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

Лабораторна робота №3

з дисципліни «Технології розробки програмного забезпечення»

Основи проектування розгортання

Виконав:
студент групи ІА-34
Швець Роман Вадимович

Перевірив:
асистент кафедри ІСТ
Мягкий Михайло Юрійович

Київ - 2025

Тема: Основи програмування розгортання

Мета: Навчитися проєктувати діаграми розгортання та компонентів для системи що проєктується, а також розробляти діаграми взаємодії, а саме діаграми послідовностей, на основі сценаріїв зроблених в попередній лабораторній роботі.

Теоретичні відомості

Діаграма розгортання UML використовується для моделювання фізичної архітектури системи. Вона показує вузли, якими можуть бути комп'ютери, сервери, мобільні пристрої або середовища виконання, а також компоненти, що на них розгортаються. Між вузлами зображуються комунікаційні зв'язки, які вказують на канали передачі даних і протоколи взаємодії. Завдяки цьому можна побачити, як програмна система розподіляється на фізичні ресурси та які залежності існують між її частинами.

Діаграма компонентів описує архітектуру системи на логічному рівні, представляючи її у вигляді модулів – інтерфейсу користувача, контролерів, сервісів, сховища даних тощо. Компоненти відображають незалежні частини програми, які можуть розроблятися та замінюватися окремо, а діаграма показує їхні взаємозв'язки та залежності. Таким чином вона допомагає структурувати систему та чітко визначити ролі окремих елементів.

Діаграма послідовностей належить до групи діаграм взаємодії та використовується для моделювання сценаріїв роботи системи. Вона демонструє динамічну поведінку, тобто порядок обміну повідомленнями між об'єктами під час виконання певного варіанта використання. Вертикальна вісь діаграми відображає плин часу, що дозволяє простежити, у якій послідовності відбуваються дії. Такі діаграми деталізують варіанти використання та показують, як саме реалізуються бізнес-процеси або функції системи.

Усі ці діаграми взаємопов'язані: варіанти використання задають функціональні вимоги, діаграма класів показує логічну структуру, діаграма компонентів структурує програмну реалізацію за модулями, діаграма розгортання визначає фізичне середовище роботи, а діаграма послідовностей демонструє динаміку сценаріїв.

Хід роботи

Діаграма розгортання відображає фізичну архітектуру системи та показує, на яких пристроях і середовищах виконання працюють її основні компоненти. Вона дозволяє візуалізувати структуру системи на рівні апаратних і програмних вузлів, а також канали взаємодії між ними.

