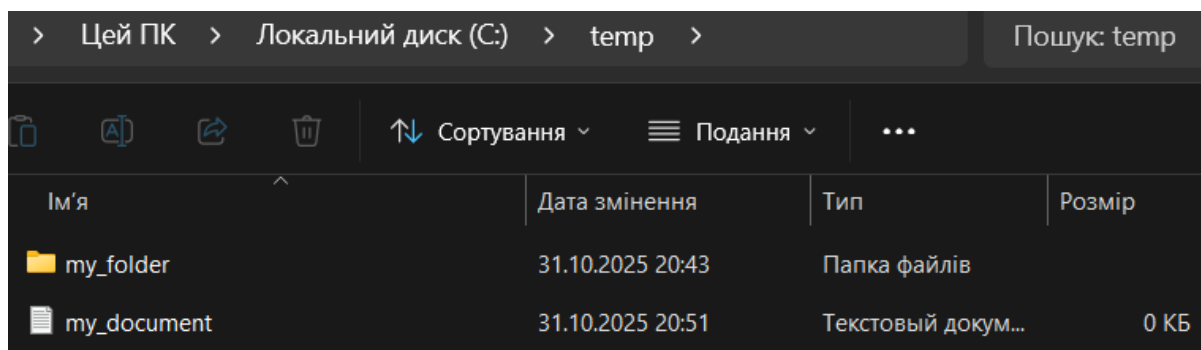
Після початкового тесту було підготовлено необхідне робоче середовище за допомогою командного рядка. Було виконано низку команд для створення каталогу C:/temp, а всередині нього - підкаталогу my\_folder та порожнього текстового файлу my\_document.txt.



```
C:\Users\arosl>cd /  
  
C:\>mkdir temp  
  
C:\>cd temp  
  
C:\temp>mkdir my_folder  
  
C:\temp>type nul > my_document.txt
```

### рис 2.6.2 - створення каталогів та файлу

Для візуального підтвердження успішності етапу підготовки вміст каталогу C:/temp було перевірено через Провідник Windows. Це підтвердило наявність як my\_folder, так і my\_document.txt. Властивості файлу вказували на розмір 0 КБ, що відповідало його порожньому стану.



**рис 2.6.3 - перевірка результатів створення**

З готовою структурою файлів програму було запущено вдруге. Цей запуск продемонстрував здатність системи коректно обробляти валідні шляхи. Вивід показав, що операції над існуючими файлом та каталогом пройшли успішно, і програма правильно визначила розмір my\_document.txt як 0 байт.

```
Starting Shell Application simulation...

1. Processing a valid file...
--- Performing operations on: my_document.txt ---
File: my_document.txt, Size: 0 bytes, Path: C:/temp/my_document.txt
-> Moving file from 'C:/temp/my_document.txt' to 'C:\temp\my_document.txt_moved'
-> Deleting file: my_document.txt
-----

2. Processing a valid directory...
--- Performing operations on: my_folder ---
Directory: my_folder, Path: C:/temp/my_folder
-> Moving directory from 'C:/temp/my_folder' to 'C:\temp\my_folder_moved'
-> Deleting directory: my_folder
-----

3. Processing a non-existent file...
Operation failed: Path does not point to a valid file: C:/temp/non_existent_file.txt

Shell Application simulation finished.
```

**рис 2.6.4 - проміжний результат**

Для подальшої перевірки взаємодії системи з файловою системою, файл my\_document.txt було змінено. За допомогою команди echo до нього було додано рядок тексту, що перезаписало порожній файл новим вмістом.

```
C:\temp>echo "YVL" > C:\temp\my_document.txt  
C:\temp>
```

