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»

Виконала:
студентка групи ІА-32
Глущенко Анастасія
Сергіївна

Перевірив:
Мягкий Михайло
Юрійович

Київ 2025

Зміст

Вступ

Метою роботи є вивчення структури шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати «Factory Method» в реалізації програмної системи.

Було обрано тему:
Installer generator (iterator, builder, factory method, bridge, interpreter, client-server)
Генератор інсталяційних пакетів повинен мати якийсь спосіб налаштування файлів, що входять в установку, установки вікон з інтерактивними можливостями (галочка – створити ярлик на робочому столі; ввести в текстове поле деякі дані, наприклад, ліцензійний ключ і т.д.). Генератор повинен вивести один файл .exe або .msi

Теоретичні відомості

1. «Abstract Factory» (Абстрактна Фабрика)

Надає інтерфейс для створення сімейств пов'язаних або залежних об'єктів без зазначення їхніх конкретних класів.

Це корисно, коли:

- Система не повинна залежати від того, як створюються, компонуються та представляються продукти.
- Система має конфігуруватися одним із сімейств продуктів (наприклад, підтримка інтерфейсу для різних ОС: Windows, MacOS, Linux).
- Сімейство взаємопов'язаних об'єктів спроєктоване для спільного використання, і ви хочете забезпечити виконання цієї умови.

Структура та реалізація

- AbstractFactory (Абстрактна Фабрика): Оголошує інтерфейс для операцій створення абстрактних продуктів.
- ConcreteFactory (Конкретна Фабрика): Реалізує операції для створення конкретних об'єктів продуктів (для певної варіації, наприклад, фабрика для стилю MacOS).
- AbstractProduct (Абстрактний Продукт): Оголошує інтерфейс для типу об'єкта-продукту.
- ConcreteProduct (Конкретний Продукт): Визначає об'єкт, що створюється відповідною конкретною фабрикою, та реалізує інтерфейс AbstractProduct.
- Client (Клієнт): Використовує виключно інтерфейси, оголошені в класах AbstractFactory та AbstractProduct.

2. «Factory Method» (Фабричний Метод)

Визначає інтерфейс для створення об'єкта, але дозволяє підкласам змінювати тип створюваного об'єкта. Дозволяє класу делегувати створення екземплярів своїм підкласам.Основна мета: Вирішення проблеми сильної

зв'язності коду, коли клієнтський код прив'язаний до конкретних класів продуктів.Застосування:

- Коли класу заздалегідь невідомо, об'єкти яких саме класів йому потрібно створювати.
- Коли клас спроєктований так, щоб об'єкти, які він створює, специфікувалися підкласами.
- Для економії системних ресурсів шляхом повторного використання вже створених об'єктів замість породження нових (наприклад, пули з'єднань).

Структура та реалізація

- Product (Продукт): Визначає інтерфейс об'єктів, які створює фабричний метод.
- ConcreteProduct (Конкретний Продукт): Реалізує інтерфейс Product.
- Creator (Творець): Оголошує фабричний метод, який повертає об'єкт типу Product. Може також містити базову реалізацію цього методу.
- ConcreteCreator (Конкретний Творець): Перевизначає фабричний метод для створення екземпляра ConcreteProduct.

3. «Memento» (Знімок)

Дозволяє зберігати та відновлювати попередній стан об'єкта, не розкриваючи подробиць його реалізації (не порушуючи інкапсуляцію).Ключова ідея: Створення "знімка" стану об'єкта, який можна зберегти окремо і використати пізніше для відкату змін.Застосування:

- Реалізація функції "Скасувати" (Undo) в редакторах та інших програмах.
- Коли пряме отримання стану об'єкта розкриває деталі реалізації та порушує інкапсуляцію.
- Створення контрольних точок (checkpoints) у довгих транзакціях.

