Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
Факультет інтелектуальних інформаційних технологій та автоматизації
(повне найменування інституту, назва факультету (відділення))
Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Бакалаврська дипломна робота на тему:

ПРОГРАМНИЙ МОДУЛЬ ДЛЯ ВІДТВОРЕННЯ МУЗИЧНИХ ФАЙЛІВ

Виконав: студент 4 курсу, групи КН-22мс
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Морозов Д.П.
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. КН
Паночишин Ю.М.
(прізвище та ініціали)
"07" 06 2024 р.
Рецензент: к.т.н., проф. каф. КСУ
Биков М.М.
(прізвище та ініціали)
"07" 06 2024 р.

Допущено до захисту
Зав. кафедри: проф., д.т.н Яровий А.А.
"07" 06 2024 р.

Вінниця ВНТУ – 2024 рік

Вінницький національний технічний університет
Факультет інтелектуальних інформаційних технологій та автоматизації
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Галузь знань – 12 "Інформаційні технології"
Спеціальність – 122 "Комп'ютерні науки"
Освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
проф., д.т.н. Яровий А.А.
" 01 " 03 2024 р

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Морозову Дмитру Павловичу

1. Тема роботи: Програмний модуль для відтворення музичних файлів.
Керівник роботи: к.т.н., доц. каф. КН Паночишин Юрій Миколайович.
Затверджені наказом вищого навчального закладу " 11 " 03 2024 року № 80

2. Термін подання студентом роботи 27.05.2024

3. Вихідні дані до роботи :
Кількість функцій – не менше 5;
Використання технології WPF;
Мова програмування – об'єктно-орієнтована;
Середовище розробки Visual Studio 2022.

4. Зміст текстової частини (перелік питань, які потрібно розробити):
вступ; обґрунтування доцільності розробки модулю для відтворення музичних файлів; проектування модулю для відтворення музичних файлів; розробка інтерфейсу та функціоналу модулю для відтворення музичних файлів; тестування розробленого модулю для відтворення музичних файлів; висновки; список використаних джерел; додатки.

5. Перелік ілюстративного матеріалу (з точним зазначенням обов'язкових креслень):
схема основної архітектури WPF; механізм прив'язки даних; архітектура .NET Framework; схема роботи патерна MVVM; загальна структура програмної реалізації модуля.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Паночишин Ю.М., к.т.н., доц. каф. КН		

7. Дата видачі завдання 07.03.2024

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва та зміст етапу	Термін виконання		Примітка
		початок	закінчення	
1	Обгрунтування доцільності розробки модулю для відтворення музичних файлів	06.05.24	10.05.24	
2	Проектування модулю для відтворення музичних файлів	11.05.24	15.05.24	
3	Розробка інтерфейсу та функціоналу модулю для відтворення музичних файлів	16.05.24	20.05.24	
4	Тестування розробленого модулю для відтворення музичних файлів	21.05.24	25.05.24	
5	Розробка інструкції користувача	26.05.24	30.05.24	
6	Аналіз отриманих результатів та формування висновків	31.05.24	02.06.24	
7	Оформлення бакалаврської дипломної роботи	03.06.24	06.06.24	

Студент (підпис) **Морозов Д.П.** (прізвище та ініціали)

Керівник роботи (підпис) **Паночишин Ю.М.** (прізвище та ініціали)

АНОТАЦІЯ

Морозов Д.П. Програмний модуль для відтворення музичних файлів. Бакалаврська дипломна робота складається з 88 сторінок формату А4, на яких є 50 рисунків, список використаних джерел містить 20 найменувань.

Метою даної бакалаврської дипломної роботи є розширення функціональних можливостей музичних відтворювачів, шляхом збагачення базового функціоналу плеєра, доданням роботи з плейлистами користувачів та доданням вбудованого еквалайзера для індивідуального налаштування звуку. Також метою роботи є розробка зручного та практичного користувацького інтерфейсу для покращення досвіду роботи користувача з програмами для відтворення музичних файлів.

Результатом є програмний модуль для відтворення музичних файлів, що реалізований у програмному середовищі Visual Studio 2022 на мові програмування C# з використанням платформи WPF.

Ключові слова: WPF, C#, еквалайзер, плейлист, аудіофайл.

ABSTRACT

Morozov D.P. Software module for playing music files. The bachelor thesis consists of 88 pages of A4 format, on which there are 50 figures, the list of used sources contains 20 items.

The purpose of this bachelor's thesis is to expand the functionality of music players by enriching the basic functionality of the player, adding work with user playlists and adding a built-in equalizer for individual sound adjustment. Also, the goal of the work is to develop a convenient and practical user interface to improve the user's experience with programs for playing music files.

The result is a software module for playing music files, implemented in the Visual Studio 2022 programming environment in the C# programming language using the WPF platform.

Keywords: WPF, C#, equalizer, playlist, audio file.

ЗМІСТ

ВСТУП

Актуальність досліджень. У сучасному світі музика займає значне місце в повсякденному житті людей, а тому зручні та функціональні музичні плеєри стають невід'ємною частиною цифрових пристроїв.

З розвитком різних платформ, користувачі очікують від програмних засобів не тільки базової функціональності, але й додаткових можливостей, таких як підтримка різноманітних аудіоформатів, налаштування еквалайзера, створення та управління плейлистами та багато іншого. Тому створення багатофункціонального програмного модуля для відтворення музичних файлів є важливим та актуальним завданням.

Крім того, конкуренція на ринку програмного забезпечення вимагає від розробників постійного вдосконалення продуктів для задоволення зростаючих потреб користувачів. Дослідження та розробка такого модуля сприятиме підвищенню рівня професійної підготовки, розвитку навичок у програмуванні та використанні сучасних технологій.

Таким чином, актуальність обраної теми дипломної роботи обумовлена необхідністю створення сучасних програмних рішень для відтворення музичних файлів, що відповідають вимогам користувачів.

Метою дослідження є розширення функціональних можливостей музичних відтворювачів, шляхом збагачення базового функціоналу плеєра, доданням роботи з плейлистами користувачів та доданням вбудованого еквалайзера для індивідуального налаштування звуку.

Також метою роботи є розробка зручного та практичного користувацького інтерфейсу для покращення досвіду роботи користувача з програмами для відтворення музичних файлів.

Об’єкт дослідження – процес комп’ютеризованого відтворення музичних файлів.

Предмет дослідження – програмні засоби для відтворення аудіофайлів та їх функціональні можливості.

Задачі дослідження:

1. дослідити технології та підходи для створення програмного модуля для відтворення музичних файлів;
2. обґрунтувати методи розробки програмного модулю;
3. обґрунтувати вибір інструментів для розробки;
4. розробити програмну реалізацію модулю для відтворення музичних файлів;
5. протестувати розроблений модуль для відтворення музичних файлів.

1 ОБҐРУНТУВАННЯ ДОЦІЛЬНОСТІ РОЗРОБКИ МОДУЛЮ ДЛЯ ВІДТВОРЕННЯ МУЗИЧНИХ ФАЙЛІВ

1.1 Аналіз предметної області

Відтворення музики на персональних комп'ютерах стало настільки звичним, що користувачі очікують високоякісних і зручних музичних плеєрів. Програмний модуль для відтворення музичних файлів є ключовим компонентом у цьому контексті, забезпечуючи користувачам функціональність, яка дозволяє насолоджуватися музикою з максимальною зручністю.

Від інтерактивного та зручного інтерфейсу до високої якості звуку, сучасні музичні плеєри повинні відповідати різноманітним вимогам користувачів та контекстам їх використання. Розробка такого програмного забезпечення вимагає планування, врахування потреб кінцевих користувачів та впровадження технологій для досягнення якості продукту та зручності його використання.

Потреби користувачів музичних плеєрів можуть варіюватися від простого відтворення музики до складніших функцій, таких як створення власних плейлистів, редагування метаданих аудіофайлів та синхронізації музичних бібліотек між різними пристроями. Деякі користувачі можуть шукати плеєр зі збалансованим відтворенням звуку, підтримкою високоякісних аудіоформатів, тоді як інші можуть бажати простого та ефективного інтерфейсу для прослуховування своєї улюбленої музики.

Програмний модуль для відтворення музичних файлів повинен ефективно використовувати системні ресурси, мінімізуючи споживання енергії та обсяг використовуваної пам'яті, щоб забезпечити плавну та стабільну роботу навіть на слабших персональних комп'ютерах.

Розробляючи програмний модуль для відтворення музичних файлів потрібно враховувати такі фактори:

– Підтримка різноманітних аудіоформатів, плеєр має забезпечувати сумісність з різними типами аудіофайлів, включаючи MP3, WAV, FLAC, AAC та інші, для оптимальної роботи з різними музичними колекціями користувачів;

– Зручний інтерфейс користувача, що повинен мати інтуїтивно зрозумілий і простий у використанні інтерфейс, що дозволяє легко орієнтуватися в музичній бібліотеці, керувати відтворенням та виконувати інші операції без зайвих зусиль;

– Розширена функціональність, крім основних операцій відтворення, користувачі очікують від плеєра можливість створення та керування плейлистами, додаткові функції керування аудіофайлами;

– Підтримка метаданих, плеєр повинен відображати важливу інформацію про музичні треки, таку як назва, виконавець, альбом, обкладинка тощо, для зручності, зрозумілості та надання приємного користувацького досвіду.

Переваги локальних музичних плеєрів:

– Контроль за музичною колекцією, коли користувач має повний контроль над своєю музикою та може організувати її на комп'ютері за своїми уподобаннями;

– Відсутність залежності від Інтернету, де відтворення музики не потребує доступу до Інтернету, що дозволяє слухати музику будь-коли;

– Краща якість звуку, так як локальні аудіофайли можуть надавати кращу якість звуку, оскільки не потребують стиснення аудіофайлів для стрімінгу.

Недоліки локальних музичних плеєрів:

– Обмеженість місця зберігання, так як розмір музичної бібліотеки обмежений обсягом пам'яті на пристрої;

– Відсутність доступу до музики з віддалених пристроїв, користувачі не можуть отримати доступ до своєї музики з інших пристроїв;

– Обмежена інтеграція з іншими сервісами, можливості інтеграції з онлайн-сервісами та стрімінговими платформами можуть бути обмеженими.

1.2 Огляд та аналіз аналогів

foobar2000 (рис. 1.1) - це безкоштовний аудіопрогравач для операційної системи Windows. Він відзначається модульною архітектурою, яка надає користувачам можливість гнучкої настройки та конфігурації. Розробником є Пітер Павловський, який раніше працював у компанії Nullsoft і створював плагіни для Winamp [1].

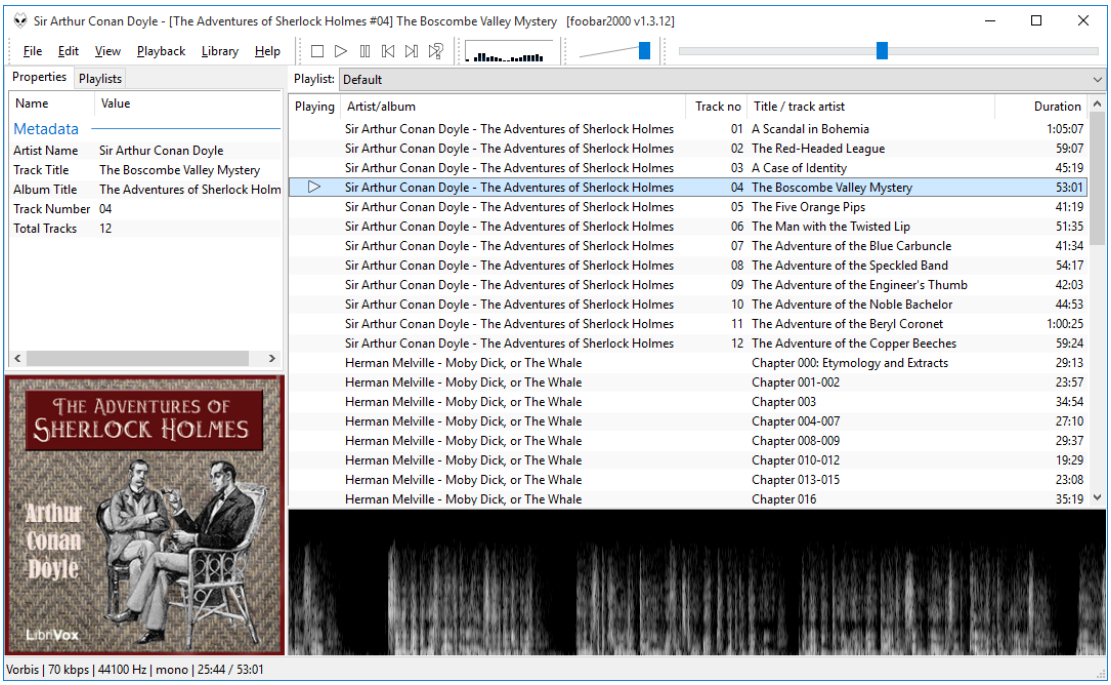

Рисунок 1.1 – Інтерфейс програвача foobar2000

Основними рисами foobar2000 є:

1. Підтримувані аудіоформати: AAC, MP3, MP4, FLAC, WavPack, CD Audio, WAV, AIFF, Musepack, WMA, Vorbis, Opus, Speex, AU, SND та інші з додатковими компонентами;
2. Безперервне відтворення;
3. Легко настроюваний макет інтерфейсу користувача;

4. Розширені можливості тегування;

5. Підтримка копіювання аудіо компакт-дисків, а також перекодування всіх підтримуваних аудіоформатів за допомогою компонента Converter;

6. Повна підтримка ReplayGain;

7. Настроювані комбінації клавіш;

8. Відкрита архітектура компонентів, що дозволяє стороннім розробникам розширювати функціональність плеєра [2].

Недоліки програми:

1. Складний інтерфейс для новачків: Інтерфейс foobar2000 може здатися складним та незручним для тих, хто використовує його вперше або не має досвіду роботи з подібними програмами. Деякі функції можуть здаватися неочевидними, що може призвести до певної складності для новачків;

2. Неінтуїтивність налаштувань: foobar2000 славиться своєю налаштовуваністю і гнучкістю, проте це може бути вадою для деяких користувачів. Налаштування програми може займати час і вимагати глибокого розуміння її функціоналу;

3. Відсутність стандартних функцій за замовчуванням: У порівнянні з іншими плеєрами, foobar2000 може бути менш функціональним за замовчуванням. Деякі стандартні функції, які можуть очікуватися від інших плеєрів, можуть бути відсутніми або потребувати налаштувань.

Windows Media Player (рис. 1.2) - це програмне забезпечення для мультимедіа, яке призначене для відтворення аудіо, відео та перегляду зображень на комп'ютерах, що працюють під управлінням операційної системи Microsoft Windows [3].

