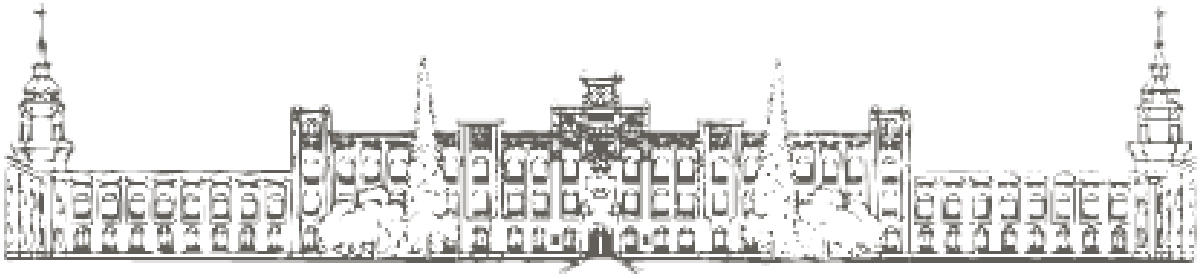


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Звіт лабораторної роботи №3

з курсу

«Технології розроблення програмного забезпечення»

Виконавець:
Нікітченко Наталя Олегівна
студентка групи ІА-33
залікова книжка № ІА-3318

«07» 11 2025 р.

Перевірив: **Мягкий М. Ю.**

Київ – 2025

3. ЛАБОРАТОРНА РОБОТА № 3

Тема: Основи проектування 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.

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

• Проаналізувати діаграми створені в попередній лабораторній роботі а також тему системи та спроєктувати діаграму розгортання використання відповідно до обраної теми лабораторного циклу.

• Розробити діаграму компонентів для проєктованої системи.

• Розробити діаграму розгортання для проєктованої системи.

• Розробити як мінімум дві діаграми послідовностей для сценаріїв прописаних в попередній лабораторній роботі.

• На основі спроєктованих діаграм розгортання та компонентів доопрацювати програмну частину системи. Реалізація системи, додатково до попередньої реалізації, повинна містити як мінімум дві візуальні форми. В системі вже повинен бути повністю реалізована архітектура (повний цикл роботи з даними від вводу на формі до збереження їх в БД і подальшій виборці з БД та відображенням на UI).

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму розгортання з описом, діаграму компонентів системи з описом, діаграми послідовностей, а також вихідний код системи, який було додано в цій лабораторній роботі.

Теоретичні відомості:

Діаграма компонентів (Component Diagram)

Діаграма компонентів описує логічну структуру програмної системи, тобто які модулі, підсистеми або компоненти її складають та як вони взаємодіють між собою. Вона показує залежності між частинами програми, а також інтерфейси, які ці компоненти надають або використовують. Такі діаграми відображають архітектуру високого рівня і демонструють, як програмні елементи об’єднані в компоненти та яким чином ці компоненти взаємозалежні.

Діаграма розгортання (Deployment Diagram)

Діаграма розгортання описує фізичне розміщення системи: на яких апаратних або програмних вузлах вона працює та як ці вузли з’єднані між собою. Вона показує, де саме виконуються програмні компоненти — на сервері, ПК, мобільному пристрої, віртуальній машині тощо. Така діаграма дає уявлення про інфраструктуру, середовище виконання, мережеві з’єднання та розподіл компонентів між апаратними елементами.

Діаграми послідовностей (Sequence Diagrams)

Діаграми послідовностей відображають динамічну поведінку системи, тобто порядок взаємодії між об’єктами під час виконання певного сценарію. Вони показують, які об’єкти беруть участь у процесі, які повідомлення або виклики методів передаються між ними, та у якій послідовності це відбувається. Такі діаграми дозволяють детально описати логіку роботи системи в часі та пов’язати її з конкретними варіантами використання.

Хід роботи:

Діаграма компонентів для проєктованої системи(Рис.1):

