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 «Теорія розробки програмного забезпечення»

Тема: «Патерни проектування»

Виконав:
Студент групи ІА-34
Цап Андрій

Перевірив:
Мягкий Михайло Юрійович

Київ – 2025

Зміст

Вступ

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

У даній лабораторній роботі було розглянуто групу поведінкових і породжуючих шаблонів, серед яких Memento, Observer, Decorator та інші. Після аналізу структури створеної системи керування проєктами було визначено, що найбільш доцільним шаблоном для інтеграції є Observer.

Підсистема управління проєктами містить велику кількість подій: зміна статусу, редагування інформації, додавання нового учасника, створення версії чи ітерації. Ці дії природно утворюють модель «сповіщення підписників», де зміна стану одного об’єкта має автоматично викликати реакції інших компонентів системи.

Теоретичні відомості

Шаблон «Abstract Factory» - призначення патерну: Шаблон

«Абстрактна фабрика» використовується для створення сімейств об'єктів без вказівки їх конкретних класів. Для цього виноситься загальний інтерфейс фабрики (AbstractFactory) і створюються його реалізації для різних сімейств продуктів.

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 біль 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.

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.

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.

Шаблон «Memento» - призначення: Шаблон використовується для збереження і відновлення стану об'єктів без порушення інкапсуляції. 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.

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.

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.

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.

Шаблон «Decorator» - призначення: Шаблон призначений для динамічного додавання функціональних можливостей об'єкту під час роботи програми. 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.

Переваги та недоліки:

+ Дозволяє мати кілька дрібних об’єктів, замість одного об’єкта «на всі випадки життя».

+ Дозволяє додавати обов’язки «на льоту».

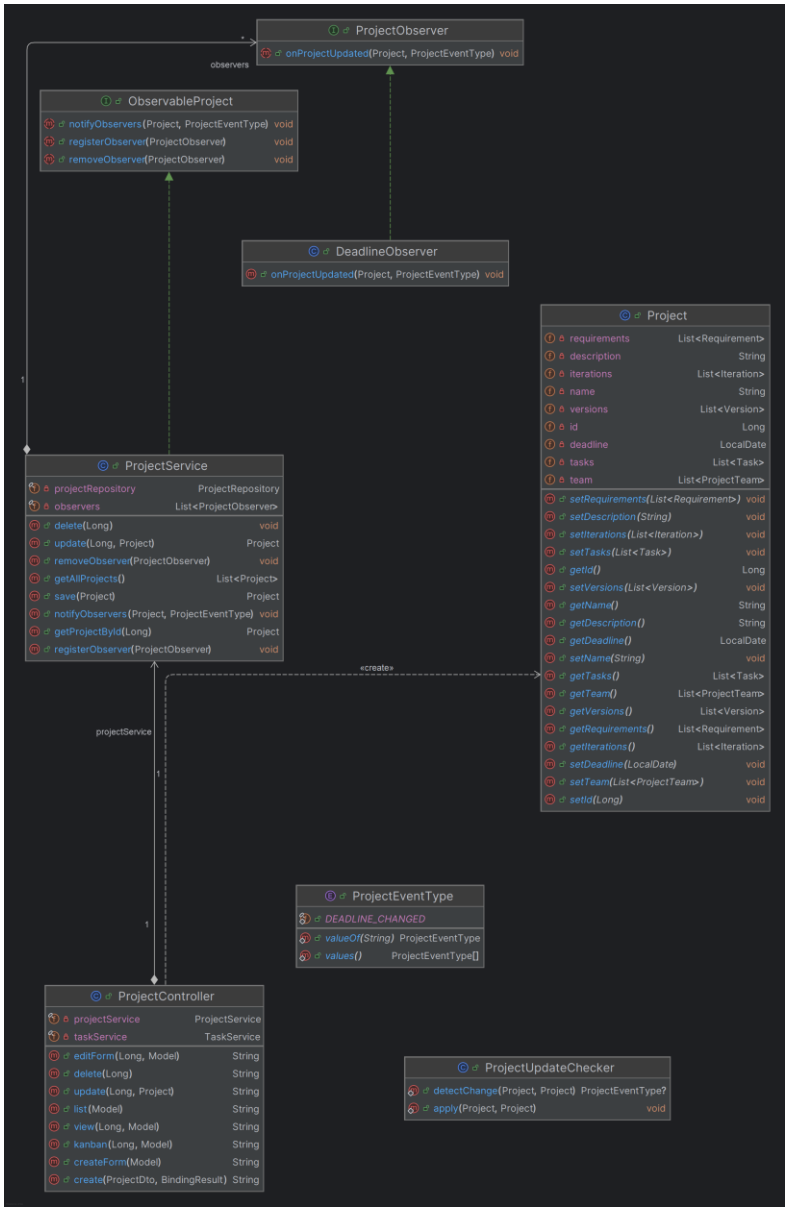
+ Більша гнучкість, ніж у спадкування.

