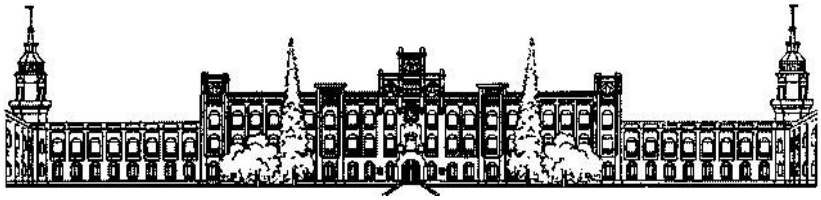


Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота №6

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Патерни проектування»

Виконав:

студент групи ІА-32

Нагорний Максим

Перевірив:

Мягкий М. Ю.

Київ 2025

Зміст

Вступ

Тема: Вступ до паттернів проектування.

Мета: Вивчити структуру шаблонів «Abstract Factory», «Factory Method», «Memento», «Observer», «Decorator» та навчитися застосовувати їх в реалізації програмної системи.

Завдання

- Ознайомитись з короткими теоретичними відомостями.

- Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

- Реалізувати один з розглянутих шаблонів за обраною темою.

- Реалізувати не менше 3-х класів відповідно до обраної теми.

- Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

14. **Архіватор** (strategy, adapter, factory method, facade, visitor, p2p)

Архіватор повинен являти собою візуальний додаток з можливістю створення і редагування архівів різного типу (.tar.gz, .zip, .rar, .ace) – додавання/ видалення файлів / папок, редагування метаданих (по можливості), перевірка checksum архівів, тестування архівів на наявність пошкоджень, розбиття архівів на частини.

Теоретичні відомості

Шаблон «Factory Method»

Патерн «Factory Method» – це породжуючий шаблон, який визначає інтерфейс для створення об’єкта, але дозволяє підкласам вирішувати, об’єкт якого саме класу створювати. Він інкапсулює логіку інстанціювання та делегує її дочірнім класам. Структура включає абстрактний клас «Творець» (Creator) з абстрактним «фабричним методом» та конкретні реалізації Творця, які повертають «Конкретні продукти» (Product). Це дозволяє системі бути гнучкою, оскільки нові типи продуктів можна додавати, не змінюючи код клієнта, який використовує базовий інтерфейс Творця. Застосовується, коли клас не може заздалегідь знати, об’єкти яких класів йому потрібно створити, або коли клас хоче делегувати відповідальність за створення об’єктів своїм підкласам.

Шаблон «Abstract Factory»

Патерн «Abstract Factory» - поруджуючий патерн, але на вищому рівні; його часто описують як «фабрику фабрик». Він надає інтерфейс для створення *сімейств* взаємопов’язаних або взаємозалежних об’єктів, не вказуючи їхні конкретні класи. Структура складається з інтерфейсу «Абстрактної фабрики», що оголошує методи для створення різних типів абстрактних продуктів. Клієнтський код працює лише з абстрактними інтерфейсами, що дозволяє легко перемикатися між різними «темами» або наборами об'єктів (наприклад, GUI для різних операційних систем), не змінюючи логіку програми.

Шаблон «Decorator»

Патерн «Decorator» – це структурний шаблон, який дозволяє динамічно додавати нові обов’язки або поведінку до об’єкта, обгортаючи його в об’єкти-декоратори. Він є гнучкою альтернативою наслідуванню для розширення функціональності. Структура вимагає, щоб і декорований об’єкт («Компонент»), і сам «Декоратор» реалізовували спільний інтерфейс. Декоратор містить посилання на об’єкт Компонента і делегує йому виклики, додаючи власну логіку до або після виклику. Це дозволяє створювати ланцюжки декораторів для комбінування функцій, причому функціональність можна додавати та видаляти під час виконання програми.

