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

ЛАБОРАТОРНА РОБОТА №2

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Основи проєктування»

Виконав:

студент групи ІА-33

Грицай Андрій

Перевірив:

асистент кафедри ІСТ

Мягкий Михайло Юрійович

Київ - 2025

Тема: Основи проектування.
Мета: Обрати зручну систему побудови UML-діаграм та навчитися будувати діаграми варіантів використання для системи що проєктується, розробляти сценарії варіантів використання та будувати діаграми класів предметної області.
Тема лабораторного циклу:

1. **Музичний програвач** (iterator, command, memento, facade, visitor, client-server)

Музичний програвач становить собою програму для програвання музичних файлів або відтворення потокової музики з можливістю створення, запам'ятовування і редагування списків програвання, перемішування/повторення (shuffle/repeat), розпізнавання різних аудіо-форматів, еквалайзер.

Теоретичні відомості

У даній лабораторній роботі для проєктування програмної системи "Музичний програвач" були застосовані фундаментальні концепції об'єктно-орієнтованого аналізу та проєктування (ООАП). Це дозволило створити надійну, гнучку та підтримувану систему. Ключовими інструментами, які було використано, є уніфікована мова моделювання (UML) та принципи проєктування баз даних.

Рівні абстракції проєктування

UML — це загальновизнаний стандарт для візуального моделювання програмних систем. За допомогою UML ми можемо специфікувати, візуалізувати та документувати архітектуру програмного забезпечення на різних рівнях абстракції:

1. Концептуальний рівень

На цьому рівні система розглядається з точки зору користувача та бізнес-логіки. Для цього була розроблена діаграма варіантів використання (Use Case Diagram). Вона визначає основні функціональні вимоги системи та її взаємодію з різними акторами, такими як Користувач (User) та Гість (Guest).

2. Логічний рівень

Цей рівень описує статичну структуру та динамічну поведінку системи.

Для опису статичної структури була створена діаграма класів (Class Diagram). Вона деталізує ключові класи предметної області (User, Song, Playlist, Album, Artist) та їх взаємозв'язки, що є основою для подальшої реалізації. Діаграма також включає сутності-репозиторії (наприклад, UserRepository, SongRepository), які відповідають за взаємодію з базою даних.

Динамічна поведінка системи (послідовність дій) описується діаграмами послідовностей (не включено в цей звіт).

3. Фізичний рівень

На цьому рівні описується фізичне розгортання компонентів системи. У нашому випадку, це включає логічну структуру бази даних, яка була

спроєктована на основі логічної моделі. Діаграма сутностей-зв'язків (ER-діаграма) показує таблиці (наприклад, User, Song, Playlist, Album, Artist), їх первинні (PK) та зовнішні (FK) ключі, що забезпечує цілісність даних та ефективну організацію зберігання інформації.

Деталізація діаграм

Діаграма варіантів використання (Use Case Diagram)

Діаграма варіантів використання є відправною точкою в аналізі вимог. Вона фіксує функціональність системи та її межі. Основними елементами діаграми є:

Актор (Actor): Будь-яка зовнішня сутність, що взаємодіє із системою. В нашому випадку це Користувач, який має свій набір прав та можливостей.

Варіант використання (Use Case): Послідовність дій, яку система виконує для досягнення певної мети для актора. Наприклад, Play Song, Create Playlist, або Search Music.

Зв'язки (Relationships): Описують взаємодію між акторами та варіантами використання, а також залежності між самими варіантами використання:

Асоціація (показує, що актор ініціює варіант використання).

Включення (<<include>>) – показує, що один варіант використання обов'язково включає в себе поведінку іншого.

Розширення (<<extend>>) – додатковий крок, який може виконуватись за потреби.

Для усунення неоднозначності та надання чітких специфікацій для розробників, кожен варіант використання деталізується за допомогою сценарію використання.

Діаграма класів (Class Diagram)

Діаграма класів є основним інструментом для моделювання статичної структури системи. Вона візуалізує класи (User, Track, Playlist, Album, Artist), їхні атрибути (дані) та операції (методи), а також зв'язки між ними.

Атрибути та операції мають рівні видимості (+ public, - private, # protected), що відображає принципи інкапсуляції.

Відносини між класами є ключовим елементом діаграми:

Асоціація: Загальний зв'язок, що показує, що об'єкти класів взаємодіють.

Для зберігання даних системи була спроєктована логічна модель бази даних. Проєктування здійснюється через нормалізацію (наприклад, 1НФ, 2НФ, 3НФ) для уникнення дублювання та забезпечення цілісності даних.

Хід роботи

Аналіз вимог та проектування діаграми варіантів використання

На початковому етапі проєктування було проведено аналіз функціональних вимог до системи. Визначено, що розроблюваний музичний плеєр має надавати користувачеві набір інструментів для організації, пошуку та відтворення аудіоконтенту, а також керування списками відтворення.

На основі аналізу виділено ключового актора системи:

- Користувач – основний актор, який взаємодіє з плеєром: переглядає бібліотеку, керує відтворенням музики, створює та редагує плейлисти.

Основні варіанти використання, що описують функціональність системи:

Перегляд музичної бібліотеки – дозволяє користувачу отримати доступ до списку треків. Цей процес включає технічну дію Отримати індекс музичної бібліотеки. Функціонал може бути розширений варіантом Додати/видалити трек з музичної бібліотеки, який обов'язково ініціює Оновити індекс музичної бібліотеки.