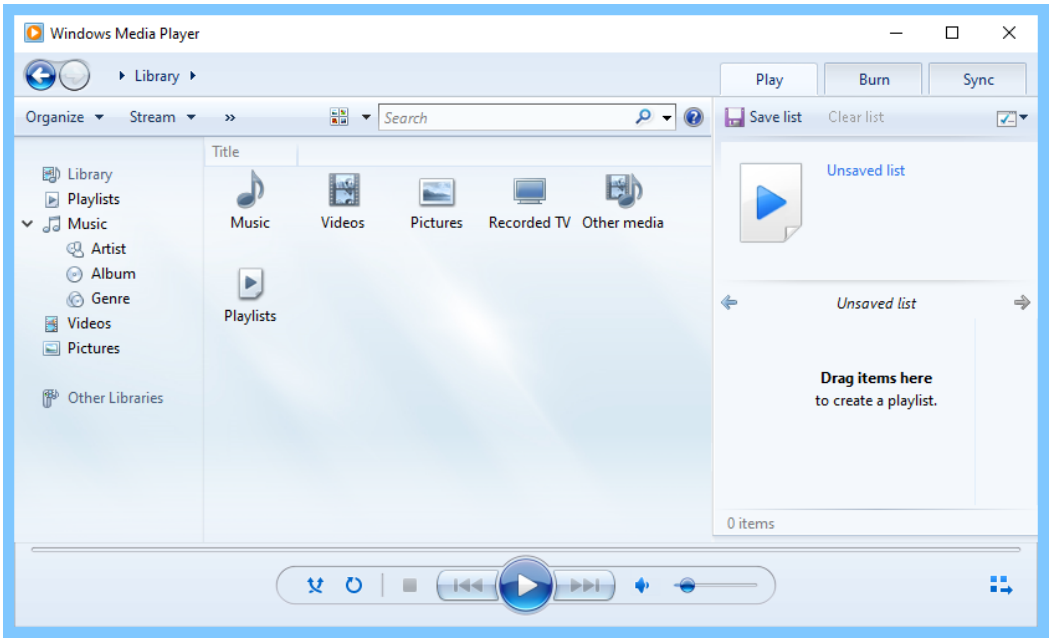

Рисунок 1.2 – Інтерфейс програвача Windows Media Player

Основними функціями WMP є:

1. Програвання аудіо та відеофайлів: WMP дозволяє відтворювати аудіо та відеофайли, що зберігаються на вашому комп'ютері. Він підтримує багато популярних аудіоформатів, таких як MP3, WAV, WMA, а також відеоформатів, таких як AVI, WMV та MPEG;

2. Організація медіабібліотеки: WMP дозволяє вам організувати вашу медіабібліотеку, скануючи ваш комп'ютер для виявлення аудіо та відеофайлів і відображаючи їх в зручному списку. Ви можете створювати плейлисти, впорядковувати файли за різними категоріями та виконувати пошук;

3. Вбудований еквалайзер і ефекти відтворення: WMP має вбудований еквалайзер та підтримує деякі звукові ефекти, які дозволяють вам налаштувати звук під свої потреби та вподобання.

Недоліки програми:

1. Обмежена підтримка форматів: Хоча WMP підтримує багато популярних форматів, він може бути обмежений в підтримці деяких менш популярних аудіо та відеоформатів. Для деяких користувачів це може стати проблемою, особливо якщо вони працюють з екзотичними форматами;

2. Відсутність регулярних оновлень: WMP вбудований в операційну систему Windows і не отримує регулярних оновлень окремо від самої операційної системи. Це означає, що нові функції та поліпшення можуть бути додані рідко або взагалі не додаватися через довгі інтервали між оновленнями Windows;

3. Обмежена функціональність порівняно з альтернативами: У порівнянні з іншими медіаплеєрами, WMP може бути менш функціональним або менш гнучким у відношенні до налаштувань та можливостей.

Dopamine (рис. 1.3) — це аудіопрогравач, який намагається зробити організацію та прослуховування музики максимально простим та красивим [4].

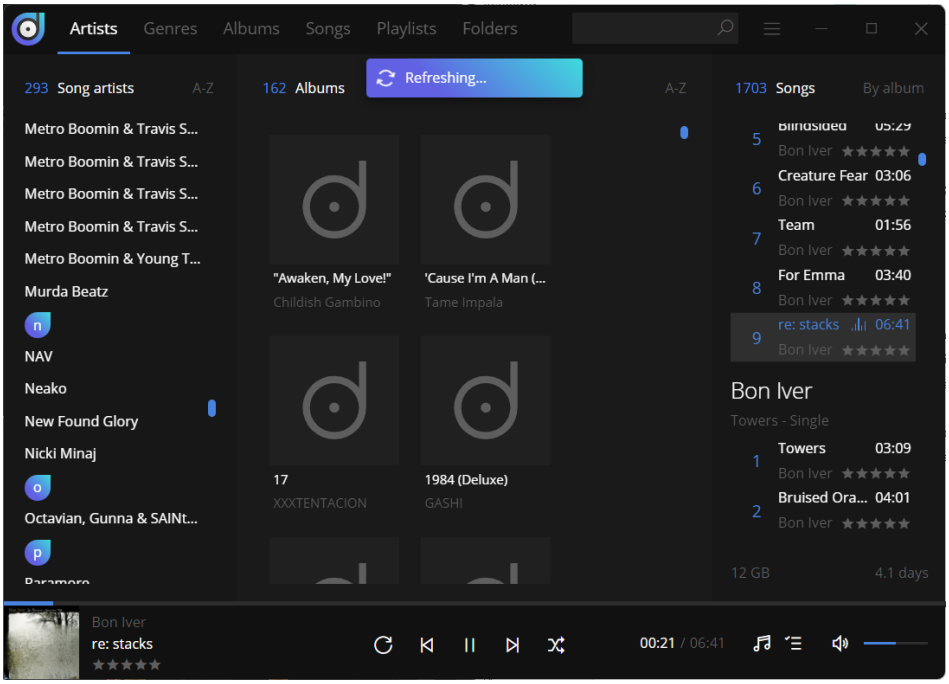

Рисунок 1.3 – Інтерфейс програвача Dopamine

Функції програми Dopamine:

1. Підтримка різних аудіо форматів: Dopamine підтримує широкий спектр аудіо форматів, включаючи MP3, OGG, FLAC, WMA, M4A, та WAV;

2. Інтуїтивний інтерфейс користувача: Простий та зручний інтерфейс для легкого користування;

3. Автоматичне завантаження метаданих: Програма автоматично завантажує та оновлює інформацію про пісні, включаючи обкладинки альбомів, назви пісень та виконавців;

4. Підтримка скробблінгу через Last.fm: Інтеграція з Last.fm для скробблінгу треків.

Недоліки програми Dopamine:

1. Обмежена підтримка платформ: Доступний тільки для Windows;

2. Відсутність еквалайзера: Немає вбудованого еквалайзера для налаштування звуку;

3. Відсутність плейлистів: Немає функції створення та управління плейлистами.

1.3 Постановка задачі та формулювання вимог

Програмні модулі для відтворення музичних файлів є ключовим інструментом для організації та відтворення музичних колекцій. Музичний плеєр може стати незамінним для меломанів та користувачів, які цінують високу якість звуку та зручний інтерфейс користування.

В даній роботі було поставлено завдання створити практичний та зручний програмний модуль для відтворення музичних файлів, що забезпечить високу якість відтворення, приємний інтерфейс та підтримку різних форматів аудіофайлів.

Щоб досягти цього програма повинна мати наступне:

1. Зручний менеджмент файлів та папок: Забезпечення користувачів інтуїтивно зрозумілим інтерфейсом для управління бібліотеками звуків;

2. Функціональна панель управління відтворенням: Надання користувачам можливості легко керувати списками відтворення через панель функціональних кнопок;

3. Приємний та простий UI дизайн: Створення інтерфейсу, що є естетично привабливим та легким у використанні;

4. Зрозумілий та практичний UX дизайн: Забезпечення користувачів логічним та зручним для навігації досвідом використання;

5. Підтримка різноманітних форматів музичних файлів: Можливість відтворення файлів у популярних форматах (wav, mp3, ogg, flac, m4a тощо);

6. Відображення метаданих файлів: Надання інформації про назву, артиста, обкладинку альбому та інші метадані аудіофайлів;

7. Робота з плейлистами: Можливість створення, збереження та завантаження списків відтворення, а також автоматичний перехід між піснями;

8. Індивідуальне налаштування звуку: Можливість зміни частот, що відтворюються, для унікального досвіду прослуховування користувачем, за його вподобанням.

1.4 Висновок до розділу 1

У цьому розділі було проведено огляд та аналіз різних програмних рішень для відтворення аудіофайлів на платформі ОС Windows. Було встановлено, що кожне з них має свої недоліки та особливості. Основною вимогою до таких програм є забезпечення можливості прослуховування файлів та надання розширених функцій, таких як підтримка різних форматів аудіофайлів, управління списками відтворення (плейлистами) та налаштування звуку.

Аналіз дозволив визначити оптимальний варіант для конкретних потреб користувача. Оцінка різних аспектів цих програм дала змогу зробити висновок про їхню ефективність та придатність для використання в різних сценаріях. На основі цього аналізу були сформульовані вимоги щодо подальшого вдосконалення програмного рішення та розробки нових функцій відповідно до потреб користувачів.

Було виконано постановку задачі та описано призначення розроблюваного програмного модулю. Після аналізу аналогів визначено напрямки подальших досліджень.

2 ПРОЕКТУВАННЯ МОДУЛЮ ДЛЯ ВІДТВОРЕННЯ МУЗИЧНИХ ФАЙЛІВ

2.1 Вибір платформи модулю

Windows Presentation Foundation (WPF) — це платформа для створення інтерфейсів користувача, яка дозволяє розробляти настільні клієнтські програми. WPF пропонує широкий спектр функцій для розробки додатків, включаючи модель додатків, ресурси, елементи управління, макет, графіку, зв'язування даних, документи та захист. Для декларативного програмування додатків WPF використовує мову розмітки Extensible Application Markup Language (XAML) [5]. В основному архітектура складається з трьох рівнів: керованого шару, некерованого шару і основного елемента операційної системи. В основному ці рівні являють собою набір збірок, які охоплюють всю структуру [6]. Основна архітектура показана на рисунку 2.1:

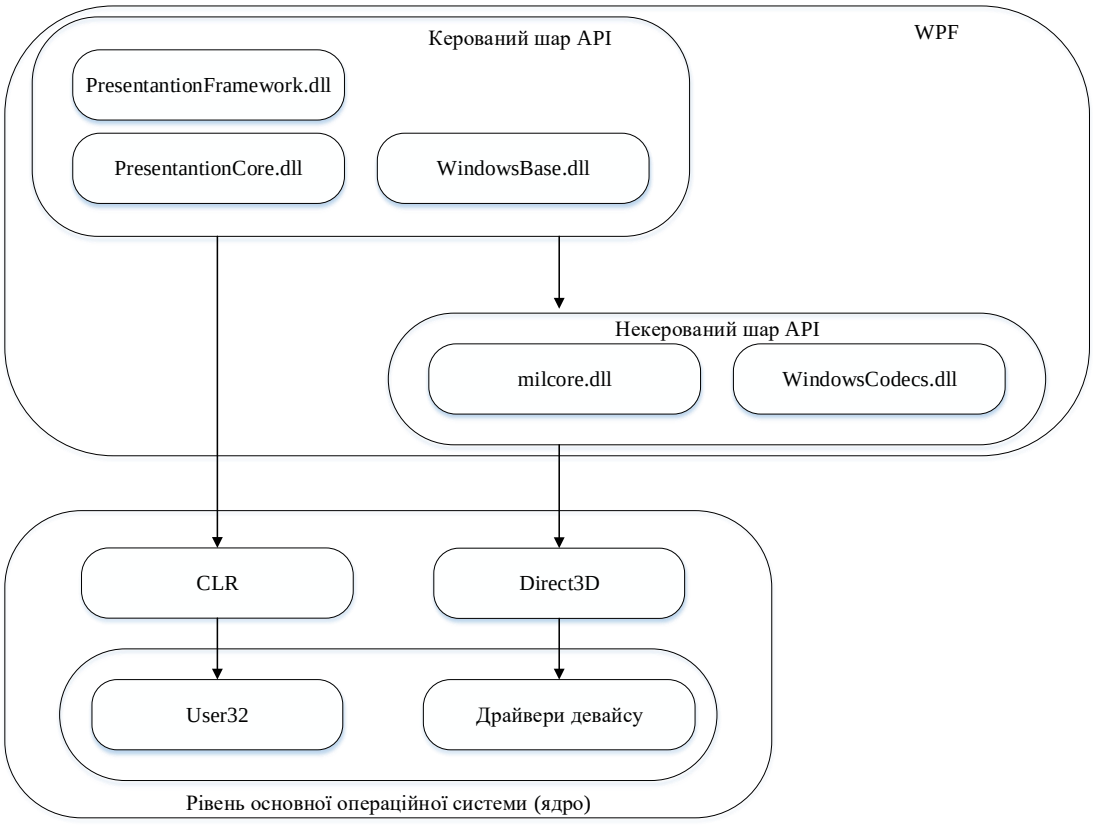

Рисунок 2.1 – Схема основної архітектури WPF

WPF створено для того, щоб дозволити розробникам створювати динамічні системи представлення, керовані даними. Кожен компонент системи побудований на основі об'єктів з набором властивостей, які визначають їх поведінку. Зв'язування даних є ключовою частиною WPF та інтегроване на кожному рівні.

Зазвичай, традиційні програми спочатку створюють інтерфейс, а потім прив'язують його до певних даних. У WPF ж, кожен аспект елемента управління формується через певний тип зв'язування даних. [7].

WPF дозволяє розробляти додатки, комбінуючи розмітку та програмний код, що схоже на підхід в ASP.NET. Зазвичай XAML використовується для визначення інтерфейсу програми, тоді як керовані мови програмування відповідають за її функціональність.

Такий поділ зовнішнього вигляду та поведінки має низку переваг:

– Витрати на розробку та обслуговування знижуються, оскільки розмітка, що визначає зовнішній вигляд, не пов'язана тісно з кодом, що зумовлює поведінку;

– Підвищується ефективність розробки, оскільки дизайнери, які займаються зовнішнім виглядом програми, можуть працювати паралельно з розробниками, що реалізують поведінку програми;

– Глобалізація та локалізація програм WPF спрощена.

Можливості взаємодії з користувачем, що забезпечуються моделлю, реалізуються за допомогою сконструйованих елементів управління. У WPF елемент управління – це загальний термін, який відноситься до категорії класів WPF, що розміщуються у вікні або на сторінці, що мають інтерфейс користувача і реалізують деяку поведінку.

Елементи управління найчастіше використовуються для визначення введення даних користувачем та реагування на нього. Система введення WPF використовує як прямі, так і перенаправлені події для підтримки введення тексту, керування фокусом та визначення положення вказівника миші.

Багато програм мають складні вимоги до обробки введення даних. WPF забезпечує систему команд, яка відділяє користувацьке введення від коду, що обробляє ці дії.

Для спрощення розробки програм, платформа WPF надає механізм прив'язки даних для автоматичного виконання цих дій. Основний елемент механізму прив'язки даних (рис. 2.2) – клас Binding, завданням якого є прив'язка елемента управління (цільового об'єкта прив'язки) до об'єкта даних (джерелу прив'язки). Цей зв'язок показаний на малюнку нижче.

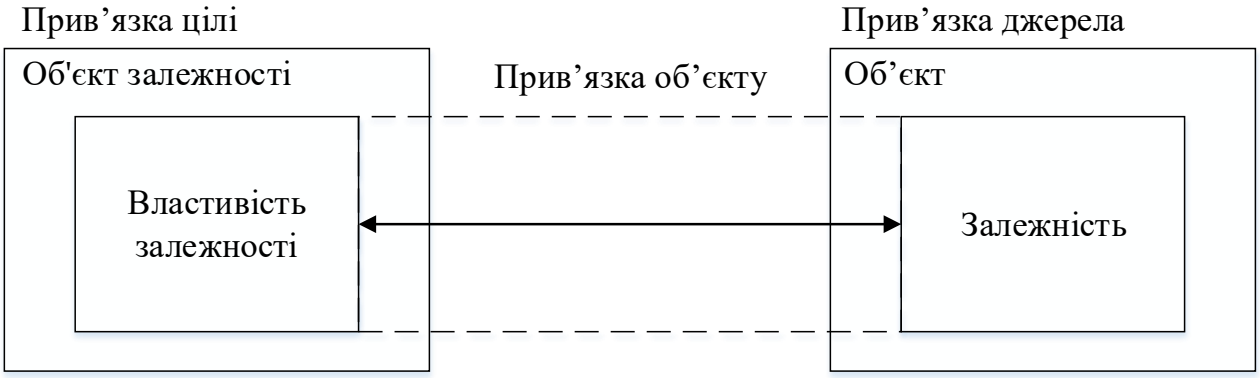

Рисунок 2.2 – Механізм прив’язки даних

Механізм прив'язки даних WPF підтримує додаткові можливості, включаючи перевірку, сортування, фільтрацію та групування. Крім того, прив'язка даних підтримує використання шаблонів даних з метою створення настроюваного інтерфейсу для пов'язаних даних, якщо інтерфейс користувача на основі стандартних елементів керування WPF не задовольняє вимогам.

Одні із способів передачі більш інформативного вмісту – використовувати аудіовізуальні засоби. WPF забезпечує спеціальну підтримку зображень, відео та звуку.

Основним призначенням більшості елементів керування WPF є відображення вмісту. У WPF тип і кількість елементів, що становлять вміст елемента управління, називаються моделлю вмісту елемента управління. Деякі елементи можуть містити лише один об'єкт і лише один тип вмісту.