**рис 2.6.5 - додавання тексту у текстовий документ**

Під час фінального тесту програму було запущено востаннє. Тепер вивід програми показав ненульовий розмір для my\_document.txt, точно відображаючи зміни, внесені на попередньому кроці. Цей результат успішно демонструє, що система коректно взаємодіє з файловою системою в реальному часі, зчитуючи актуальні атрибути файлів.

```
Starting Shell Application simulation...
```

```
1. Processing a valid file...
```

```
--- Performing operations on: my_document.txt ---
```

```
File: my_document.txt, Size: 8 bytes, Path: C:/temp/my_document.txt
```

```
-> Moving file from 'C:/temp/my_document.txt' to 'C:\temp\my_document.txt_moved'
```

```
-> Deleting file: my_document.txt
```

```
-----
```

```
2. Processing a valid directory...
```

```
--- Performing operations on: my_folder ---
```

```
Directory: my_folder, Path: C:/temp/my_folder
```

```
-> Moving directory from 'C:/temp/my_folder' to 'C:\temp\my_folder_moved'
```

```
-> Deleting directory: my_folder
```

```
-----
```

```
3. Processing a non-existent file...
```

```
Operation failed: Path does not point to a valid file: C:/temp/non_existent_file.txt
```

```
Shell Application simulation finished.
```

**рис 2.6.6 - фінальний результат**



### 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».

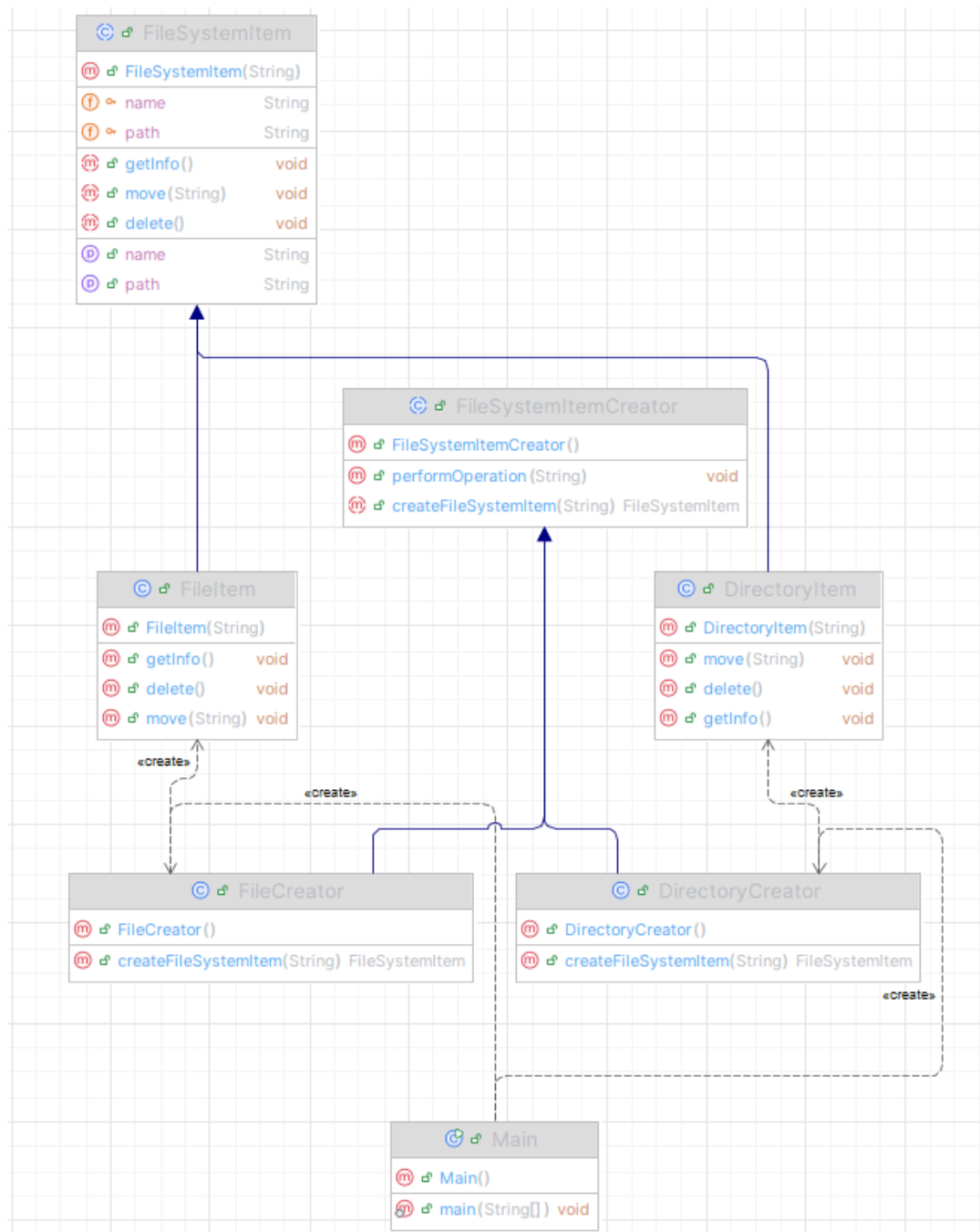


рис 3.1 - діаграма класів

Діаграма складається з трьох основних логічних блоків: ієрархії продуктів, ієрархії творців та клієнтського коду.