Відтворення музики – основна функція системи, яка включає дію Вибрати трек. Вибір треку, у свою чергу, включає Запит на файл. Процес відтворення може бути розширений додатковими налаштуваннями:

- Використання еквалайзера – для налаштування частот звуку.
- Перемішування – для зміни порядку відтворення.
- Повторення треку – для циклічного програвання композиції.

Керування плейлистами – функція для організації власних списків відтворення. Вона розширюється наступними діями:

- Створення плейлиста
- Видалення плейлиста
- Додавання треку до плейлиста
- Видалення треку з плейлиста

Для відображення логічних зв'язків між варіантами використання було застосовано відношення `<<include>>` та `<<extend>>`. Наприклад, відтворення музики неможливе без вибору треку (include), але використання еквалайзера є опціональним розширенням (extend).

Результатом цього етапу є діаграма варіантів використання, яка візуалізує межі системи та взаємодію користувача з функціоналом музичного плеєра.

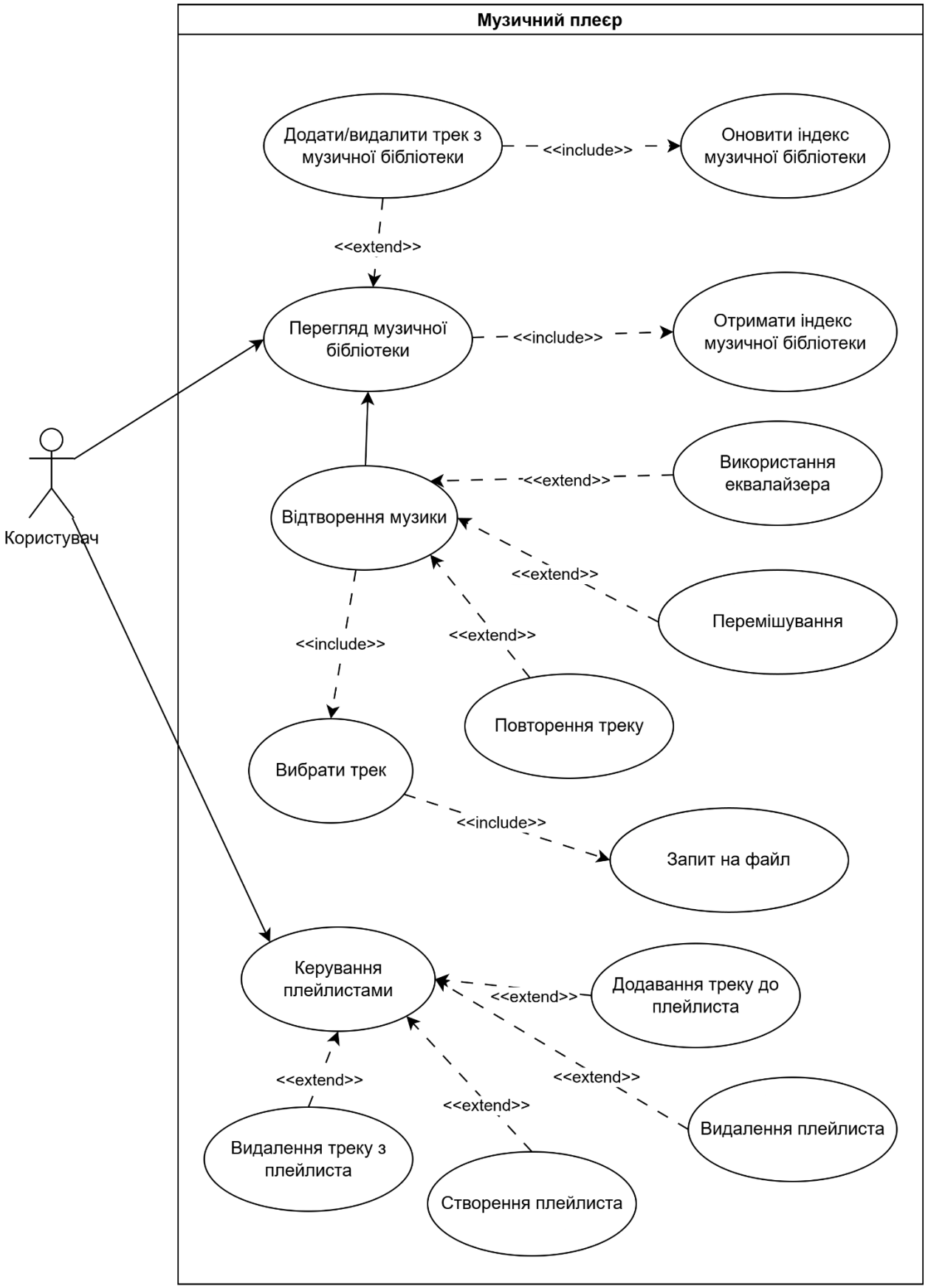

Рис. 1. Діаграма варіантів використання системи

Розробка сценаріїв варіантів використання

Сценарій №1: Відтворення музичного треку

- Передумови: Користувач має запущений музичний програвач і доступ до медіа-бібліотеки.
- Постумови:
    - Успіх: Обраний трек відтворюється, інформація про нього (назва, виконавець, обкладинка) відображається на екрані.
    - Провал: Відтворення не розпочато через відсутність файлу або помилку доступу.
- Взаємодіючі сторони: Користувач (User), Система “Music Player”.
- Короткий опис: Користувач обирає трек із бібліотеки для відтворення. Система завантажує файл, передає його до аудіоплеєра та запускає програвання.
- Основний перебіг подій:
    - Користувач відкриває бібліотеку пісень (View Media Library).
    - Система отримує список доступних треків (Retrieve Media Library Index).
    - Користувач обирає трек (Select Track for Playback).
    - Система завантажує вибраний файл (Request and Stream File).
    - Відтворення розпочинається (Manage Playback).
    - Користувач може скористатися еквалайзером (Use Equalizer) або ввімкнути випадкове/повторне відтворення (Shuffle/Repeat Playback).
- Винятки:
    - Виняток №1: На кроці 3 користувач скасовує вибір — операцію перервано.
    - Виняток №2: На кроці 4 система не може знайти або відкрити файл — повідомлення: «Файл недоступний або пошкоджений».
    - Виняток №3: На кроці 5 виникає помилка декодування аудіо — система повідомляє: «Непідтримуваний формат файлу».

Сценарій №2: Керування плейлистом (Оновлений)

- Передумови: Користувач запустив застосунок і має доступ до власної бібліотеки пісень.
- Постумови:
    - Успіх: Плейлист створено, оновлено або видалено відповідно до дій користувача.
    - Провал: Зміни не застосовано (наприклад, через скасування дії користувачем).
- Взаємодіючі сторони: Користувач (User), Система “Music Player”.
- Короткий опис: Користувач керує списками відтворення: створює нові, видаляє існуючі, а також додає або прибирає окремі треки з плейлиста.

- Основний перебіг подій:
    - Користувач відкриває розділ керування плейлистами (Manage Playlists).
    - Користувач обирає дію: створити новий список (Create Playlist) або редагувати існуючий.
    - Система відображає вміст обраного плейлиста та список доступних треків.
    - Користувач виконує одну з дій розширення:
        - Додає трек до списку (Add Track to Playlist).
        - Видаляє трек зі списку (Remove Track from Playlist).