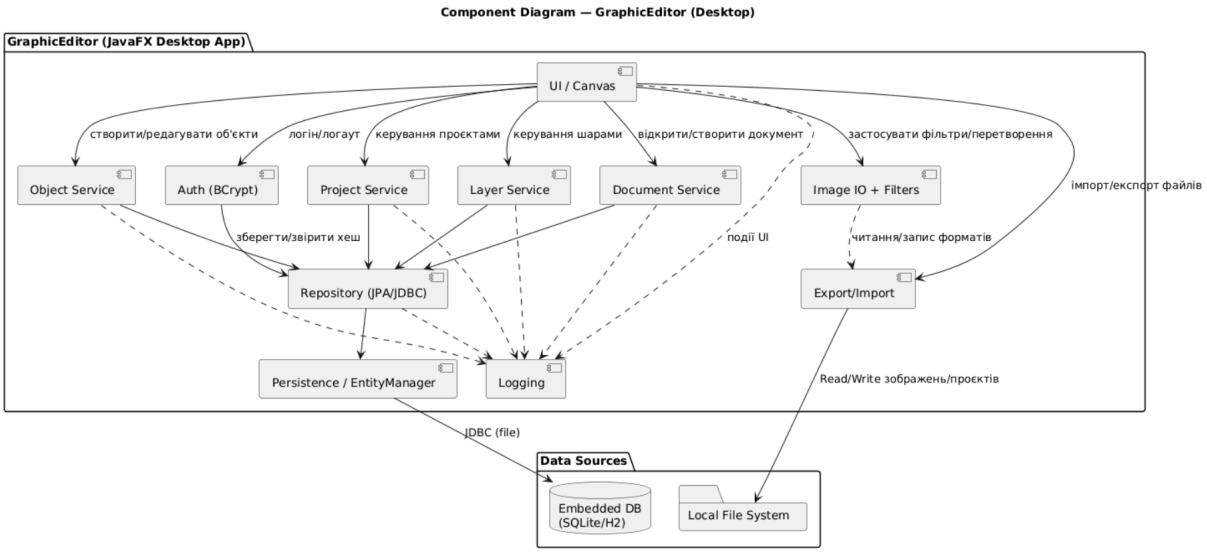


Рис.1 -Діаграма компонентів для проєктованої системи

Діаграма компонентів відображає внутрішню архітектуру настільного графічного редактора GraphicEditor, побудованого за модульним підходом. Центральним елементом системи є компонент UI/Canvas, який забезпечує взаємодію користувача з інтерфейсом, обробку команд та відображення графічного полотна. Інтерфейс напряму використовує сервіси системи: *Project Service*, *Document Service*, *Layer Service* та *Object Service*, які відповідають за керування проєктами, документами, шарами та графічними об’єктами відповідно. Авторизація в застосунку реалізована компонентом Auth (BCrypt), який здійснює хешування та перевірку паролів і працює через компонент Repository (JPA/JDBC), що забезпечує доступ до локальної бази даних.

Компонент Image IO + Filters відповідає за відкриття, збереження та обробку зображень різних форматів (JPEG, PNG, TIFF, PSD тощо) та передає ці операції компоненту Export/Import, який взаємодіє з локальною файловою системою для імпорту вихідних файлів та експорту результатів

редагування. Для збереження даних застосунок використовує компонент Persistence/EntityManager, який працює через JDBC-файлове підключення до вбудованої бази даних *SQLite/H2*. Усі компоненти надсилають службові події до модуля Logging, який записує інформацію про роботу програми.

У такий спосіб діаграма демонструє чітку багаторівневу структуру: інтерфейс користувача → бізнес-логіка (сервіси) → доступ до даних → файлове та базове зберігання. Архітектура забезпечує розширюваність, простоту модифікації та зручність локальної роботи без необхідності підключення до сервера.

Діаграма розгортання для проєктованої системи:

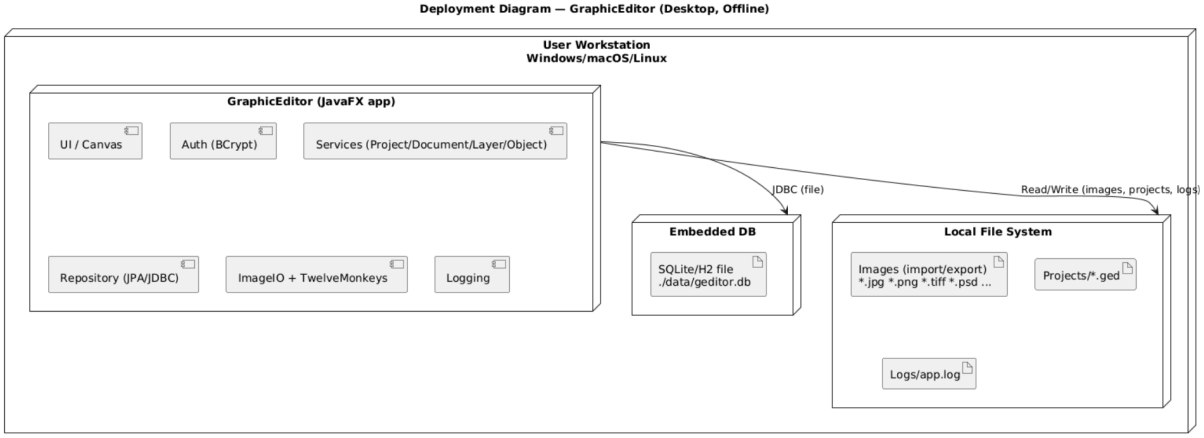


Рис.2 - Діаграма розгортання для проєктованої системи

Діаграма розгортання демонструє структуру та фізичне розміщення компонентів настільного застосунку GraphicEditor, який працює локально на робочій станції користувача (Windows/macOS/Linux). Застосунок реалізовано як автономний JavaFX-додаток, що містить інтерфейс користувача (UI/Canvas), модуль авторизації (BCrypt), сервіси бізнес-логіки (керування проєктами, документами, шарами та графічними

об’єктами), компонент для роботи з даними (Repository через JPA/JDBC), модуль обробки зображень (ImageIO + TwelveMonkeys) та підсистему журналювання (Logging).

Для зберігання внутрішніх даних система використовує вбудовану файлову базу даних SQLite або H2, до якої доступ здійснюється через JDBC у файловому режимі. Операції імпорту та експорту зображень, збереження проєктів у власному форматі (*.ged), а також ведення журналу подій виконуються через локальну файлову систему. Усі елементи застосунку функціонують на одному вузлі — робочому комп’ютері користувача, що забезпечує повністю офлайн-роботу, швидкий доступ до ресурсів і відсутність залежності від мережі чи серверної інфраструктури.