### 1. Ієрархія Продуктів (Product Hierarchy):

- `FileSystemItem` є **абстрактним продуктом**. Він визначає загальний інтерфейс для всіх об'єктів файлової системи.
- `FileItem` та `DirectoryItem` є **конкретними продуктами**, що успадковують `FileSystemItem`. Зв'язок успадкування показаний **суцільною лінією з трикутною стрілкою**.

### 2. Ієрархія Творців (Creator Hierarchy):

- `FileSystemItemCreator` виступає як **абстрактний творець**. Він оголошує фабричний метод `createFileSystemItem()` і містить загальну логіку `performOperation`, яка залежить від абстрактного продукту.
- `FileCreator` та `DirectoryCreator` є **конкретними творцями**. Вони реалізують фабричний метод для створення відповідних конкретних продуктів.

### 3. Ключові Залежності:

- **Залежність створення («create»):** Пунктирні лінії з цією міткою є найважливішими на діаграмі. Вони візуалізують суть шаблону: `FileCreator` створює `FileItem`, а `DirectoryCreator` створює `DirectoryItem`. Це показує, що відповідальність за створення конкретного об'єкта делегована конкретній фабриці.
- Зв'язок між `FileSystemItemCreator` та `FileSystemItem` показує, що логіка творця не залежить від конкретних реалізацій продукту, що забезпечує гнучкість системи.
- **Клієнт (Main):** Клас `Main` показаний як клієнт, що залежить від конкретних творців для ініціалізації процесу.