    - Система зберігає зміни у структурі плейлиста.
    - Користувач отримує підтвердження про успішне оновлення плейлиста.
- Альтернативні потоки (Видалення плейлиста):
    - На кроці 2 користувач обирає видалення всього списку (Delete Playlist).
    - Система запитує підтвердження.
    - Після підтвердження плейлист зникає зі списку доступних.
- Винятки:
    - Виняток №1: На кроці 4 користувач намагається додати трек, який вже є в плейлисті — система повідомляє: «Трек вже додано».
    - Виняток №2: Під час збереження (крок 5) виникає системна помилка запису файлу плейлиста — система повідомляє: «Не вдалося зберегти зміни у плейлисті».

Сценарій №3: Оновлення бібліотеки пісень

- Передумови: Користувач запустив застосунок і бажає оновити вміст бібліотеки.
- Постумови:
    - Успіх: Медіа-бібліотека оновлена, додано нові пісні або видалено непотрібні.
    - Провал: Оновлення не виконано через помилку доступу до файлів.
- Взаємодіючі сторони: Користувач (User), Система “Music Player”.
- Короткий опис: Користувач додає нові пісні до бібліотеки або видаляє непотрібні. Система оновлює структуру бібліотеки та оновлює індекс.
- Основний перебіг подій:
    - Користувач обирає функцію Add/Remove Song from Library.
    - Система відкриває діалогове вікно для вибору файлів.
    - Користувач додає або видаляє обрані треки.
    - Система оновлює індекс бібліотеки (Update Media Library Index).
    - Оновлена бібліотека відображається у View Media Library.

- Винятки:
    - Виняток №1: Користувач натискає “Скасувати” → операцію перервано.
    - Виняток №2: Один або кілька файлів недоступні — система повідомляє: «Неможливо додати вибрані файли».
    - Виняток №3: На кроці 4 сталася помилка запису в бібліотеку — система повідомляє: «Помилка оновлення індексу медіа-бібліотеки».

Проектування діаграми класів предметної області

На наступному етапі проєктування була розроблена деталізована діаграма класів предметної області, що відображає не лише основні сутності системи, а і їхні атрибути та поведінку через визначені методи. Ця модель слугує логічним каркасом для майбутньої реалізації музичного програвача.

MusicPlayer (Музичний програвач) — центральна сутність, що керує процесом відтворення та станом системи. Клас містить атрибути для збереження поточних налаштувань, таких як гучність (volume), поточний час (startAtTime) та режими відтворення (isRepeat, isShuffleGlobal). Методи play(), pause(), stop(), next() та previous() забезпечують безпосередній контроль над програванням медіа. Крім того, клас відповідає за збереження та відновлення стану через методи saveState() та restoreState().

User (Користувач) — сутність, що представляє зареєстрованого користувача системи. Вона інкапсулює облікові дані: ім'я (name), електронну пошту (email) та пароль (password). Користувач володіє колекцією плейлистів, що відображено через композиційний зв'язок зі списком об'єктів Playlist.

Library (Бібліотека) — клас, що виступає основним сховищем аудіозаписів. Він реалізує інтерфейс Playable, що дозволяє програвати весь вміст бібліотеки. Клас містить колекцію треків та надає методи для маніпуляції ними: addTrack() для додавання нових композицій та removeTrack() для їх видалення.

Playlist (Плейлист) — сутність для організації користувацьких добірок музики. Як і бібліотека, реалізує інтерфейс Playable. Містить ідентифікатор, назву та список посилань на треки. Методи класу дозволяють додавати та видаляти треки зі списку відтворення.