Шаблон «Observer»

Патерн «Observer» – це поведінковий шаблон, який встановлює залежність «один-до-багатьох» між об’єктами. Коли один об’єкт, відомий як «Суб’єкт» (Subject) або «Видавець» (Observable), змінює

свій стан, усі залежні від нього об’єкти, «Спостерігачі» (Observers), автоматично сповіщуються та оновлюються. Структура передбачає, що Суб’єкт веде список своїх Спостерігачів і надає методи для їх додавання (attach) та видалення (detach), а також метод notify(). Інтерфейс Спостерігача має метод update(), який Суб’єкт викликає під час сповіщення. Цей патерн широко використовується в системах, керованих подіями, та для реалізації зв’язку між моделлю (Model) та представленням (View) в архітектурі MVC.

Шаблон «Memento»

Патерн «Memento» – це поведінковий шаблон, який дозволяє фіксувати та зберігати внутрішній стан об’єкта (не порушуючи інкапсуляцію), щоб до нього можна було повернутися пізніше. Це є основою для реалізації механізмів скасування (Undo) або повернення до контрольних точок. Структура включає три ролі: «Творець» (Originator) – об’єкт, стан якого потрібно зберегти, він вміє створювати «Знімок» та відновлювати свій стан з нього; «Знімок» (Memento) – об’єкт, що зберігає стан Творця (його вміст має бути прихованим від інших); та «Опікун» (Caretaker) – об’єкт, який зберігає Знімок, але не модифікує і не читає його, він лише запитує Знімок у Творця та повертає його назад на вимогу.

Хід виконання

Для розробки системи архіватора було обрано шаблон проєктування «Factory method». Вибір цього шаблону обумовлений необхідністю створення сімейств об'єктів без вказівки їх конкретних класів.

Суть шаблону «Адаптер» полягає у тому, що створюється інтерфейс для створення об’єкту, який дозволяє підкласам вирішувати, об’єкт якого саме класу створювати . Таким чином, наш архіватор може працювати з різними типами архівування, не змінюючи основну логіку програми. Адаптер перехоплює виклик методу і перенаправляє його на відповідний метод сторонньої бібліотеки, перетворюючи формат виклику у потрібний для системи.

Для реалізації цього шаблону ми виконали такі дії:

1. Створено загальний інтерфейс StrategyArchive, цей інтерфейс визначає метод для архівування
2. Створено клас ZipStrategyArchive, що імплементується від StrategyArchive та перевизначає метод compress
3. Створено клас TarGzStrategyArchive, що імплементується від StrategyArchive та перевизначає метод compress

4. Створено клас RarStrategyArchive, що імплементується від StrategyArchive та перевизначає метод compress
5. Створено клас ArchiveFactory, який оголошує фабричний метод
6. Створено клас RarArchiveFactory, що імплементується від ArchiveFactor
7. Створено клас TarArchiveFactory, що імплементується від ArchiveFactor
8. Створено клас ZipArchiveFactory, що імплементується від ArchiveFactor

Переваги обраного шаблону:
- Можливість роботи з кожним типом архівування окремо не впливаючи, на інші типи архівування;
- Можливість додавати нові типи архівування без зміни вже написаного коду;
- Зменшення залежностей між класами, підтримка IoC, тепер модулі верхнього рівня не залежать від модулів нижнього рівня;
- Робота зійснюється через 1 інтерфейс;
- Підвищення гнучкості та розширюваності системи;

На рис. 1 наведена модифікована діаграма класів із представленням використання шаблону в реалізації системи