## 4. Висновки

**Висновки:** В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано патерн проектування «Фабричний метод» для

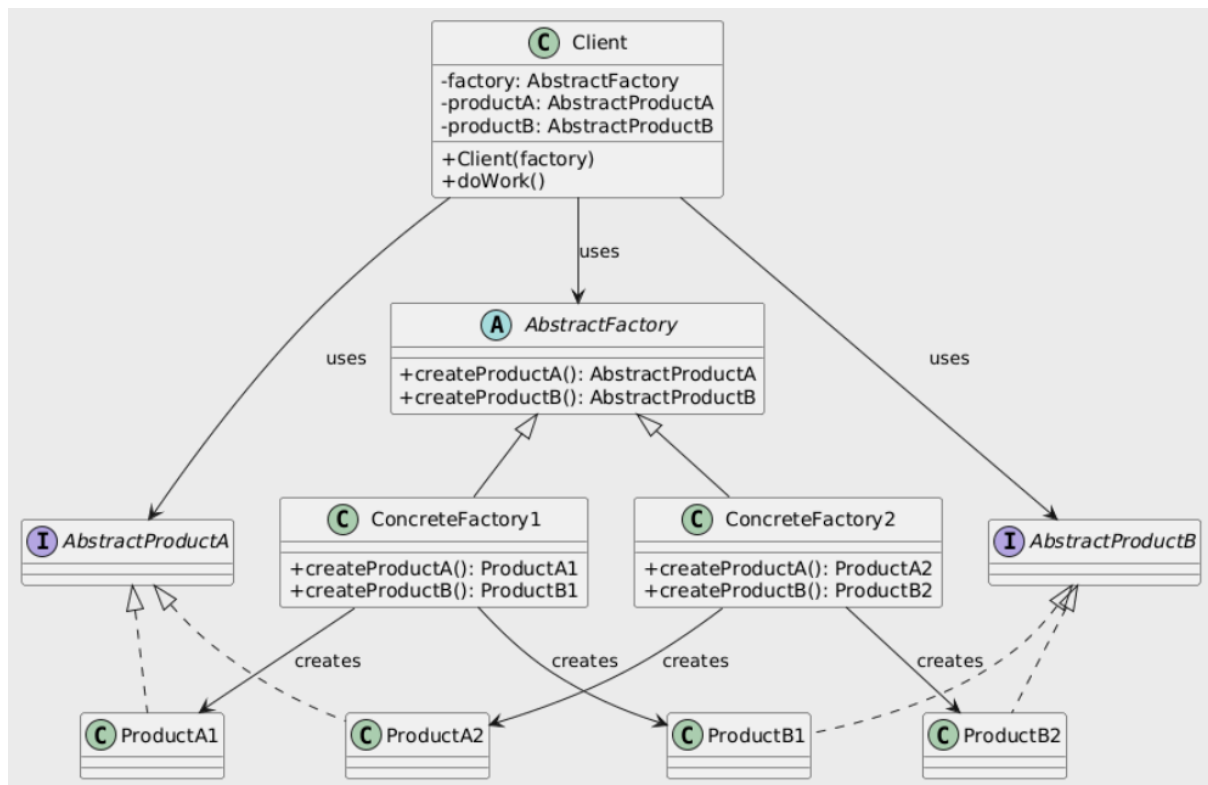
вирішення задачі гнучкого створення об'єктів у програмі-оболонці. Була створена розширювана архітектура, що складається з паралельних ієрархій класів: продуктів (FileItem, DirectoryItem) та творців (FileCreator, DirectoryCreator). Особливу увагу було приділено реалізації валідації в конструкторах конкретних продуктів. Впровадження перевірки на існування та відповідність типу об'єкта у файловій системі забезпечило надійність системи, гарантуючи, що фабрики створюють лише коректні та валідні екземпляри, та коректно обробляють помилки. Впровадження абстрактного класу FileSystemItemCreator у ролі «Творця» дозволило централізувати загальну логіку роботи з об'єктами в методі performOperation та повністю абстрагувати клієнтський код від процесів ініціалізації конкретних класів. Практична реалізація продемонструвала ключові переваги патерну: гнучкість системи, що дозволяє легко додавати нові типи об'єктів без зміни існуючого коду; слабке зв'язування між клієнтом та конкретними продуктами; та чітке дотримання принципу відкритості/закритості. Таким чином, мета лабораторної роботи була повністю досягнута, а ефективність та архітектурні переваги патерну «Фабричний метод» були підтверджені на практиці.

## Контрольні запитання

### 1. Яке призначення шаблону «Абстрактна фабрика»?

Призначення шаблону «Абстрактна фабрика» - надати інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів, не вказуючи їхні конкретні класи. Цей шаблон дозволяє клієнтському коду працювати з об'єктами через абстрактні інтерфейси, що робить систему незалежною від того, як саме ці об'єкти створюються та компонуються.