Track (Трек) — базовий інформаційний об'єкт, що представляє окремий аудіофайл. Він містить метадані: назву (title), виконавця (artist), альбом (album), тривалість (duration) та фізичний шлях до файлу (filePath). Треки є основними елементами, що агрегуються в бібліотеці та на які посилаються плейлисти.

Допоміжні елементи:

Equalizer (Еквалайзер): Відповідає за налаштування параметрів звучання. Зберігає налаштування частотних смуг (currentGains) та надає методи для їх зміни (setBandGain()) або відновлення збережених налаштувань (restoreGains()).

Playable (Відтворюваний): Інтерфейс, що визначає контракт поведінки для будь-якої сутності, яку можна програвати (бібліотека або плейлист). Він уніфікує методи навігації (next(), previous()) та керування порядком відтворення (setShuffle(), isShuffle()).

Таким чином, розроблена діаграма класів чітко розподіляє обов'язки між об'єктами системи. Визначення методів та атрибутів дозволяє детально описати взаємодію між плеєром, медіатекою та користувачем, створюючи міцний фундамент для подальшої програмної реалізації архітектури музичного програвача.

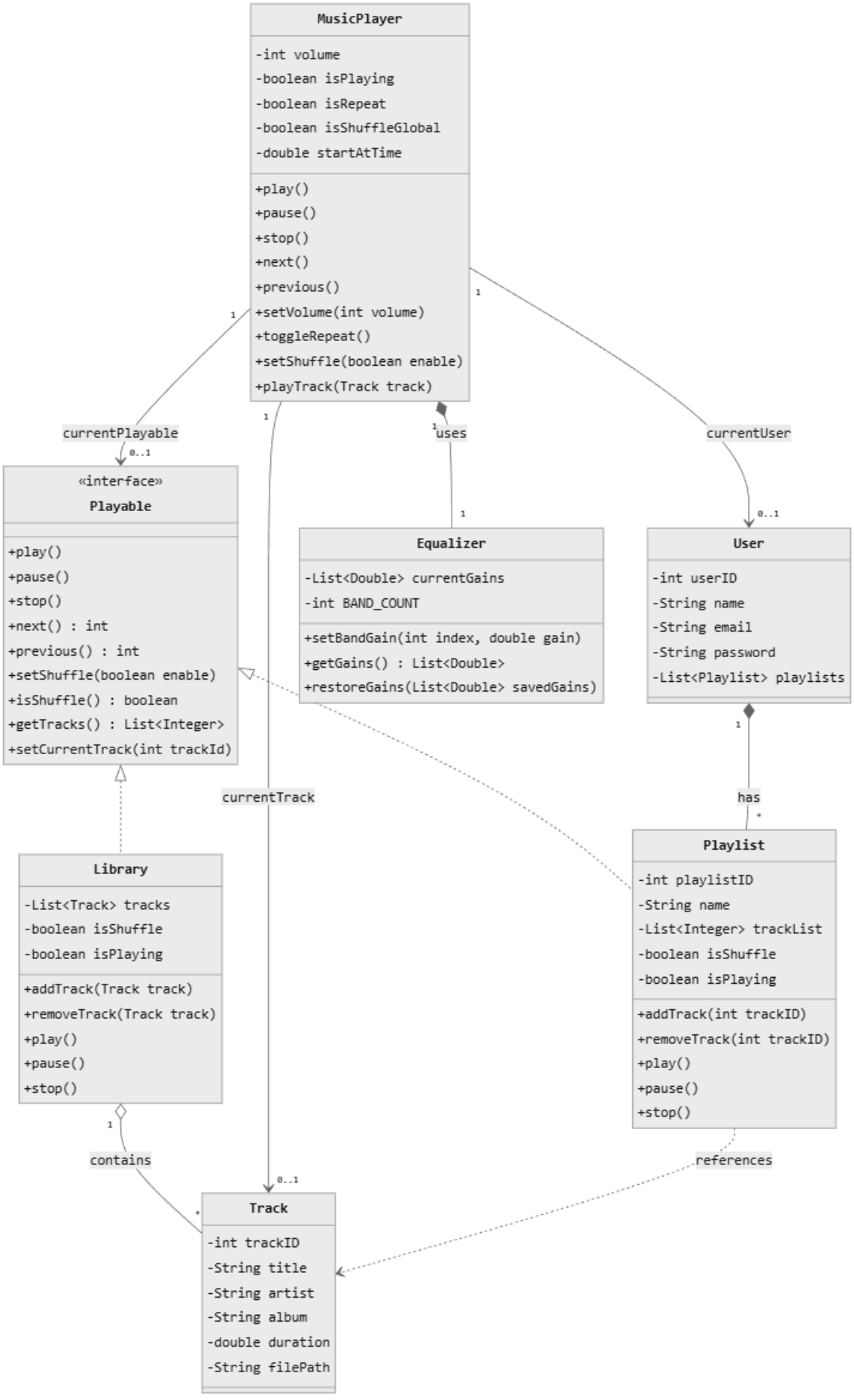

Рис. 2. Діаграма класів предметної області

Проектування структури бази даних та реалізація патерну Repository

Для зберігання та управління інформацією про користувачів, музичні композиції, плейлисти та їхні зв’язки була розроблена реляційна база даних, структура якої представлена на ER-діаграмі. Модель бази даних відображає основні сутності предметної області та способи їхньої взаємодії.

users — зберігає облікові дані користувачів системи: ім’я, унікальну електронну пошту та пароль для автентифікації.

tracks — містить детальну інформацію про аудіофайли, включаючи назву композиції, виконавця, альбом, тривалість відтворення та шлях до фізичного файлу в системі.

playlists — описує колекції треків (списки відтворення), створені користувачами, зберігаючи їх ідентифікатори та назви.