- Велика кількість крихітних класів.

- Важко конфігурувати об’єкти, які загорнуто в декілька обгорток одночасно.

Хід роботи

Діаграма класів для реалізації шаблону Observer:



1. **ObservableProject** – інтерфейс, який описує поведінку "суб’єкта" (Subject).

Він визначає методи:

registerObserver(), removeObserver(), notifyObservers() — необхідні для керування списком спостерігачів.

2. **ProjectObserver** – інтерфейс спостерігача (Observer)

Містить метод onProjectUpdated(Project, ProjectEventType), який викликається при зміні стану проєкту.

3. **DeadlineObserver** – конкретний спостерігач (ConcreteObserver).

Реагує на зміну дедлайну проєкту та виконує дії сповіщення.

4. **ProjectService** – «конкретний суб'єкт» (Concrete Subject).

Керує списком спостерігачів і викликає notifyObservers() при зміні стану проєкту.

5. **ProjectEventType** – перелік типів подій, які можуть статися з проєктом).

Дозволяє спостерігачам розуміти, на яку саме зміну вони реагують.

6. **ProjectUpdateChecker** – допоміжний клас (utility), який визначає, які саме поля проєкту були змінені, та повертає відповідний тип події.
7. **Project** – доменна модель, над якою працює сервіс і яку відстежують спостерігачі.
8. **ProjectController** – клас, який взаємодіє з користувачем через веб-інтерфейс.

Він викликає ProjectService, тим самим ініціюючи створення, редагування та видалення проєкту, а також запуск механізму Observer.

Переваги використання

Гнучке реагування на зміну стану

У модулі роботи з проєктами різні частини системи можуть по-різному реагувати на зміну проєкту.
Завдяки Observer можна додати новий тип реакції, просто створивши новий клас-спостерігач, не змінюючи ProjectService чи Project.

Зменшення кількості залежностей

Без Observer сервіс довелось би «знати» про всі модулі, які треба сповіщати.

Чистіша архітектура

Observer дозволяє винести логіку реакції на зміну стану в окремі класи.

Недоліки використання

Ускладнення початкового розуміння архітектури

На відміну від прямого виклику методів, механізм сповіщення виглядає менш очевидно.

Можливі труднощі з відстеженням взаємодій

При великій кількості спостерігачів складніше зрозуміти, який саме клас виконує яку реакцію, особливо якщо зміна проєкту запускає кілька ланцюгів подій.

Вихідні коди реалізації шаблону Observer

DeadlineObserver.java

Run / Debug Configurations

```java
package org.example.projectmanagementsoftware.pattern.observer.implementations;

import org.example.projectmanagementsoftware.domain.Project;
import org.example.projectmanagementsoftware.pattern.observer.enums.ProjectEventType;
import org.example.projectmanagementsoftware.pattern.observer.interfaces.ProjectObserver;
import org.springframework.stereotype.Component;

@Component  new *
public class DeadlineObserver implements ProjectObserver {

    @Override  1 usage  new *
    public void onProjectUpdated(Project project, ProjectEventType eventType) {
        if (eventType == ProjectEventType.DEADLINE_CHANGED) {
            System.out.println("Дедлайн змінено: " + project.getDeadline());
        }
    }
}

```

Рис 1 - Код DeadlineObserver

ObservableProject.java

```java
package org.example.projectmanagementsoftware.pattern.observer.interfaces;

import org.example.projectmanagementsoftware.domain.Project;
import org.example.projectmanagementsoftware.pattern.observer.enums.ProjectEventType;

public interface ObservableProject {  2 usages  1 implementation  new *

    void registerObserver(ProjectObserver observer);  no usages  1 implementation  new *

    void removeObserver(ProjectObserver observer);  no usages  1 implementation  new *

    void notifyObservers(Project project, ProjectEventType eventType);  1 usage  1 impleme
}