## 2. Нарисуйте структуру шаблону «Абстрактна фабрика».



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

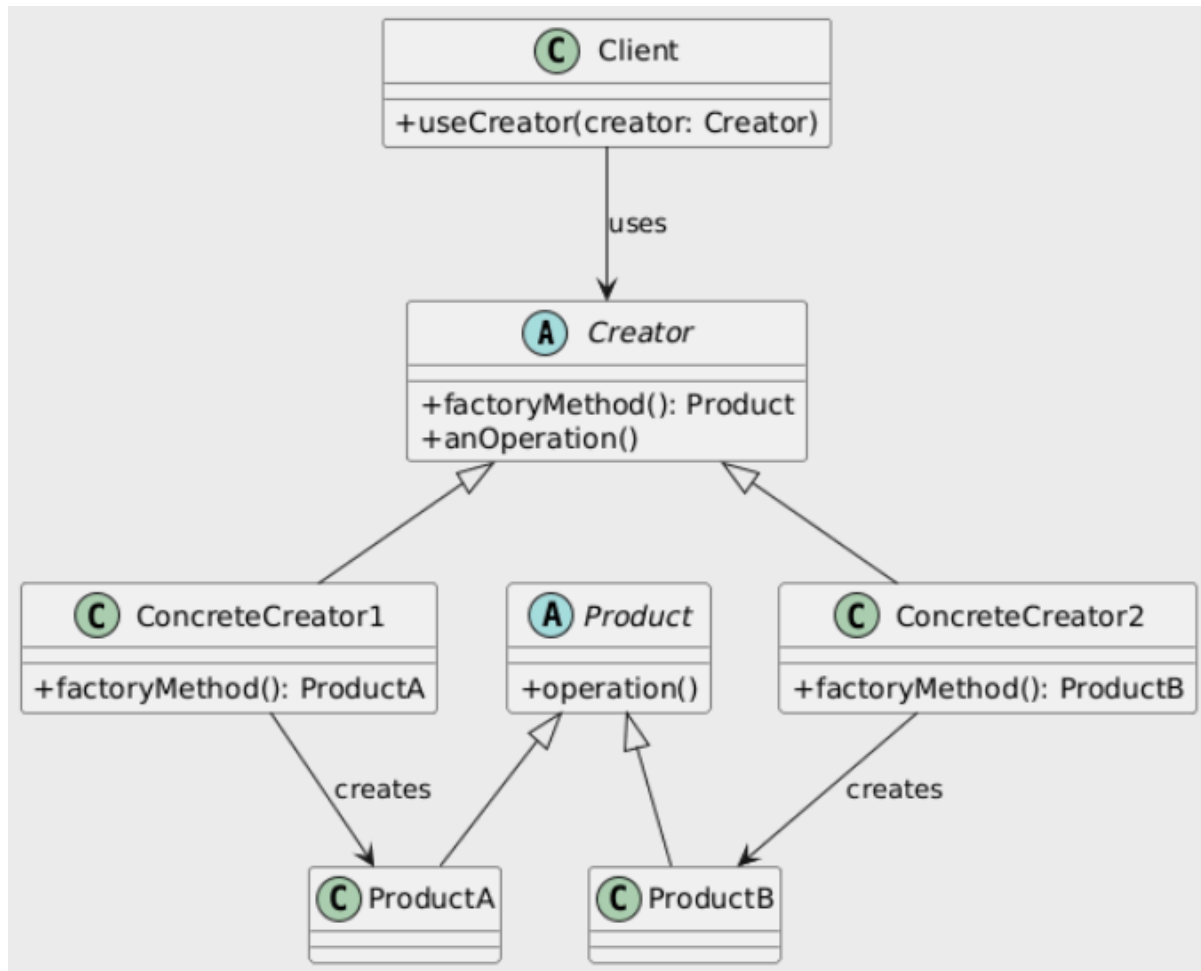
- **AbstractFactory (Абстрактна фабрика):** Оголошує інтерфейс для операцій, що створюють абстрактні продукти.
- **ConcreteFactory (Конкретна фабрика):** Реалізує операції для створення конкретних продуктів. Кожна конкретна фабрика відповідає за створення певного сімейства продуктів.
- **AbstractProduct (Абстрактний продукт):** Оголошує інтерфейс для типу об'єкта-продукту.
- **ConcreteProduct (Конкретний продукт):** Реалізує інтерфейс абстрактного продукту; це об'єкт, що створюється конкретною фабрикою.
- **Client (Клієнт):** Використовує тільки інтерфейси **AbstractFactory** та **AbstractProduct**.

**Взаємодія:** Клієнт створює екземпляр конкретної фабрики, а потім використовує її методи для створення об'єктів-продуктів. Клієнт не знає, які саме конкретні продукти створюються, оскільки працює з ними через абстрактні інтерфейси.

#### 4. Яке призначення шаблону «Фабричний метод»?

Призначення шаблону «Фабричний метод» - визначити інтерфейс для створення **одного об'єкта**, але дозволити дочірнім класам вирішувати, екземпляр якого саме класу створювати. Таким чином, шаблон делегує операцію створення екземпляра дочірнім класам.

#### 5. Нарисуйте структуру шаблону «Фабричний метод».



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

- **Product (Продукт):** Визначає інтерфейс об'єктів, які створює фабричний метод.
- **ConcreteProduct (Конкретний продукт):** Реалізує інтерфейс **Product**.
- **Creator (Творець):** Оголошує фабричний метод, що повертає об'єкт типу **Product**. Може містити загальну логіку, яка працює з продуктами.

- **ConcreteCreator (Конкретний творець):** Перевизначає фабричний метод для створення екземпляра ConcreteProduct.