Хоча основним призначенням розмітки XAML є визначення зовнішнього вигляду програми, її також можна використовувати для деяких аспектів поведінки програми. Один із прикладів - зміна зовнішнього вигляду програми за допомогою тригерів при виконанні користувачем певних дій.

.NET Framework — це платформа для розробки програмного забезпечення, яка дозволяє створювати та запускати програми в середовищі Windows. Це оригінальна реалізація .NET, що підтримує запуск веб-сайтів, сервісів, настільних додатків та іншого в Windows. Два основні компоненти .NET Framework — бібліотека класів .NET Framework і Common Language Runtime. Common Language Runtime (CLR) є виконуючим механізмом, який керує роботою програм, забезпечуючи управління потоками, збирання сміття, безпеку типів, обробку винятків тощо. Бібліотека класів містить набір API і типів для загальної функціональності, включаючи роботу з рядками, датами, числами та інше. Вона також забезпечує API для роботи з файлами, підключення до баз даних, малювання тощо.

Код у .NET Framework компілюється в загальну проміжну мову (CIL), яка є незалежною від мови програмування. Зкомпільований код зберігається у збірках з розширенням .dll або .exe. Під час запуску програми CLR завантажує збірку і за допомогою свого JIT-компілятора перетворює її на машинний код, що може виконуватися на конкретній архітектурі комп'ютера, на якому програма працює (рис. 2.3) [8].

.NET і .NET Framework мають багато однакових компонентів, і ви можете спільно використовувати код для обох. Деякі ключові відмінності включають:

– .NET є кросплатформним і працює на Linux, macOS і Windows. .NET Framework працює лише в Windows;

– .NET є відкритим кодом і приймає внески від спільноти. Вихідний код .NET Framework доступний, але не вимагає прямих внесків;

– Усі інновації відбуваються в .NET;

– .NET Framework включено до складу Windows і автоматично оновлюється на всій машині через Windows Update. .NET поставляється окремо.

Рисунок 2.3 – Архітектура .NET Framework

2.2 Патерни та технології програмного модулю

Патерн Model-View-ViewModel (MVVM) — це шаблон проектування, який розділяє логіку програми та елементи керування інтерфейсом користувача. MVVM організовує код і розбиває програми на модулі, що спрощує розробку,

оновлення та повторне використання коду. Шаблон широко використовується в Windows і веб-графічних додатках.

MVVM застосовується в Windows Presentation Foundation (WPF) і Silverlight, інтернет-еквіваленті мультимедійного плагіну Microsoft WPF, які працюють на платформі .NET [9].

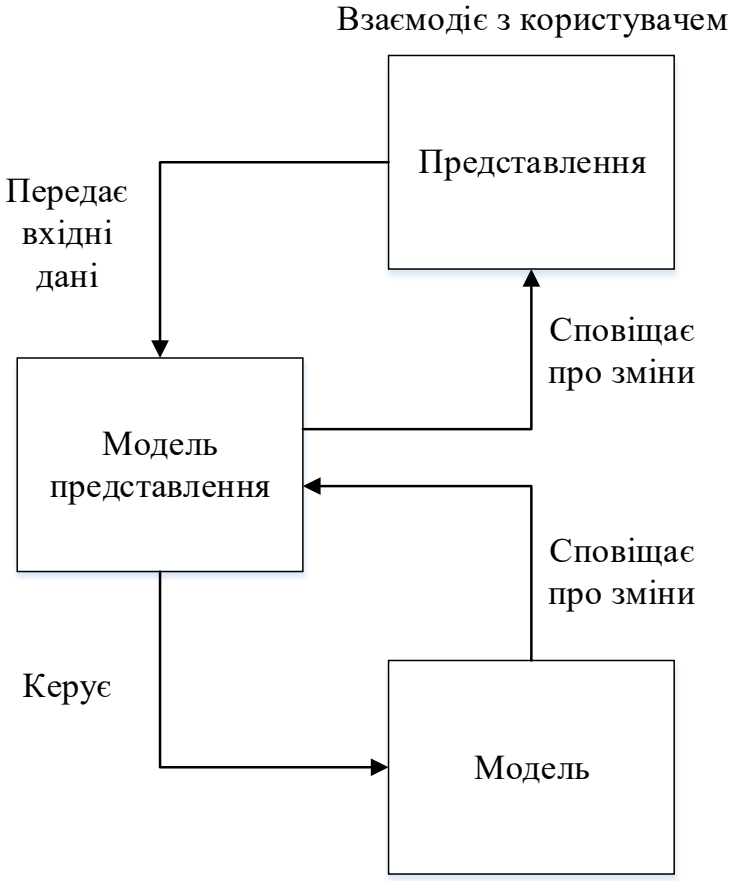

Рисунок 2.4 – Схема роботи патерна MVVM

Шаблон MVVM забезпечує чітке розділення бізнес-логіки програми та її інтерфейсу користувача (UI). Це розділення вирішує багато проблем розробки та полегшує тестування, підтримку і розвиток програм. Крім того, це покращує можливості повторного використання коду і сприяє ефективній співпраці між розробниками та дизайнерами UI. MVVM складається з трьох основних компонентів: моделі, представлення та моделі представлення, кожен з яких виконує свою специфічну роль.

View. View відповідає за структуру, компонування та вигляд, який користувач бачить на екрані. У більшості випадків це визначається в XAML, але іноді може містити інтерфейсну логіку, наприклад, анімацію. Моделі представлення відповідають за стан елементів інтерфейсу, наприклад, доступність команд. Це реалізується через прив'язку елементів до властивостей моделі, а не через код. У відповідь на події, такі як натискання кнопки, код у моделі представлення може виконувати дії, які можуть бути прив'язані до команд або подій у представленні.

Model. Класи моделі, що невізуальні, вбирають у себе дані програми і представляють модель домену програми. Зазвичай ця модель включає дані разом з бізнес-логікою та логікою перевірки. Об'єкти моделі можуть бути об'єктами передачі даних (DTO), звичайними об'єктами CLR (POCO), а також згенерованими об'єктами та проксі-об'єктами. Ці класи моделі зазвичай використовуються разом зі службами або репозиторіями, які забезпечують доступ до даних і кешування.

ViewModel. Модель представлення визначає властивості та команди, які можуть бути прив'язані до даних в представленні і сповіщає про будь-які зміни стану через відповідні події. Функціональні можливості, запропоновані інтерфейсом користувача, визначаються властивостями та командами моделі представлення. Щоб підтримувати продуктивність, модель перегляду використовує асинхронні методи для операцій вводу-виводу та сповіщає представлення про зміни властивостей через події. Модель представлення також координує взаємодію з класами моделі, часто у формі "один-до-багатьох" взаємозв'язку. Щоб забезпечити легке використання даних у представленні, модель перегляду час від часу виконує перетворення даних або використовує конвертери для спеціального форматування. Усі ці дії спрямовані на забезпечення ефективного і двостороннього зв'язку даних з представленням.

Шаблон MVVM полегшує розробку, спрощує тестування і забезпечує кращу співпрацю між розробниками і дизайнерами. Підходить для покращення

повторного використання коду та забезпечує чітке розподіл бізнес-логіки і презентації. [10].

Для безпосередньої роботи з музичними файлами було використано бібліотеку NAudio.

NAudio – це аудіоінструмент з відкритим вихідним кодом .NET, що містить різноманітні класи для роботи з аудіофайлами та пристроями в Windows [11]. Його ключовими характеристиками є:

– Стандартне читання та запис файлів WAV;

– Архітектура потокового потоку хвиль, що включає різні мікшери, перетворення форматів і деякі основні ефекти;

– Повний доступ до встановлених кодеків ACM;

– Можна отримати доступ до окремих зразків WAV;

– Відтворення та запис аудіо за допомогою WinMM API;

– Обробка стиснених файлів WAV за допомогою;

– Широка обробка MIDI-файлів;

– Можливість отримувати MIDI-події;

– Читання звукових шрифтів;

– Кодування файлів OGG;

– Різні експериментальні функції DSP;

– Вибір елементів керування Windows Forms для графічних інтерфейсів, пов’язаних із аудіо;

– Читання барабанної карти Cakewalk;

– Зчитування файлу семплера RGCAudio sfz;

– Виведення звуку за допомогою DirectSound;

– Аудіо вихід за допомогою ASIO;

– Обгортки API Vista CoreAudio;

– Обгортки Media Object DirectX.

Була використана бібліотека TagLib для роботи з метаданими музичних файлів. TagLib дозволяє читати та редагувати метадані різних аудіоформатів, таких як MP3, Ogg Vorbis, FLAC, і багато інших. Вона відома своєю швидкодією,

чистотою коду, простотою використання та потужними можливостями. Ось деякі з її ключових переваг:

– Висока швидкість: Згідно з тестами, TagLib працює значно швидше за інші бібліотеки, такі як id3lib та libvorbisfile, особливо при читанні тегів;

– Чистий код: TagLib написаний на чистому об'єктно-орієнтованому C++ з використанням стилю програмування, який поширений в середовищах розробки KDE та Qt;

– Простота використання: Бібліотека пропонує високорівневий інтерфейс, який дозволяє легко взаємодіяти з різними форматами файлів, ігноруючи їхні внутрішні відмінності;

– Потужні можливості: Для тих, хто бажає глибокого розуміння, TagLib надає доступ до реалізації окремих форматів файлів і інструменти для розширеної обробки метаданих аудіо;

– Документація: Кожен елемент TagLib, включаючи класи, простори імен, функції та перерахування, має детально описану документацію;

– Підтримка Unicode: TagLib підтримує стандарти Unicode, що дозволяє працювати з різними мовами та символами;

– Розширюваність: Бібліотека дозволяє розширити свої можливості для підтримки конкретних функцій, що можуть знадобитися у власних додатках;

– Незалежність від інструментарію: TagLib, хоча створена з використанням стилю програмування KDE та C++, не має зовнішніх залежностей і надає прив'язки C у стилі Glib [12].

2.3 Висновок до розділу 2

У цьому розділі обґрунтовано вибір платформи для модуля та описано використовувані патерни й технології. Платформа WPF була обрана через її здатність підтримувати комплексну розробку клієнтських додатків з використанням XAML для декларативного програмування інтерфейсів. WPF

забезпечує багатофункціональні можливості для створення графічних інтерфейсів, анімації та мультимедіа, що є важливими для розробки модуля відтворення музичних файлів.

Застосування патерну MVVM дозволяє відокремити бізнес-логіку від візуальної частини додатка, що покращує структуру коду та спрощує процес розробки і тестування. Для роботи з аудіофайлами та їх метаданими використовуються бібліотеки NAudio і TagLib, що забезпечує ефективну обробку та відтворення музичних файлів.

Схематичні зображення архітектури WPF, механізму прив’язки даних та схеми роботи патерну MVVM допомагають візуалізувати структуру системи для кращого розуміння її роботи. Таким чином, обрані платформа, технології та патерни забезпечують ефективну розробку та підтримку модуля для відтворення музичних файлів, відповідаючи вимогам сучасних програмних рішень.

3 РОЗРОБКА ФУНКЦІОНАЛЬНИХ КОМАНД ТА ІНТЕРФЕЙСУ КОРИСТУВАЧА ПРОГРАМНОГО МОДУЛЯ

3.1 Обґрунтування вибору інструментів для реалізації

Для розробки програмного модулю слід визначитися з мовою програмування. Проаналізуємо найбільш популярні варіанти.

Java — це мова програмування, побудована на об’єктно-орієнтованому підході і зорієнтована на класи. Її особливість полягає в тому, що вона розроблена з найменшими можливими залежностями від конкретної реалізації. Це дає можливість розробникам писати програми один раз і запускати їх будь-де (WORA). Іншими словами, скомпільований код на Java може працювати на будь-якій платформі, що підтримує Java, без необхідності повторної компіляції. [13].

Java, одна з найпопулярніших мов програмування, використовується для розробки різноманітних веб-додатків протягом понад двох десятиліть. Мільйони програм побудовані на Java, яка є багатоплатформною, об’єктно-орієнтованою та мережево-орієнтованою мовою. Ця швидка, безпечна та надійна мова програмування знайшла застосування у багатьох галузях, включаючи:

– Розробка ігор: Багато популярних ігор побудовані на Java, включаючи ті, що використовують передові технології, такі як машинне навчання і віртуальна реальність;

– Хмарні обчислення: Java, відома як WORA - Write Once and Run Anywhere, ідеально підходить для хмарних застосунків, які працюють на різних платформах;

– Великі дані: Java використовується для обробки складних та великих обсягів даних в реальному часі;

– Штучний інтелект: Завдяки своїм бібліотекам машинного навчання, Java є потужним інструментом для розробки програм зі штучним інтелектом;

– Інтернет речей: Java використовується для програмування датчиків та інших пристроїв Інтернету речей.

Java здобула популярність завдяки своїй зручності використання, а також через:

– Наявність високоякісних навчальних ресурсів;

– Вбудовані функції та бібліотеки, що полегшують розробку програм;

– Активну спільноту користувачів та підтримку;

– Високоякісні інструменти розробки;

– Незалежність від платформи, що дозволяє запускати код на різних пристроях;

– Вбудовану безпеку, яка забезпечує безпечне виконання коду в середовищі Java [14].

C# представляє собою сучасну мову програмування загального призначення, яка здатна вирішувати різноманітні завдання та цілі, що охоплюють широкий спектр професій. Хоча C# в основному використовується у середовищі Windows .NET, він також може бути застосований до платформ з відкритим кодом. Ця дуже універсальна мова програмування має об’єктно-орієнтовану структуру і, хоча є відносно новою для гри, вона вже має своїх прихильників [15]. Використання C# розглядається у таких контекстах, як:

– Розробка програмного забезпечення: C# створено компанією Microsoft для власних потреб, що пояснює її популярність для настільних додатків Windows. Для оптимального функціонування програм на C# необхідна платформа Windows .NET, що робить її першочерговим вибором для розробки програм, що відповідають архітектурі Microsoft;

– Створення ігор: C# визнано однією з найкращих мов програмування для геймерських проектів і широко використовується для створення популярних ігор, за допомогою Unity Game Engine;

– Розробка веб-сайтів: C# активно застосовується для розробки професійних динамічних веб-сайтів на платформі .NET або в середовищі з

відкритим кодом. Навіть якщо ви не прихильник архітектури Microsoft, ви можете використовувати C# для створення повнофункціонального веб-сайту. Завдяки об’єктно-орієнтованій природі мови, веб-сайти, розроблені на C#, відрізняються високою ефективністю, легкістю масштабування та простотою обслуговування.

C# має кілька переваг, які роблять його популярною мовою програмування. Хоча його відносно легко вивчити, особливо якщо ви знайомі з іншими мовами C, C# можна використовувати для розробки багатьох різних типів програм, програмного забезпечення та платформ. Ось кілька факторів, які роблять C# популярною мовою програмування [16]:

– Швидший розвиток. У C# є функції, які допомагають скоротити час розробки, а ефективний і простий код, який не складно читати, допомагає обмежити час, витрачений на налагодження;

– Велика громада. C# використовується в усьому світі, і розробники C# готові допомогти, якщо у вас виникли проблеми з кодом;

– Висока масштабованість. Це полегшує вам підтримку та розширення ваших проектів.

Python є високорівневою інтерпретованою мовою програмування з об’єктно-орієнтованою структурою та динамічною семантикою. Його вбудовані структури даних, спільно з динамічною типізацією та зв'язуванням, роблять його дуже зручним для швидкої розробки програм та використання у якості мови сценаріїв або з’єднувальної мови для існуючих компонентів [17].

Переваги використання Python включають:

– Легкий для вивчення синтаксис, який підкреслює читабельність і знижує вартість обслуговування програми;

– Підтримка модулів та пакетів, що сприяє модульності програми та повторному використанню коду;

– Доступність інтерпретатора Python та обширної стандартної бібліотеки у вихідному або двійковому вигляді безкоштовно для всіх основних платформ і можливість вільного поширення.

