Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №4

з дисципліни «Технології розробки програмного забезпечення»

Вступ до паттернів програмування

Виконав:
студент групи ІА-34
Швець Роман Вадимович

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ - 2025

Тема: Вступ до паттернів програмування

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи.

Теоретичні відомості

Шаблони проектування – це стандартизовані рішення типових завдань, що виникають у процесі розробки програмного забезпечення. Вони не є готовим кодом, а скоріше описом архітектурного підходу, який можна реалізувати у будь-якій мові програмування. Використання шаблонів дозволяє підвищити повторне використання коду, узгодженість структури програми та спростити її подальшу модифікацію й підтримку.

Шаблони проектування поділяються на три основні категорії:

- Породжувальні (Creational) – відповідають за створення об’єктів і приховують логіку інстанціювання (наприклад, Singleton, Factory, Builder).
- Структурні (Structural) – описують способи поєднання класів і об’єктів у більші структури (наприклад, Adapter, Proxy, Composite).
- Поведінкові (Behavioral) – визначають способи взаємодії між об’єктами та розподіл відповідальності (наприклад, Strategy, State, Iterator).

Singleton (Одинак) – породжувальний шаблон, який забезпечує створення лише одного екземпляра класу та надає глобальну точку доступу до нього. Часто використовується для керування спільними ресурсами, такими як підключення до бази даних, налаштування програми або системний журнал. Основна ідея – зробити конструктор приватним і створювати екземпляр лише через статичний метод.

Iterator (Ітератор) – поведінковий шаблон, що забезпечує зручний спосіб послідовного доступу до елементів колекції без розкриття її внутрішньої реалізації. Ітератор інкапсулює логіку обходу, тому один і той самий інтерфейс може використовуватись для різних структур даних. Це підвищує узгодженість і спрощує роботу з колекціями.

Proxy (Проксі) – структурний шаблон, який створює об’єкт-замісник для іншого об’єкта з метою контролю доступу до нього. Проксі може виконувати додаткові функції – кешування результатів, перевірку прав доступу, відкладене

створення ресурсів або логування викликів. Таким чином, він додає поведінку без зміни основного класу.

State (Стан) – поведінковий шаблон, що дозволяє об’єкту змінювати свою поведінку залежно від поточного стану. Кожен стан оформлюється як окремий клас, а головний об’єкт делегує виконання завдань активному стану. Це робить код чистішим і усуває великі конструкції if або switch.

Strategy (Стратегія) – поведінковий шаблон, який дозволяє визначати групу схожих алгоритмів, інкапсулювати їх у власні класи та робити взаємозамінними. Клієнтський код може динамічно вибирати потрібну стратегію залежно від умов виконання. Такий підхід спрощує розширення системи – нові алгоритми можна додавати без зміни існуючого коду.

Хід роботи

Для реалізації функціональності графічного редактора було обрано шаблон проєктування «Стратегія» (Strategy), оскільки він найкраще підходить для задачі застосування різних ефектів до зображення. У цій системі користувач може обрати один із кількох алгоритмів обробки – чорно-білий, сепія або негатив – і застосувати його до відкритого файлу без зміни основної логіки програми. Завдяки використанню шаблону Strategy можна легко додавати нові ефекти, не змінюючи код контролера або сервісу, а лише створюючи новий клас, який реалізує спільний інтерфейс. Реалізація включає інтерфейс EffectStrategy, окремі класи для кожного ефекту, сервіс, що динамічно вибирає відповідну стратегію, контролер для обробки запитів і репозиторій для взаємодії з базою даних. Такий підхід забезпечує модульність, розширюваність і гнучкість системи.

Створимо діаграму класів, на якій зображено використання шаблону проектування Strategy у структурі системи графічного редактора. Вона демонструє взаємодію між основними компонентами програми – інтерфейсом користувача, контролером, сервісом ефектів, репозиторієм та класами зображень. Шаблон дозволяє динамічно змінювати алгоритм обробки зображення, підключаючи різні

стратегії ефектів без зміни основної логіки застосунку.