**Взаємодія:** Клієнт працює з екземпляром ConcreteCreator через інтерфейс Creator і викликає його методи. Сам Creator використовує фабричний метод, реалізований у ConcreteCreator, для створення продукту.

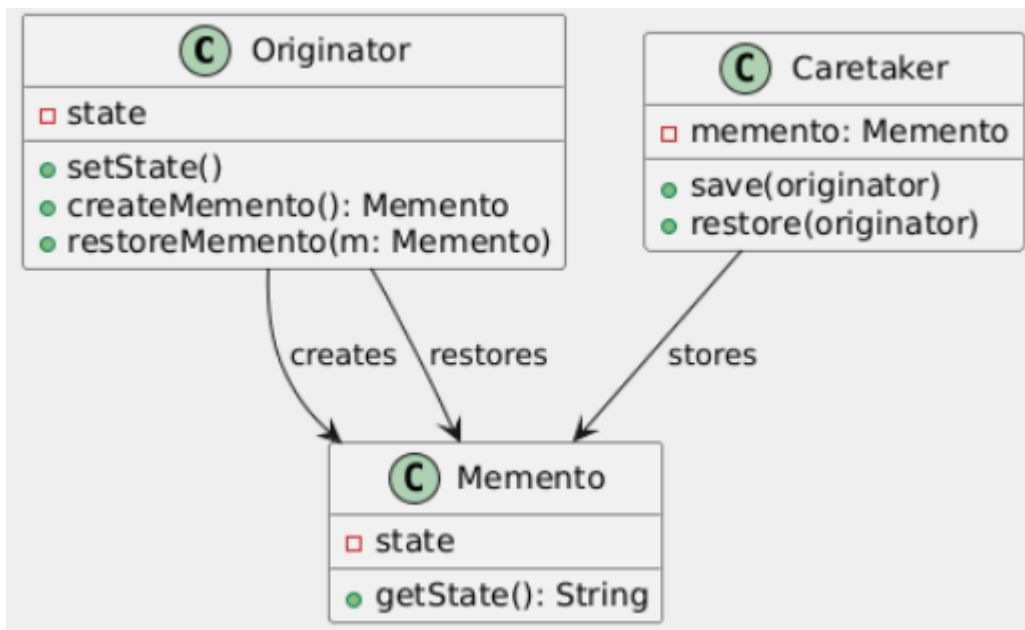
## **7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?**

- «Фабричний метод» призначений для створення **одного** об'єкта. «Абстрактна фабрика» - для створення **сімейства** пов'язаних об'єктів.
- «Фабричний метод» реалізується через успадкування (дочірній клас перевизначає один метод). «Абстрактна фабрика» зазвичай реалізується через композицію (клієнт використовує об'єкт фабрики, який передається як параметр або створюється на місці).
- «Фабричний метод» є простішим шаблоном. «Абстрактна фабрика» є більш складним і часто використовує декілька «фабричних методів» для створення продуктів свого сімейства.

## **8. Яке призначення шаблону «Знімок»?**

Призначення шаблону «Знімок» (Memento) - зафіксувати та винести за межі об'єкта його внутрішній стан так, щоб пізніше можна було відновити цей стан, не порушуючи інкапсуляцію.