Розглянемо деякі з поширених способів використання цієї мови програмування [18]:

– Аналіз даних і машинне навчання: Python є незамінним інструментом у сфері науки про дані, дозволяючи проводити складні статистичні обчислення, створювати візуалізації даних, розробляти алгоритми машинного навчання та оброблювати дані;

– Веб-розробка: Python визнаний лідером у створенні бекенду веб-сайтів та додатків, забезпечуючи надійну обробку даних, зв'язок з базами даних та забезпечення безпеки;

– Автоматизація або сценарії: Python дозволяє ефективно автоматизувати рутинні задачі та створювати сценарії для автоматичного виконання різних операцій;

– Тестування програмного забезпечення та створення прототипів: Python використовується для високоефективного тестування нових продуктів і функцій, а також для створення прототипів програмного забезпечення.

Враховуючи вищесказану інформацію, мовою програмування для розробки на Windows програмного модулю було обрано C#.

3.2 Розробка класу для роботи з командами

Команди оголошуються в класі MainWindowViewModel і ініціалізуються методом LoadCommands, який присвоює їм нові значення RelayCommand. Цей метод викликається в конструкторі класу.

Для використання команд у проекті реалізовано клас RelayCommand, що зображено на рисунку 3.1, який реалізує інтерфейс ICommand.

```
    public class RelayCommand : ICommand
    {
        private readonly Action<object> _execute;
        private readonly Predicate<object> _canExecute;

        public RelayCommand(Action<object> execute, Predicate<object>
canExecute)
        {
            _execute = execute;
            _canExecute = canExecute;
        }
        public bool CanExecute(object parameter)
        {
            bool result = _canExecute == null ? true :
_canExecute(parameter);
            return result;
        }

        public void Execute(object parameter)
        {
            _execute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
    }
```

Рисунок 3.1 – Клас RelayCommand

Клас RelayCommand реалізує інтерфейс ICommand і призначений для спрощення роботи з командами в WPF (Windows Presentation Foundation) додатках. Команди використовуються для зв'язування дій користувача (наприклад, натискання кнопки) з логікою в коді, забезпечуючи тим самим реалізацію патерну MVVM (Model-View-ViewModel).

У класі реалізовано два методи: Execute і CanExecute, а також подію CanExecuteChanged.

– Метод Execute визначає дію, яка виконується при виклику команди;

– Метод CanExecute визначає, чи може ця команда бути виконана в даний момент;

– Подія CanExecuteChanged спрацьовує при змінах у виконанні команди.

RelayCommand спрощує роботу з командами, дозволяючи легко реалізовувати дії та перевірку можливості їх виконання, що є важливим для побудови додатків з чистою архітектурою та зручним управлінням станом інтерфейсу користувача.

3.3 Розробка команд меню

Для додавання музичного файлу одного за одним до списку відтворення використовується за допомогою команди CommandPlaylistAddFile. Ця команда реалізується за допомогою методів CanPlaylistAddFile (рис. 3.2) та PlaylistAddFile (рис. 3.3).

Метод CanPlaylistAddFile визначає, чи можна додавати файл до списку відтворення. Він перевіряє поточний стан відтворення (_playbackState). Якщо відтворення припинено (PlaybackState.Stopped), то метод повертає true, що дозволяє додавати файл до списку відтворення. Інакше, якщо відтворення активне, метод повертає false, що показує, що файл не може бути доданий до списку відтворення.

```
private bool CanPlaylistAddFile(object p)
{
    if (_playbackState == PlaybackState.Stopped)
    {
        return true;
    }
    return false;
}
```

Рисунок 3.2 - Метод CanPlaylistAddFile

Метод PlaylistAddFile виконує додавання музичного файлу до списку відтворення. Спочатку створюється екземпляр класу OpenFileDialog (ofd), який

встановлює фільтр для відображення лише аудіофайлів з розширеннями .wav, .mp3, .wma, .ogg, .flac, .m4a. Після відкриття діалогового вікна користувач може вибрати файл. Якщо вибір файлу підтверджено користувачем, створюється новий об'єкт класу Composition, який представляє музичну композицію з вказаним шляхом до файлу (ofd.FileName), назвою композиції та обкладинкою (якщо такі дані доступні). Потім ця композиція додається до списку відтворення.

```
        private void PlaylistAddFile(object p)
        {
            var ofd = new OpenFileDialog
            {
                Filter = "Audio files (*.wav, *.mp3, *.wma, *.ogg, *.flac, *.m4a) | *.wav;
*.mp3; *.wma; *.ogg; *.flac, *.m4a"
            };
            var result = ofd.ShowDialog();
            if (result == true)
            {
                var composition = new Composition(ofd.FileName,
                    MetadataReaderHelper.Titles_Changed(ofd.FileName),
                    MetadataReaderHelper.Image_Changed(ofd.FileName));
                Playlist.Add(composition);
            }
        }
```

Рисунок 3.3 - Метод PlaylistAddFile

Однак додавання файлів по одному може займати значну кількість часу. Щоб уникнути цього, використовується команда CommandPlaylistAddFolder, яка дозволяє завантажити весь каталог музичних файлів одразу в плейлист. Це значно прискорює взаємодію з музичною бібліотекою. Сама команда складається з методів PlaylistAddFolder (рис. 3.4) та CanPlaylistAddFolder (рис. 3.5).

Метод PlaylistAddFolder додає всі музичні файли з обраної користувачем папки до списку відтворення. Спочатку користувач обирає папку за допомогою діалогового вікна, після чого програма отримує список всіх музичних файлів у цій папці та її підпапках. Лише файли з підтримуваними форматами (наприклад,

.wav, .mp3) додаються до списку відтворення. Кожен файл обробляється для отримання короткої назви та додавання до списку відтворення.

```
private void PlaylistAddFolder(object p)
{
    var cofd = new CommonOpenFileDialog
    {
        IsFolderPicker = true
    };
    var result = cofd.ShowDialog();
    if (result == CommonFileDialogResult.Ok)
    {
        var folderName = cofd.FileName;
        var audioFiles = Directory
            .EnumerateFiles(folderName, "*.*", SearchOption.AllDirectories)
            .Where(f => f.EndsWith(".wav") ||
            f.EndsWith(".mp3") || f.EndsWith(".wma") ||
            f.EndsWith(".ogg") || f.EndsWith(".flac") ||
            f.EndsWith(".m4a"));
        foreach (var audioFile in audioFiles)
        {
            var pathRemove = audioFile.PathRemove();
            var shortenedTitleName = pathRemove.Remove(pathRemove.Length - 4);
            var composition = new Composition(audioFile,
                MetadataReaderHelper.Titles_Changed(audioFile),
                MetadataReaderHelper.Image_Changed(audioFile));
            Playlist.Add(composition);
        }
        Playlist = new ObservableCollection<Composition>
            (Playlist.OrderBy(z => z.ShortenedTitleName).ToList());
    }
}
```

Рисунок 3.4 - Метод PlaylistAddFolder

Метод CanPlaylistAddFolder визначає, чи можна додавати папку до списку відтворення. Він перевіряє поточний стан відтворення (_playbackState). Якщо відтворення припинено (PlaybackState.Stopped), то метод повертає true, що дозволяє додавати папку до списку відтворення. Інакше, якщо відтворення активне, метод повертає false, що показує, що папка не може бути додана до списку відтворення.

```
private bool CanPlaylistAddFolder(object p)
{
    if (_playbackState == PlaybackState.Stopped)
    {
        return true;
    }
    return false;
}
```

Рисунок 3.5 - Метод CanPlaylistAddFolder

Якщо потрібно зберегти список відтворення, можна скористатися функціоналом плеєра, який дозволяє зберегти поточний список відтворення. Це можна зробити за допомогою команди CommandPlaylistSave, яка включає методи CanPlaylistSave і PlaylistSave (рис. 3.6).

Метод PlaylistSave відповідає за збереження плейлисту. Спочатку він створює діалогове вікно збереження файлу за допомогою класу SaveFileDialog. Користувач може обрати місце та назву файлу для збереження плейлисту. Якщо користувач підтверджує свій вибір, то виконується операція збереження плейлисту за допомогою об'єкта SaverPlaylist. Цей об'єкт отримує список композицій (Playlist) та шлях до файлу для збереження, який обрано користувачем.

```
private void PlaylistSave(object p)
{
    var sfd = new SaveFileDialog
    {
        CreatePrompt = false,
        OverwritePrompt = true,
        Filter = "PLAYLIST files (*.playlist) | *.playlist"
    };
    if (sfd.ShowDialog() == true)
    {
        var ps = new SaverPlaylist();
        ps.Save(Playlist, sfd.FileName);
    }
}
```

Рисунок 3.6 - Метод PlaylistSave

Метод CanPlaylistSave завжди повертає true, щоб завжди дозволяти користувачеві зберігати плейлист.

Для початку відтворення плейлиста композицій його потрібно завантажити в програму. Цю операцію допомагає використання команди CommandPlaylistLoad, яка включає в себе методи PlaylistLoad (рис. 3.7) та CanPlaylistLoad.

Метод PlaylistLoad відповідає за завантаження плейлисту з файлу. Він створює діалогове вікно відкриття файлу за допомогою класу OpenFileDialog. Користувач обирає файл плейлисту, після чого плейлист завантажується з цього файлу за допомогою класу PlaylistUpload. Після завантаження плейлисту він конвертується в ObservableCollection і призначається властивості Playlist.

Метод CanPlaylistLoad також завжди повертає true, щоб завжди дозволяти користувачеві завантажувати плейлисти.

```
private void PlaylistLoad(object p)
{
    var ofd = new OpenFileDialog
    {
        Filter = "PLAYLIST files (*.playlist) | *.playlist"
    };
    if (ofd.ShowDialog() == true)
    {
        Playlist = new PlaylistUpload().Load(ofd.FileName)
            .ToObservableCollection();
    }
    StopPlayback(null);
}
```

Рисунок 3.7 - Метод PlaylistLoad

Якщо потрібно закрити вікно музичного плеєра, використовується команда CommandPlayerExit. Вона включає методи PlayerExit (рис. 3.8) і CanPlayerExit.

Метод PlayerExit відповідає за закриття вікна музичного плеєра. Він спершу перевіряє, чи існують компоненти медіа. Якщо так, вони видаляються

(dispose). Після цього викликається метод Shutdown() з поточного екземпляра додатку (Application.Current), щоб закрити програму.

```
private void PlayerExit(object p)
{
    _mediaComponents?.Dispose();

    Application.Current.Shutdown();
}
```

Рисунок 3.8 – Метод PlayerExit

Метод CanPlayerExit завжди повертає true, що вказує на те, що вихід з програми завжди можливий.

3.4 Розробка команд програвача

Щоб запустити вибрану композицію зі списку відтворення, використовується команда CommandPlaybackStart, яка включає методи StartPlayback, зображений на рисунку 3.9 і CanStartPlayback.

Метод CanStartPlayback перевіряє, чи можливо розпочати відтворення композиції. Він повертає true, якщо поточна вибрана композиція (CurrentlySelectedComposition) не null, інакше повертає false.

Метод StartPlayback запускає відтворення вибраної композиції. Короткий опис функцій:

– Перевіряє, чи вибрана композиція (CurrentlySelectedComposition) не є null.

– Якщо вибрана композиція відрізняється від поточної, створює новий об'єкт MediaComponents для відтворення з необхідними налаштуваннями та обробниками подій.

– Якщо поточна композиція грає, зупиняє її (StopPlayback) та робить паузу.

– Якщо відтворення зупинене, створює новий об'єкт MediaComponents і оновлює тривалість та поточну композицію.

– Запускає або ставить на паузу відтворення через _mediaComponents.TogglePlayPause(VolumeCurrent).

```
        private void StartPlayback(object p)
        {
            if (CurrentlySelectedComposition != null)
            {
                if (CurrentlyPlayingComposition != CurrentlySelectedComposition)
                {
                    if (CurrentlyPlayingComposition == null || _playbackState ==
PlaybackState.Stopped)
                    {
                        _mediaComponents = new
MediaComponents(CurrentlySelectedComposition.Filepath, VolumeCurrent, _bands)
                        {
                            PlaybackStopType =
PlaybackStopTypes.PlaybackStoppedReachingEndOfFile
                        };
                        _mediaComponents.PlaybackPaused += MediaComponents_PlaybackPaused;
                        _mediaComponents.PlaybackResumed += MediaComponents_PlaybackResumed;
                        _mediaComponents.PlaybackStopped += MediaComponents_PlaybackStopped;
                        CurrentCompositionLength = _mediaComponents.GetLengthInSeconds();
                        CurrentlyPlayingComposition = CurrentlySelectedComposition;
                        Duration = GetAudioDuration(CurrentlySelectedComposition.Filepath);
                        CurrentDuration = _mediaComponents.GetSeconds();

                    }
                    else
                    {
                        if (_mediaComponents != null)
                        {
                            StopPlayback(null);
                            Thread.Sleep(100);
                        }
                    }
                }
                if (_playbackState == PlaybackState.Stopped)
                {
                    _mediaComponents = new
MediaComponents(CurrentlySelectedComposition.Filepath, VolumeCurrent, _bands)
                    {
                        PlaybackStopType = PlaybackStopTypes.PlaybackStoppedReachingEndOfFile
                    };
                    _mediaComponents.PlaybackPaused += MediaComponents_PlaybackPaused;
                    _mediaComponents.PlaybackResumed += MediaComponents_PlaybackResumed;
                    _mediaComponents.PlaybackStopped += MediaComponents_PlaybackStopped;
                    CurrentCompositionLength = _mediaComponents.GetLengthInSeconds();
                    CurrentlyPlayingComposition = CurrentlySelectedComposition;
                    Duration = GetAudioDuration(CurrentlySelectedComposition.Filepath);
                    CurrentDuration = _mediaComponents.GetSeconds();

                }
                PropertyChanged += OnPropertyChanged;
                _mediaComponents.TogglePlayPause(VolumeCurrent);
            }
        }
```

Рисунок 3.9 – Метод StartPlayback

Для повної зупинки композиції використовується команда CommandPlaybackStop, яка включає методи StopPlayback, зображений на рисунку 3.10 і CanStopPlayback.

Метод StopPlayback зупиняє відтворення композиції, якщо _mediaComponents існує, встановлює тип зупинки на PlaybackStoppedByUser і викликає Stop() та змінює стан PlayStop на false і обнуляє CurrentDuration.

Метод CanStopPlayback повертає true, якщо відтворення перебуває в стані Playing або Paused, інакше повертає false.

```
private void StopPlayback(object p)
{
    if (_mediaComponents != null)
    {
        _mediaComponents.PlaybackStopType = PlaybackStopTypes.PlaybackStoppedByUser;
        _mediaComponents.Stop();
    }
    PlayStop = false;
    CurrentDuration = TimeSpan.Zero;
}
```

Рисунок 3.10 – Метод StopPlayback

Команда BackToStart при натисканні кнопки повертає поточну позицію музичного файлу на початок (0). Кнопка активна, коли трек відтворюється. Ця команда реалізована методами RewindToStart (рис. 3.11) і CanRewindToStart.

Метод RewindToStart повертає поточну позицію відтворення на початок музичного файлу. Спочатку він перевіряє, чи об'єкт _mediaComponents не є null. Якщо об'єкт існує, метод встановлює тип зупинки відтворення на PlaybackWentToStartByUser і викликає метод SetPosition з параметром 0, щоб повернутись на початок файлу.

Метод CanRewindToStart визначає, чи можливо повернутись до початку відтворення, повертаючи true, якщо поточний стан відтворення (_playbackState) є Playing, і false в іншому випадку.

```
        private void RewindToStart(object p)
        {
            if (_mediaComponents != null)
            {
                _mediaComponents.PlaybackStopType =
PlaybackStopTypes.PlaybackWentToStartByUser;
                _mediaComponents.SetPosition(_mediaComponents.GetLengthInSeconds());
            }
        }
```

Рисунок 3.11 – Метод RewindToStart

Команда CommandGoToEnd при натисканні кнопки переміщує поточну позицію композиції до кінця файлу, встановлюючи її на останню секунду пісні. Кнопка активна, коли трек відтворюється. Ця команда реалізована методами ForwardToEnd (рис. 3.12) і CanForwardToEnd.