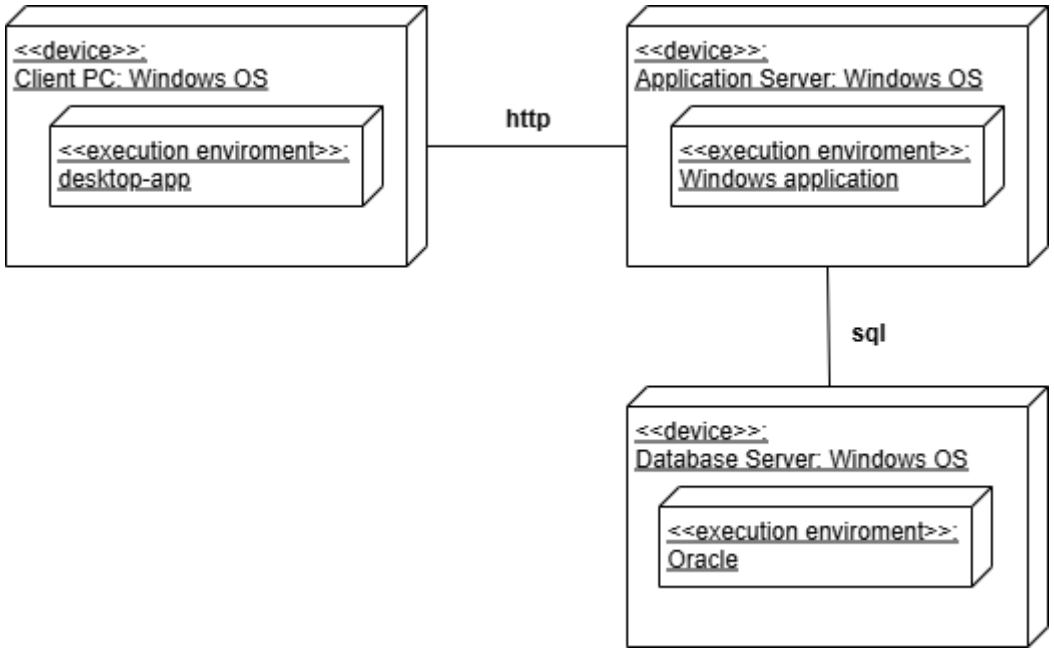

Рис. 1 – Діаграма розгортання

На діаграмі розгортання представлено три основні вузли системи: клієнтський комп’ютер із .exe-застосунком графічного редактора, сервер застосунку з бізнес-логікою та сервер бази даних SQLite. Клієнт відповідає за інтерфейс і взаємодію з користувачем, сервер застосунку виконує роль «посередника», що опрацьовує команди та координує роботу, а база даних виступає як сховище, де зберігається інформація про зображення, шари та колажі. Така архітектура нагадує команду: користувач – це капітан, який дає накази, сервер застосунку – офіцер, що організовує дії, а база даних — архів, де зберігаються всі документи й записи для виконання завдань.

Далі спроектуємо діаграму компонентів для нашого графічного редактора:

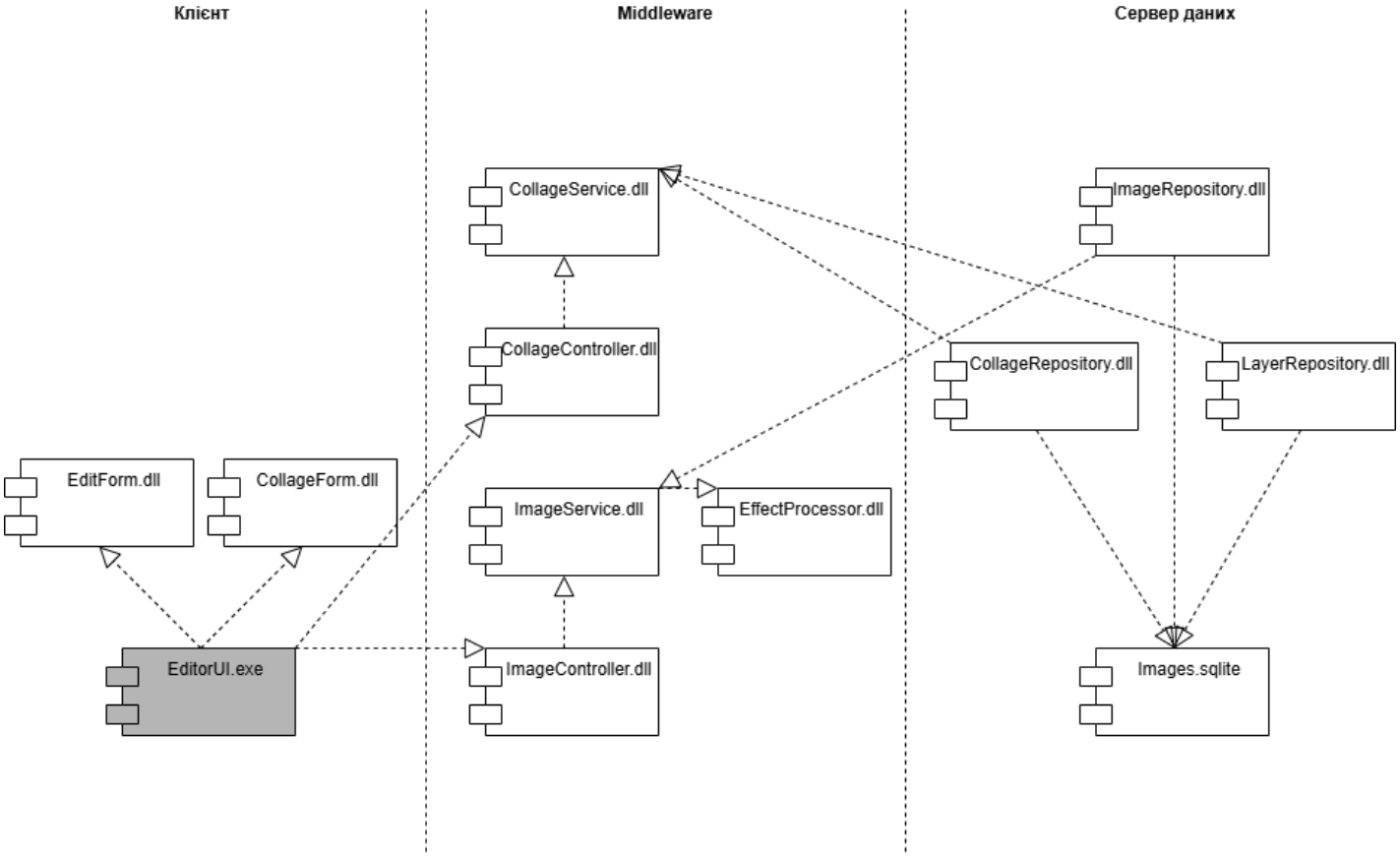

Рис. 2 – Діаграма компонентів

Діаграма компонентів відображає архітектуру графічного редактора та взаємодію його основних частин. На клієнтському рівні розташований виконуваний файл EditorUI.exe, який працює з контролерами ImageController.dll та CollageController.dll. Контролери викликають сервіси – ImageService.dll і CollageService.dll, що відповідають за обробку зображень, ефектів і колажів. Для застосування ефектів використовується окремий модуль EffectProcessor.dll. Збереження даних забезпечують репозиторії (ImageRepository.dll, CollageRepository.dll, LayerRepository.dll), які підключені до єдиної бази SQLiteDB.

Тепер створимо діаграму послідовностей для двох сценаріїв («Відкрити і редагувати зображення» та «Створити колаж»:

Варіант використання 1: «Застосувати ефект до зображення»

Передумови: Відкрите зображення у редакторі.

Постумови:

Якщо виконано успішно – система відображає зображення з вибраним ефектом (Grayscale, Sepia або Invert).

Якщо сталася помилка — зображення залишається без змін.

Взаємодіючі сторони: Користувач, Редактор зображень.

Короткий опис: Користувач застосовує один із доступних ефектів до відкритого зображення.

Основний потік подій:

1. Користувач відкриває зображення.
2. У меню вибирає потрібний ефект.
3. Система застосовує ефект до поточного зображення.
4. У вікні попереднього перегляду відображається результат.

Винятки:

Обрано ефект, несумісний із поточним форматом зображення → повідомлення про помилку.