Діаграма послідовностей №1 — Створення нового проєкту та полотна(Рис.3):

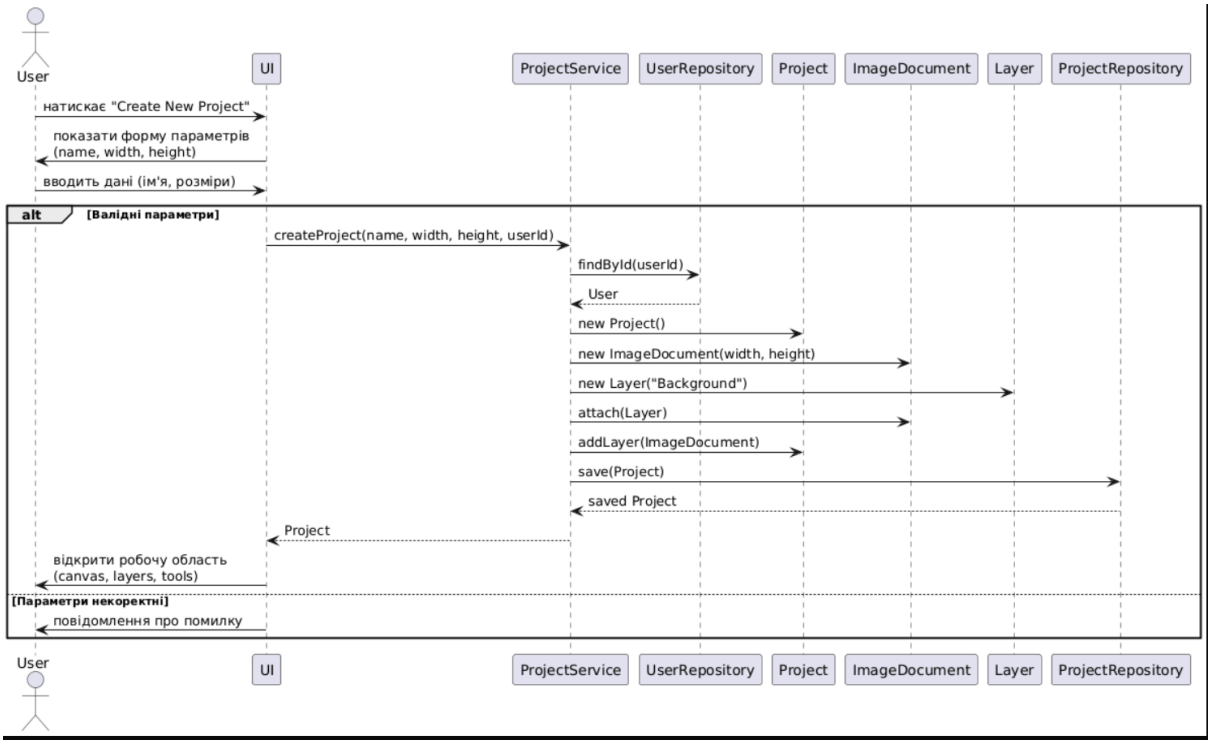


Рис.3 - **Діаграма послідовностей №1**

На поданій діаграмі послідовностей зображено процес створення нового проєкту в графічному редакторі. Сценарій починається з того, що користувач натискає кнопку **«Create New Project»**, після чого інтерфейс (UI) показує форму введення параметрів документа — назви проєкту та розмірів полотна. Користувач вводить необхідні дані, і якщо параметри коректні, UI передає їх у **ProjectService** методом createProject(...).

ProjectService отримує дані та викликає **UserRepository** для пошуку користувача за його ідентифікатором. Після повернення об’єкта User сервіс створює новий об’єкт **Project**, пов’язує з ним **ImageDocument** з указаними width/height та створює стартовий шар **Layer** (наприклад, "Background"). Створені модельні об’єкти зв’язуються між собою: документ прикріплюється до проєкту, шар додається до документа, а сам проєкт — у список проєктів користувача (user.projects.add(Project)).

Далі ProjectService зберігає проєкт через **ProjectRepository**, після чого повертає результат у UI, який відкриває робочу область редагування (canvas, layers, tools).

У випадку, якщо параметри документа некоректні, UI одразу повертає користувачеві повідомлення про помилку, і процес створення проєкту не продовжується.

Діаграма послідовностей №2 — Додавання фігури та тексту в активний шар(Рис.4):