Метод ForwardToEnd переміщує поточну позицію відтворення до кінця музичного файлу. Він перевіряє, чи об'єкт _mediaComponents не є null. Якщо об'єкт існує, метод встановлює тип зупинки відтворення на PlaybackStoppedReachingEndOfFile і викликає метод SetPosition з параметром, що дорівнює довжині файлу, щоб перейти до кінця файлу.

```
        private void ForwardToEnd(object p)
        {
            if (_mediaComponents != null)
            {
                _mediaComponents.PlaybackStopType =
PlaybackStopTypes.PlaybackStoppedReachingEndOfFile;
                _mediaComponents.SetPosition(_mediaComponents.GetLengthInSeconds());
            }
        }
```

Рисунок 3.12 – Метод ForwardToEnd

Метод CanForwardToEnd визначає, чи можливо перейти до кінця відтворення, повертаючи true, якщо поточний стан відтворення (_playbackState) є Playing, і false в іншому випадку.

Для створення унікальних і непередбачуваних послідовностей у списку відтворення використовується команда CommandShuffle, яка реалізована методами Shuffle (рис. 3.13) і CanShuffle. Ця команда стає доступною тільки після повної зупинки відтворення.

Метод Shuffle перемішує композиції в списку відтворення. Він викликає метод Shuffle для поточного списку відтворення, щоб перемішати його, і встановлює значення PlayStop на false, що зупиняє відтворення.

```
private void Shuffle(object p)
{
    {
        Playlist = Playlist.Shuffle();
        PlayStop = false;
    }
}
```

Рисунок 3.13 – Метод Shuffle

Метод CanShuffle визначає, чи можливо перемішати список відтворення, повертаючи true, якщо поточний стан відтворення (_playbackState) є Stopped, і false в іншому випадку.

3.5 Розробка функції еквалайзеру

Розглянемо реалізацію еквалайзера для аудіопрогравача. Розглянутий фрагмент коду визначає клас для управління еквалайзером. Він містить різні властивості та методи для роботи з окремими смугами еквалайзера. На рисунку 3.14 зображено EqualizerBand[] Bands: Це масив об'єктів типу EqualizerBand,

який представляє смуги еквалайзера з різними параметрами, такими як ширина, частота та підсилення.

```
public EqualizerBand[] Bands
{
    get { return _bands; }
    set
    {
        if (_bands == value) return;
        _bands = value;
        OnPropertyChanged(nameof(Bands));
    }
}
```

Рисунок 3.14 – Масив об'єктів типу EqualizerBand

Розглянемо властивості Band1-Band8: Кожна з цих властивостей відповідає одній із смуг еквалайзера. На рисунку 3.15 зображено властивості Band1. Вони дозволяють отримувати і задавати значення підсилення для відповідної смуги. При зміні значення властивості відбувається перевірка на те, чи нове значення відрізняється від попереднього. Якщо воно відрізняється, викликається подія OnPropertyChanged, що дозволяє оновити інтерфейс користувача.

```
public float Band1
{
    get => _bands[0].Gain;
    set
    {
        if (_bands[0].Gain != value)
        {
            _bands[0].Gain = value;
            OnPropertyChanged("Band1");
        }
    }
}
```

Рисунок 3.15 – Властивості Band1

Також налаштування еквалайзеру мають властивості MinimumGain та MaximumGain: Вони визначають мінімальне та максимальне значення підсилення для смуг еквалайзера та встановлені в -10 та 10 дБ відповідно.

3.6 Розробка подій та властивостей функціональних команд

Подія CommandPressMouseClick активується, коли натискається кнопка миші на слайдері для його управління, тобто для можливості переміщення в певне положення. Її методи: CompositionControlMouseDown (рис. 3.16) та CanCompositionControlMouseDown.

```
private void CompositionControlMouseDown(object p)
{
    if (mediaComponents != null)
    {
        mediaComponents.Pause();
    }
}
```

Рисунок 3.16 – Метод CompositionControlMouseDown

Подія CommandReleaseMouseClick активується при відпусканні кнопки миші на слайдері для його контролю. Вона має методи CompositionControlMouseUp та CanCompositionControlMouseUp.

Подія CommandVolumeControl запускається при зміні значення слайдера гучності. Її методи: VolumeControlValueChanged (рис. 3.17) та CanVolumeControlValueChanged.

```
private void VolumeControlValueChanged(object p)
{
    if (mediaComponents != null)
    {
        mediaComponents.SetVolume(CurrentVolume);
    }
}
```

Рисунок 3.17 – Метод VolumeControlValueChanged

Подія CommandVolumeMute працює так, що гучність зменшується до нуля при натисканні на кнопку. Вона має два методи: VolumeMute (рис. 3.18) та CanVolumeMute.

```
private void VolumeMute(object p)
{
    if (_mediaComponents != null)
    {
        if (_mediaComponents.GetVolume() == 0)
        {
            _mediaComponents.SetVolume(_savedVolumeLevel);
        }
        else
        {
            _savedVolumeLevel = _mediaComponents.GetVolume();
            _mediaComponents.SetVolume(0);
        }
    }
}
```

Рисунок 3.18 – Метод VolumeMute

Для забезпечення простого доступу до полів класу та структур, а також їх встановлення, використовуються такі властивості:

CurrentTrackPosition (рис. 3.19) — використовується для збереження поточної позиції під час відтворення звуку, в секундах;

```
public double CurrentCompositionPosition
{
    get { return _currentCompositionPosition; }
    set
    {
        if (value.Equals(_currentCompositionPosition)) return;
        _currentCompositionPosition = value;
        OnPropertyChanged(nameof(CurrentCompositionPosition));
    }
}
```

Рисунок 3.19 – Властивість CurrentTrackPosition

CurrentVolume (рис. 3.20) — представляє гучність;

```
public float VolumeCurrent
{
    get { return _volumeCurrent; }
    set
    {
        if (value == _volumeCurrent) return;
        _volumeCurrent = value;
        OnPropertyChanged(nameof(VolumeCurrent));
    }
}
```

Рисунок 3.20 – Властивість CurrentVolume

Playlist (рис. 3.21) — представляє список відтворення;

```
public ObservableCollection<Composition> Playlist
    {
        get { return _playlist; }
        set
        {
            if (Equals(value, _playlist)) return;
            _playlist = value;
            OnPropertyChanged(nameof(Playlist));
        }
    }
```

Рисунок 3.21 – Властивість Playlist

CurrentlySelectedComposition (рис. 3.22) — представляє поточно вибраний трек у списку відтворення;

```
public Composition CurrentlySelectedComposition
        {
            get { return _currentlySelectedComposition; }
            set
            {
                if (Equals(value, _currentlySelectedComposition)) return;
                _currentlySelectedComposition = value;
                actionImage.Invoke(_currentlySelectedComposition?.Filepath);
                actionArtist.Invoke(_currentlySelectedComposition?.Filepath);
                actionAlbum.Invoke(_currentlySelectedComposition?.Filepath);
                actionTitle.Invoke(_currentlySelectedComposition?.Filepath);
                OnPropertyChanged(nameof(CurrentlySelectedComposition));
            }
        }
```

Рисунок 3.22 – Властивість CurrentlySelectedComposition

CurrentlyPlayingComposition (рис. 3.23) — представляє трек, що зараз відтворюється;

```
public Composition CurrentlyPlayingComposition
{
    get { return _currentlyPlayingComposition; }
    set
    {
        if (Equals(value, _currentlyPlayingComposition)) return;
        _currentlyPlayingComposition = value;
        OnPropertyChanged(nameof(CurrentlyPlayingComposition));
    }
}
```

Рисунок 3.23 – Властивість CurrentlyPlayingComposition

Duration (рис. 3.24) та CurrentDuration (рис. 3.25) — представляють повний час треку та поточний час композиції відповідно.

```
public TimeSpan Duration
{
    get { return _duration; }
    set
    {
        if (value == _duration) return;
        _duration = value;
        OnPropertyChanged(nameof(Duration));
    }
}
```

Рисунок 3.24 – Властивість Duration

```
public TimeSpan CurrentDuration
{
    get { return _currentDuration; }
    set
    {
        if (value.Equals(_currentDuration)) return;
        _currentDuration = value;
        OnPropertyChanged(nameof(CurrentDuration));
    }
}
```

Рисунок 3.25 – Властивість CurrentDuration

3.7 Розробка інтерфейсу користувача

Для створення інтерфейсу користувача використовується бібліотека Material Design In XAML. Щоб використовувати ці бібліотеки, потрібно встановити пакети MaterialDesignThemes та MaterialDesignColors через диспетчер пакетів, як показано на рисунку 3.26.

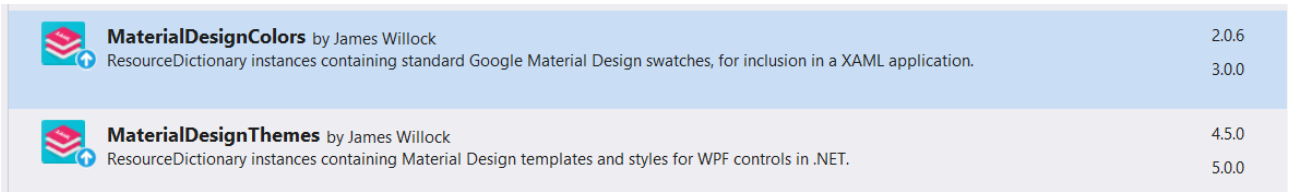

Рисунок 3.26 – Пакети у диспетчері пакетів

Покажемо фрагмент класу App.xaml, а саме реалізацію Material Design в проекті на рисунку 3.27.

```
<ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;
          component/Themes/MaterialDesignTheme.Light.xaml" />
        <ResourceDictionary Source="pack://application:,,,/MaterialDesignThemes.Wpf;
          component/Themes/MaterialDesignTheme.Defaults.xaml" />
        <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;
          component/Themes/Recommended/Primary/MaterialDesignColor.DeepPurple.xaml" />
        <ResourceDictionary Source="pack://application:,,,/MaterialDesignColors;
          component/Themes/Recommended/Accent/MaterialDesignColor.Lime.xaml" />
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Рисунок 3.27 – Реалізація Material Design в App.xaml

Додамо до класу MainWindow.xaml код для реалізації Material Design в проекті, цей фрагмент вказано на рисунку 3.28.