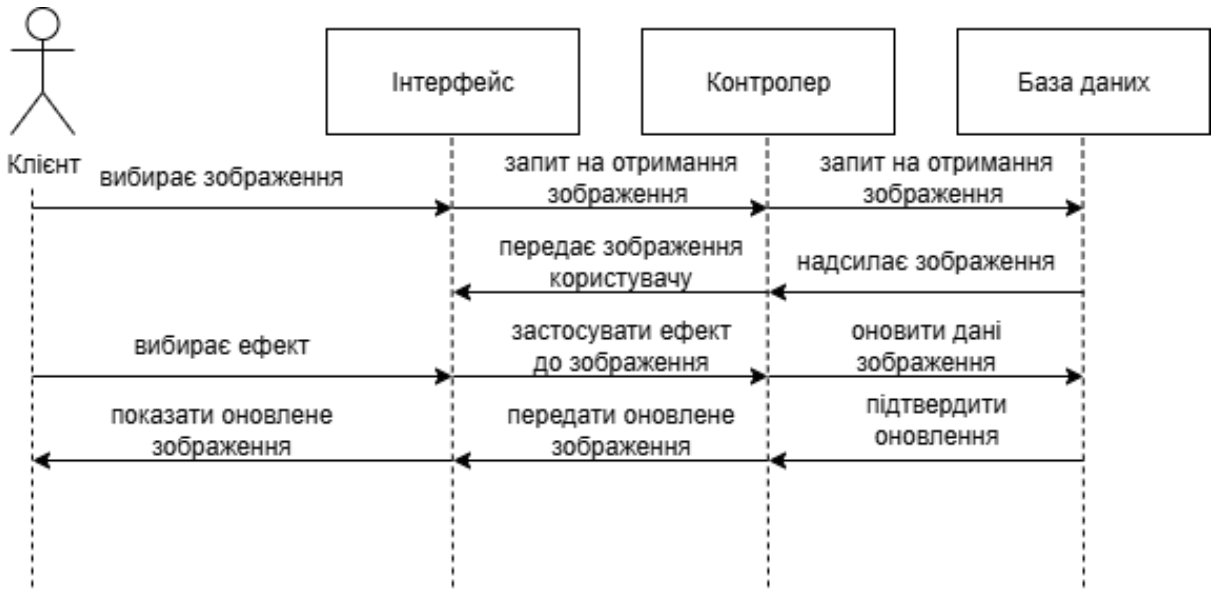

Рисунок 3 – Діаграма послідовностей (сценарій 1)

Варіант використання 3: «Створити колаж»

Передумови: Є кілька відкритих зображень або доданих шарів.

Постумови:

Якщо виконано успішно – створено колаж з обраних зображень/шарів.

Якщо сталася помилка – колаж не створюється.

Взаємодіючі сторони: Користувач, Редактор зображень.

Короткий опис: Користувач створює колаж, комбінуючи кілька шарів або зображень.

Основний потік подій:

1. Користувач запускає команду «Створити колаж».
2. Система пропонує вибрати зображення/шари для колажу.
3. Користувач додає потрібні елементи.
4. Система генерує колаж.
5. У вікні попереднього перегляду відображається результат.

Винятки:

Вибрано занадто мало зображень (наприклад, одне) → повідомлення про помилку.