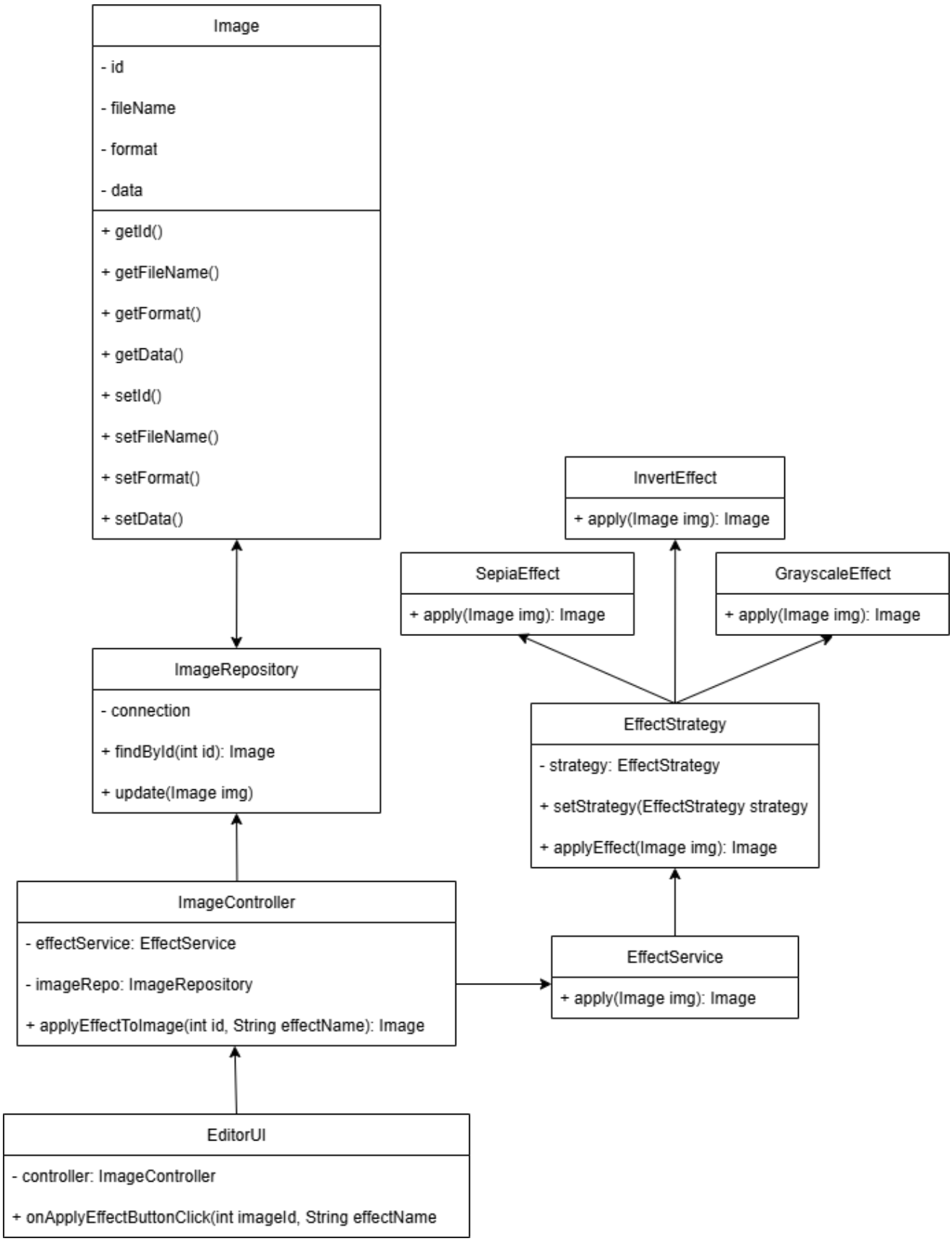

Рисунок 4.1 – Діаграма класів для Strategy

Як видно з діаграми, EditorUI ініціює обробку зображення через ImageController, який координує роботу між EffectService та ImageRepository. Клас EffectService реалізує вибір і застосування конкретної стратегії ефекту (GrayscaleEffect, SepiaEffect, InvertEffect), що наслідують спільний інтерфейс EffectStrategy. Така структура забезпечує гнучкість, розширюваність і полегшує підтримку системи, оскільки додавання нових ефектів не потребує зміни вже наявного коду.