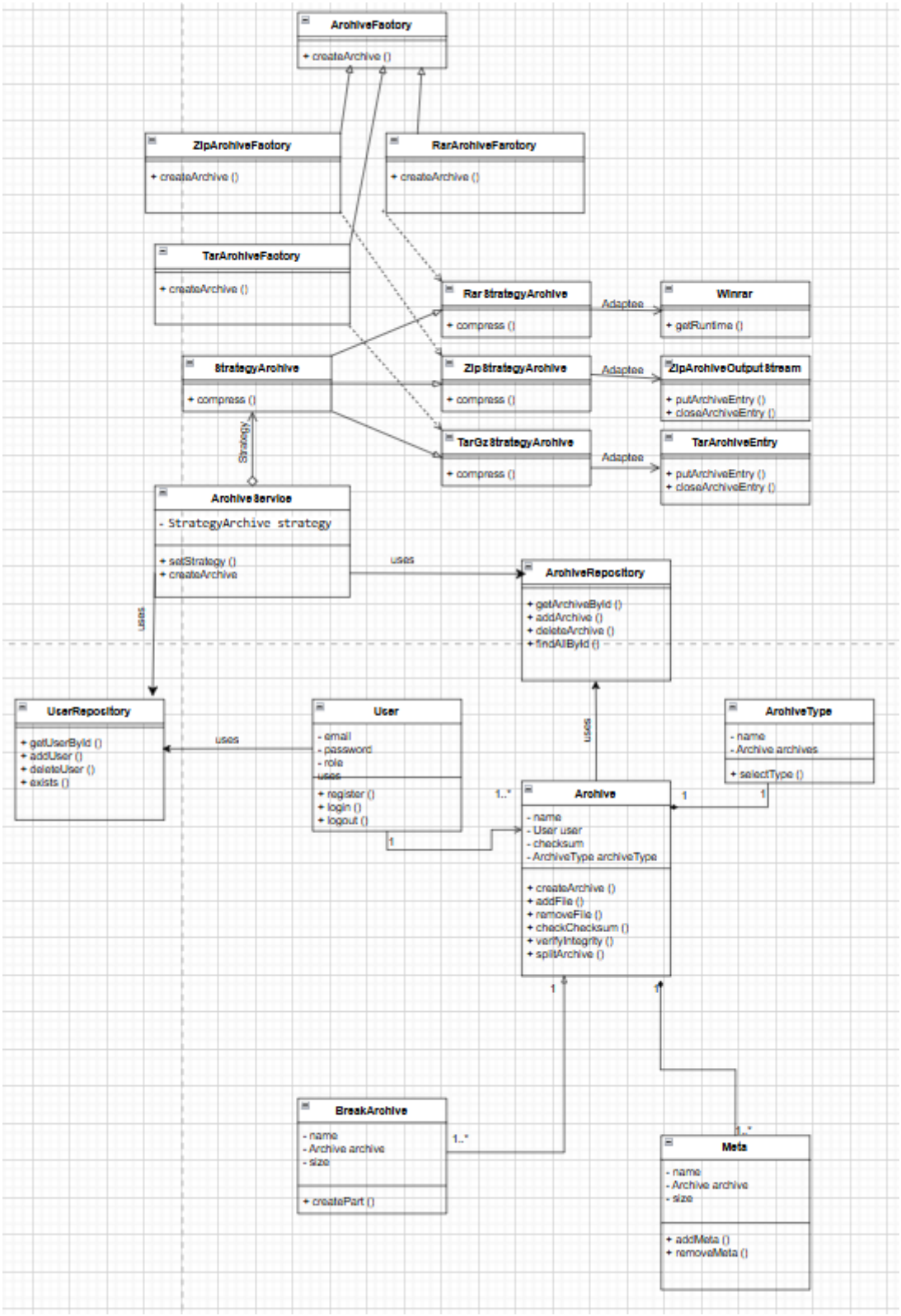


Рис. 1. Діаграма класів

Код програми

```
package com.example.demo.factory;
import com.example.demo.strategy.StrategyArchive;
public abstract class ArchiveFactory {
    public abstract StrategyArchive createArchive();
}
package com.example.demo.factory;
import com.example.demo.strategy.*;
public class ArchiveStrategyFactory {
    public static StrategyArchive create(String type) {
        return switch (type.toLowerCase()) {
            case "zip" -> new ZipStrategyArchive();
            case "tar.gz" -> new TarGzStrategyArchive();
            case "rar" -> new RarStrategyArchive("C:\\Program Files\\WinRAR\\WinRAR.exe");
            default -> throw new IllegalArgumentException("Невідомий тип архіву: " + type);
        };
    }
}
package com.example.demo.factory;
import com.example.demo.strategy.StrategyArchive;
import com.example.demo.strategy.RarStrategyArchive;
public class RarArchiveFactory extends ArchiveFactory {
    private final String winrarPath;
    public RarArchiveFactory(String winrarPath) {
        this.winrarPath = winrarPath;
    }
    @Override
    public StrategyArchive createArchive() {
        return new RarStrategyArchive(winrarPath);
    }
}
package com.example.demo.factory;

import com.example.demo.strategy.StrategyArchive;
```

```
import com.example.demo.strategy.ZipStrategyArchive;
public class ZipArchiveFactory extends ArchiveFactory {
    @Override
    public StrategyArchive createArchive() {
        return new ZipStrategyArchive();
    }
}
package com.example.demo.factory;
import com.example.demo.strategy.StrategyArchive;
import com.example.demo.strategy.TarGzStrategyArchive;
public class TarArchiveFactory extends ArchiveFactory {
    @Override
    public StrategyArchive createArchive() {
        return new TarGzStrategyArchive();
    }
}
```