```

Рис 2 - Код ObservableProject

ProjectObserver.java

```java
package org.example.projectmanagementsoftware.pattern.observer.interfaces;

import org.example.projectmanagementsoftware.domain.Project;
import org.example.projectmanagementsoftware.pattern.observer.enums.ProjectEventType;

public interface ProjectObserver {  8 usages  1 implementation  new *

    void onProjectUpdated(Project project, ProjectEventType eventType);  1 usage  1 implem
}
```

Рис 3 - Код ProjectObserver

ProjectUpdateChecker.java

```java
package org.example.projectmanagementsoftware.pattern.observer.utils;

import org.example.projectmanagementsoftware.domain.Project;
import org.example.projectmanagementsoftware.pattern.observer.enums.ProjectEventType;

public class ProjectUpdateChecker {  3 usages  new *

    public static ProjectEventType detectChange(Project oldP, Project newP) {  1 usage
        if (oldP.getDeadline() != null
                && newP.getDeadline() != null
                && !oldP.getDeadline().equals(newP.getDeadline())) {
            return ProjectEventType.DEADLINE_CHANGED;
        }
        return null;
    }

    public static void apply(Project oldP, Project newP) {  1 usage  new *
        oldP.setDeadline(newP.getDeadline());
    }
}
```

Рис 4 - Код ProjectUpdateChecker

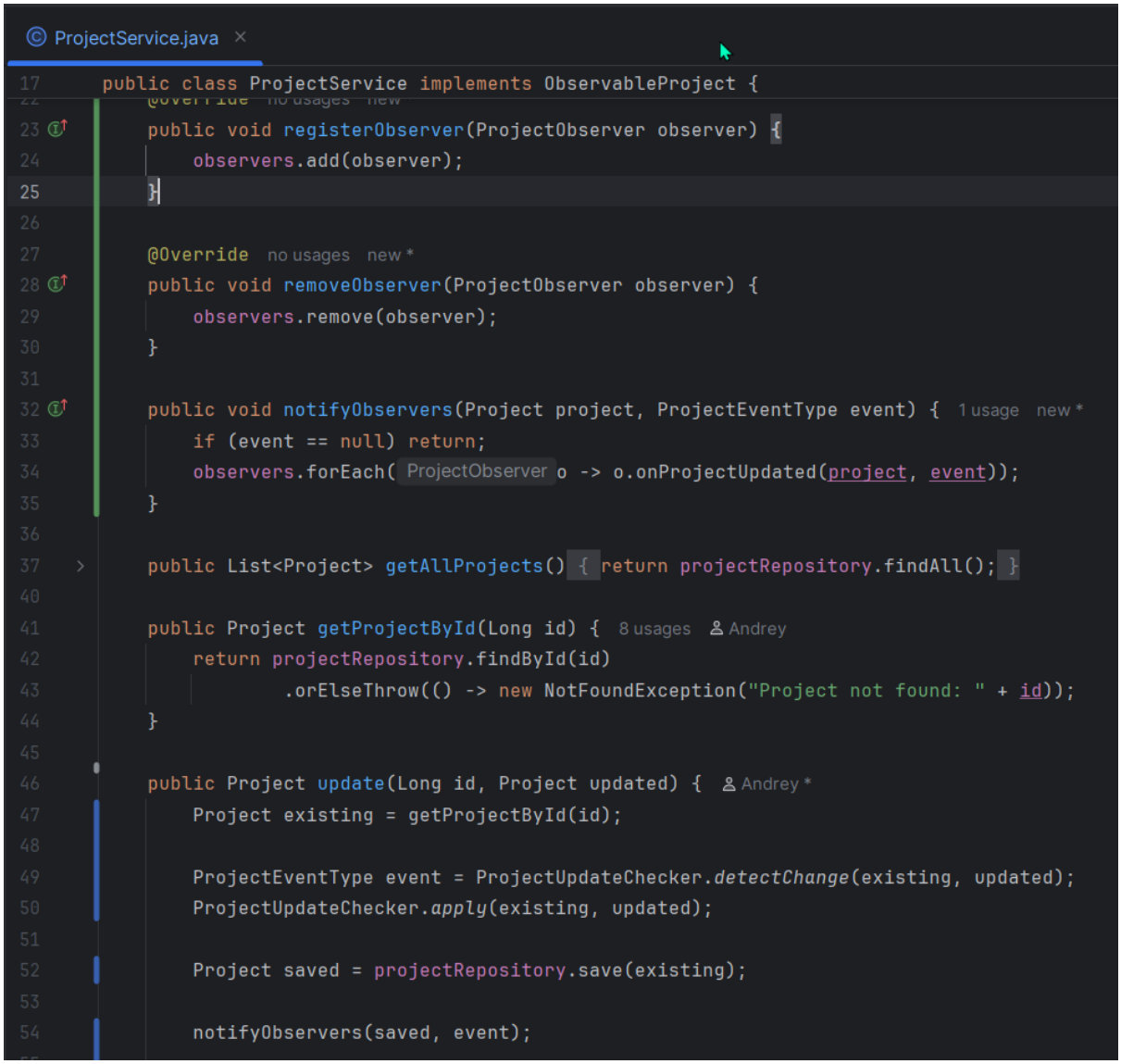


Рис 5 - Змінені методи з класу ProjectService

Висновки

Під час виконання лабораторної роботи були опрацьовані шаблони Abstract Factory, Factory Method, Memento, Observer та Decorator. Для кожного з них було визначено основні принципи роботи, особливості використання, переваги та можливі недоліки. На основі аналізу архітектури програмної системи було встановлено, що найбільш доцільним для впровадження є Observer.

Використання Observer у модулі управління проєктами дозволило відокремити логіку опрацювання подій від основних бізнес-класів, уникнути надмірної зв’язності та створити механізм, який легко розширюється додаванням нових спостерігачів. Такий підхід підвищує масштабованість і гнучкість системи, робить її архітектуру чіткішою й простішою в підтримці.

Контрольні запитання

1. Яке призначення шаблону «Абстрактна фабрика»?

Створювати сімейства пов’язаних об’єктів без прив’язки до їх конкретних класів.

2. Нарисуйте структуру шаблону «Абстрактна фабрика».

Клієнт → Абстрактна фабрика → Конкретні фабрики → Конкретні продукти.

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