Структура та реалізація

- Originator (Творець): Об'єкт, стан якого потрібно зберігати. Створює Memento, що містить знімок його поточного стану, і використовує Memento для відновлення стану.
- Memento (Знімок): Зберігає внутрішній стан об'єкта Originator. Захищає збережені дані від доступу будь-яких об'єктів, крім Originator.
- Caretaker (Опікун): Відповідає за зберігання знімків Memento (наприклад, у стеку історії), але не має права змінювати їх або заглядати всередину.

4. «Observer» (Спостерігач)

Визначає залежність «один до багатьох» між об'єктами таким чином, що коли один об'єкт змінює свій стан, усі залежні від нього об'єкти отримують повідомлення про це та оновлюються автоматично.Основна ідея: Механізм підписки, що дозволяє об'єктам стежити за подіями в інших об'єктах.Застосування:

- Коли зміна стану одного об'єкта вимагає зміни інших, і ви не знаєте заздалегідь, скільки таких об'єктів буде.
- Реалізація архітектури MVC (Model-View-Controller), де представлення (View) оновлюється при зміні моделі (Model).
- Системи обробки подій (Event Handling).

Структура та реалізація

- Subject (Суб'єкт / Видавець): Знає своїх спостерігачів. Надає інтерфейс для приєднання (attach) та від'єднання (detach) об'єктів Observer.
- Observer (Спостерігач): Визначає інтерфейс оновлення для об'єктів, які мають бути повідомлені про зміни в суб'єкті (зазвичай метод update()).

- ConcreteSubject (Конкретний Суб'єкт): Зберігає стан, цікавий ConcreteObserver. Надсилає повідомлення спостерігачам, коли стан змінюється.
- ConcreteObserver (Конкретний Спостерігач): Зберігає посилання на об'єкт ConcreteSubject, зберігає стан, який має бути узгоджений із суб'єктом, та реалізує інтерфейс оновлення.

5. «Decorator» (Декоратор)

Дозволяє динамічно додавати об'єктам нову функціональність, загортаючи їх у корисні "обгортки". Це гнучка альтернатива успадкуванню для розширення функціональності.Ключова ідея: Об'єкт поміщається всередину іншого об'єкта-обгортки, який додає свою поведінку до або після виклику основного об'єкта.Застосування:

- Коли потрібно додавати обов'язки до окремих об'єктів динамічно та прозоро для клієнтського коду, не зачіпаючи інші об'єкти того ж класу.
- Коли розширення шляхом успадкування є неможливим або призводить до вибухового зростання кількості підкласів.
- Для додавання функцій, які можна вмикати або вимикати під час виконання (наприклад, смуги прокрутки у вікні, шифрування потоку даних).

Структура та реалізація

- Component (Компонент): Визначає інтерфейс для об'єктів, на які можна динамічно покладати додаткові обов'язки.
- ConcreteComponent (Конкретний Компонент): Визначає об'єкт, до якого можна додати нові функції.
- Decorator (Декоратор): Зберігає посилання на об'єкт Component і визначає інтерфейс, що відповідає інтерфейсу Component.
- ConcreteDecorator (Конкретний Декоратор): Додає нові обов'язки (функціональність) до компонента.

Хід роботи

1. Шаблон проектування «Factory Method»

Призначення:

Шаблон Factory Method (Фабричний метод) використовується для визначення інтерфейсу створення об'єктів, при цьому дозволяючи підкласам (конкретним реалізаціям) вирішувати, який саме клас інстанціювати. Він забезпечує створення продуктів через спільний інтерфейс, абстрагуючи клієнтський код від конкретних класів, що дозволяє системі бути незалежною від процесу створення, компонування та представлення об'єктів.

Проблема, яку вирішує шаблон:

У системах, які повинні створювати різні варіанти схожих продуктів, часто виникає проблема жорсткої залежності коду від конкретних класів. Якщо клієнтський код (наприклад, сервіс генерації) безпосередньо створює екземпляри генераторів (через оператор new), то додавання нового типу продукту вимагає зміни існуючого коду, що порушує принцип відкритості/закритості (Open/Closed Principle). Це призводить до заплутаних конструкцій if-else або switch при виборі типу генерації та ускладнює розширення системи.