```
xmlns:materialDesign="http://materialdesigninxaml.net/winfx/xaml/themes"
```

Рисунок 3.28 – Реалізація Material Design в MainWindow.xaml

Розглянемо загальну структуру проекта, що наведена на рисунку 3.29. Проєкт складається з двох основних частин: загальних компонентів (Common) та основного додатку (MusPlayerApp). Common містить загальні компоненти, такі як методи розширення для ObservableCollection в ObservableCollectionExtensionMethods.cs.

MusPlayerApp складається з підкаталогів, які включають: AudioOperations для обробки медіа, Command для шаблону команд, Helpers для допоміжних класів, Image для зображень, Models для даних, Services для управління

плейлистами, ViewModels для моделей представлення, та Views для інтерфейсу користувача.

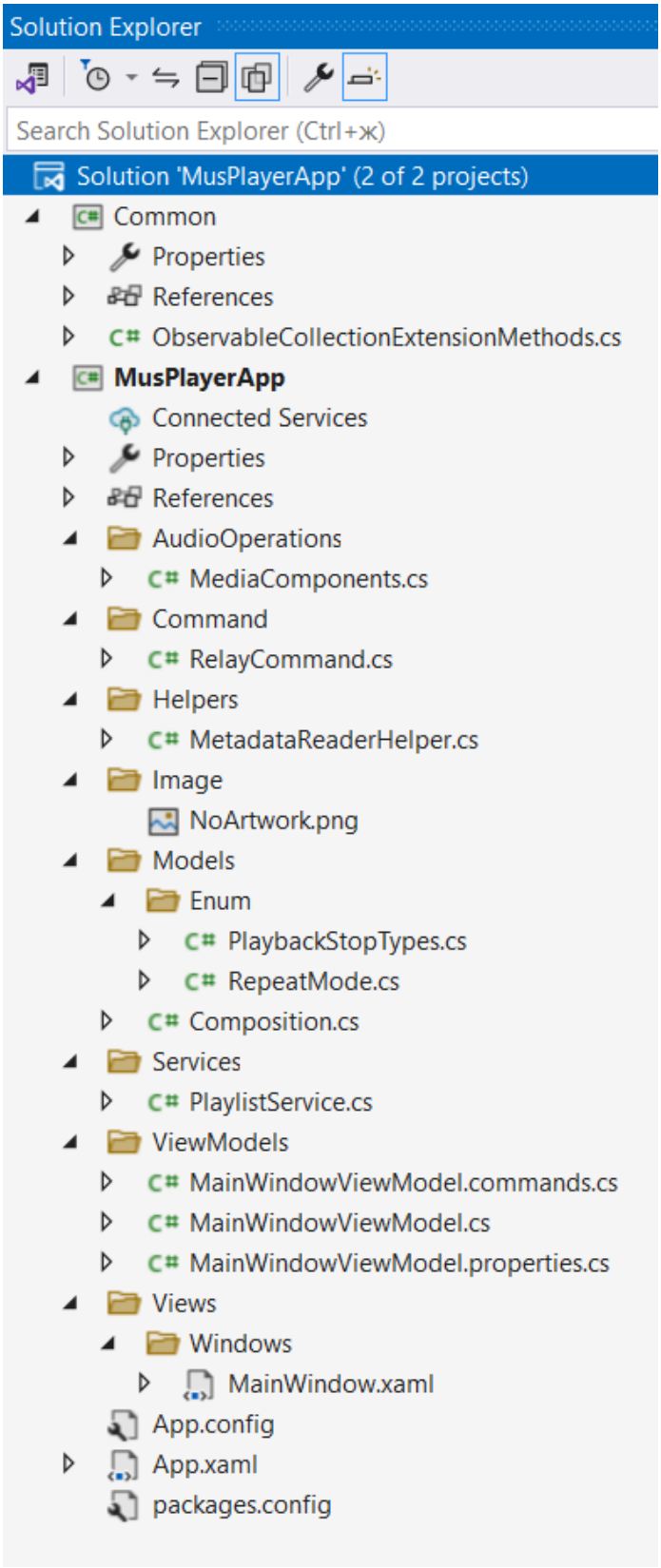

Рисунок 3.29 – Загальна структура програмної реалізації модуля

3.8 Висновок до розділу 3

В даному розділі обґрунтовано вибір мови програмування для втілення функціональних команд та користувацького інтерфейсу і розглянуто основні етапи розробки.

Обрано мову програмування C#, так як для розробки програмних модулів на Windows вона надає певні переваги.

Розглянуто розробку та імплементацію функціональних команд, команд меню, команд програвача та функцій еквалізації, подій та властивостей. Показано реалізацію патерна MVVM, що дозволяє відокремити бізнес-логіку від візуальної частини додатка, що покращує структуру коду та спрощує процес розробки і тестування. Розглянуто загальну структуру проекту.

Описано розробку користувацького інтерфейсу за допомогою Material Design в WPF. Цей підхід допомагає створити інтерфейс, який не лише привабливий з зовнішнього вигляду, але й забезпечує зручну та ефективну взаємодію з користувачем.

4 ТЕСТУВАННЯ РОЗРОБЛЕНИХ ФУНКЦІОНАЛЬНИХ КОМАНД ТА ІНТЕРФЕЙСУ КОРИСТУВАЧА

4.1 Тестування роботи списку відтворення

Щоб розпочати виконання даного тесту, використаємо кнопку меню, яка дозволяє додати один аудіофайл до списку відтворення.

На рисунку 4.1 зображено інтерфейс списку програвання з кнопками меню, без музичних файлів. Всі кнопки відображені коректно.

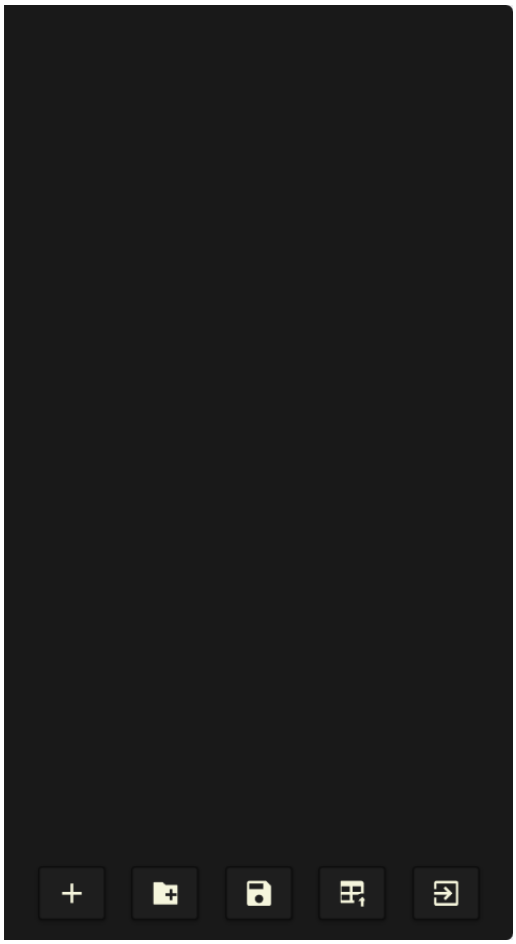

Рисунок 4.1 – Інтерфейс списку програвання без музичних файлів

За допомогою першої кнопки меню додамо до списку відтворення одну композицію. На рисунку 4.2 показано додавання файлу після натискання на першу кнопку меню, яка додає один аудіофайл.

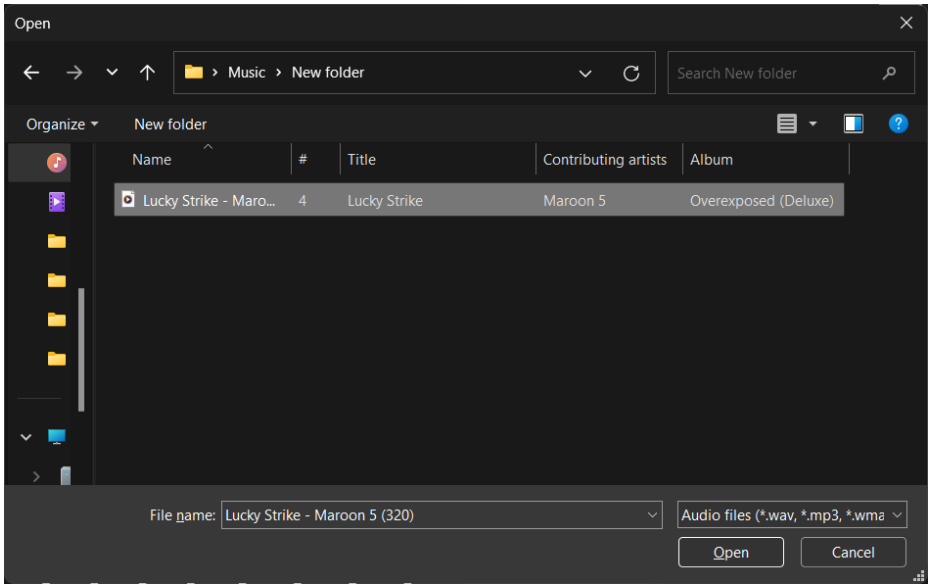

Рисунок 4.2 – Вибір аудіофайлу в діалоговому вікні

Список відтворення поповнився на одну композицію, дані відображені вірно, що видно на рисунку 4.3.

Рисунок 4.3 – Інтерфейс списку програвання з музичним файлом

Тепер використаємо другу кнопку меню, яка відповідає за додавання всіх композицій з папки, що може бути зручно для підбірок, колекцій та альбомів.

На рисунку 4.4 показано вибір папки з аудіофайлами в діалоговому вікні.

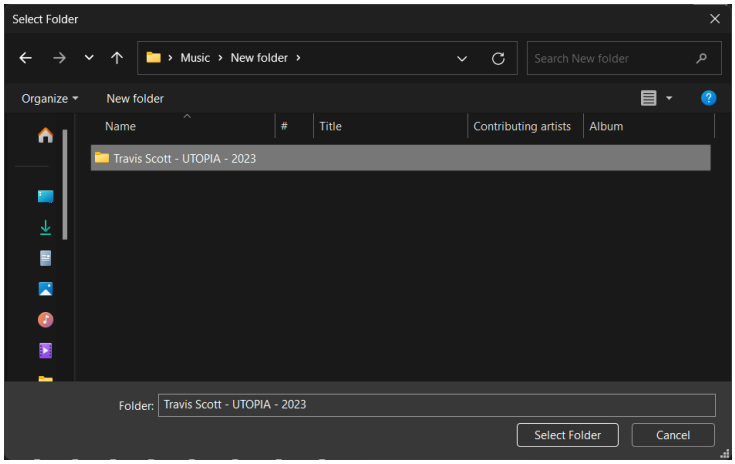

Рисунок 4.4 – Вибір папки (альбому) з аудіофайлами в діалоговому вікні

До списку відтворення додалась та кількість композицій, що знаходилась в папці, що видно на рисунку 4.5.

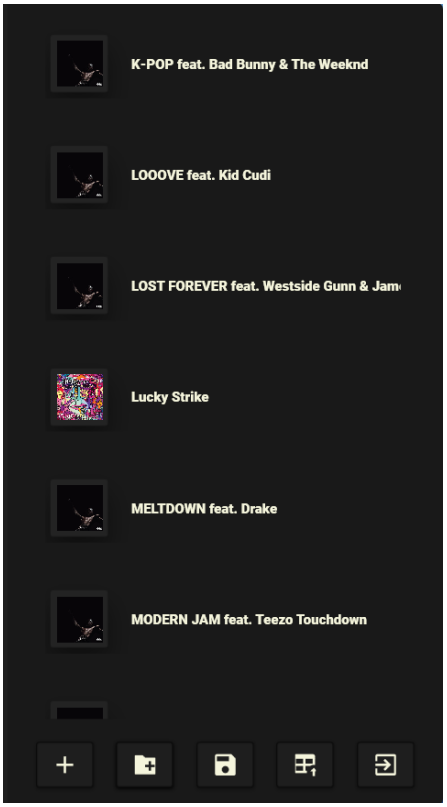

Рисунок 4.5 – Інтерфейс списку програвання з музичними файлами

З додаванням файлу та альбому файлів різних виконавців, було створено унікальний список відтворення, який можна зберегти за бажанням користувача третьою кнопкою меню, яка відповідає за збереження плейлиста. Протестуємо цей механізм, на рисунку 4.6 зображено зберігання файлу формату .playlist.

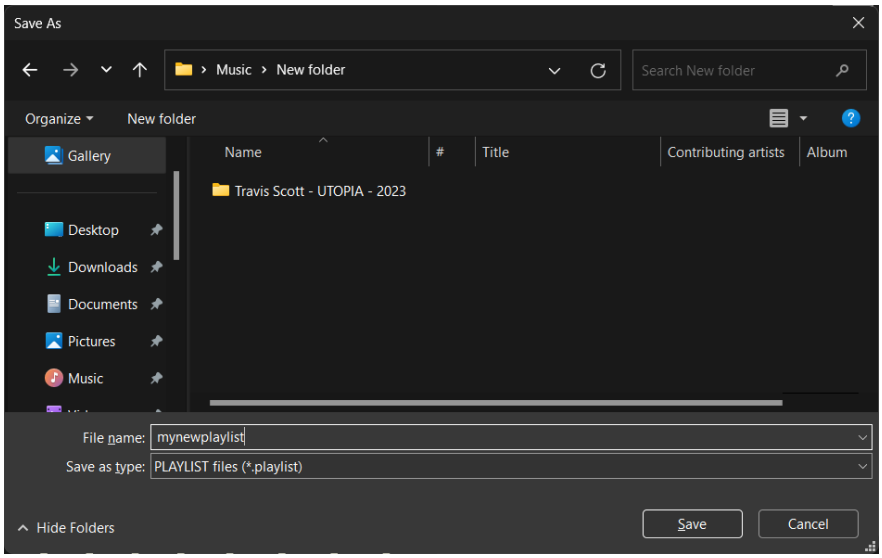

Рисунок 4.6 – Збереження плейлиста в діалоговому вікні

За допомогою четвертої кнопки меню оберемо функцію завантаження плейлиста у список відтворення, на рисунку 4.7 зображено завантаження плейлиста у діалоговому вікні.

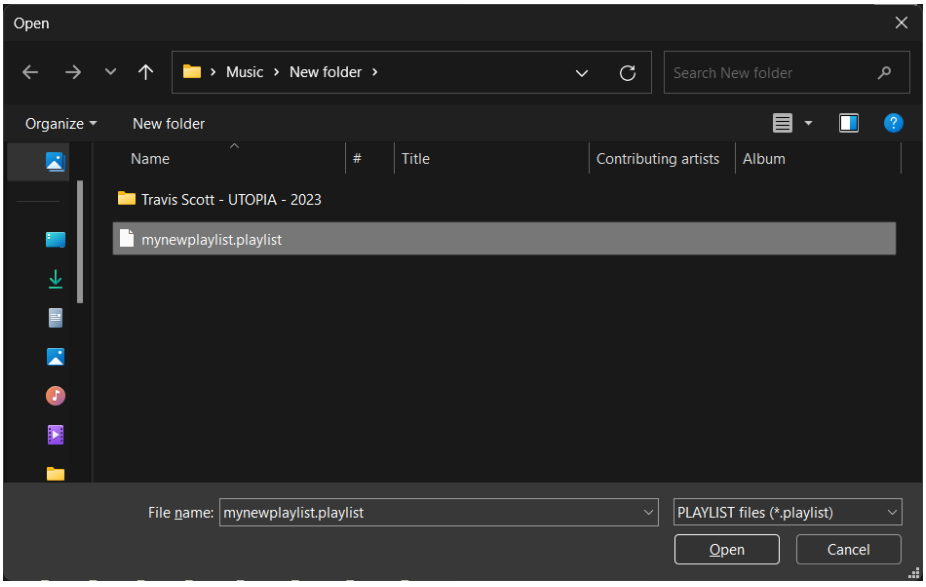

Рисунок 4.7 – Завантаження плейлиста в діалоговому вікні

Результат завантаження плейлиста у список відтворення модулю показано на рисунку 4.8.

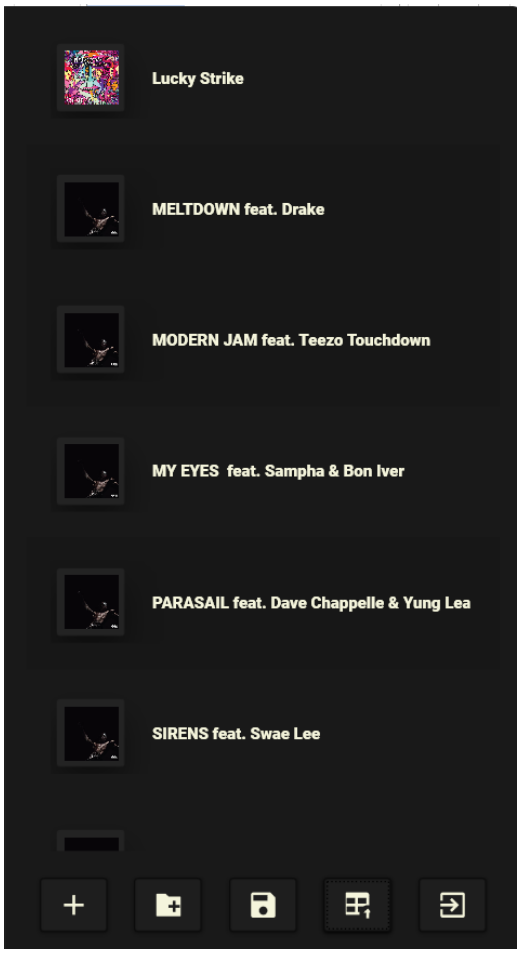

Рисунок 4.8 – Інтерфейс списку програвання з музичними файлами з плейлиста

Розглянемо також функцію, що запобігає втручанню у список відтворення під час програвання аудіофайлу. У програвачі на даний момент відтворюється пісня, кнопки меню в цей час показані на рисунку 4.9.

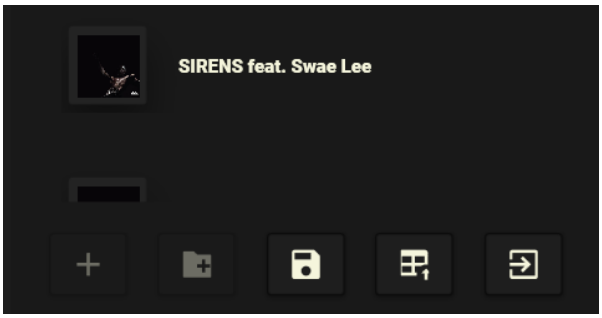

Рисунок 4.9 – Вимкнення функцій додавання під час програвання

Остання кнопка меню відповідає за вихід з програми, вона працює коректно. Отже, у ході тестування було виявлено, що всі кнопки меню та функції закладені в них працюють належним чином, також чітко працює і список відтворення програмного модулю.

4.2 Тестування роботи програвача

Після того як зі списку відтворення було вибрано пісню, інформація про її стан переходить до програвача, отже починає відтворюватись аудіофайл та показуються метадані файлу, що містять інформацію про назву композиції, виконавця, назву альбому, обкладинку та інші. Результат відтворення файлу продемонстровано на рисунку 4.10.

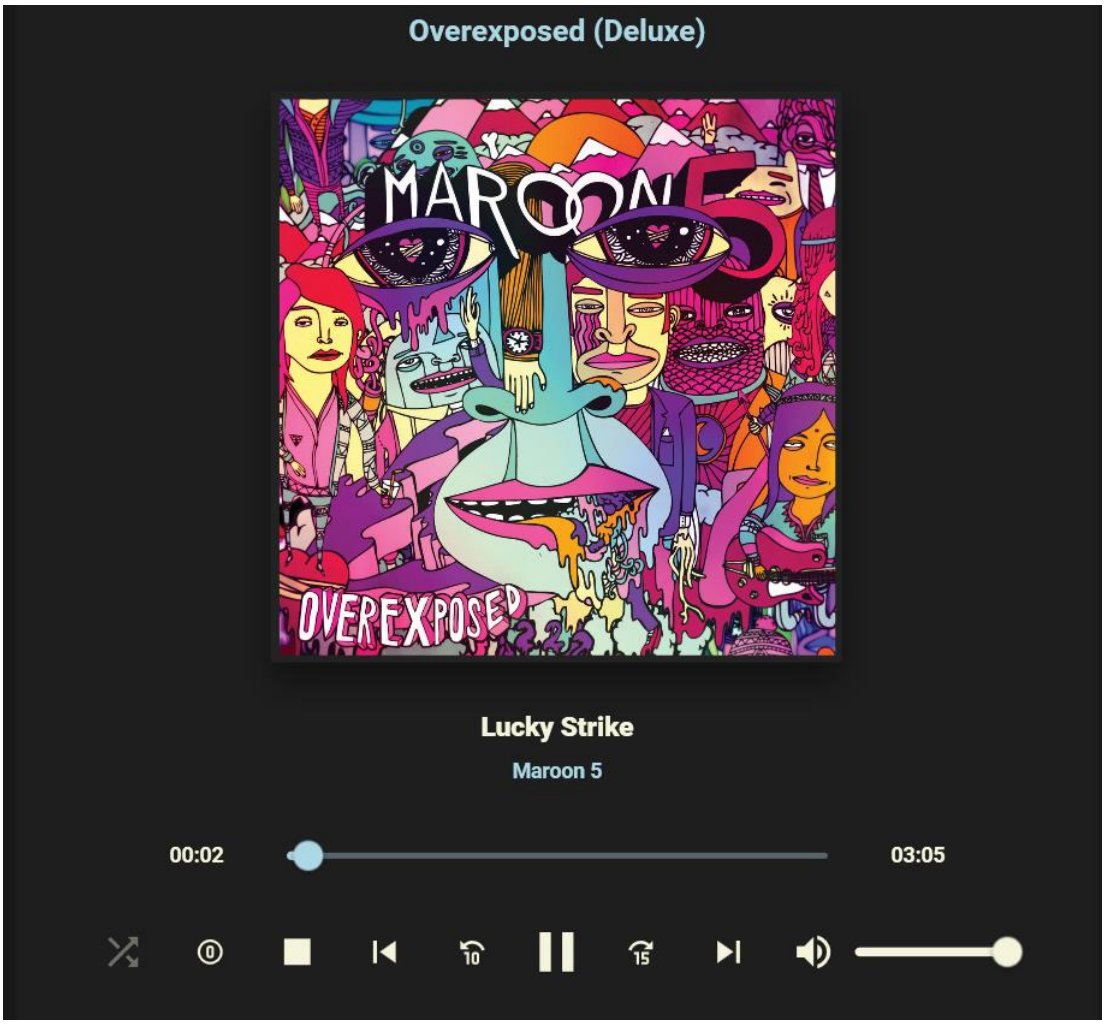

Рисунок 4.10 – Відтворення аудіофайлу у програвачі

Переконавшись в коректній роботі відтворення файлу та відображення його даних, перейдемо до тестування функціональних команд програвача. Під час натиснення на паузу, відтворення призупиняється, та певна кількість функцій блокується для запобігання випадкового натиснення, це зображено на рисунку 4.11.

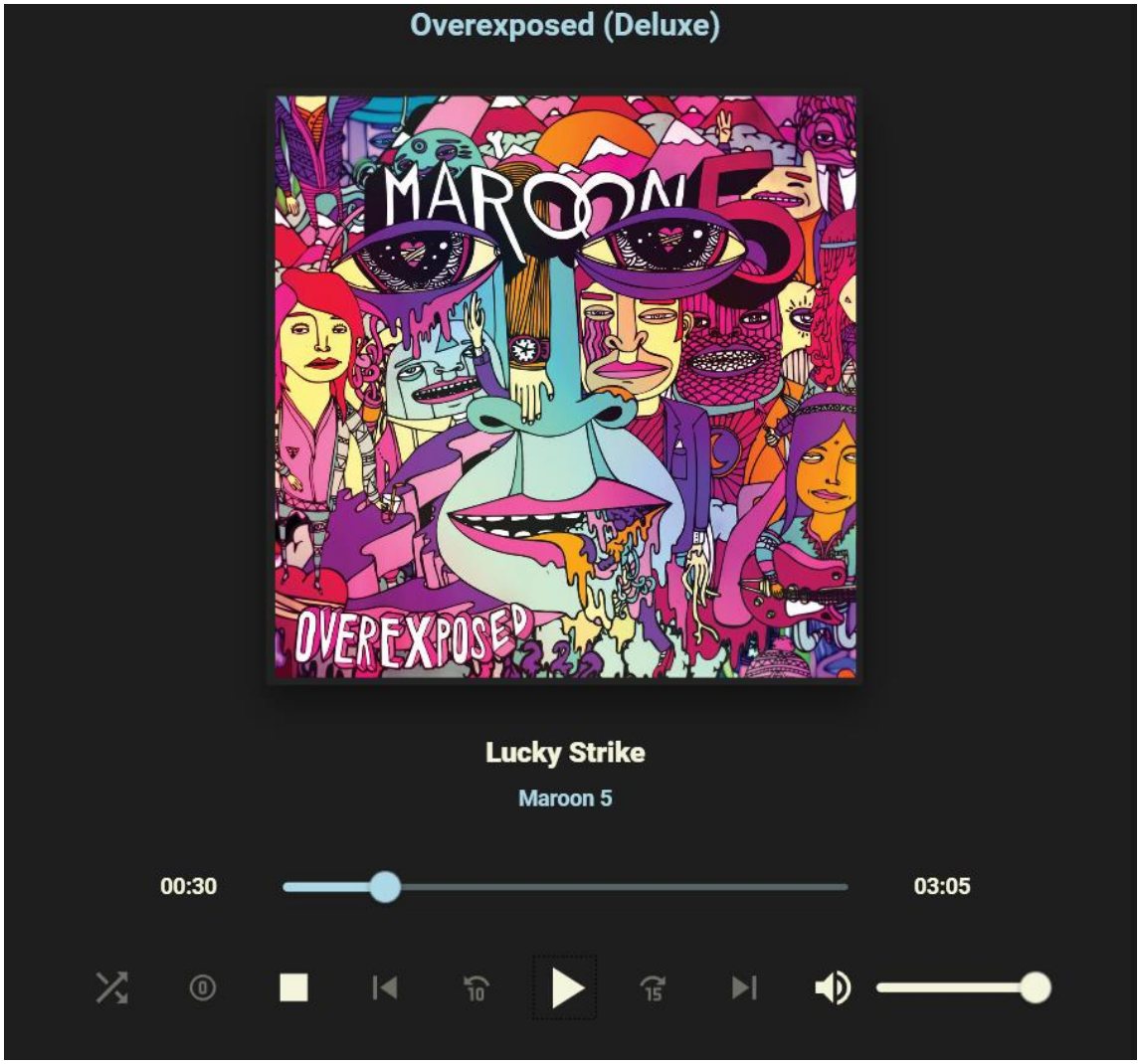

Рисунок 4.11 – Призупинення аудіофайлу у програвачі

Також протестуємо функцію повної зупинки роботи з аудіофайлом, її робота показано на рисунку 4.12.

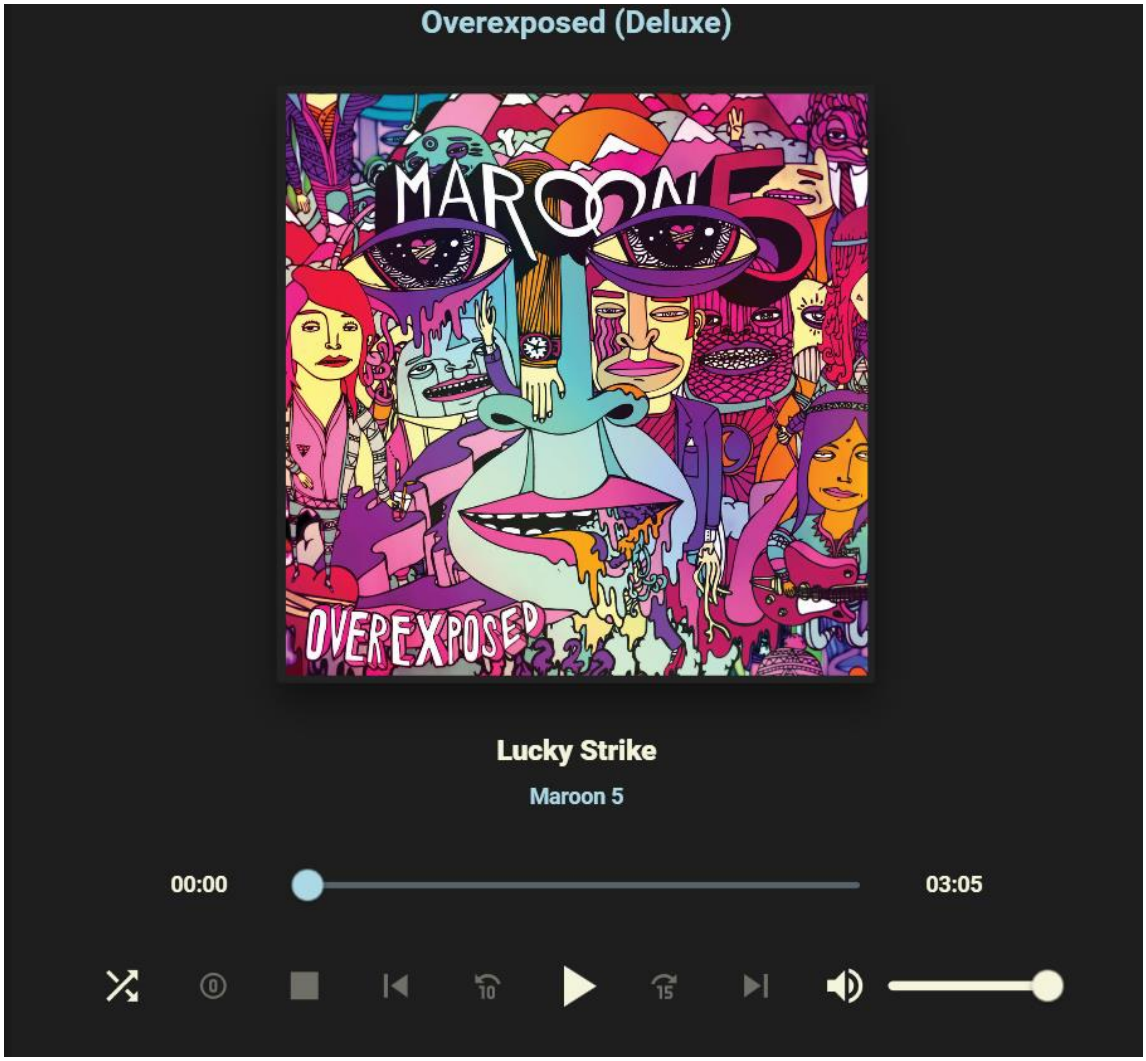

Рисунок 4.12 – Повна зупинка аудіофайлу у програвачі

Було також протестовано роботу інших кнопок та повзунків, до яких прив’язані функціональні команди. Кожна функція виконується належним чином.

4.3 Тестування роботи еквалайзера

Еквалайзер - основний метод обробки звуку, що дозволяє налаштовувати частоти аудіосигналу згідно з вашими уподобаннями або середовищем прослуховування. Використовуючи еквалайзер, можна виділяти або приглушувати певні звуки, щоб досягти гармонійного звучання. Регулюючи параметри еквалайзера, забезпечується чіткість і якість звуку, а також баланс

міксу. Еквалайзер дозволяє регулювати амплітуду і частотну характеристику аудіосигналу, забезпечуючи можливість підвищення або зниження інтенсивності певних частотних складових звуку, таких як баси, тарілки або вокал [19].

Стандартний вигляд еквалайзера показано на рисунку 4.13.

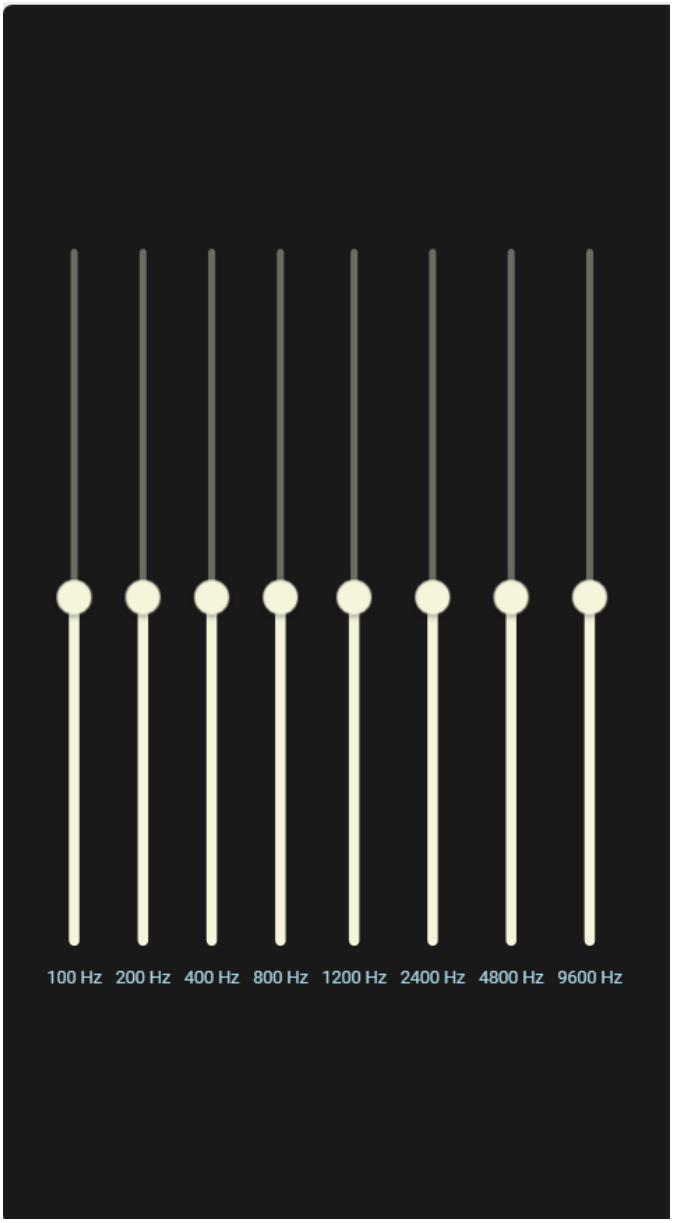

Рисунок 4.13 – Стандартний вигляд еквалайзера

Оберемо в списку відтворення музичний файл та протестуємо роботу еквалайзера частот. Результат налаштувань зображено на рисунку 4.14.

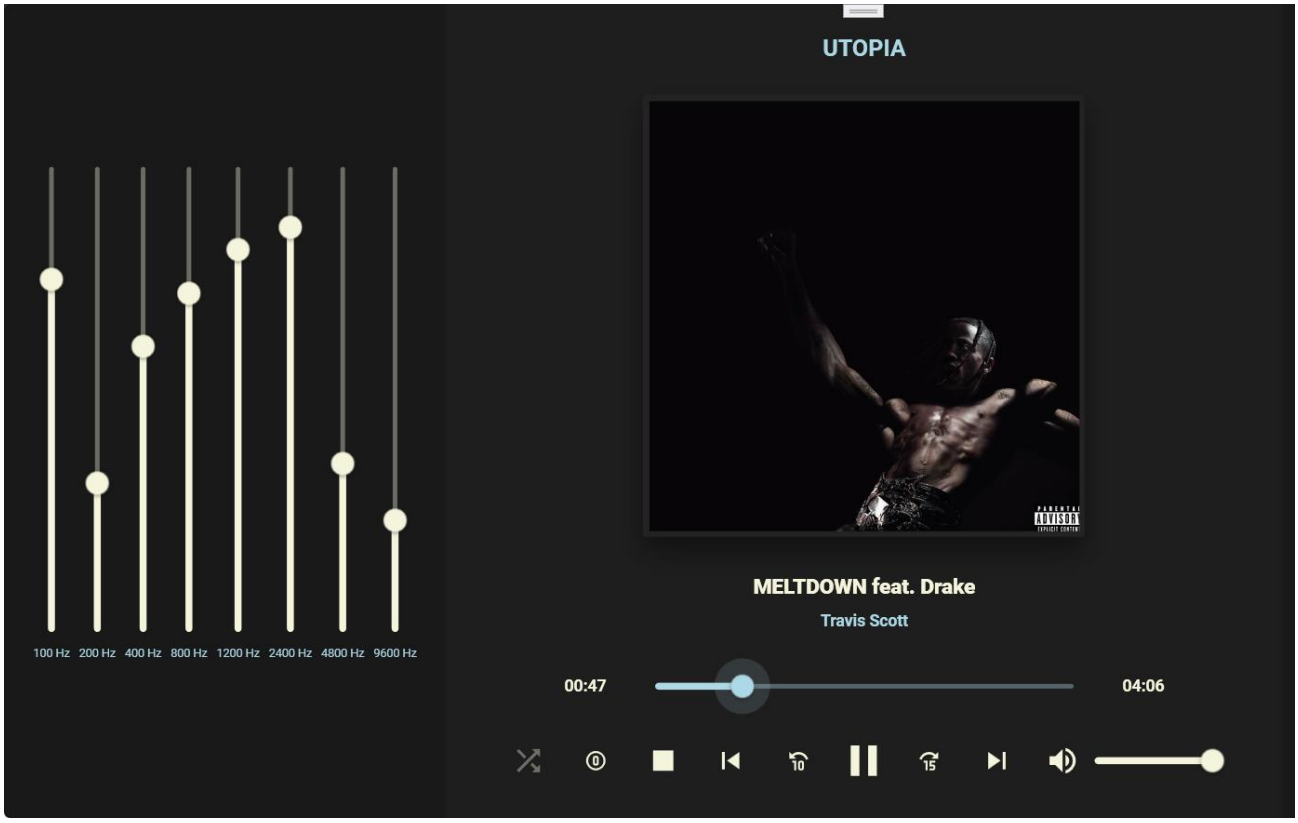

Рисунок 4.14 – Відтворення файлу зі зміненими частотами в еквалайзері

При прослуховуванні можна помітити, що звук змінився порівняно зі стандартним еквалайзером, відповідно до налаштувань, з чого можна зробити висновок, що еквалайзер працює коректно.

На даному етапі тестування програмного модулю можна вважати завершеним.

4.4 Порівняння функціональних можливостей з програмою-аналогом

У таблиці 4.1 порівняємо функціональні можливості розробленого програмного модулю для відтворення музичних файлів з його програмою-аналогом.