- Абстрактна фабрика – оголошує методи створення продуктів.
- Конкретні фабрики – реалізують створення конкретних продуктів.
- Абстрактні продукти – інтерфейси продуктів.
- Конкретні продукти – реалізація цих інтерфейсів.
- Клієнт використовує тільки абстракції і не залежить від конкретних класів.

4. Яке призначення шаблону «Фабричний метод»?

Перенести логіку створення об’єкта в підкласи, дозволяючи їм вирішувати, який продукт створювати.

5. Нарисуйте структуру шаблону «Фабричний метод».

Творець (Creator) → метод factoryMethod() → Конкретні творці → Конкретні продукти.

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

- Creator – оголошує фабричний метод.

- Concrete Creator – перевизначає фабричний метод і створює конкретний продукт.
- Product – інтерфейс продукту.
- Concrete Product – реалізація продукту.
- Creator працює через інтерфейс продукту, не знаючи деталі конкретних класів.

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

Абстрактна фабрика створює **цілі сімейства** продуктів.

Фабричний метод створює **один продукт**, але делегує вибір підкласам.

Абстрактна фабрика використовує композицію; фабричний метод – наслідування.

8. Яке призначення шаблону «Знімок»?

Зберігати та відновлювати внутрішній стан об’єкта, не порушуючи інкапсуляції.

9. Нарисуйте структуру шаблону «Знімок».

Походжає так:
Originator → створює Memento
Caretaker → зберігає Memento
Memento → містить стан

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

- Originator – створює й відновлює стан.
- Memento – зберігає стан.
- Caretaker – тримає Memento, але не читає його.

- Originator віддає й бере стан через Memento, інші класи стан не бачать.

11. Яке призначення шаблону «Декоратор»?

Додавати нову функціональність об’єкту без зміни його класу, обгортаючи його в інші об’єкти.

12. Нарисуйте структуру шаблону «Декоратор».

Компонент (інтерфейс) → Конкретний компонент
↑
Декоратор (базовий клас) → Конкретні декоратори

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

- Component – інтерфейс.
- Concrete Component – основний об’єкт.
- Decorator – містить посилання на Component.
- Concrete Decorators – додають поведінку та викликають компонент через композицію.

14. Які є обмеження використання шаблону «декоратор»?

- Збільшується кількість дрібних класів.
- Важче дебажити, бо поведінка "розмазана" по обгортках.
- Порядок накладання декораторів має значення
- Може ускладнитися читання коду, якщо декораторів багато.