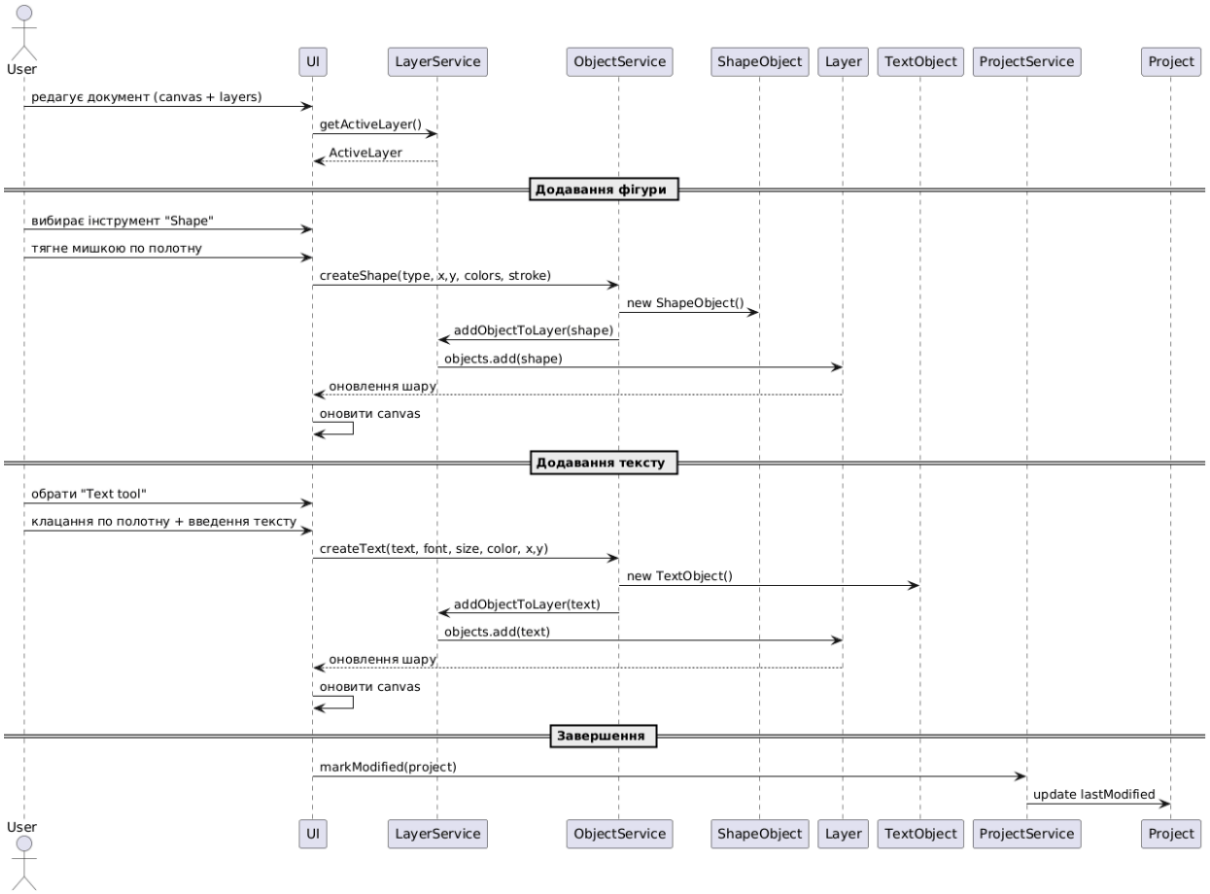


Рис.4 - **Діаграма послідовностей №2**

На діаграмі подано процеси, що відбуваються під час редагування документа в графічному редакторі, коли користувач додає фігуру та текстовий об’єкт у вибраний шар. Сценарій починається з того, що користувач працює у вікні редагування (canvas + список шарів), після чого UI запитує в LayerService активний шар. Після вибору інструмента «Shape» користувач тягне мишкою по полотну, і UI передає параметри створення фігури у LayerService методом createShape(...). LayerService

делегує створення нового графічного об’єкта в ObjectService, який ініціалізує новий ShapeObject і повертає його.

Сервіс шару додає нову фігуру до поточного шару (addObjectToLayer), а шар додає її у свій список об’єктів (objects.add(shape)), після чого UI оновлює відображення полотна.

Далі користувач вибирає інструмент «Text», клацає по полотну та вводить текст. UI викликає createText(...), і ObjectService створює новий TextObject, який так само додається у список об’єктів шару. Після кожного доданого елемента UI повторно рендерить оновлене полотно.

У фіналі UI викликає markModified(project), передаючи запит до ProjectService, який оновлює поле lastModified проєкту, позначаючи зміни як несохранені. Таким чином, діаграма відображає повну взаємодію між користувачем, UI, сервісами, моделями об’єктів і проєктом під час виконання операцій редагування шару.

На основі спроєктованих діаграм розгортання та компонентів доопрацювано програмну частину системи.

Було створено JpaBaseRepository.java(Рис.5):.

```java
package com.example.graphiceditor.repository;

import com.example.graphiceditor.persistence.JpaUtil;
import jakarta.persistence.EntityManager;
import java.util.List;

public abstract class JpaBaseRepository<T, ID> {  3 usages  3 inheritors  new *
    private final Class<T> type;  4 usages
    protected JpaBaseRepository(Class<T> type) { this.type = type; }  3 usages  new *

    public T save(T entity) {  6 usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try {
            em.getTransaction().begin();
            T merged = em.merge(entity);
            em.getTransaction().commit();
            return merged;
        } finally { em.close(); }
    }

    public T find(ID id) {  no usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try { return em.find(type, id); }
        finally { em.close(); }
    }

    public List<T> findAll() {  no usages  new *
        EntityManager em = JpaUtil.emf().createEntityManager();
        try {
            return em.createQuery( s: "select e from " + type.getSimpleName() + " e", type).getResultList();
        } finally { em.close(); }
    }
```

Рис.5 - JpaBaseRepository.java.

