Міністерство освіти і науки України
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»

Виконав:
Тимчук Владислав
ІА-34
Перевірив:
Мягкий М. Ю.

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 Практикуму:

Музичний програвач (iterator, command, memento, facade, visitor, clientserver)
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 аудіоформатів, еквалайзер.

Вступ

Основна мета цієї роботи полягає в тому, щоб пройти всі ключові етапи проєктування: від аналізу вимог користувача та візуалізації їх у вигляді UML-діаграм до практичної реалізації архітектури. Завдання передбачає не лише проєктування логічної структури та бази даних, а й написання коду основних класів із застосуванням сучасного шаблону проєктування *Repository*.

Зміст

Хід роботи

1. Діаграма варіантів використання (Use Case Diagram)

Діаграма варіантів використання показує, як користувачі взаємодіють із системою для досягнення своїх цілей.

Актори:

- **Користувач (User):** Основна дійова особа, яка слухає музику та керує нею.

Основні варіанти використання:

- **Керувати відтворенням:** Основна функція, що включає програвання, паузу, зупинку та перемикання треків.
- **Керувати плейлістами:** Дозволяє користувачеві створювати плейлисти, додавати та видаляти з них треки.
- **Керувати бібліотекою:** Додавання нових треків до загальної медіатеки.
- **Шукати музику:** Пошук треків, альбомів чи виконавців у бібліотеці.

Опис діаграми:

- **Користувач** є головним актором.
- Він може **Керувати відтворенням**, що включає в себе більш конкретні дії: відтворити трек, поставити на паузу та перемикнути трек.
- Також він може **Керувати плейлистами**, що включає у себе: Створити плейлист, Додати трек до плейліста та видалити трек з плейлиста.
- Дві незалежні функції – це **Керувати бібліотекою** (наприклад, імпортувати файли) та **Шукати музику**.