У моєму проекті генератора інсталяторів існує потреба створювати різні артефакти на основі одного й того ж JAR-файлу: повноцінний виконуваний файл (.exe) або лише файл конфігурації (.xml) для подальшого ручного використання. Логіка створення цих двох сутностей різна, але для клієнта (контролера або білдера) це єдина дія — "згенерувати результат". Без використання Factory Method довелося б писати жорстку логіку вибору між генераторами всередині InstallerBuilder, що ускладнило б додавання нових форматів у майбутньому (наприклад, інсталяторів для Linux).

Як шаблон вирішує проблему:

Шаблон Factory Method вирішує цю проблему шляхом створення спільного інтерфейсу InstallerFactory, який декларує метод генерації. Конкретні реалізації цього інтерфейсу (ExeInstallerFactory, Launch4jConfigFactory) містять специфічну логіку створення конкретного продукту. Клієнтський код працює лише з інтерфейсом і не знає деталей реалізації. Вибір потрібної фабрики відбувається динамічно на основі типу інсталятора.

Процес створення об’єкта виглядає так:

2. Існує спільний інтерфейс InstallerFactory, який гарантує, що будь-яка фабрика вміє робити дві речі: повідомляти свій тип (supportsType) та генерувати продукт (generateInstaller).

3. При старті програми всі доступні реалізації фабрик реєструються в спеціальній мапі (Map), де ключем є тип інсталятора (InstallerType), а значенням — відповідна фабрика.

4. Клієнтський клас звертається до цієї мапи, отримує потрібну реалізацію за типом і викликає метод генерації, не знаючи, який саме клас виконує роботу.

Як реалізовано в проекті:

Інтерфейс InstallerFactory визначає контракт для всіх генераторів.

Клас ExeInstallerFactory реалізує цей інтерфейс для створення .exe файлів (використовуючи всередині ExeGenerator).

Клас Launch4jConfigFactory реалізує інтерфейс для генерації лише XML-конфігурації.

Клас InstallerBuilder виступає клієнтом: він автоматично отримує список усіх фабрик через Spring Dependency Injection, зберігає їх у Map<InstallerType, InstallerFactory> і делегує виконання потрібній фабриці залежно від обраного типу.

Таким чином, шаблон Factory Method забезпечує:

- Слабку зв'язність (Loose Coupling): Клієнтський код не залежить від конкретних класів генераторів.
- Розширюваність: Додавання нового типу інсталятора (наприклад, MSI або DEB) потребує лише створення нового класу фабрики без зміни існуючого коду InstallerBuilder.
- Дотримання принципу єдиної відповідальності (SRP): Логіка створення кожного типу продукту винесена в окремий клас.
- Поліморфізм: Можливість обробляти різні типи генерації однаковим чином через єдиний інтерфейс.

2. Діаграма класів, яка представляє використання шаблону «Factory Method»