Таблиця 4.1 – Порівняння функцій розробленого програмного модулю з його програмою-аналогом

Функції	Аналог (Dopamine)	Розроблений модуль
Відтворення форматів WAV, MP3, OGG VORBIS, FLAC, WMA та M4A/AAC	+	+
Безпосереднє відтворення файлів з папок	+	+
Безперервне відтворення аудіофайлів	+	+
Відображення даних аудіофайлів (назва, обкладинка, автор, тощо)	+	+
Робота над файлами (зміна таймлайну, перемішування списку, повтор, пауза)	+	+
Індивідуальне налаштування звуку (еквалайзер)	-	+
Можливість створення унікальних плейлистів користувача	-	+

Із таблиці видно, що розроблений програмний модуль має перевагу над програмою-аналогом в наявності таких функціональних можливостях, як індивідуальне налаштування звуку (еквалайзер) та можливість створення унікальних плейлистів користувача.

Таким чином, можна зробити висновок, що розроблений програмний модуль має більшу кількість функцій порівняно з аналогом. Тобто основна мета

роботи досягнута – функціональні можливості програм для відтворення музичних файлів розширено.

4.5 Висновок до розділу 4

У цьому розділі було протестовано розроблений користувацький інтерфейс та функціональні команди програмного модулю для відтворення музичних файлів. Протестовано роботу кнопок меню, як наприклад додавання файлу чи створення плейлиста, діалогових вікон та списку відтворення в частині вікна меню. Проведено тестування вікна програвача, його функцій, завантаження метаданих з даних файлу, роботу кнопок та повзунків, як наприклад регулювання звуку чи періоду часу на таймлайні.

Також протестовано роботу еквалайзеру, для індивідуального налаштування звуку. Всі вищеназвані функції працюють коректно. Було порівняно функціональні можливості розробленого програмного модулю з програмою-аналогом, в ході порівняння було доведено збільшення функціональних можливостей з 5 до 7.

Отже, цикл розробки програмного модулю можна вважати завершеним.

ВИСНОВКИ

В результаті виконання бакалаврської дипломної роботи було розроблено функціональну частину та користувацький інтерфейс програмного модулю для відтворення музичних файлів.

Було здійснено глибокий огляд та аналіз різноманітних програмних рішень для відтворення аудіофайлів на платформі ОС Windows. Метою цього аналізу було виявлення унікальних особливостей та обмежень кожного із рішень. Переважно, важливими вимогами є забезпечення можливості відтворення аудіофайлів та надання розширених функцій, таких як підтримка різних форматів та управління списками відтворення, наявність налаштування звуку індивідуально.

На основі цього аналізу було визначено оптимальний вибір для конкретних потреб користувача та сформульовано вимоги до подальшого вдосконалення програмного рішення. Після цього було обґрунтовано вибір платформи та описано використані патерни й технології. WPF була обрана за можливість підтримки комплексної розробки клієнтських додатків, а патерн MVVM дозволяє ефективно відокремити бізнес-логіку від візуальної частини додатка.

Використання підходу Material Design для створення користувацького інтерфейсу дозволяє забезпечити не лише естетично привабливий дизайн, а й зручну та інтуїтивно зрозумілу взаємодію з програмою. Тестування розробленого інтерфейсу та функціональності дало позитивні результати, підтверджуючи коректну програмного модулю для відтворення музичних файлів.

Мета бакалаврської дипломної роботи була досягнута, було розширено функціональні можливості розробленого програмного модулю для відтворення музичних файлів з 5 до 7, в порівнянні з програмою-аналогом, та інтерфейс користувача відповідає вимогам, всі поставлені завдання були повністю виконані, пояснювальна записка виконана відповідно до методичних рекомендацій, які визначені для написання бакалаврської дипломної роботи [20].

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. foobar2000 [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Foobar2000
2. foobar2000 [Електронний ресурс]. Режим доступу: https://www.foobar2000.org
3. What is Windows Media Player (WMP)? [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/what-is-windows-media-player/
4. Digimezzo software [Електронний ресурс]. Режим доступу: https://digimezzo.github.io/site/software
5. Get started (WPF) [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/getting-started/?view=netframeworkdesktop-4.8
6. Overview of Windows Presentation Foundation (WPF) Architecture [Електронний ресурс]. Режим доступу: https://www.c-sharpcorner.com/UploadFile/819f33/overview-of-windows-presentation-foundation-wpf-architectu/
7. WPF Architecture [Електронний ресурс]. Режим доступу: https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2022
8. What is .NET Framework? [Електронний ресурс]. Режим доступу: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework
9. Model-View-ViewModel (MVVM). [Електронний ресурс]. Режим доступу: https://www.techtarget.com/whatis/definition/Model-View-ViewModel
10. Model-View-ViewModel (MVVM) [Електронний ресурс]. Режим доступу: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm
11. Introducing NAudio - .NET Audio Toolkit. [Електронний ресурс]. Режим доступу: https://markheath.net/post/introducing-naudio-net-audio-toolkit
12. TagLib. [Електронний ресурс]. Режим доступу: https://taglib.org/
13. Introduction to Java [Електронний ресурс]. Режим доступу: https://www.geeksforgeeks.org/introduction-to-java/

14. What is Java?. [Електронний ресурс]. Режим доступу: https://aws.amazon.com/what-is/java/
15. What is C# Programming? A Beginner's Guide. [Електронний ресурс]. Режим доступу: https://www.pluralsight.com/blog/software-development/everything-you-need-to-know-about-c-
16. C# Programming: What It Is, How It's Used + How to Learn It [Електронний ресурс]. Режим доступу: https://www.coursera.org/articles/c-sharp
17. What is Python? Executive Summary. [Електронний ресурс]. Режим доступу: https://www.python.org/doc/essays/blurb/
18. What Is Python Used For? A Beginner’s Guide. [Електронний ресурс]. Режим доступу: https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python
19. Fine-Tuning Frequencies: What is Equalization in Audio? [Електронний ресурс]. Режим доступу: https://www.practical-music-production.com/what-is-equalization-in-audio/
20. Методичні вказівки до виконання бакалаврської дипломної роботи для студентів денної та заочної форм навчання спеціальності 122 - «Комп`ютерні науки» [Електронний ресурс]. Режим доступу: https://iq.vntu.edu.ua/method/

ДОДАТКИ

ДОДАТОК А (ОБОВ'ЯЗКОВИЙ). ПРОТОКОЛ ПЕРЕВІРКИ ДИПЛОМНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Програмний модуль для відтворення музичних файлів

Тип роботи: бакалаврська дипломна робота
(БДР, МКР)

Підрозділ кафедра комп'ютерних наук, ФІТА
(кафедра, факультет)

Показники звіту подібності Unicheck

Оригінальність 99,1% Схожість 0,9%

Аналіз звіту подібності (відмітити потрібне):

- ✓ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.
- □ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її виконання автором. Роботу направити на розгляд експертної комісії кафедри.
- □ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку Озеранський В.С.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck щодо роботи.

Автор роботи Морозов Д.П.

Керівник роботи Паночишин Ю.М.

ДОДАТОК Б (ОБОВ'ЯЗКОВИЙ). ФРАГМЕНТ ЛІСТИНГУ КОДУ