Висновок

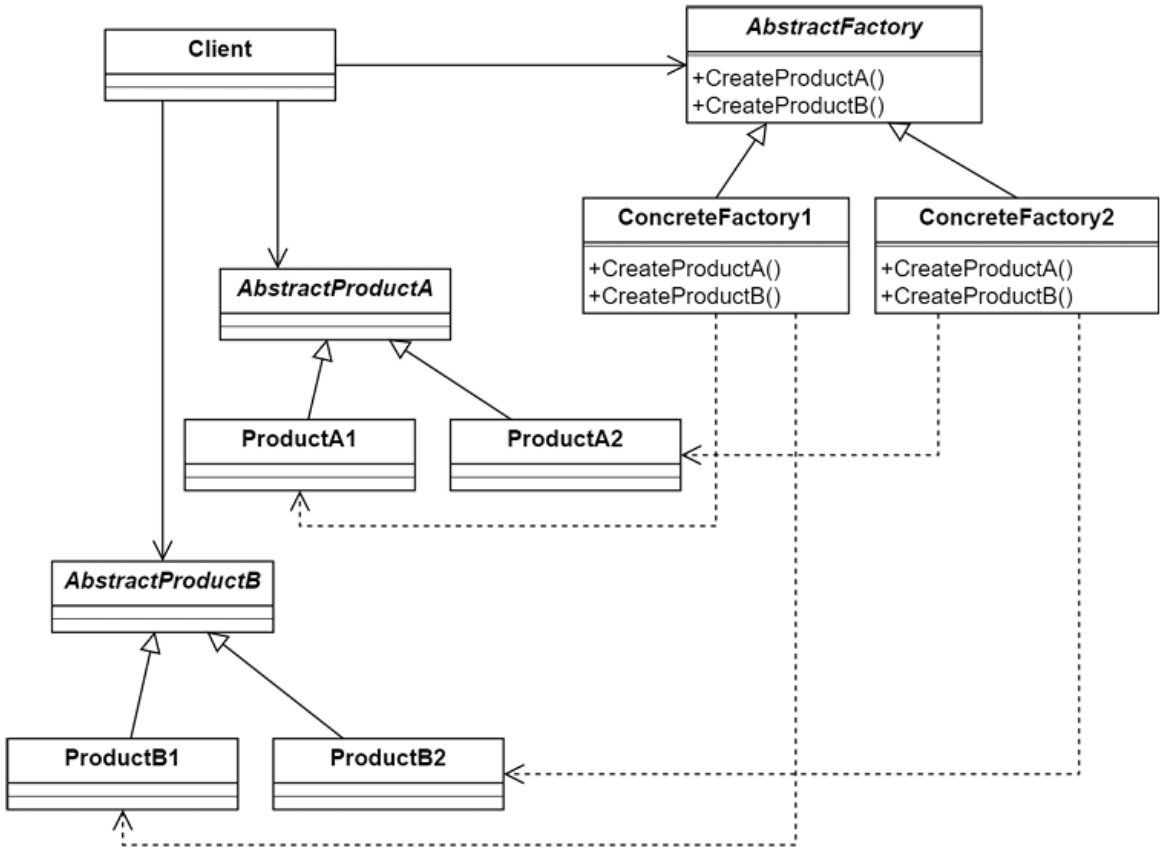
Під час виконання лабораторної роботи було реалізовано шаблон «Factory method» згідно теми «архіватор», під час реалізації було розроблено загальний інтерфейс ArchiveFactory, який визначає метод createArchive() для архівування та розроблено класи – наслідники ZipArchiveFactory, TarArchiveFactory, RarArchiveFactory для реалізації логіки архівування різних типів та останнім етапом було модифікації діаграми класів з представленням використання шаблону «Factory method».