Між основними таблицями реалізовані зв’язки типу «багато-до-багатьох» за допомогою проміжних таблиць:

users_tracks — фіксує, які треки додав до своєї бібліотеки конкретний користувач.

users_playlists — визначає приналежність плейлистів конкретним користувачам.

playlist_tracks — зберігає інформацію про те, які треки входять до складу конкретного плейлиста.

Така структура забезпечує логічну цілісність даних, дозволяє одному треку бути частиною різних плейлистів та бібліотек різних користувачів без дублювання інформації, а також гарантує гнучкість у подальшому масштабуванні системи.

Для взаємодії з базою даних у проєкті реалізовано патерн Repository, який забезпечує відокремлення бізнес-логіки застосунку від низькорівневих деталей доступу до даних. Кожна сутність має власний репозиторій (UserRepository, TrackRepository, PlaylistRepository), що спрощує роботу з даними через уніфіковані інтерфейси. Це підвищує масштабованість, підтримуваність та зручність подальшої модифікації програмного коду.

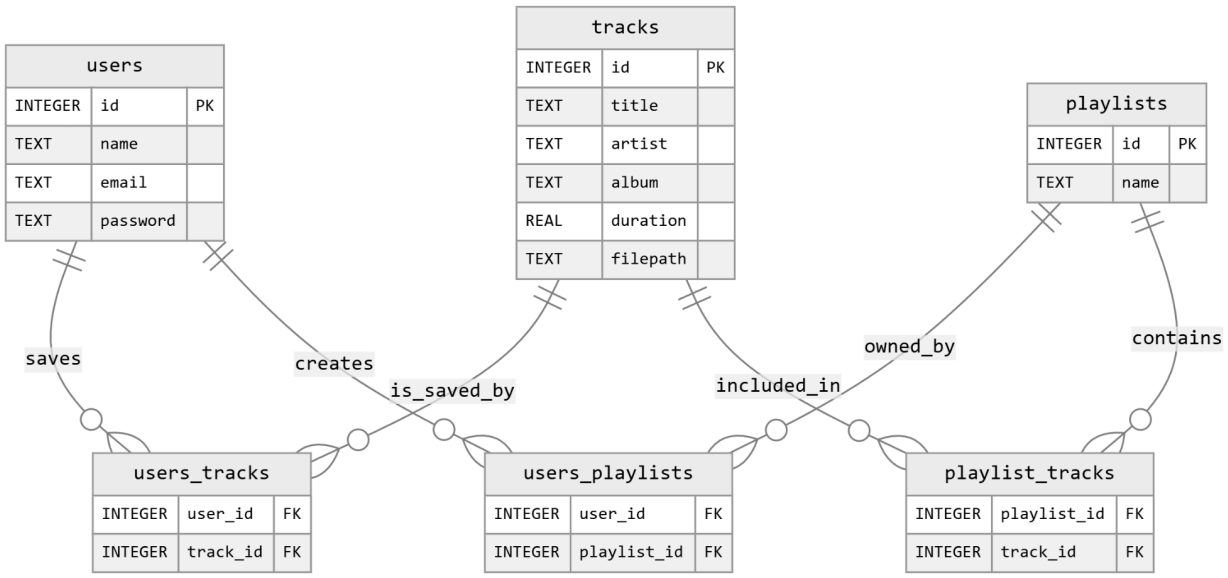

Рис. 3. Схема даних

Проектування діаграми класів реалізованої системи

Фінальна діаграма класів відображає архітектуру реалізованої частини системи, яка поєднує сутності предметної області, шар доступу до даних (репозиторії), ключові методи та зв’язки між об’єктами. Вона показує структуру програмного забезпечення з точки зору об’єктно-орієнтованого проєктування, демонструючи, як реалізована бізнес-логіка музичного програвача. Побудована модель забезпечує гнучкість, масштабованість та чітке розмежування відповідальності між компонентами.

Основні елементи діаграми:

- MusicPlayer – центральний клас, що керує процесом відтворення та станом системи. Зберігає налаштування гучності, поточний час треку та режими (повтор, перемішування). Реалізує методи управління медіа (play(), pause(), stop(), next()) та методи збереження стану (saveState()).
- User – клас, що зберігає інформацію про користувача (ідентифікатор, ім’я, email, пароль) та містить список створених ним плейлистів.
- Track – базовий елемент системи, який представляє аудіофайл. Містить метадані: назву, виконавця, альбом, тривалість та шлях до файлу.
- Playlist – сутність для організації користувацьких добірок музики. Реалізує інтерфейс Playable, що дозволяє програвати його вміст, та надає методи для додавання (addTrack()) або видалення треків.
- Library – представляє повну колекцію треків, доступних у системі. Як і плейлист, реалізує інтерфейс Playable.
- Equalizer – відповідає за налаштування параметрів звучання, зберігаючи рівні підсилення для різних частотних смуг.

- IRepository та його реалізації (UserRepository, TrackRepository, PlaylistRepository) – шар доступу до даних. Забезпечують виконання операцій пошуку (findByID, findAll), оновлення (update) та видалення (delete) сутностей у базі даних, ізолюючи бізнес-логіку від SQL-запитів.