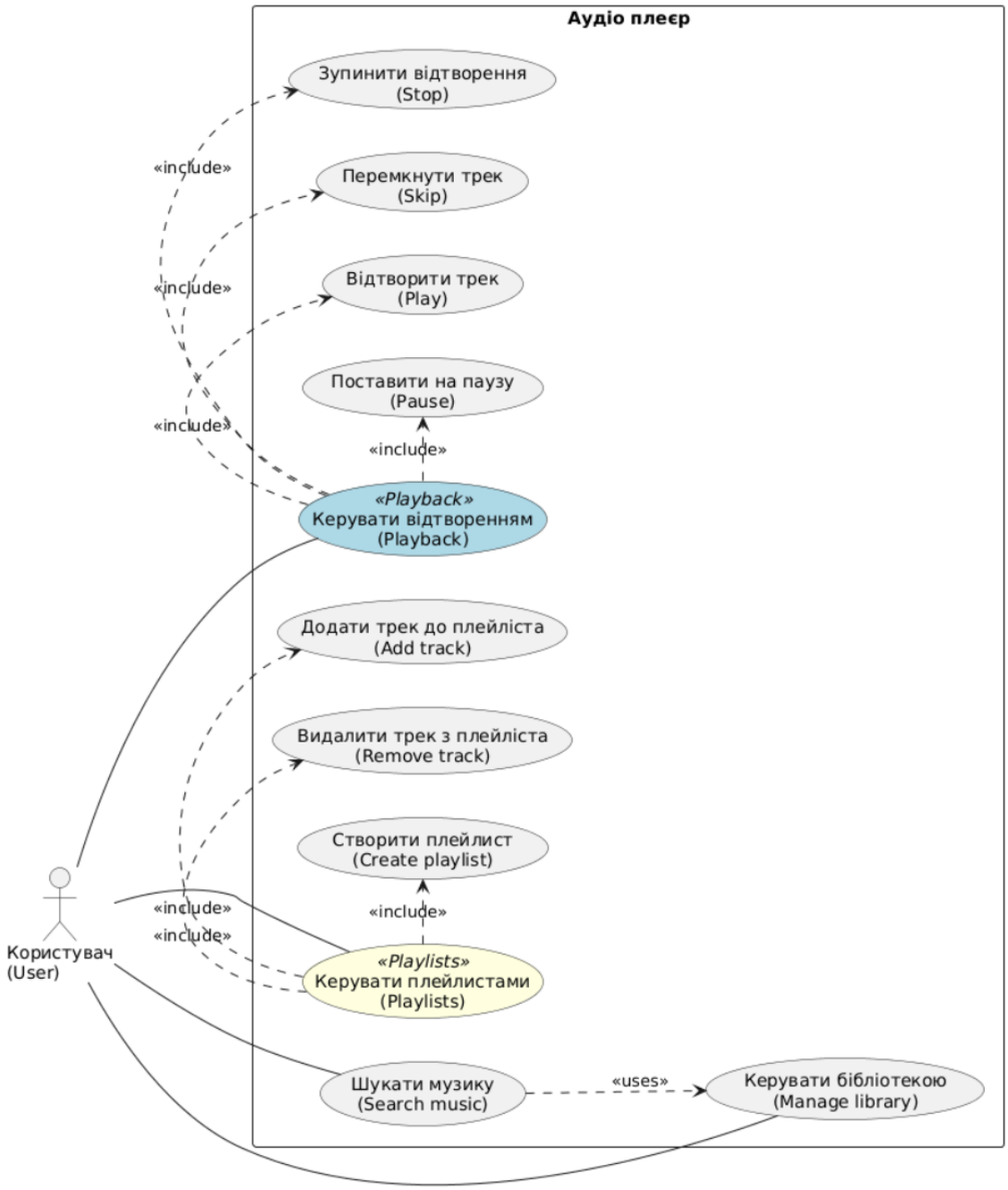

Рис.1 – Діаграма варіантів використання

2. Діаграма класів предметної області (Domain Model Class Diagram)

Ця діаграма відображає основні сутності системи та зв'язки між ними на концептуальному рівні.

Основні класи:

- **Track (Трек):** Представляє один аудіофайл. Має атрибути: title (назва), duration (тривалість), filePath (шлях до файлу).

- **Playlist (Плейлист):** Колекція треків. Має назву (name).
- **Artist (Виконавець):** Представляє музичного виконавця. Має ім'я (name).
- **Album (Альбом):** Колекція треків одного виконавця. Має назву (title) та рік випуску (year).

Зв'язки (Associations):

- Один **Виконавець** може мати багато **Альбомів** (1 до N).
- Один **Альбом** належить одному **Виконавцю** (N до 1).
- Один **Альбом** містить багато **Треків** (1 до N).
- Один **Трек** належить одному **Альбому**.
- Між **Плейлістом** і **Треком** існує зв'язок N до N: один трек може бути в кількох плейлістах, а один плейліст може містити багато треків.