Питання до лабораторної роботи

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

   Шаблон «Абстрактна фабрика» (Abstract Factory) призначений для створення сімейств взаємопов'язаних об'єктів без необхідності вказувати їх конкретні класи. Він дозволяє клієнтському коду працювати з об'єктами через спільні інтерфейси, забезпечуючи незалежність від конкретних реалізацій. Таким чином, зміна всієї групи продуктів може виконуватись просто шляхом підміни фабрики, не змінюючи основну логіку програми.

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



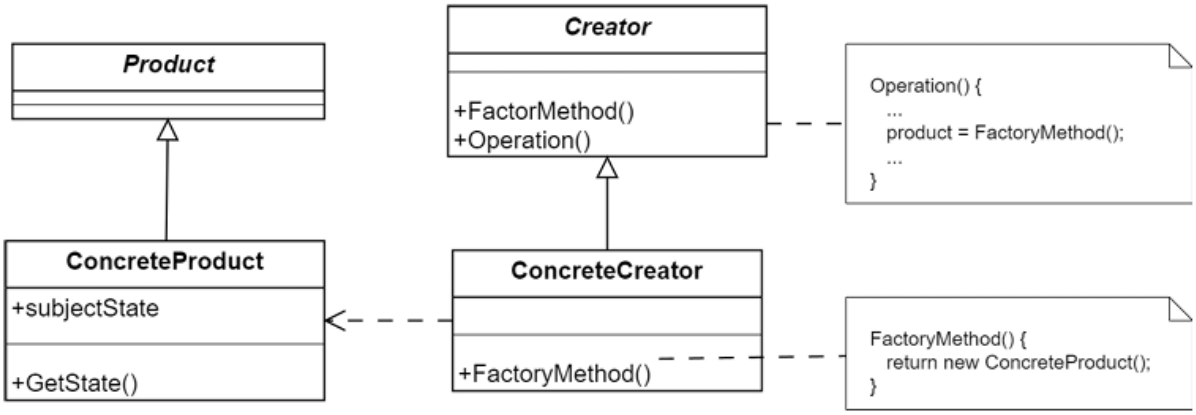
3. Які класи входять в шаблон «Абстрактна фабрика», та яка між ними взаємодія?

До складу шаблону входять класи AbstractFactory, ConcreteFactory, AbstractProduct та ConcreteProduct. Абстрактна фабрика визначає інтерфейс створення об’єктів, конкретна фабрика реалізує цей інтерфейс, створюючи об’єкти певного сімейства продуктів. Абстрактні продукти визначають спільну поведінку для всіх типів об’єктів, а конкретні продукти реалізують цю поведінку. Клієнт звертається до фабрики через абстрактний інтерфейс, отримує створені продукти й використовує їх без знання конкретних реалізацій.

4. Яке призначення шаблону «Фабричний метод»?

Шаблон «Фабричний метод» (Factory Method) призначений для делегування створення об’єктів підкласам. Він дозволяє базовому класу визначити загальний інтерфейс для створення об’єктів, але сам процес створення переносить у підкласи, які вирішують, який саме об’єкт створювати. Таким чином, фабричний метод забезпечує гнучкість і розширюваність, дозволяючи додавати нові типи об’єктів без зміни коду базового класу.

5. Нарисуйте структуру шаблону «Фабричний метод».



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

У шаблоні беруть участь класи Product, ConcreteProduct, Creator і ConcreteCreator. Клас Product визначає інтерфейс для всіх об’єктів, які можуть бути створені. ConcreteProduct є конкретною реалізацією цього інтерфейсу. Creator містить фабричний метод factoryMethod(), який повертає об’єкт типу Product, а ConcreteCreator перевизначає цей метод, створюючи конкретний продукт. Клієнт звертається до Creator і викликає його метод, не знаючи, який саме продукт створюється всередині — це забезпечує незалежність коду від конкретних класів.