Вихідний код UI частини:

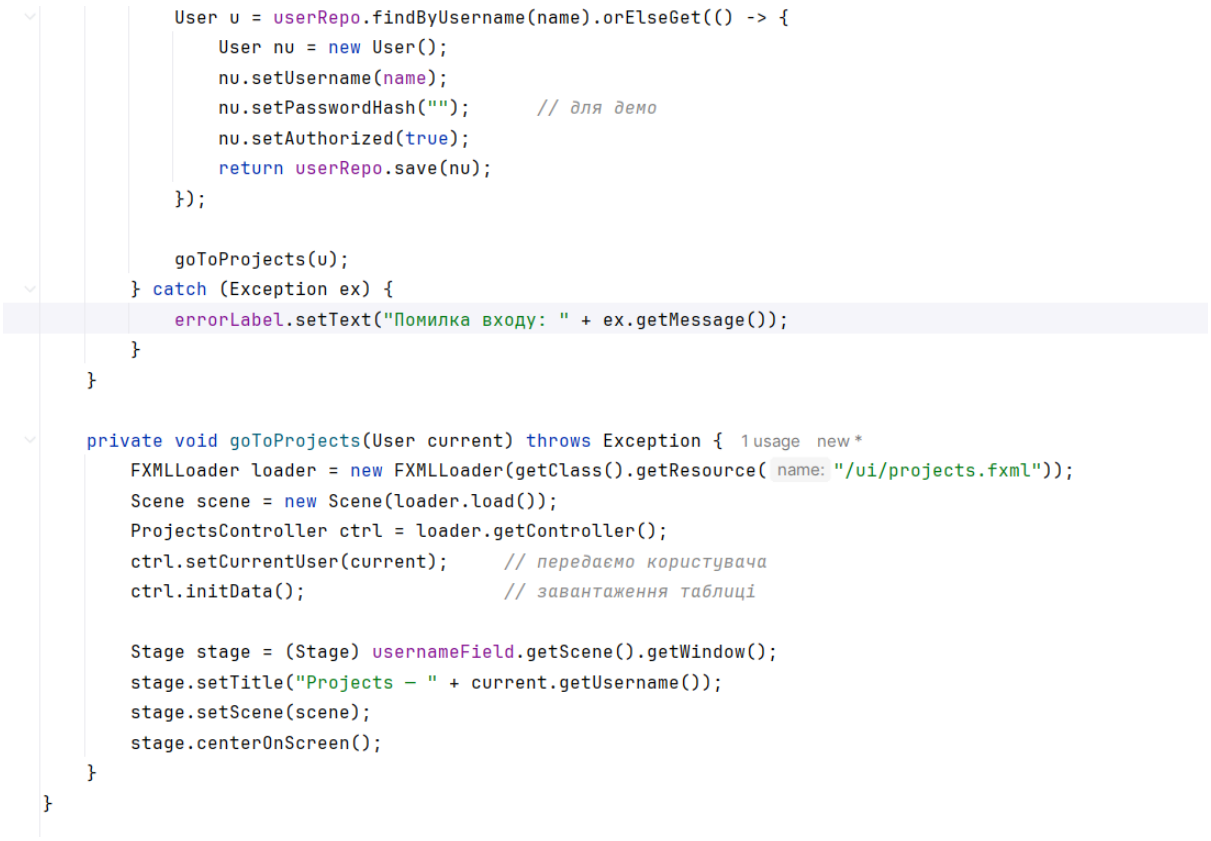
FXML-файли для двох форм:

- login.fxml
- projects.fxml

Java-контролери(Рис.6-12):

```java
package com.example.graphiceditor.ui;

import com.example.graphiceditor.model.User;
import com.example.graphiceditor.repository.UserRepository;
import com.example.graphiceditor.service.AuthService;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.Scene;
import javafx.fxml.FXMLLoader;
import javafx.stage.Stage;

public class LoginController {  1 usage  new *

    @FXML private TextField usernameField;
    @FXML private Label errorLabel;

    private final AuthService authService = new AuthService();  no usages
    private final UserRepository userRepo = new UserRepository();  2 usages

    @FXML  new *
    private void onLogin() {
        String name = usernameField.getText() == null ? "" : usernameField.getText().trim();
        if (name.isEmpty()) {
            errorLabel.setText("Введіть ім'я користувача");
            return;
        }
        try {
            User u = userRepo.findByUsername(name).orElseGet(() -> {
                User nu = new User();
                nu.setUsername(name);
                nu.setPasswordHash("");      // для демо
```

Рис.6 - **LoginController**



```
            User u = userRepo.findByUsername(name).orElseGet(() -> {
                User nu = new User();
                nu.setUsername(name);
                nu.setPasswordHash("");      // для демо
                nu.setAuthorized(true);
                return userRepo.save(nu);
            });

            goToProjects(u);
        } catch (Exception ex) {
            errorLabel.setText("Помилка входу: " + ex.getMessage());
        }
    }

    private void goToProjects(User current) throws Exception {  1 usage  new *
        FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/ui/projects.fxml"));
        Scene scene = new Scene(loader.load());
        ProjectsController ctrl = loader.getController();
        ctrl.setCurrentUser(current);     // передаємо користувача
        ctrl.initData();                  // завантаження таблиці

        Stage stage = (Stage) usernameField.getScene().getWindow();
        stage.setTitle("Projects — " + current.getUsername());
        stage.setScene(scene);
        stage.centerOnScreen();
    }
}
```

Рис.7 - **LoginController**

Клас LoginController(Рис.6-7), що розташований у пакеті com.example.graphiceditor.ui, реалізує логіку сценарію «Авторизація користувача та перехід до екрана управління проєктами» у клієнтській частині системи «Графічний редактор». Клас є контролером JavaFX-форми логіну та відповідає за обробку введених користувачем даних, валідацію і ініціацію переходу до наступного вікна застосунку.