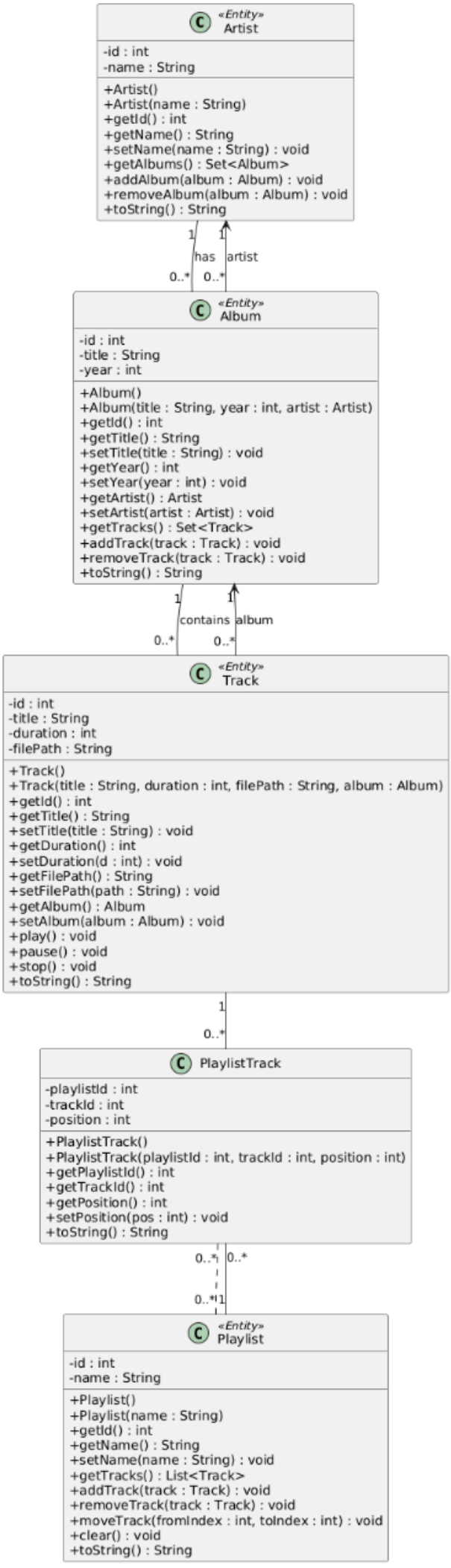
«Entity»
Artist
-id : int
-name : String
+Artist()
+Artist(name : String)
+getId() : int
+getName() : String
+setName(name : String) : void
+getAlbums() : Set<Album>
+addAlbum(album : Album) : void
+removeAlbum(album : Album) : void
+toString() : String
1
has
0..*
1
artist
0..*
«Entity»
Album
-id : int
-title : String
-year : int
+Album()
+Album(title : String, year : int, artist : Artist)
+getId() : int
+getTitle() : String
+setTitle(title : String) : void
+getYear() : int
+setYear(year : int) : void
+getArtist() : Artist
+setArtist(artist : Artist) : void
+getTracks() : Set<Track>
+addTrack(track : Track) : void
+removeTrack(track : Track) : void
+toString() : String
1
contains
0..*
1
album
0..*
«Entity»
Track
-id : int
-title : String
-duration : int
-filePath : String
+Track()
+Track(title : String, duration : int, filePath : String, album : Album)
+getId() : int
+getTitle() : String
+setTitle(title : String) : void
+getDuration() : int
+setDuration(d : int) : void
+getFilePath() : String
+setFilePath(path : String) : void
+getAlbum() : Album
+setAlbum(album : Album) : void
+play() : void
+pause() : void
+stop() : void
+toString() : String
1
0..*
PlaylistTrack
-playlistId : int
-trackId : int
-position : int
+PlaylistTrack()
+PlaylistTrack(playlistId : int, trackId : int, position : int)
+getPlaylistId() : int
+getTrackId() : int
+getPosition() : int
+setPosition(pos : int) : void
+toString() : String
0..*
0..*
0..*
1
«Entity»
Playlist
-id : int
-name : String
+Playlist()
+Playlist(name : String)
+getId() : int
+getName() : String
+setName(name : String) : void
+getTracks() : List<Track>
+addTrack(track : Track) : void
+removeTrack(track : Track) : void
+moveTrack(fromIndex : int, toIndex : int) : void
+clear() : void
+toString() : String

Рис. 2 – Діаграма класів

3. Сценарії використання (Use Case Scenarios)

Сценарій 1: Створення нового плейлиста

Поле	Опис
Назва	Створення нового плейлиста
Актор	Користувач User
Передумова	Користувач знаходиться в головному вікні програми.
Основний потік	1. Користувач натискає кнопку "Створити плейлист". 2. Система відкриває діалогове вікно з полем для введення назви. 3. Користувач вводить назву "Улюблена музика" та підтверджує дію. 4. Система зберігає новий порожній плейлист з указаною назвою. 5. Система відображає новий плейліст у списку плейлистів.
Постумова	У системі створено новий плейлист "Улюблена музика".
Винятки	3а. Користувач вводить назву, яка вже існує. 4а. Система показує повідомлення про помилку: "Плейлист з такою назвою вже існує". 5а. Система повертає користувача до кроку 3.

Сценарій 2: Додавання треку до плейліста

Поле	Опис
Назва	Додавання треку до плейліста
Актор	Користувач User
Передумова	Користувач бачить список треків у бібліотеці. Існує хоча б один плейлист.
Основний потік	1. Користувач знаходить потрібний трек у бібліотеці. 2. Користувач викликає контекстне меню для цього треку 3. У меню користувач обирає опцію "Додати до плейлиста". 4. Система відображає список існуючих плейлистів. 5. Користувач обирає плейлист "Улюблена музика". 6. Система додає посилання на трек до обраного плейлиста. 7. Система виводить сповіщення "Трек успішно додано".
Постумова	Обраний трек тепер є частиною плейлиста "Улюблена музика".
Винятки	6а. Трек вже існує в цьому плейлисті. 7а. Система не додає дублікат і виводить повідомлення "Цей трек вже є у плейлисті".