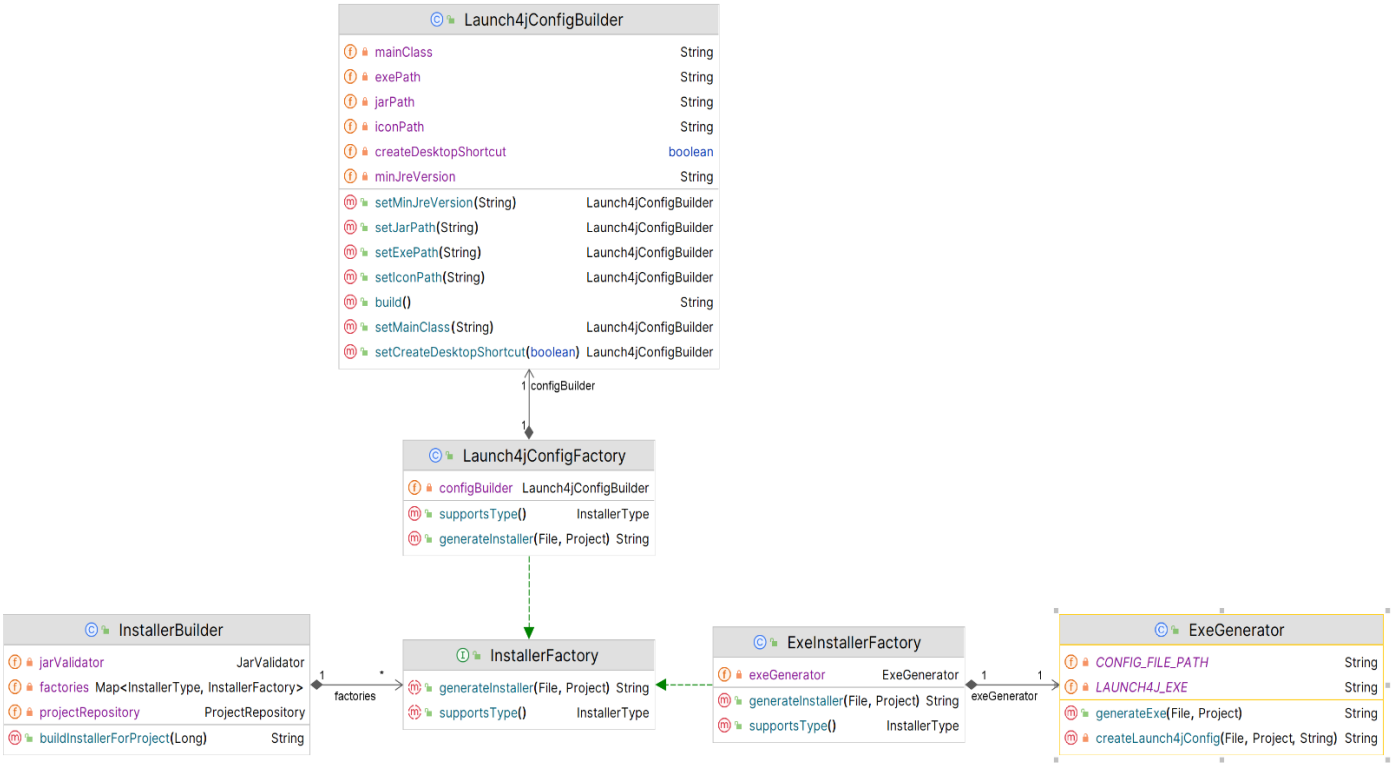


Рисунок 1 - Діаграма класів

Опис діаграми класів реалізації шаблону Factory Method

На діаграмі зображено структуру класів, що беруть участь у реалізації шаблону проектування Factory Method, який використовується для

створення різних типів інсталяційних артефактів (виконуваних файлів або конфігурацій) через єдиний інтерфейс.

Інтерфейс **InstallerFactory** виступає в ролі абстрактного творця (Creator). Він визначає контракт для всіх фабрик, декларуючи метод generateInstaller, який відповідає за створення продукту, та метод supportsType, що дозволяє ідентифікувати тип інсталятора. Цей інтерфейс забезпечує поліморфізм, дозволяючи клієнтському коду працювати з будь-якою конкретною фабрикою однаковим чином.

Класи ExeInstallerFactory та Launch4jConfigFactory є конкретними реалізаціями (Concrete Creators) інтерфейсу InstallerFactory.

ExeInstallerFactory реалізує логіку створення повноцінного .exe файлу. Для виконання цього завдання цей клас має залежність від сервісу ExeGenerator.

Launch4jConfigFactory реалізує логіку генерації XML-конфігурації для Launch4j. Він використовує Launch4jConfigBuilder для формування вмісту файлу.

Зв'язок між цими класами та InstallerFactory є відношенням реалізації (implements).