Вихідні коди класів системи

Лістинг 1 – клас Image

```
import jakarta.persistence.*;

@Entity
@Table(name = "images")
public class Image {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "file_name", nullable = false)
    private String fileName;

    @Column(nullable = false)
    private String format; // SVG, EPS

    @Lob
    @Column(nullable = false)
    private byte[] data;

    public Image() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getFileName() { return fileName; }
    public void setFileName(String fileName) { this.fileName = fileName; }

    public String getFormat() { return format; }
    public void setFormat(String format) { this.format = format; }

    public byte[] getData() { return data; }
    public void setData(byte[] data) { this.data = data; }
}
```

Лістинг 2 – Інтерфейс EffectStrategy

```
public interface EffectStrategy {
    Image apply(Image img);
}
```

Лістинг 3 – клас GrayscaleEffect

```
public class GrayscaleEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: чорно-білий (Grayscale)");
        // Реалізація перетворення кольорів у градації сірого
        return img;
    }
}
```

Лістинг 4 – клас SepiaEffect

```
public class SepiaEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: сепія (Sepia)");
        // Реалізація перетворення зображення в сепію
        return img;
    }
}
```

Лістинг 5 – клас InvertEffect

```
public class InvertEffect implements EffectStrategy {

    @Override
    public Image apply(Image img) {
        System.out.println("Застосовано ефект: негатив (Invert)");
        // Реалізація інверсії кольорів
        return img;
    }
}
```

Лістинг 6 – клас EffectService

```
public class EffectService {

    private EffectStrategy strategy;

    public void setStrategy(EffectStrategy strategy) {
        this.strategy = strategy;
    }

    public Image applyEffect(Image img) {
        if (strategy == null) {
            throw new IllegalStateException("Ефект не вибрано!");
        }
        return strategy.apply(img);
    }
}
```

Лістинг 7 – клас Layer

```
import jakarta.persistence.*;

@Entity
@Table(name = "layers")
public class Layer {

    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @ManyToOne
    @JoinColumn(name = "image_id", nullable = false)
    private Image image;

    @Column(name = "layer_order")
    private int layerOrder;

    public Layer() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public Image getImage() { return image; }
    public void setImage(Image image) { this.image = image; }

    public int getLayerOrder() { return layerOrder; }
    public void setLayerOrder(int layerOrder) { this.layerOrder = layerOrder; }
}
```

Лістинг 8 – клас Collage

```
import jakarta.persistence.*;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name = "collages")
public class Collage {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(nullable = false)
    private String title;

    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
    @JoinColumn(name = "collage_id")
    private List<Layer> layers = new ArrayList<>();

    public Collage() {}

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }

    public List<Layer> getLayers() { return layers; }
    public void setLayers(List<Layer> layers) { this.layers = layers; }

    public void addLayer(Layer layer) {
        layers.add(layer);
    }

    public void removeLayer(Layer layer) {
        layers.remove(layer);
    }
}
```

Лістинг 9 – клас models.sql

```
CREATE TABLE Images (
    Id INTEGER PRIMARY KEY,
    FileName VARCHAR(255) NOT NULL,
    Format VARCHAR(50) NOT NULL,
    Data BLOB,
    EffectApplied VARCHAR(50)
);

CREATE TABLE Collages (
    Id INTEGER PRIMARY KEY,
    Title VARCHAR(255) NOT NULL
);

CREATE TABLE CollageImages (
    CollageId INTEGER NOT NULL,
    ImageId INTEGER NOT NULL,
    LayerOrder INTEGER,
    PRIMARY KEY (CollageId, ImageId),
    FOREIGN KEY (CollageId) REFERENCES Collages(Id),
    FOREIGN KEY (ImageId) REFERENCES Images(Id)
);
```