Сценарій 3: Відтворення треку

Поле	Опис
Назва	Відтворення треку
Актор	Користувач User
Передумова	Користувач знаходиться у вікні, де видно список треків (бібліотека або плейлист).
Основний потік	1. Користувач двічі клацає лівою кнопкою миші по треку. 2. Система перевіряє доступність файлу за вказаним шляхом. 3. Система починає відтворення аудіофайлу. 4. В інтерфейсі плеєра відображається назва треку, що грає, та прогрес відтворення.
Постумова	Музика грає, інтерфейс оновлено.

Винятки	2а. Файл за вказаним шляхом не знайдено. 3а. Система виводить повідомлення про помилку: "Не вдалося знайти файл треку". 4а. Відтворення не починається.

4. Основні класи та структура бази даних

- **Artists:** `Id`, `Name`
- **Albums:** `Id`, `Title`, `Year`, `ArtistId` (FK)
- **Tracks:** `Id`, `Title`, `Duration`, `FilePath`, `AlbumId` (FK)
- **Playlists:** `Id`, `Name`
- **PlaylistTracks:** `PlaylistId, TrackId`

Track.java:

```
import javax.persistence.*;

import java.util.Set;

@Entity

@Table(name = "Tracks")

public class Track {

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    private int id;

    private String title;

    private int duration;

    private String filePath;

    @ManyToOne

    @JoinColumn(name = "AlbumId")

    private Album album;

    @ManyToMany(mappedBy = "tracks")

    private Set<Playlist> playlists;

    // геттери сетери
```

```
}
```

Playlist.java:

```
import javax.persistence.*;
import java.util.Set;
@Entity
@Table(name = "Playlists")
public class Playlist {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    @ManyToMany
    @JoinTable(
        name = "PlaylistTracks",
        joinColumns = @JoinColumn(name = "PlaylistId"),
        inverseJoinColumns = @JoinColumn(name = "TrackId")
    )
    private Set<Track> tracks;

    // геттери та сеттери
}
```

Короткий опис коду:

- **@Entity** позначає клас як сутність, пов'язану з таблицею.
- **@Id** та **@GeneratedValue** — визначають первинний ключ.
- **@ManyToOne** та **@ManyToMany** — налаштовують зв'язки між сутностями.
- **@JoinTable** — явно вказує на таблицю-посередник `PlaylistTracks` для зв'язку "багато-до-багатьох".

4.1 Шаблон Repository

При використанні фреймворку **Spring for Java**, шаблон Repository реалізується дуже зручно через **Spring Data JPA**. Замість того, шоб писати реалізацію власноруч, створимо інтерфейс, який наслідує JpaRepository. Spring автоматично надасть реалізацію всіх базових CRUD-операцій (Create, Read, Update, Delete).

TrackRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;

import java.util.List;



@Repository

public interface TrackRepository extends JpaRepository<Track, Integer> {

}
```

PlaylistRepository.java

```
import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.stereotype.Repository;