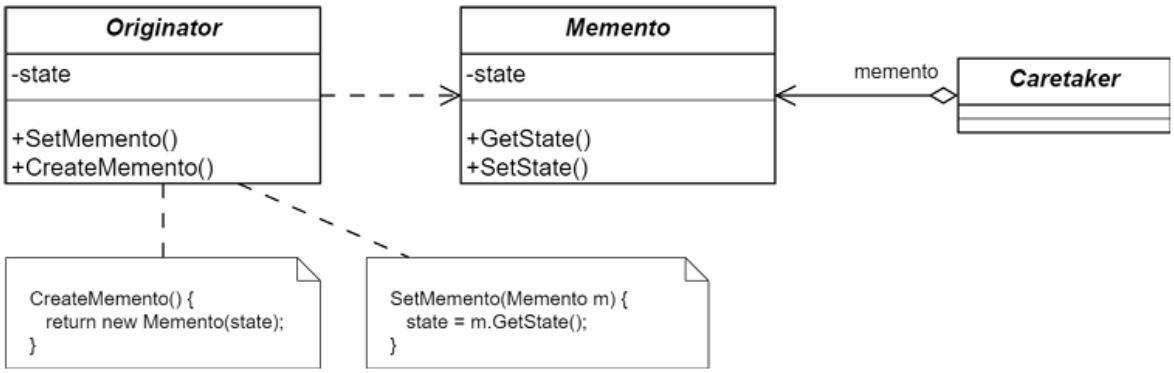
7. Чим відрізняється шаблон «Абстрактна фабрика» від «Фабричний метод»?

Основна відмінність полягає у рівні абстракції та кількості створюваних об’єктів. Шаблон «Фабричний метод» створює об’єкти одного типу, делегуючи цей процес підкласам. Натомість «Абстрактна фабрика» створює цілі сімейства взаємопов’язаних об’єктів. Таким чином, «Абстрактна фабрика» часто реалізується за допомогою кількох фабричних методів і використовується на вищому рівні абстракції.

8. Яке призначення шаблону «Знімок»?

Шаблон «Знімок» (Memento) призначений для збереження внутрішнього стану об’єкта таким чином, щоб згодом можна було відновити його без порушення інкапсуляції. Він дозволяє реалізувати механізм «скасування» дій або повернення до попереднього стану об’єкта, зберігаючи його історію змін.

9. Нарисуйте структуру шаблону «Знімок».