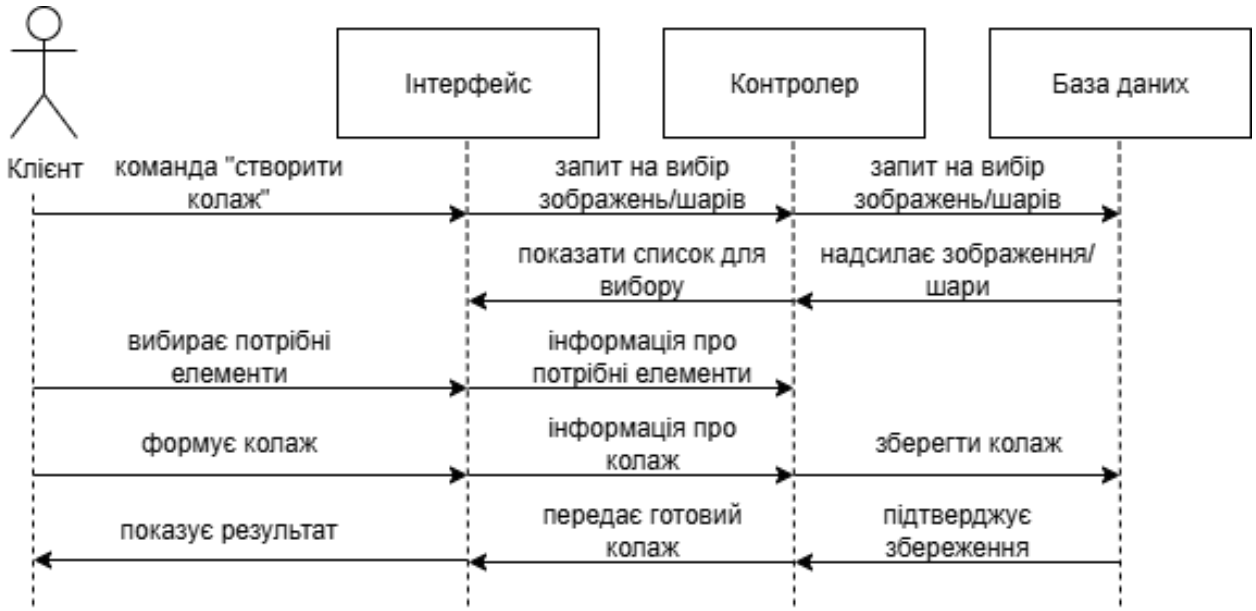

Рисунок 4 – Діаграма послідовностей (сценарій 3)

На діаграмах послідовностей зображено взаємодію між основними компонентами системи під час виконання трьох ключових сценаріїв: застосування ефектів, керування шарами та створення колажів.

Усі сценарії починаються з ініціативи користувача через графічний інтерфейс. Інтерфейс передає запити контролеру, який обробляє їх, звертається до бази даних і повертає результати для відображення. Така структура забезпечує логічний розподіл функцій між рівнями системи, узгодженість дій та зручність у подальшому масштабуванні застосунку.

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 – абстрактний клас Effect

```
import jakarta.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "effects")
public abstract class Effect {

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

    @Column(nullable = false)
    private String name;

    public Effect() {}

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

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Лістинг 3 – клас GrayscaleEffect

```
import jakarta.persistence.*;

@Entity
@Table(name = "grayscale_effects")
public class GrayscaleEffect extends Effect {

    @Column(nullable = false)
    private boolean applied;

    public GrayscaleEffect() {}

    public boolean isApplied() { return applied; }
    public void setApplied(boolean applied) { this.applied = applied; }
}
```

Лістинг 4 – клас SepiaEffect

```
import jakarta.persistence.*;

@Entity
@Table(name = "sepia_effects")
public class SepiaEffect extends Effect {

    @Column(nullable = false)
    private boolean applied;

    public SepiaEffect() {}

    public boolean isApplied() { return applied; }
    public void setApplied(boolean applied) { this.applied = applied; }
}
```

Лістинг 5 – клас InvertEffect

```
import jakarta.persistence.*;

@Entity
@Table(name = "invert_effects")
public class InvertEffect extends Effect {

    @Column(nullable = false)
    private boolean applied;

    public InvertEffect() {}

    public boolean isApplied() { return applied; }
    public void setApplied(boolean applied) { this.applied = applied; }
}
```

Лістинг 6 – 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 7 – 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 8 – 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 9 – Клас ImageRepository

```
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 10 – Клас EffectService

```
public class EffectService {

    public Image applyEffect(Image img, String effectName) {
        switch (effectName.toLowerCase()) {
            case "grayscale":
```

```
                return applyGrayscale(img);
            case "sepia":
                return applySepia(img);
            case "invert":
                return applyInvert(img);
            default:
                throw new IllegalArgumentException("Unknown effect: " +
effectName);
        }
    }

    private Image applyGrayscale(Image img) {
        // Логіка перетворення байтів у градації сірого
        return img;
    }

    private Image applySepia(Image img) {
        // Логіка сепії
        return img;
    }

    private Image applyInvert(Image img) {
        // Інверсія кольорів
        return img;
    }
}
```