@Repository

public interface PlaylistRepository extends JpaRepository<Playlist, Integer>
{

    // Spring сам реалізує всі базові CRUD-операції

}
```

4.2 Діаграма класів для реалізованої системи

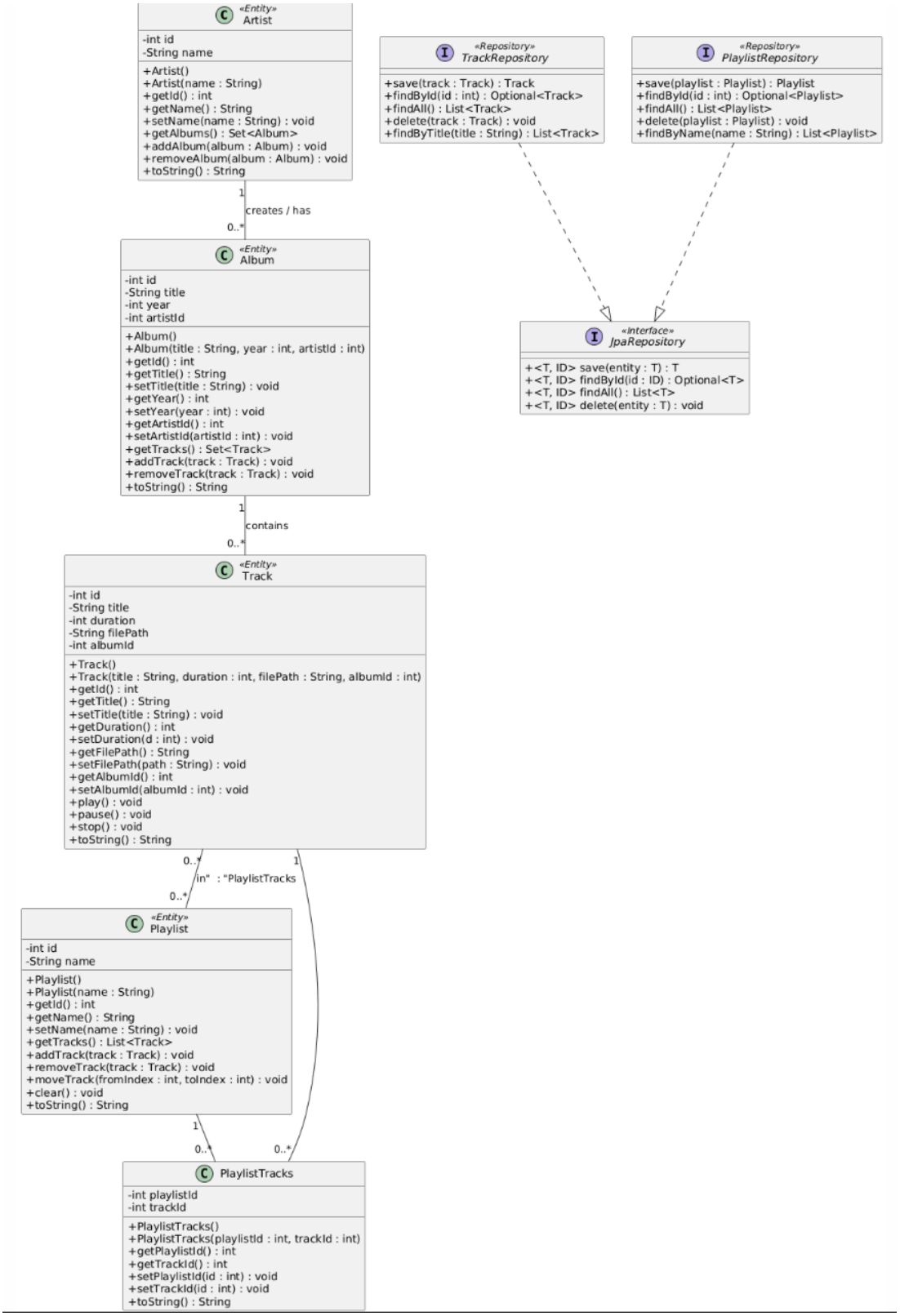

Рис.3 – Діаграма класів реалізованої системи

Висновок

Під час виконання цієї лабораторної роботи було пройдено повний цикл проєктування програмної системи "Аудіо плеєр". На основі спроєктованої предметної області була розроблена логічна структура бази даних та реалізовані основні класи-сутності на мові Java з використанням JPA.

КОНТРОЛЬНІ ЗАПИТАННЯ

1. Що таке UML?

UML (Unified Modeling Language) — це уніфікована графічна мова для візуалізації, специфікації, конструювання та документування програмних систем. Вона надає стандартний набір діаграм та нотацій, що дозволяє розробникам, аналітикам та замовникам однозначно розуміти архітектуру, процеси та структуру майбутнього програмного продукту, незалежно від мови програмування.

2. Що таке діаграма класів UML?

Діаграма класів — це статична структурна діаграма UML, яка описує структуру системи, показуючи її класи, їхні атрибути (поля), операції (методи) та зв'язки між класами. Вона є своєрідним "кресленням" системи, що демонструє, з яких логічних блоків вона складається і як ці блоки взаємодіють між собою на рівні коду.

3. Які діаграми UML називають канонічними?

Канонічними діаграмами UML називають стандартний набір із 14 типів діаграм. Вони поділяються на дві великі категорії: **структурні діаграми** (7 штук, наприклад, діаграма класів, компонентів), які описують статичні аспекти системи, та **поведінкові діаграми** (7 штук, наприклад, діаграма варіантів використання, послідовності), які моделюють динамічні аспекти та процеси.

4. Що таке діаграма варіантів використання?

Діаграма варіантів використання (Use Case Diagram) — це поведінкова діаграма, яка моделює взаємодію між зовнішніми користувачами (акторами) та системою. Вона візуалізує, які функції (варіанти використання) система надає акторам, і є чудовим інструментом для визначення та погодження функціональних вимог до системи на ранніх етапах розробки.

5. Що таке варіант використання?

Варіант використання (Use Case) — це опис послідовності дій, які система виконує у відповідь на запит від актора для досягнення певної мети. По суті, це одна конкретна функція системи, розглянута з точки зору користувача, наприклад, "Створити плейлист", "Додати трек до бібліотеки" або "Здійснити пошук".

6. Які відношення можуть бути відображені на діаграмі використання?

На діаграмі варіантів використання можуть бути відображені три основні типи відношень: **<<include>>** (включення), коли один варіант використання завжди містить у собі інший; **<<extend>>** (розширення), коли один варіант використання може доповнювати інший за певних умов; та **асоціація**, яка просто показує зв'язок між актором та варіантом використання.

7. Що таке сценарій?