Лістинг 10 – Клас ImageRepository

```
import java.sql.*;

public class ImageRepository {

    private final Connection connection;

    public ImageRepository(Connection connection) {
        this.connection = connection;
    }

    public Image findById(int id) throws SQLException {
        String query = "SELECT * FROM Images WHERE Id = ?";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setInt(1, id);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            Image img = new Image();
            img.setId(rs.getInt("Id"));
            img.setFileName(rs.getString("FileName"));
            img.setFormat(rs.getString("Format"));
            img.setData(rs.getBytes("Data"));
            return img;
        }
        return null;
    }

    public void update(Image img) throws SQLException {
        String query = "UPDATE Images SET Data = ? WHERE Id = ?";
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setBytes(1, img.getData());
        stmt.setInt(2, img.getId());
        stmt.executeUpdate();
    }
}
```

Лістинг 11 – Клас ImageController

```
public class ImageController {

    private final ImageRepository imageRepo;
    private final EffectService effectService;

    public ImageController(ImageRepository repo, EffectService service) {
        this.imageRepo = repo;
        this.effectService = service;
    }

    public Image applyEffectToImage(int imageId, String effectName) throws
SQLException {
        Image img = imageRepo.findById(imageId);
        if (img == null) {
            throw new IllegalArgumentException("Image not found with ID: " +
imageId);
        }

        EffectStrategy strategy;
        switch (effectName.toLowerCase()) {
            case "grayscale":
                strategy = new GrayscaleEffect();
                break;
            case "sepia":
                strategy = new SepiaEffect();
                break;
            case "invert":
                strategy = new InvertEffect();
                break;
            default:
                throw new IllegalArgumentException("Unknown effect: " +
effectName);
        }

        effectService.setStrategy(strategy);
        Image updated = effectService.applyEffect(img);
        imageRepo.update(updated);
        return updated;
    }
}
```

Лістинг 12 – Імітація форми

```
public class EditorUI {

    private final ImageController controller;

    public EditorUI(ImageController controller) {
        this.controller = controller;
    }

    public void onApplyEffectButtonClick(int imageId, String effectName) {
        try {
            Image result = controller.applyEffectToImage(imageId, effectName);
            System.out.println("Ефект " + effectName + " успішно застосовано!");
        } catch (Exception e) {
            System.err.println("Помилка під час застосування ефекту: " +
e.getMessage());
        }
    }
}
```

У поданих лістингах реалізовано приклад використання шаблону

проєктування Strategy (Стратегія) в системі графічного редактора. Інтерфейс EffectStrategy визначає загальний метод apply() для застосування будь-якого ефекту, а класи GrayscaleEffect, SepiaEffect та InvertEffect реалізують цей інтерфейс, кожен — із власною логікою обробки зображення. Клас EffectService виступає контекстом стратегії, дозволяючи динамічно змінювати алгоритм застосування ефекту під час виконання програми. ImageController координує роботу сервісу та репозиторію, обираючи потрібну стратегію відповідно до дій користувача. ImageRepository забезпечує збереження й оновлення зображень у базі даних, а EditorUI імітує взаємодію користувача з інтерфейсом, ініціюючи застосування ефектів. Така структура демонструє чітке розділення відповідальностей і забезпечує гнучкість розширення функціональності програми без зміни її основного коду.