До складу класу входять поля, зв’язані з елементами FXML-інтерфейсу: текстове поле usernameField для введення імені користувача та мітка errorLabel для відображення повідомлень про помилки. Також у класі створюються екземпляри сервісу автентифікації AuthService та репозиторію користувачів UserRepository, що інкапсулюють доступ до даних і бізнес-логіку роботи з об’єктами типу User.

Основний метод onLogin() виконує зчитування імені користувача з текстового поля, його нормалізацію (усунення порожніх значень і зайвих пропусків) та перевірку на непорожність. У разі коректного введення здійснюється пошук користувача в репозиторії за іменем; якщо запис не знайдено, створюється новий користувач із заданим ім’ям, порожнім хешем пароля (демонстраційний режим) та ознакою авторизованого доступу, після чого цей користувач зберігається. У випадку виникнення виняткових ситуацій (помилки доступу до даних тощо) текст помилки виводиться на інтерфейс через errorLabel.

Метод goToProjects(User current) відповідає за перехід до форми управління проєктами. Він завантажує FXML-ресурс projects.fxml за допомогою FXMLLoader, створює нову сцену, отримує контролер ProjectsController і передає йому поточного користувача через метод setCurrentUser(), а також викликає initData() для ініціалізації вмісту (наприклад, заповнення таблиці проєктів). Після цього на поточному вікні (Stage) змінюється сцена та заголовок, що відображає ім’я авторизованого користувача. Такий підхід забезпечує розділення відповідальностей між контролерами та узгоджений обмін даними між екранами програми.

```java
package com.example.graphiceditor.ui;

import com.example.graphiceditor.model.Project;
import com.example.graphiceditor.model.User;
import com.example.graphiceditor.repository.ProjectRepository;
import com.example.graphiceditor.service.EditorService;
import javafx.collections.FXCollections;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.layout.GridPane;
import javafx.util.StringConverter;

import java.time.format.DateTimeFormatter;
import java.util.List;

public class ProjectsController {  2 usages  new *

    @FXML private TableView<Project> table;
    @FXML private TableColumn<Project, String> nameCol;
    @FXML private TableColumn<Project, String> sizeCol;
    @FXML private TableColumn<Project, String> modifiedCol;
    @FXML private Label statusLabel;

    private final ProjectRepository projectRepo = new ProjectRepository();  1 usage
    private final EditorService editor = new EditorService();  1 usage

    private User currentUser;  4 usages


    public void setCurrentUser(User u) { this.currentUser = u; }  1 usage  new *
```

Рис.8 - **ProjectsController**

```java
@FXML  new *
private void initialize() {
    DateTimeFormatter fmt = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    nameCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
            .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() ->
                    cd.getValue().getName())));

    sizeCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
            .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() -> {
                var doc = cd.getValue().getDocument();
                return (doc == null) ? "-" : (doc.getWidth() + " x " + doc.getHeight());
            })));

    modifiedCol.setCellValueFactory(cd -> javafx.beans.property.SimpleStringProperty
            .stringExpression(javafx.beans.binding.Bindings.createStringBinding(() ->
                    cd.getValue().getLastModified() == null ? "-" : fmt.format(cd.getValue().getLastModified()))
}

public void initData() {  1 usage  new *
    refresh();
}
```

Рис.9 - **ProjectsController**

```
@FXML  3 usages  new *
public void refresh() {
    if (currentUser == null) return;
    List<Project> list = projectRepo.findByUserId(currentUser.getId());
    table.setItems(FXCollections.observableList(list));
    statusLabel.setText("Loaded " + list.size() + " projects");
}

@FXML  new *
private void onNewProject() {
    Dialog<ProjectParams> dlg = buildNewProjectDialog();
    dlg.showAndWait().ifPresent(params -> {
        try {
            Project p = editor.createProjectWithDocument(currentUser, params.name, params.w, params.h);
            statusLabel.setText("Created project: " + p.getName());
            refresh(); // перечитати з БД
        } catch (Exception ex) {
            new Alert(Alert.AlertType.ERROR,  s: "Помилка створення: " + ex.getMessage()).showAndWait();
        }
    });
}
```

Рис.10 - **ProjectsController**

```
@FXML  new *
private void onOpenProject() {
    Project sel = table.getSelectionModel().getSelectedItem();
    if (sel == null) { new Alert(Alert.AlertType.INFORMATION, s: "Оберіть проєкт").showAndWait(); return; }
    new Alert(Alert.AlertType.INFORMATION, s: "Відкриваємо: " + sel.getName()).showAndWait();
}

@FXML  new *
private void onDeleteProject() {
    Project sel = table.getSelectionModel().getSelectedItem();
    if (sel == null) { new Alert(Alert.AlertType.INFORMATION, s: "Оберіть проєкт").showAndWait(); return; }
    new Alert(Alert.AlertType.INFORMATION, s: "(демо) Видалення ще не реалізовано").showAndWait();
}
```