Сценарій — це один конкретний шлях або екземпляр виконання варіанту використання. Якщо варіант використання — це загальний опис функції (напр., "Пошук треку"), то сценарії описують конкретні ситуації: "Успішний пошук треку за назвою", "Пошук не дав результатів", "Помилка під час пошуку через відсутність з'єднання".

8. Що таке діаграма класів?

Діаграма класів — це центральна структурна діаграма в UML, що детально описує внутрішню будову системи. Вона показує набір класів, інтерфейсів та їхніх компонентів (атрибутів і методів), а також статичні зв'язки між ними, такі як асоціація, успадкування та залежність, формуючи логічний каркас програмного коду.

9. Які зв’язки між класами ви знаєте?

Існують кілька основних типів зв'язків між класами: **асоціація** (класи пов'язані один з одним), **агрегація** (відношення "є частиною", де частина може існувати окремо), **композиція** (сувора форма агрегації, де частина не може існувати без цілого), **успадкування** (один клас є нащадком іншого) та **залежність** (один клас використовує інший).

10. Чим відрізняється композиція від агрегації?

Композиція відрізняється від **агрегації** тривалістю життя компонентів. При агрегації об'єкт-"частина" може існувати незалежно від об'єкта-"цілого"

(наприклад, колесо може існувати окремо від машини). При композиції "частина" жорстко прив'язана до "цілого" і знищується разом з ним (наприклад, кімната не може існувати без будинку).

11. Чим відрізняються зв'язки типу агрегації від зв'язків композиції на діаграмах класів?

На діаграмах класів ці зв'язки візуально відрізняються ромбом на кінці лінії біля класу-"цілого". Для **агрегації** використовується порожній (незафарбований) ромб ◇, що символізує слабший зв'язок. Для **композиції** використовується зафарбований (чорний) ромб ◆, що вказує на сильний, життєво залежний зв'язок.

12. Що являють собою нормальні форми баз даних?

Нормальні форми (НФ) — це набір правил і вимог до проєктування реляційних баз даних, спрямованих на зменшення надлишковості даних та усунення аномалій (помилок) при їх оновленні, додаванні чи видаленні. Процес приведення бази даних до цих форм називається нормалізацією, і найчастіше використовуються перші три форми (1НФ, 2НФ, 3НФ).

13. Що таке фізична модель бази даних? Логічна?

Логічна модель бази даних описує дані та зв'язки між ними на абстрактному рівні, без прив'язки до конкретної системи управління базами даних (СУБД). **Фізична модель**, навпаки, є конкретною реалізацією логічної моделі для обраної СУБД. Вона включає точні назви таблиць і стовпців, типи даних, індекси, обмеження та інші технічні деталі.

14. Який взаємозв'язок між таблицями БД та програмними класами?

Взаємозв'язок між таблицями БД та програмними класами зазвичай реалізується за допомогою технології **ORM (Object-Relational Mapping)**. За цією парадигмою, кожна таблиця в базі даних відповідає одному класу в програмному коді, кожен рядок у таблиці — одному об'єкту (екземпляру) цього класу, а кожен стовпець — полю (атрибуту) цього класу.

ДОДАТКИ

UML-код діаграми класів та варіантів використання:

```
@startuml
left to right direction
skinparam usecase {
```

```
  BackgroundColor<<Playback>> LightBlue
  BackgroundColor<<Playlists>> LightYellow
}

actor "Користувач\n(User)" as User

rectangle "Аудіо плеєр" {
  usecase "Керувати відтворенням\n(Playback)" as UC_Playback <<Playback>>
  usecase "Відтворити трек\n(Play)" as UC_Play
  usecase "Поставити на паузу\n(Pause)" as UC_Pause
  usecase "Зупинити відтворення\n(Stop)" as UC_Stop
  usecase "Перемкнути трек\n(Skip)" as UC_Skip

  usecase "Керувати плейлистами\n(Playlists)" as UC_Playlists <<Playlists>>
  usecase "Створити плейлист\n(Create playlist)" as UC_CreatePL
  usecase "Додати трек до плейліста\n(Add track)" as UC_AddToPL
  usecase "Видалити трек з плейліста\n(Remove track)" as UC_RemoveFromPL

  usecase "Керувати бібліотекою\n(Manage library)" as UC_Library
  usecase "Шукати музику\n(Search music)" as UC_Search
}

User -- UC_Playback
User -- UC_Playlists
User -- UC_Library
User -- UC_Search

UC_Playback .> UC_Play : <<include>>
UC_Playback .> UC_Pause : <<include>>
UC_Playback .> UC_Stop : <<include>>
UC_Playback .> UC_Skip : <<include>>

UC_Playlists .> UC_CreatePL : <<include>>
UC_Playlists .> UC_AddToPL : <<include>>
UC_Playlists .> UC_RemoveFromPL : <<include>>

UC_Search ..> UC_Library : <<uses>>

@enduml




@startuml
skinparam classAttributeIconSize 0
```

```
class Artist <<Entity>> {
  - int id
  - String name
}

class Album <<Entity>> {
  - int id
  - String title
  - int year
  - int artistId
}

class Track <<Entity>> {
  - int id
  - String title
  - int duration
  - String filePath
  - int albumId
}

class Playlist <<Entity>> {
  - int id
  - String name
}

class PlaylistTracks {
  - int playlistId
  - int trackId
}

interface JpaRepository <<Interface>> {
  ' generic: JpaRepository<T, ID>
}

interface TrackRepository <<Repository>> {
}

interface PlaylistRepository <<Repository>> {
}

Artist "1" -- "0..*" Album : "creates / has"
Album "1" -- "0..*" Track : "contains"
Track "0..*" -- "0..*" Playlist : "in"  : "PlaylistTracks"
' Альтернативно показати таблицю-посередник
Playlist "1" -- "0..*" PlaylistTracks
Track "1" -- "0..*" PlaylistTracks
```

```
TrackRepository ..|> JpaRepository
PlaylistRepository ..|> JpaRepository

@enduml


@startuml
skinparam classAttributeIconSize 0

class Artist <<Entity>> {
  - id : int
  - name : String
}

class Album <<Entity>> {
  - id : int
  - title : String
  - year : int
}

class Track <<Entity>> {
  - id : int
  - title : String
  - duration : int
  - filePath : String
}

class Playlist <<Entity>> {
  - id : int
  - name : String
}

class PlaylistTrack {
  - playlistId : int
  - trackId : int
}

Artist "1" -- "0..*" Album : has
Album  "1" -- "0..*" Track : contains

Album "0..*" --> "1" Artist : artist
Track "0..*" --> "1" Album : album

Playlist "1" -- "0..*" PlaylistTrack
Track    "1" -- "0..*" PlaylistTrack
PlaylistTrack "0..*" .. "0..*" Playlist : ""
```

```
@enduml
```