## 9. Нарисуйте структуру шаблону «Знімок».



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

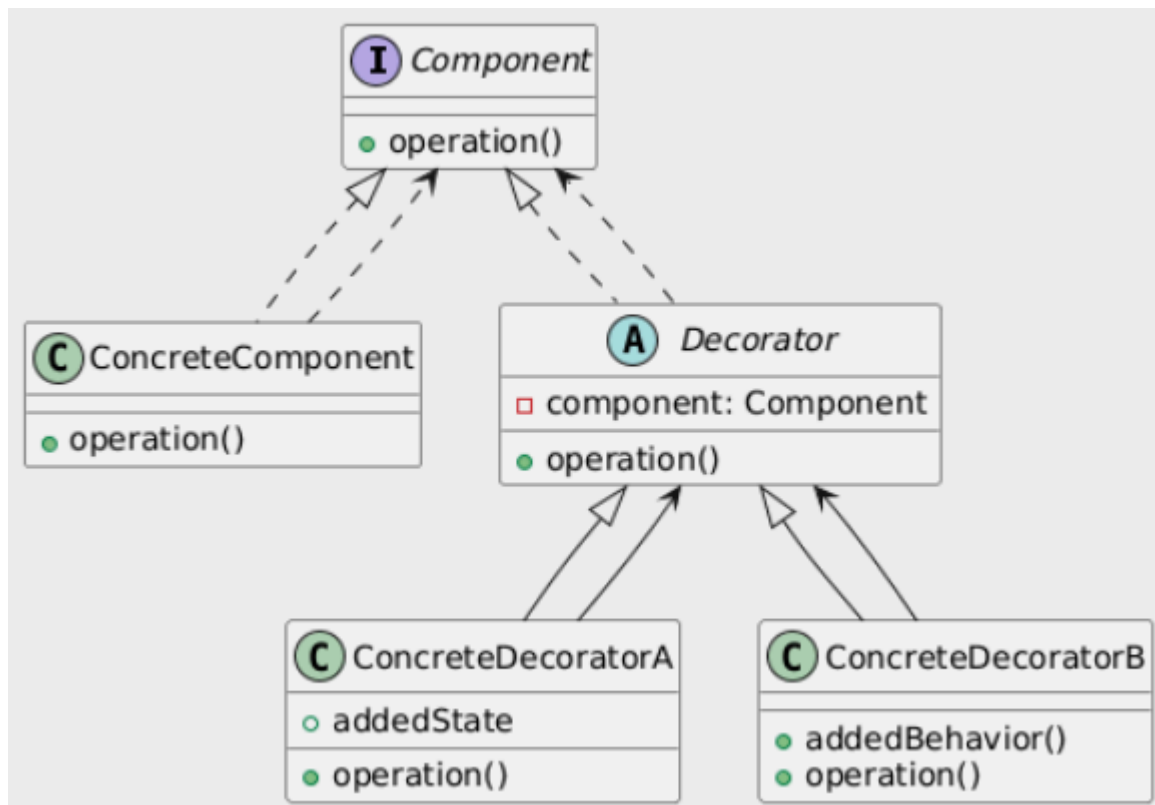
- **Originator (Творець):** Об'єкт, стан якого потрібно зберегти. Він створює знімок (Memento) і використовує його для відновлення свого стану.
- **Memento (Знімок):** Зберігає внутрішній стан Originator. Знімок має два інтерфейси: один для Caretaker (обмежений), інший для Originator (повний).
- **Caretaker (Опікун):** Зберігає об'єкт Memento, але не має доступу до його вмісту. Він відповідає за збереження та повернення знімка творцю.

**Взаємодія:** Caretaker просить Originator створити знімок і зберігає його. Коли потрібно відновити стан, Caretaker повертає знімок Originator, а той відновлює свій стан з даних знімка.

## 11. Яке призначення шаблону «Декоратор»?

Призначення шаблону «Декоратор» - динамічно додавати об'єкту нові обов'язки (функціональність). Декоратор є гнучкою альтернативою успадкуванню для розширення можливостей класів.

## 12. Нарисуйте структуру шаблону «Декоратор».



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

- **Component (Компонент):** Визначає інтерфейс для об'єктів, яким можна динамічно додавати нові обов'язки.
- **ConcreteComponent (Конкретний компонент):** Клас, функціональність якого розширюється.
- **Decorator (Декоратор):** Абстрактний клас, що зберігає посилання на об'єкт `Component` і реалізує його інтерфейс.
- **ConcreteDecorator (Конкретний декоратор):** Додає нові обов'язки до компонента.

**Взаємодія:** Об'єкт `ConcreteComponent` "обгортається" одним або кількома об'єктами `ConcreteDecorator`. Клієнт працює з усією структурою через інтерфейс `Component`, не розрізняючи, чи це вихідний об'єкт, чи декоратор.

## 14. Які є обмеження використання шаблону «декоратор»?

- Використання декоратора може призвести до появи великої кількості маленьких класів, що ускладнює розуміння системи.



- Код для створення об'єкта, обгорнутого в декілька декораторів, може бути складним і важким для налагодження.
- Декоратор і об'єкт, який він обгортає, не є одним і тим самим об'єктом (перевірка `a == b` поверне `false`), що може спричинити несподівану поведінку, якщо код покладається на ідентичність об'єктів.