Рис.11 - **ProjectsController**

```
    private Dialog<ProjectParams> buildNewProjectDialog() {
        Dialog<ProjectParams> d = new Dialog<>();
        d.setTitle("New Project");

        Label nameL = new Label( s: "Name:");
        TextField nameF = new TextField( s: "My Project");

        Label wL = new Label( s: "Width:");
        Spinner<Integer> wSp = new Spinner<>( i: 1,  i1: 10000,  i2: 1280);

        Label hL = new Label( s: "Height:");
        Spinner<Integer> hSp = new Spinner<>( i: 1,  i1: 10000,  i2: 720);

        GridPane gp = new GridPane();
        gp.setHgap(10); gp.setVgap(10);
        gp.addRow( i: 0, nameL, nameF);
        gp.addRow( i: 1, wL, wSp);
        gp.addRow( i: 2, hL, hSp);
        d.getDialogPane().setContent(gp);

        d.getDialogPane().getButtonTypes().addAll(ButtonType.OK, ButtonType.CANCEL);
        d.setResultConverter(bt -> bt == ButtonType.OK
                ? new ProjectParams(nameF.getText(), wSp.getValue(), hSp.getValue())
                : null);
        return d;
    }

    private record ProjectParams(String name, int w, int h) {}

}
```

Рис.12 - **ProjectsController**

Клас ProjectsController(Рис.8-12), що розташований у пакеті com.example.graphiceditor.ui, відповідає за керування вікном роботи з проєктами користувача у системі «Графічний редактор». Він реалізує сценарії перегляду списку проєктів, створення нового проєкту та (у демонстраційному режимі) відкриття й видалення вибраного проєкту. Клас є JavaFX-контролером для FXML-форми, яка містить таблицю проєктів та службові елементи інтерфейсу.

До складу контролера входять елементи, позначені анотацією @FXML: таблиця TableView<Project> із трьома колонками (nameCol, sizeCol, modifiedCol) та мітка стану statusLabel для відображення службових повідомлень (наприклад, кількості завантажених проєктів). Для доступу до даних використовується репозиторій ProjectRepository, а для бізнес-логіки створення нового проєкту разом із документом — сервіс EditorService. Поле currentUser з методом setCurrentUser(User u) зберігає поточного авторизованого користувача і використовується як контекст для вибірки проєктів.

Метод initialize() викликається автоматично під час ініціалізації контролера і налаштовує відображення колонок таблиці. Для колонки назви проєкту прив'язується рядкове представлення Project.getName(). Колонка розміру формує текст у вигляді «ширина x висота» на основі пов'язаного документа (або «-», якщо документ відсутній). Колонка дати модифікації використовує форматування через DateTimeFormatter за шаблоном yyyy-MM-dd HH:mm:ss, а при відсутності дати відображає «-». Метод initData() викликає refresh(), який виконує завантаження списку проєктів поточного користувача з репозиторію (findByUserId) та встановлює отриманий список як вміст таблиці, одночасно оновлюючи statusLabel.

Обробники подій користувача реалізовані через методи, позначені @FXML. Метод onNewProject() формує діалогове вікно створення нового проєкту за допомогою приватного методу buildNewProjectDialog(), де користувач задає назву, ширину та висоту документа. Після підтвердження діалогу параметри пакуються у запис ProjectParams і передаються до сервісу editor.createProjectWithDocument(), який створює та зберігає новий проєкт; надалі викликається refresh() для оновлення таблиці. У разі помилки показується вікно Alert з повідомленням про помилку. Метод

onOpenProject() перевіряє наявність вибраного рядка в таблиці і в демонстраційному режимі виводить інформаційний діалог із назвою проєкту, що «відкривається». Метод onDeleteProject() наразі реалізований як заглушка й повідомляє користувачу, що видалення ще не реалізовано. Приватний метод buildNewProjectDialog() створює діалог із елементами введення (назва, ширина, висота) на основі GridPane, додає стандартні кнопки OK/CANCEL і налаштовує перетворення результату в екземпляр ProjectParams.

```
package com.example.graphiceditor.ui;

import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class ViewRouter {  4 usages  new *

    private static Stage primaryStage;  7 usages

    public static void setPrimaryStage(Stage stage) {  1 usage  new *
        primaryStage = stage;
    }

    public static void show(String fxmlPath, String title) {  1 usage  new *
        try {
            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPath));
            Parent root = loader.load();

            primaryStage.setTitle(title);
            primaryStage.setScene(new Scene(root));
            primaryStage.show();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
        }
    }
```

Рис.13 - **ViewRouter**

```
    public static <T> T showAndGetController(String fxmlPath, String title) {  no usages  new *
        try {
            FXMLLoader loader = new FXMLLoader(ViewRouter.class.getResource(fxmlPath));
            Parent root = loader.load();

            primaryStage.setTitle(title);
            primaryStage.setScene(new Scene(root));
            primaryStage.show();

            return loader.getController();
        } catch (Exception ex) {
            ex.printStackTrace();
            throw new RuntimeException("Cannot load FXML: " + fxmlPath, ex);
        }
    }
}
```

Рис.14 - **ViewRouter**

Клас ViewRouter(Рис.13-14), розташований у пакеті com.example.graphiceditor.ui, виконує роль централізованого навігатора між вікнами (формами) JavaFX у застосунку «Графічний редактор». Його основне призначення — інкапсулювати логіку завантаження FXML-ресурсів, створення сцен та відображення їх на головному вікні програми (primaryStage), забезпечуючи єдиний вхідний пункт для перемикання представлень (view).