Клас **InstallerBuilder** виступає в ролі клієнта, який використовує фабричний метод. Він не створює екземпляри генераторів напряму через оператор new, а зберігає колекцію доступних фабрик у вигляді Map<InstallerType, InstallerFactory>. Це дозволяє йому динамічно вибирати потрібну реалізацію фабрики під час виконання програми на основі типу інсталятора, не знаючи деталей її внутрішньої роботи. Клас InstallerBuilder має залежність (асоціацію) до інтерфейсу InstallerFactory.

Enum **InstallerType** використовується як ідентифікатор для розрізнення типів інсталяторів (наприклад, EXE, CONFIG_ONLY). Він слугує ключем для вибору відповідної фабрики у клієнтському класі та повертається методом supportsType у конкретних фабриках.

Класи **ExeGenerator** та **Launch4jConfigBuilder** є допоміжними компонентами, які використовуються конкретними фабриками для виконання фактичної роботи з генерації файлів. Вони інкапсулюють низькорівневу логіку створення .exe та .xml файлів відповідно.

Ця діаграма відображає архітектуру, побудовану на принципі інверсії залежностей (Dependency Inversion), де високорівневий модуль InstallerBuilder залежить від абстракції InstallerFactory, а не від конкретних класів генерації. Це забезпечує гнучкість системи та легкість додавання нових типів інсталяторів без зміни існуючого коду.

3. Фрагменти коду реалізації шаблону «Factory Method»

InstallerType.java

```
package com.example.installer.service.installer;

public enum InstallerType {
    EXE,
    CONFIG_ONLY
}
```

InstallerFactory

```
package com.example.installer.service.installer;

import com.example.installer.File;
import com.example.installer.Project;

public interface InstallerFactory {
    InstallerType supportsType();

    String generateInstaller(File jarFile, Project project) throws
Exception;
}
```

ExeInstallerFactory

```
package com.example.installer.service.installer;

import com.example.installer.File;
import com.example.installer.Project;
import org.springframework.stereotype.Service;

@Service
public class ExeInstallerFactory implements InstallerFactory {

    private final ExeGenerator exeGenerator;
```

```
    public ExeInstallerFactory(ExeGenerator exeGenerator) {
        this.exeGenerator = exeGenerator;
    }

    @Override
    public InstallerType supportsType() {
        return InstallerType.EXE;
    }

    @Override
    public String generateInstaller(File jarFile, Project project) throws
Exception {
        return exeGenerator.generateExe(jarFile, project);
    }
}
```

Launch4jConfigFactory

```
package com.example.installer.service.installer;

import com.example.installer.File;
import com.example.installer.Project;
import org.springframework.stereotype.Service;

@Service
public class Launch4jConfigFactory implements InstallerFactory {

    private final Launch4jConfigBuilder configBuilder;

    public Launch4jConfigFactory(Launch4jConfigBuilder configBuilder) {
        this.configBuilder = configBuilder;
    }

    @Override
    public InstallerType supportsType() {
        return InstallerType.CONFIG_ONLY;
    }

    @Override
    public String generateInstaller(File jarFile, Project project) throws
Exception {
        String mainClass = project.getInstallerSettings().getMainClass();
        return configBuilder.setJarPath(jarFile.getPath())
                .setExePath(jarFile.getPath().replace(".jar", ".exe"))
                .setMainClass(mainClass)
                .setMinJreVersion("16")

.setCreateDesktopShortcut(project.getInstallerSettings().isCreateDesktopSho
rtcut())
                .build();
    }
}
```