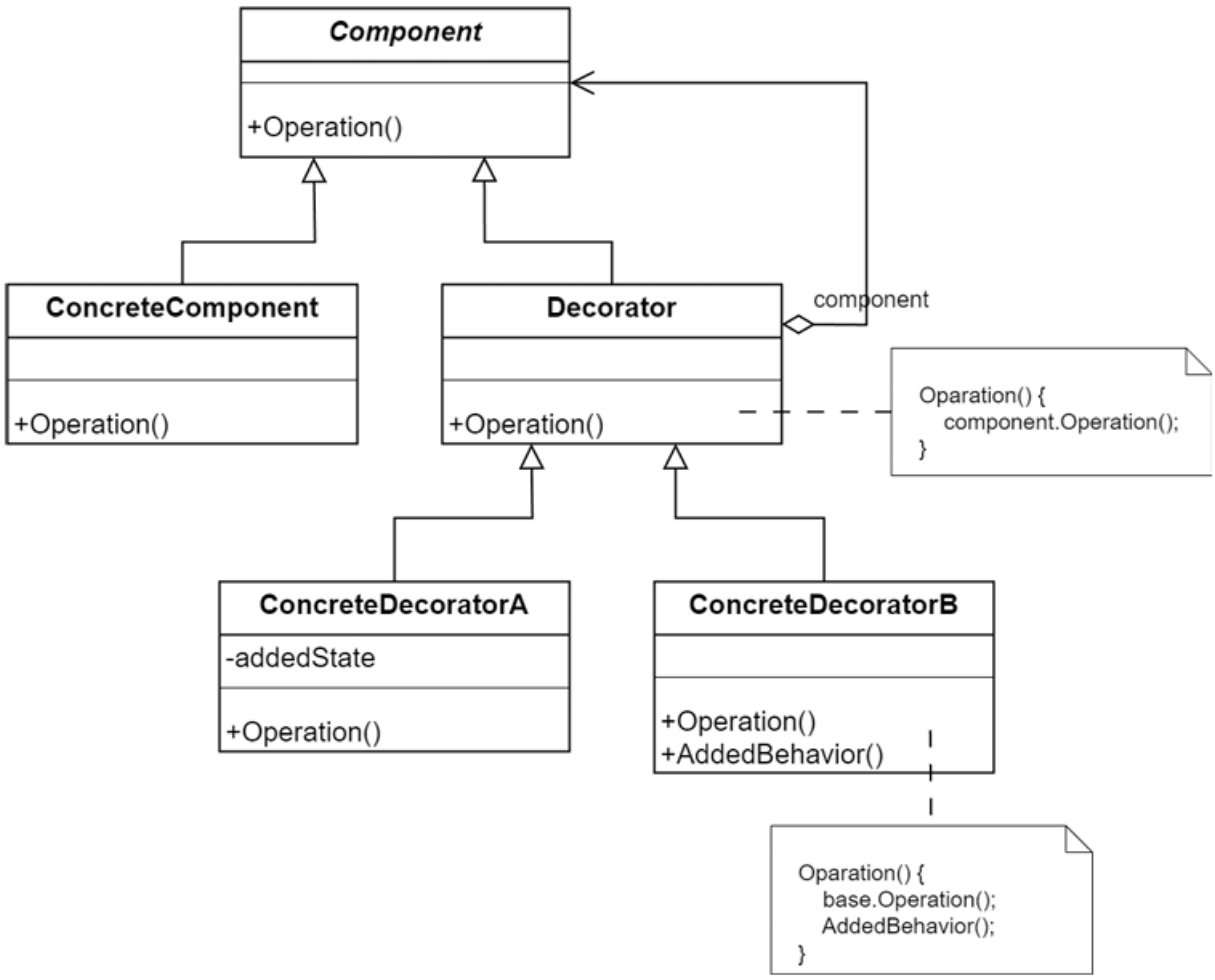
10. Які класи входять в шаблон «Знімок», та яка між ними взаємодія?

Originator відповідає за створення та відновлення об’єктів-знімків (Memento). Клас Memento зберігає копію внутрішнього стану об’єкта. Caretaker зберігає знімок і у разі потреби передає його назад до Originator для відновлення попереднього стану. Таким чином, клієнтська частина системи може зберігати історію змін об’єкта, не знаючи його внутрішньої структури.

11. Яке призначення шаблону «Декоратор»?

Шаблон «Декоратор» (Decorator) використовується для динамічного розширення функціональності об’єктів без зміни їхнього коду. Він дозволяє «обгорнути» базовий об’єкт додатковими класами-декораторами, які реалізують ту саму поведінку, але додають нові можливості. Це забезпечує гнучке розширення системи, дотримуючись принципу відкритості/закритості.

12. Нарисуйте структуру шаблону «Декоратор».



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

Компонент є базовим інтерфейсом, який реалізують усі об’єкти, що можуть бути декоровані. Конкретний компонент реалізує основну функціональність. Декоратор містить посилання на об’єкт типу компонент і делегує йому виконання основних методів. Конкретний декоратор перевизначає або розширює ці методи, додаючи нову поведінку. Таким чином, можна створювати довільну кількість обгорток, динамічно змінюючи функціональність об’єкта.

14. Які є обмеження використання шаблону «декоратор»?

Основним обмеженням шаблону «Декоратор» є ускладнення структури програми — велика кількість дрібних класів може зробити систему менш зрозумілою. Також виникає проблема ініціалізації, коли необхідно правильно комбінувати кілька декораторів. Крім того, важливо, щоб усі декоратори підтримували спільний інтерфейс, інакше зникне прозорість використання.