Зв’язки між класами:

- MusicPlayer використовує об’єкти, що реалізують інтерфейс Playable (поточний плейлист або бібліотеку), керує поточним треком та взаємодіє з Equalizer для обробки звуку.
- User володіє Playlist-ами, а Library агрегує в собі об’єкти Track.
- Репозиторії (PlaylistRepository, TrackRepository, UserRepository) відповідають за менеджмент відповідних сутностей, забезпечуючи їх збереження та відновлення з бази даних.

Таким чином, фінальна діаграма демонструє узгоджену систему взаємодії класів, де MusicPlayer виступає контролером логіки відтворення, сутності (User, Track, Playlist) моделюють дані предметної області, а репозиторії забезпечують надійний зв'язок зі сховищем даних. Така архітектура сприяє легкості підтримки коду та можливості подальшого розширення функціонала.

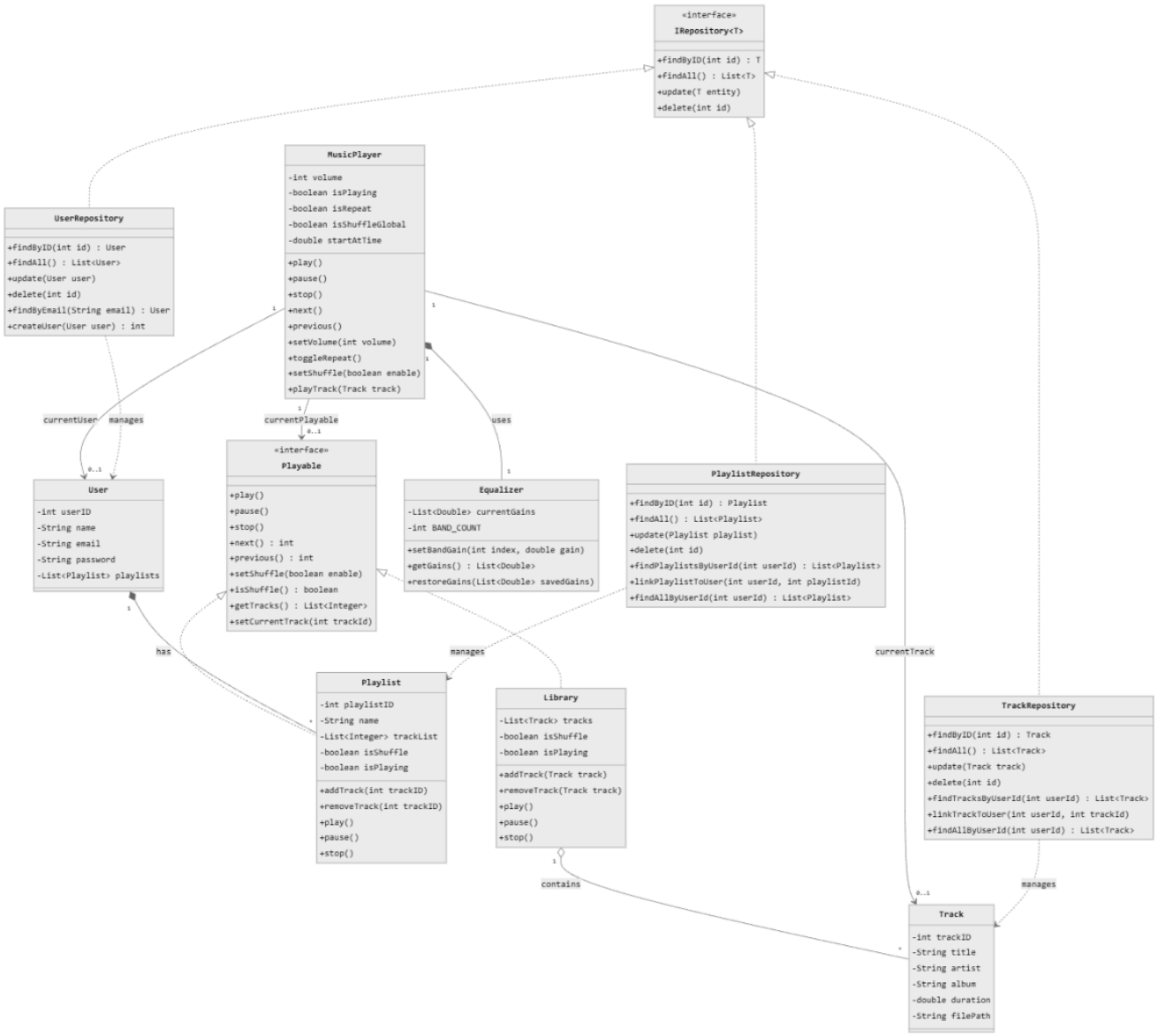

Рис. 4. Діаграма класів реалізованої системи

Лістинг коду реалізованих класів системи

```
public class Equalizer implements Serializable {
    private List<Double> currentGains;
    private static final int BAND_COUNT = 10;

    public Equalizer() {
    }

    public void attachToPlayer(MediaPlayer player) {
    }

    public void setBandGain(int index, double gain,
MediaPlayer currentPlayer) {
    }

    public List<Double> getGains() {
```

```
        return null;
    }

    public void restoreGains(List<Double> savedGains)
{
    }
}

public class Library implements Playable {
    private List<Track> tracks;
    private AudioIterator<Track> iterator;
    private boolean isShuffle = false;
    private boolean isPlaying = false;

    public Library() {
    }

    public void addTrack(Track track) {
    }

    public void removeTrack(Track track) {
    }

    private void updateIterator() {
    }

    @Override
    public void play() {
    }

    @Override
    public void pause() {
    }

    @Override
    public void stop() {
    }

    @Override
    public int next() {
        return -1;
```

```
    }

    @Override
    public int previous() {
        return -1;
    }

    @Override
    public void setShuffle(boolean enable) {
    }

    @Override
    public boolean isShuffle() {
        return false;
    }

    @Override
    public List<Integer> getTracks() {
        return null;
    }

    public List<Track> getFullTrackList() {
        return null;
    }

    @Override
    public void setCurrentTrack(int trackId) {
    }
}

public class MusicPlayer {
    private int volume;
    private boolean isPlaying;
    private boolean isRepeat;
    private Playable currentPlayable;
    private Track currentTrack;
    private User currentUser;
    private double startAtTime = 0.0;
    private Runnable onTrackChanged;
    private MediaPlayer mediaPlayer;
    private boolean isShuffleGlobal = false;
```

```
    private final Equalizer equalizer;
    private final UserService userService;
    private final TrackService trackService;
    private final PlaylistService playlistService;

    public MusicPlayer(UserService userService,
TrackService trackService, PlaylistService
playlistService) {
        this.userService = userService;
        this.trackService = trackService;
        this.playlistService = playlistService;
        this.equalizer = new Equalizer();
    }

    public void play() {
    }

    private void loadNextValidTrack() {
    }

    public void pause() {
    }

    public void setShuffle(boolean enable) {
    }

    public boolean isShuffle() {
        return false;
    }

    public void setPlayableSource(Playable playable)
{
    }

    public boolean isRepeat() {
        return false;
    }

    public void stop() {
    }
```

```
    public void next() {
    }

    public void previous() {
    }

    public void setVolume(int volume) {
    }

    public Track getCurrentTrack() {
        return null;
    }

    public void toggleRepeat() {
    }

    public void playTrack(Track track) {
    }

    public boolean isPlaying() {
        return false;
    }

    public User getCurrentUser() {
        return null;
    }

    public void loginUser(int userId) {
    }

    public Equalizer getEqualizer() {
        return null;
    }

    public void updateEqBand(int index, double gain)
{
    }

    private void playCurrentTrackInternal() {
    }
```

```
    public PlayerMemento saveState() {
        return null;
    }

    public void restoreState(PlayerMemento memento) {
    }

    public void setOnTrackChanged(Runnable callback)
{
    }
}

public interface Playable {
    void play();
    void pause();
    void stop();
    int next();
    int previous();
    void setShuffle(boolean enable);
    boolean isShuffle();
    List<Integer> getTracks();
    void setCurrentTrack(int trackId);
}

public class Playlist implements Playable,
Serializable {
    private int playlistID;
    private String name;
    private List<Integer> trackList;
    private transient AudioIterator<Integer>
iterator;
    private boolean isShuffle = false;
    private boolean isPlaying = false;

    public Playlist(int playlistID, String name) {
    }

    private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException {
    }
```

```
public void addTrack(int trackID) {
}

public void removeTrack(int trackID) {
}

private void updateIterator() {
}

@Override