InstallerBuilder

```
package com.example.installer.service.installer;

import com.example.installer.File;
import com.example.installer.Project;
```

```java
import com.example.installer.repository.ProjectRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

@Service
public class InstallerBuilder {

    private final JarValidator jarValidator;
    private final ProjectRepository projectRepository;

    private final Map<InstallerType, InstallerFactory> factories;

    public InstallerBuilder(JarValidator jarValidator,
                            ProjectRepository projectRepository,
                            List<InstallerFactory> factoryList) {
        this.jarValidator = jarValidator;
        this.projectRepository = projectRepository;

        this.factories = factoryList.stream()
                .collect(Collectors.toMap(InstallerFactory::supportsType, f
-> f));
    }

    @Transactional
    public String buildInstallerForProject(Long projectId) throws Exception
{
        Project project = projectRepository.findById(projectId)
                .orElseThrow(() -> new IllegalArgumentException("Project
not found"));

        ProjectFileIterator it = new
ProjectFileIterator(project.getFiles());
        List<File> jars = new ArrayList<>();

        while (it.hasNext()) {
            File f = it.next();
            if (jarValidator.isValidJar(f)) {
                jars.add(f);
            }
        }

        if (jars.isEmpty())
            throw new IllegalStateException("No JAR file found in
project.");
        if (jars.size() > 1)
            throw new IllegalStateException("Only one JAR file allowed.");

        File jarFile = jars.get(0);

        InstallerType targetType = InstallerType.EXE;

        InstallerFactory factory = factories.get(targetType);

        if (factory == null) {
            throw new UnsupportedOperationException("Type not supported: "
+ targetType);
        }
```

```
        return factory.generateInstaller(jarFile, project);
    }
}
```

У проекті використано вдосконалену версію патерну Factory Method, адаптовану під Spring Framework:

- Поліморфізм: Клієнт (InstallerBuilder) працює з інтерфейсом InstallerFactory, не знаючи деталей реалізації.
- Розширюваність (Open/Closed Principle): Щоб додати новий тип інсталятора (наприклад, для Linux), достатньо створити новий клас, що імплементує InstallerFactory, і додати тип в InstallerType. Змінювати код InstallerBuilder не потрібно.
- Інверсія управління (IoC): Spring автоматично знаходить усі реалізації фабрик і впроваджує їх у клієнтський клас.

Висновки

На даному етапі реалізації проєкту «Installer Generator» було успішно впроваджено шаблон проектування «Factory Method», який дозволив уніфікувати процес створення різних типів інсталяційних артефактів та забезпечити гнучкість архітектури системи.

Було розроблено спільний інтерфейс InstallerFactory, який став базою для конкретних реалізацій: ExeInstallerFactory для створення виконуваних файлів та Launch4jConfigFactory для генерації XML-конфігурацій. Клас InstallerBuilder було модифіковано для роботи з абстракціями, що дозволило реалізувати динамічний вибір необхідної фабрики під час виконання програми на основі типу інсталятора (InstallerType).

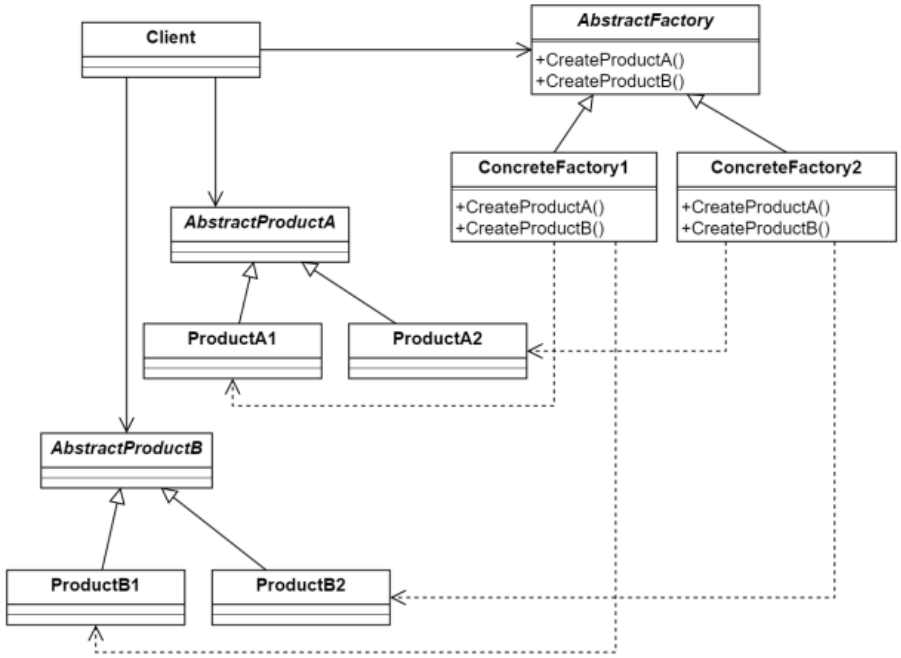
Таким чином, на поточному етапі було:

- реалізовано шаблон проектування Factory Method для абстрагування процесу створення інсталяторів;
- забезпечено дотримання принципу відкритості/закритості (Open/Closed Principle), що дозволяє додавати нові формати інсталяторів без зміни існуючого коду;
- знижено зв'язність компонентів системи (loose coupling) шляхом використання інверсії залежностей;
- налагоджено механізм автоматичної реєстрації та вибору фабрик засобами Spring Framework.

Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»? Надати інтерфейс для створення сімейств пов'язаних або залежних об'єктів, не специфікуючи їхніх конкретних класів.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».



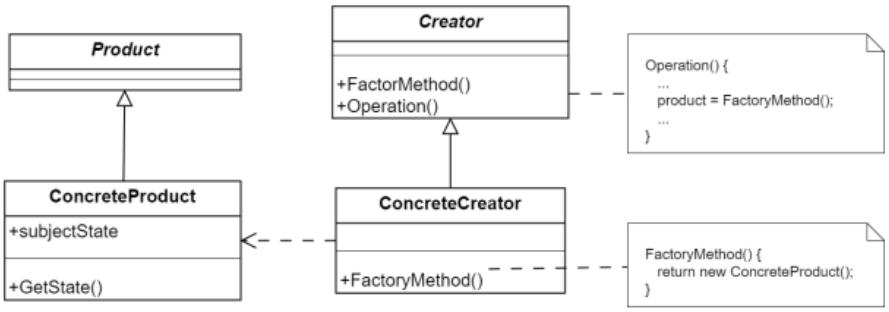
3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

- **AbstractFactory:** оголошує інтерфейс для операцій створення абстрактних продуктів.
- **ConcreteFactory:** реалізує операції для створення конкретних продуктів.
- **AbstractProduct:** оголошує інтерфейс для типу об'єкта-продукту.
- **ConcreteProduct:** визначає об'єкт-продукт, що створюється відповідною конкретною фабрикою.
- **Client:** використовує лише інтерфейси, оголошені в класах AbstractFactory та AbstractProduct.

4. Яке призначення шаблону «Фабричний метод»? Визначити інтерфейс для створення об'єкта, але дозволити підкласам вирішувати, який саме клас

інстанціювати. Шаблон дозволяє класу делегувати створення об'єктів підкласам.

5. Нарисуйте структуру шаблону «Фабричний метод».



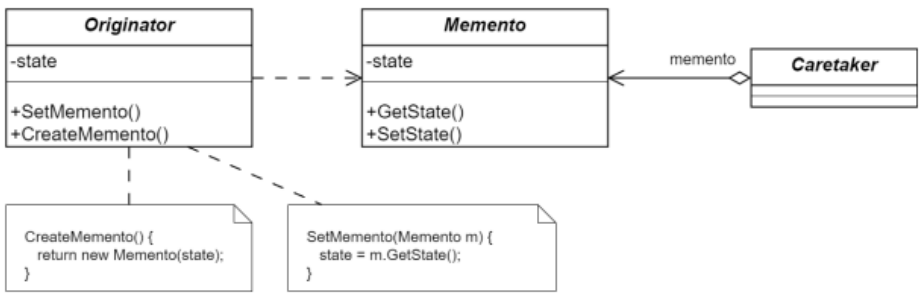
6. Які класи входять в шаблон «Фабричний метод», та яка між ними взаємодія?