Висновок:

У ході виконання лабораторної роботи було розглянуто та реалізовано один із базових шаблонів проєктування – Strategy (Стратегія), застосований у контексті системи графічного редактора. Реалізація продемонструвала, як за допомогою цього шаблону можна відокремити алгоритми обробки зображень від основної логіки програми, забезпечивши можливість легко додавати нові ефекти без зміни існуючого коду. У проєкті створено класи для трьох ефектів (чорно-білий, сепія, негатив), сервіс обробки, контролер, репозиторій та інтерфейс користувача, що разом утворюють повний цикл взаємодії між рівнями системи – від введення даних користувачем до збереження змін у базі. Використання патерну Strategy підвищило модульність і масштабованість програми, зробивши архітектуру більш гнучкою, надійною та зручною для подальшого розвитку.

Відповіді на контрольні запитання:

1. Що таке шаблон проектування?

Шаблон проектування – це перевірене архітектурне рішення типових проблем програмування, яке описує стандартний спосіб організації класів і об’єктів для досягнення певної мети.

2. Навіщо використовувати шаблони проектування?

Вони допомагають створювати гнучкий, зрозумілий і підтримуваний код, зменшують дублювання рішень і полегшують спільну роботу розробників завдяки уніфікованим підходам.

3. Яке призначення шаблону «Стратегія»?

Шаблон Стратегія дає змогу змінювати алгоритм роботи об’єкта під час виконання, не змінюючи його структури. Клас делегує виконання певної операції об’єктам, які реалізують спільний інтерфейс.

4. Нарисуйте структуру шаблону «Стратегія».

Context → Strategy (інтерфейс)

Strategy ← ConcreteStrategyA

Strategy ← ConcreteStrategyB

5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

Context – клас, що містить посилання на стратегію.

Strategy – спільний інтерфейс алгоритмів.

ConcreteStrategy – конкретні реалізації стратегій.

Контекст викликає методи стратегії, не знаючи, яку саме реалізацію використовує.

6. Яке призначення шаблону «Стан»?

Шаблон Стан дозволяє об’єкту змінювати свою поведінку залежно від внутрішнього стану, створюючи враження, ніби він належить до різних класів.

7. Нарисуйте структуру шаблону «Стан».

Context → State (інтерфейс)

State ← ConcreteStateA

State ← ConcreteStateB

8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

Context – зберігає поточний стан і делегує йому поведінку.

State – інтерфейс для всіх можливих станів.

ConcreteState – реалізації поведінки для кожного стану.

Context може змінювати об’єкт стану під час виконання.

9. Яке призначення шаблону «Ітератор»?

Шаблон Ітератор забезпечує послідовний доступ до елементів колекції без розкриття її внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».

Iterator → ConcreteIterator

Iterator → Aggregate (інтерфейс колекції)

Aggregate ← ConcreteAggregate

11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

Iterator – інтерфейс для обходу колекції.

ConcreteIterator – реалізація логіки обходу.

Aggregate – інтерфейс колекції.

ConcreteAggregate – конкретна колекція, яка створює ітератор.

Ітератор послідовно повертає елементи колекції через стандартні методи (next(), hasNext()).

12. В чому полягає ідея шаблона «Одинак»?

Одинак (Singleton) гарантує, що у програмі існує лише один екземпляр певного класу, і надає глобальну точку доступу до нього.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Тому що він створює глобальний стан і сильну залежність між компонентами, що ускладнює тестування, масштабування і підтримку програми.

14. Яке призначення шаблону «Проксі»?

Шаблон Проксі надає сурогатний об’єкт, який контролює доступ до іншого об’єкта – наприклад, додаючи кешування, перевірку прав або відкладене створення.

15. Нарисуйте структуру шаблону «Проксі».

Client → Subject (інтерфейс)

Subject ← RealSubject

Subject ← Proxy → RealSubject

16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

Subject – спільний інтерфейс для реального об’єкта та проксі.

RealSubject – основний об’єкт, до якого здійснюється доступ.

Proxy – посередник, який керує доступом до RealSubject.

Клієнт працює з проксі, не знаючи, що звертається не напряму до справжнього об’єкта.