У класі використовується статичне поле primaryStage, яке зберігає посилання на головний об’єкт Stage застосунку. Метод setPrimaryStage(Stage stage) призначений для початкової ініціалізації цього поля, зазвичай викликається з методу start(...) головного класу JavaFX. Таким чином, ViewRouter не створює нові вікна, а керує вже існуючим головним вікном, змінюючи його вміст (сцену) відповідно до запитів.

Метод show(String fxmlPath, String title) завантажує FXML-ресурс, розташований за шляхом fxmlPath, за допомогою FXMLLoader, створює кореневий вузол сцени (Parent root), після чого формує новий об’єкт Scene

і встановлює його на primaryStage. Додатково оновлюється заголовок вікна (setTitle(title)), а саме вікно відображається на екрані через виклик show(). У разі виникнення помилок завантаження FXML (некоректний шлях, помилка у файлі розмітки тощо) генерується виняток типу RuntimeException з пояснювальним повідомленням, а стек викликів виводиться у консоль для спрощення діагностики.

Узагальнений метод showAndGetController(String fxmlPath, String title) реалізує подібний сценарій, але додатково повертає контролер завантаженої FXML-форми. Після завантаження ресурсу і встановлення нової сцени на primaryStage, метод викликає loader.getController() і повертає посилання на контролер, тип якого задається як параметр-узагальнення <T>. Це дозволяє викликаючому коду одразу отримати доступ до відповідного контролера (наприклад, для передачі даних, ініціалізації стану або виклику спеціалізованих методів), не порушуючи інкапсуляції логіки відображення. Такий підхід сприяє централізації навігації між вікнами та підвищує узгодженість роботи графічного інтерфейсу.

Висновки:

У ході виконання лабораторної роботи було реалізовано повноцінну програмну частину клієнтської частини системи GraphicEditor, яка включає роботу з даними, реалізацію бізнес-логіки, репозиторіїв, сервісів та побудову графічного інтерфейсу через JavaFX.

Було створено та налаштовано архітектуру застосунку, що охоплює всі рівні: модель, сховище даних (JPA/Hibernate), сервер логіки (сервіси) та

користувацький інтерфейс. Застосунок забезпечує повний цикл роботи з даними: створення користувача, створення проєкту з графічним документом, додавання шарів, збереження цих даних у базі, а також подальше їх отримання та відображення в UI.

Було реалізовано дві візуальні форми відповідно до вимог:

- LoginForm для входу користувача;
- ProjectsForm для перегляду, створення та керування проєктами авторизованого користувача.

Для навігації між формами створено компонент ViewRouter, який інкапсулює логіку перемикання сцен та дозволяє контролерам відкривати нові вікна без дублювання коду.

Усі діаграми — розгортання, компонентів, послідовностей — використані як основа для побудови системи, а структура проєкту відповідає сучасним принципам модульності, перевикористання та розділення відповідальностей.
Реалізована система демонструє коректну взаємодію UI з бізнес-логікою та нормально функціонує як повноцінний застосунок.

Контрольні питання:

1.Діаграма розгортання — це UML-діаграма, яка відображає фізичне розміщення програмних компонентів системи на апаратних та програмних вузлах. Вона показує, на яких пристроях працює система, які середовища виконання використовуються, а також як ці вузли з’єднані між собою.

2.На діаграмі розгортання розрізняють два основні види вузлів: апаратні вузли (наприклад, ПК, сервер, мобільний пристрій) та програмні або

віртуальні вузли (контейнери виконання, операційні системи, JVM, Docker-контейнери).

3.На діаграмі розгортання використовують два основні типи зв’язків: комунікаційні зв’язки, які показують мережеву взаємодію між вузлами, та зв’язки розгортання, які відображають, які компоненти встановлені або працюють на конкретному вузлі.

4.На діаграмі компонентів присутні самі компоненти системи, їхні інтерфейси, реалізації, залежності, а також допоміжні артефакти, бібліотеки чи модулі, від яких залежить система.

5.Зв’язки на діаграмі компонентів представляють собою залежності між компонентами. Вони показують, який компонент використовує інший, які інтерфейси надаються або вимагаються, а також схему взаємодії між частинами системи.

6.До діаграм взаємодії у UML належать діаграми послідовностей, діаграми комунікації (колаборації), діаграми часу та оглядові діаграми взаємодії.

7.Діаграма послідовностей призначена для відображення порядку взаємодії між об’єктами у часі. Вона показує, які об’єкти беруть участь у виконанні сценарію та які повідомлення надсилаються між ними.

8.На діаграмі послідовностей можуть бути такі ключові елементи: об’єкти та їхні життєві лінії, повідомлення між об’єктами, повернення результатів, альтернативні гілки, цикли, опціональні блоки, а також області активності.

9.Діаграми послідовностей тісно пов’язані з діаграмами варіантів використання, оскільки для кожного варіанта використання зазвичай створюється відповідна діаграма послідовностей. Вона деталізує, як саме реалізується конкретний сценарій, описаний у діаграмі варіантів використання.

10.Діаграми послідовностей пов’язані з діаграмами класів тим, що об’єкти, які фігурують у діаграмах послідовностей, є екземплярами класів із діаграми класів. Виклики методів у діаграмі послідовностей підтверджують наявність цих методів у класах, а взаємодія між об’єктами відповідає зв’язкам між класами.