- **Product:** визначає інтерфейс об'єктів, які створює фабричний метод.
- **ConcreteProduct:** реалізує інтерфейс Product.
- **Creator:** оголошує фабричний метод, який повертає об'єкт типу Product.
- **ConcreteCreator:** заміщує фабричний метод, щоб він повертав екземпляр ConcreteProduct.

7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»? «Фабричний метод» використовує наслідування та створює лише один продукт, тоді як «Абстрактна фабрика» використовує композицію об'єктів для створення сімейств продуктів.

8. Яке призначення шаблону «Знімок»? Дозволяє, не порушуючи інкапсуляцію, зафіксувати та зберегти внутрішній стан об'єкта так, щоб пізніше цей об'єкт можна було відновити у цьому стані.

9. Нарисуйте структуру шаблону «Знімок».

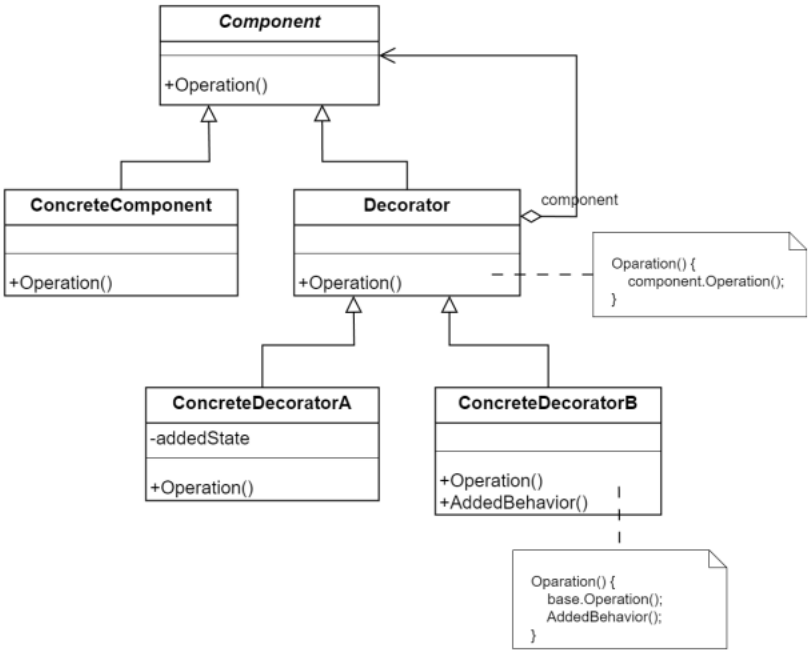


10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

- **Originator:** створює знімок (memento), що містить його поточний внутрішній стан, і може відновити стан зі знімка.
- **Memento:** зберігає внутрішній стан Originator.
- **Caretaker:** відповідає за збереження знімка, але не виконує операцій над ним і не досліджує його вміст.

11. Яке призначення шаблону «Декоратор»? Динамічно додавати об'єктам нові обов'язки. Цей шаблон забезпечує гнучку альтернативу успадкуванню для розширення функціональності.

12. Нарисуйте структуру шаблону «Декоратор».



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

- **Component:** визначає інтерфейс для об'єктів, на які можуть бути динамічно покладені обов'язки.
- **ConcreteComponent:** визначає об'єкт, на який покладаються додаткові обов'язки.
- **Decorator:** зберігає посилання на об'єкт Component і визначає інтерфейс, що відповідає інтерфейсу Component.
- **ConcreteDecorator:** додає обов'язки до компонента.

14. Які є обмеження використання шаблону «декоратор»? Може призвести до створення великої кількості дрібних об'єктів, що ускладнює налагодження. Також складно ініціалізувати та налаштувати об'єкт, загорнутий у багато шарів декораторів, а сам декоратор не є ідентичним декорованому об'єкту (проблема ідентичності об'єкта).