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);
        }
        Image updated = effectService.applyEffect(img, effectName);
        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 повного циклу обробки зображення в системі графічного редактора. Клас ImageRepository відповідає за роботу з базою даних – пошук і оновлення зображень. Клас EffectService реалізує логіку застосування вибраних користувачем ефектів (чорно-білий, сепія, негатив). Клас ImageController координує обмін даними між інтерфейсом користувача, сервісами та репозиторієм, забезпечуючи обробку запитів і збереження результатів у базі даних. Нарешті, EditorUI моделює взаємодію користувача з системою, ініціюючи обробку зображення через контролер. Така архітектура демонструє розподіл функцій між рівнями застосунку – інтерфейсом, логікою та доступом до даних, що підвищує зручність підтримки та масштабування програми.

Висновок:

У ході виконання лабораторної роботи було розроблено діаграму розгортання, діаграму компонентів та дві діаграми послідовностей для ключових сценаріїв роботи графічного редактора: застосування ефектів та створення колажу. На основі цих моделей було вдосконалено архітектуру системи та реалізовано приклад повного циклу обробки даних – від дії користувача до збереження змін у базі даних. Додані класи контролера, сервісів та репозиторіїв забезпечують чітке розділення відповідальності між рівнями системи. Отримані результати дозволили сформувати завершену модель програмного застосунку, що поєднує структурне, поведінкове та фізичне представлення компонентів графічного редактора.

Відповіді на контрольні запитання:

1. Що собою становить діаграма розгортання?

Діаграма розгортання (Deployment Diagram) – це UML-діаграма, яка показує фізичне розміщення програмних компонентів системи на апаратних вузлах. Вона описує, як програма функціонує у реальному середовищі – на яких пристроях,

серверах або клієнтських машинах розгорнуті її частини.

2. Які бувають види вузлів на діаграмі розгортання?

Основні типи вузлів – це апаратні вузли (наприклад, сервер, комп'ютер користувача, мобільний пристрій) і програмні вузли (середовище виконання, сервер додатків, база даних тощо).

3. Які бувають зв'язки на діаграмі розгортання?

На діаграмі розгортання використовуються зв'язки типу асоціації (для позначення взаємодії вузлів) і залежності (для показу відносин між компонентами, розгорнутими на цих вузлах).

4. Які елементи присутні на діаграмі компонентів?

На діаграмі компонентів зображаються компоненти системи (модулі, бібліотеки, сервіси), інтерфейси (точки взаємодії) та зв'язки залежності між ними.

5. Що становлять собою зв'язки на діаграмі компонентів?

Зв'язки на діаграмі компонентів відображають залежності між модулями – який компонент використовує інший або надає йому певний інтерфейс.

6. Які бувають види діаграм взаємодії?

До діаграм взаємодії належать діаграма послідовностей (Sequence Diagram) і діаграма комунікацій (Communication Diagram). Вони описують обмін повідомленнями між об'єктами системи.

7. Для чого призначена діаграма послідовностей?

Діаграма послідовностей показує порядок викликів методів та обмін повідомленнями між об'єктами або компонентами системи в межах певного сценарію використання.

8. Які ключові елементи можуть бути на діаграмі послідовностей?

Основні елементи – актор, об'єкти або класи, лінії життя (lifelines), повідомлення (виклики методів) і блоки управління (alt, loop, opt).

9. Як діаграми послідовностей пов'язані з діаграмами варіантів використання?

Діаграми послідовностей деталізують сценарії, що описані у діаграмах варіантів використання. Вони показують, як саме реалізується кожен варіант використання на рівні взаємодії об'єктів.

10. Як діаграми послідовностей пов’язані з діаграмами класів?

Діаграми послідовностей використовують класи, визначені на діаграмі класів, як активні об’єкти, що взаємодіють між собою. Таким чином, вони відображають динамічну поведінку об’єктів, створених на основі структурної моделі класів.