public void play() {
}

@Override
public void pause() {
}

@Override
public void stop() {
}

@Override
public int next() {
    return -1;
}

@Override
public int previous() {
    return -1;
}

@Override
public void setShuffle(boolean enable) {
}

@Override
public boolean isShuffle() {
    return false;
}

@Override
```

```
    public List<Integer> getTracks() {
        return null;
    }

    public int getPlaylistID() {
        return 0;
    }

    public void setPlaylistID(int playlistID) {
    }

    public String getName() {
        return null;
    }

    public void setName(String name) {
    }

    @Override
    public String toString() {
        return null;
    }

    @Override
    public void setCurrentTrack(int trackId) {
    }
}

public class Track implements Serializable {
    private int trackID;
    private String title;
    private String artist;
    private String album;
    private double duration;
    private String filePath;

    public Track() {
    }
```

```
    public Track(int trackID, String title, String
artist, String album, double duration, String
filePath) {
    }

    public int getTrackID() {
        return 0;
    }

    public void setTrackID(int trackID) {
    }

    public String getTitle() {
        return null;
    }

    public void setTitle(String title) {
    }

    public String getArtist() {
        return null;
    }

    public void setArtist(String artist) {
    }

    public String getAlbum() {
        return null;
    }

    public void setAlbum(String album) {
    }

    public double getDuration() {
        return 0.0;
    }

    public void setDuration(double duration) {
    }

    public String getFilePath() {
```

```
        return null;
    }

    public void setFilePath(String filePath) {
    }

    @Override
    public String toString() {
        return null;
    }

    @Override
    public boolean equals(Object o) {
        return false;
    }

    @Override
    public int hashCode() {
        return 0;
    }
}

public class User implements Serializable {
    private int userID;
    private String name;
    private String email;
    private String password;
    private List<Playlist> playlists;

    public User() {
    }

    public User(int userID, String name, String
email, String password) {
    }

    public int getUserID() {
        return 0;
    }
    public void setUserID(int userID) {
    }
```

```
    public String getName() {
        return null;
    }
    public void setName(String name) {
    }

    public String getEmail() {
        return null;
    }
    public void setEmail(String email) {
    }

    public String getPassword() {
        return null;
    }
    public void setPassword(String password) {
    }

    public List<Playlist> getPlaylists() {
        return null;
    }
    public void setPlaylists(List<Playlist>
playlists) {
    }

    @Override
    public String toString() {
        return null;
    }

    @Override
    public boolean equals(Object o) {
        return false;
    }

    @Override
    public int hashCode() {
        return 0;
    }
}
```

```
public interface IRepository<T> {
    T findByID(int id);
    List<T> findAll();
    void update(T entity);
    void delete(int id);
}

public class PlaylistRepository implements
IRepository<Playlist> {
    @Override
    public Playlist findByID(int id) {
        return null;
    }

    @Override
    public List<Playlist> findAll() {
        return null;
    }

    @Override
    public void update(Playlist playlist) {
    }

    private void savePlaylistTracks(Playlist
playlist, Connection conn) throws SQLException {
    }

    @Override
    public void delete(int id) {
    }

    private void loadPlaylistTracks(Playlist
playlist, Connection conn) throws SQLException {
    }

    public List<Playlist> findPlaylistsByUserId(int
userId) {
        return null;
    }
```

```
    public void linkPlaylistToUser(int userId, int
playlistId) {
    }

    public List<Playlist> findAllByUserId(int userId)
{
        return null;
    }
}

public class TrackRepository implements
IRepository<Track> {
    @Override
    public Track findByID(int id) {
        return null;
    }

    @Override
    public List<Track> findAll() {
        return null;
    }

    @Override
    public void update(Track track) {
    }

    @Override
    public void delete(int id) {
    }

    private Track mapRowToTrack(ResultSet rs) throws
SQLException {
        return null;
    }

    public List<Track> findTracksByUserId(int userId)
{
        return null;
    }
```

```
    public void linkTrackToUser(int userId, int
trackId) {
    }

    public List<Track> findAllByUserId(int userId) {
        return null;
    }
}

public class UserRepository implements
IRepository<User> {
    @Override
    public User findByID(int id) {
        return null;
    }

    public User findByEmail(String email) {
        return null;
    }

    @Override
    public List<User> findAll() {
        return null;
    }

    public int createUser(User user) {
        return -1;
    }

    @Override
    public void update(User user) {
    }

    @Override
    public void delete(int id) {
    }

    private User mapRowToUser(ResultSet rs) throws
SQLException {
        return null;
    }
```

}

Висновок: під час виконання лабораторної роботи були успішно опановані та застосовані ключові інструменти та методології об'єктно-орієнтованого аналізу та проєктування (ООАП) на прикладі розробки архітектури веб-сервісу для автоматизованої перевірки програм на мові Java. На першому етапі було проведено аналіз вимог і побудовано діаграму варіантів використання , що дозволило чітко визначити функціональність системи та ролі її користувачів. Це стало основою для подальшого детального проєктування. Далі, на основі вимог, було розроблено модель предметної області та деталізовану діаграму класів . Ці моделі не тільки визначили ключові сутності системи, але й заклали архітектурні рішення, що відповідають принципам модульності та масштабованості. Для взаємодії з даними був спроєктований та реалізований патерн Repository, який забезпечує ізоляцію бізнес-логіки від деталей сховища даних.

Питання до лабораторної роботи

1. Що таке UML? UML (Unified Modeling Language) — це стандартизована мова для візуального моделювання програмного забезпечення.
2. Що таке діаграма класів UML?
Діаграма класів — це тип діаграми UML, що моделює статичну структуру системи, показуючи класи, їхні атрибути, операції та зв'язки.
3. Які діаграми UML називають канонічними?
Зазвичай так називають 5 основних діаграм: варіантів використання, класів, послідовності, станів та діяльності.
4. Що таке діаграма варіантів використання?
Діаграма варіантів використання фіксує функціональність системи та її межі, показуючи взаємодію акторів із системою.
5. Що таке варіант використання?
Варіант використання — це послідовність дій, яку система виконує для досягнення мети актора.
6. Які відношення можуть бути відображені на діаграмі використання?
На діаграмі використання відображають відношення асоціації, <> (включення) та <> (розширення).
7. Що таке сценарій?
Сценарій — це покроковий текстовий опис варіанту використання, що включає передумови, потік подій та винятки.
8. Що таке діаграма класів?
Діаграма класів моделює статичну структуру системи, візуалізуючи класи, їхні атрибути, операції та зв'язки між ними.
9. Які зв’язки між класами ви знаєте?
Основні типи зв'язків між класами: асоціація, агрегація, композиція, залежність та успадкування.

10. Чим відрізняється композиція від агрегації?
Відмінність між композицією та агрегацією полягає в управлінні життєвим циклом. При агрегації "частини" можуть існувати окремо від "цілого", а при композиції — ні.
11. Чим відрізняється зв'язки типу агрегації від зв'язків композиції на діаграмах класів?
На діаграмах класів агрегація позначається незафарбованим ромбом , а композиція — зафарбованим з боку класу-"цілого".
12. Що являють собою нормальні форми баз даних? Нормальні форми — це правила проєктування баз даних для усунення надлишковості та забезпечення цілісності даних.
13. Що таке фізична модель бази даних? Логічна?
Логічна модель бази даних описує, що за дані зберігаються (таблиці, зв'язки). Фізична модель описує, як вони зберігаються (типи даних, індекси).
14. Який взаємозв'язок між таблицями БД та програмними класами?
Класи є програмним представленням сутностей, а таблиці БД зберігають їхні дані. Патерн Repository використовується для зв'язку між ними, абстрагуючи доступ до даних від бізнес-логіки.