```
using Common;
using Microsoft.Win32;
using Microsoft.WindowsAPICodePack.Dialogs;
using Microsoft.WindowsAPICodePack.Shell;
using Microsoft.WindowsAPICodePack.Shell.PropertySystem;
using MusPlayerApp.Annotations;
using MusPlayerApp.Helpers;
using MusPlayerApp.Models;
using MusPlayerApp.Models.Enum;
using MusPlayerApp.Services;
using NAudio.Extras;
using NAudio.Wave;
using Newtonsoft.Json;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Threading;
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;

namespace MusPlayerApp.ViewModels
{
   public partial class MainWindowViewModel : INotifyPropertyChanged
   {
      public event PropertyChangedEventHandler PropertyChanged;
      private PlaybackState _playbackState;
      private ObservableCollection<Composition> _playlist;
      private Composition _currentlyPlayingComposition;
      private Composition _currentlySelectedComposition;
      private MediaComponents _mediaComponents;
      private bool playStop;
      private double _currentCompositionLength;
      private double _currentCompositionPosition;
      private float _volumeCurrent;
      private TimeSpan _duration;
      private TimeSpan _currentDuration;
      private EqualizerBand[] _bands;
      public ImageSource _actionImage;
      private string _actionArtist;
      private string _actionAlbum;
      private string _actionTitle;
      private RepeatMode _repeatMode;

      public MainWindowViewModel()
```

```
    {
        Application.Current.MainWindow.Closing += MainWindow_Closing;
        LoadCommands();
        Playlist = new ObservableCollection<Composition>();
        _playbackState = PlaybackState.Stopped;
        VolumeCurrent = 1;
        var timer = new System.Timers.Timer
        {
            Interval = 500
        };
        timer.Elapsed += Timer_Elapsed;
        timer.Start();
        PlayStop = false;
        RepeatMode = RepeatMode.Off;

        _bands = new EqualizerBand[]
        {
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 100, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 200, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 400, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 800, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 1200, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 2400, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 4800, Gain = 0},
            new EqualizerBand {Bandwidth = 0.8f, Frequency = 9600, Gain = 0},
        };
    }

    private void GetInfo()
    {
        ActionArtist =
MetadataReaderHelper.Artist_Changed(_currentlySelectedComposition?.Filepath);
        ActionAlbum =
MetadataReaderHelper.Album_Changed(_currentlySelectedComposition?.Filepath);
        ActionTitle =
MetadataReaderHelper.Titles_Changed(_currentlySelectedComposition?.Filepath);
    }

    private void GetImage()
    {
        ActionImage =
MetadataReaderHelper.Image_Changed(_currentlySelectedComposition?.Filepath);
        OnPropertyChanged(nameof(ActionImage));
    }

    private TimeSpan GetAudioDuration(string filePath)
    {
        using var shell = ShellObject.FromParsingName(filePath);
        IShellProperty prop = shell.Properties.System.Media.Duration;
        var t = (ulong)prop.ValueAsObject;
        return TimeSpan.FromTicks((long)t);
    }
```

```
        private void UpdateSeekBar()
        {
            if (_playbackState == PlaybackState.Playing)
            {
                CurrentCompositionPosition = _mediaComponents.GetPositionInSeconds();
                CurrentDuration = _mediaComponents.GetSeconds();
            }
        }

        private void Timer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
        {
            UpdateSeekBar();
        }

        private void MainWindow_Closing(object sender, CancelEventArgs e)
        {
            _mediaComponents?.Dispose();
        }

        private void MediaComponents_PlaybackStopped()
        {
            _playbackState = PlaybackState.Stopped;
            CommandManager.InvalidateRequerySuggested();
            CurrentCompositionPosition = 0;

            if (_mediaComponents.PlaybackStopType == PlaybackStopTypes.PlaybackStoppedReachingEndOfFile)
            {
                if (RepeatMode == RepeatMode.RepeatOne)
                {
                    StartPlayback(null);
                }
                else if (RepeatMode == RepeatMode.Off)
                {
                    CurrentlySelectedComposition = Playlist.NextItem(CurrentlyPlayingComposition);
                    StartPlayback(null);
                }
            }
            else if (_mediaComponents.PlaybackStopType == PlaybackStopTypes.PlaybackWentToStartByUser)
            {
                CurrentlySelectedComposition = Playlist.PreviousItem(CurrentlyPlayingComposition);
                StartPlayback(null);
            }
            else if (_mediaComponents.PlaybackStopType == PlaybackStopTypes.PlaybackStoppedByUser)
            {
                if (CurrentlySelectedComposition != CurrentlyPlayingComposition)
                {
                    StartPlayback(null);
                }
```

```
        }
    }

    private void MediaComponents_PlaybackResumed()
    {
        _playbackState = PlaybackState.Playing;
    }

    private void MediaComponents_PlaybackPaused()
    {
        _playbackState = PlaybackState.Paused;
    }

    private void PlayerExit(object p)
    {
        _mediaComponents?.Dispose();

        Application.Current.Shutdown();
    }

    private bool CanPlayerExit(object p)
    {
        return true;
    }

    private void HideWindow(object p)
    {
        Application.Current.MainWindow.Hide();
    }

    private bool CanHideWindow(object p)
    {
        return true;
    }

    private void PlaylistAddFile(object p)
    {
        var ofd = new OpenFileDialog
        {
            Filter = "Audio files (*.wav, *.mp3, *.wma, *.ogg, *.flac, *.m4a) | *.wav; *.mp3; *.wma;
*.ogg; *.flac, *.m4a"
        };
        var result = ofd.ShowDialog();
        if (result == true)
        {
            var composition = new Composition(ofd.FileName,
                MetadataReaderHelper.Titles_Changed(ofd.FileName),
                MetadataReaderHelper.Image_Changed(ofd.FileName));
            Playlist.Add(composition);
        }
    }
```

```
private bool CanPlaylistAddFile(object p)
{
    if (_playbackState == PlaybackState.Stopped)
    {
        return true;
    }
    return false;
}

private void PlaylistAddFolder(object p)
{
    var cofd = new CommonOpenFileDialog
    {
        IsFolderPicker = true
    };
    var result = cofd.ShowDialog();
    if (result == CommonFileDialogResult.Ok)
    {
        var folderName = cofd.FileName;
        var audioFiles = Directory
            .EnumerateFiles(folderName, "*.*", SearchOption.AllDirectories)
            .Where(f => f.EndsWith(".wav") ||
            f.EndsWith(".mp3") || f.EndsWith(".wma") ||
            f.EndsWith(".ogg") || f.EndsWith(".flac") ||
            f.EndsWith(".m4a"));
        foreach (var audioFile in audioFiles)
        {
            var shortenedTitleName = MetadataReaderHelper.Titles_Changed(audioFile);
            var composition = new Composition(audioFile,
                shortenedTitleName,
                MetadataReaderHelper.Image_Changed(audioFile));
            Playlist.Add(composition);
        }
        Playlist = new ObservableCollection<Composition>
            (Playlist.OrderBy(z => z.ShortenedTitleName).ToList());
    }
}

private bool CanPlaylistAddFolder(object p)
{
    if (_playbackState == PlaybackState.Stopped)
    {
        return true;
    }
    return false;
}

private void PlaylistSave(object p)
{
    var sfd = new SaveFileDialog
    {
        CreatePrompt = false,
```

```
            OverwritePrompt = true,
            Filter = "PLAYLIST files (*.playlist) | *.playlist"
        };
        if (sfd.ShowDialog() == true)
        {
            new PlaylistService().Save(Playlist, sfd.FileName);
        }
    }

    private bool CanPlaylistSave(object p)
    {
        return true;
    }

    private void PlaylistLoad(object p)
    {
        var ofd = new OpenFileDialog
        {
            Filter = "PLAYLIST files (*.playlist) | *.playlist"
        };
        if (ofd.ShowDialog() == true)
        {
            Playlist = new PlaylistService().Load(ofd.FileName)
                .ToObservableCollection();
        }
        StopPlayback(null);
    }

    private bool CanPlaylistLoad(object p)
    {
        return true;
    }

    private void RewindToStart(object p)
    {
        if (_mediaComponents != null)
        {
            _mediaComponents.PlaybackStopType                                    =
PlaybackStopTypes.PlaybackWentToStartByUser;
            _mediaComponents.SetPosition(_mediaComponents.GetLengthInSeconds());
        }
    }
    private bool CanRewindToStart(object p)
    {
        if (_playbackState == PlaybackState.Playing)
        {
            return true;
        }
        return false;
    }

    private void StartPlayback(object p)
```

```
    {
        if (CurrentlySelectedComposition != null)
        {
            if (CurrentlyPlayingComposition != CurrentlySelectedComposition)
            {
                if (CurrentlyPlayingComposition == null || _playbackState == PlaybackState.Stopped)
                {
                    _mediaComponents = new MediaComponents(CurrentlySelectedComposition.Filepath, VolumeCurrent, _bands)
                    {
                        PlaybackStopType = PlaybackStopTypes.PlaybackStoppedReachingEndOfFile
                    };
                    _mediaComponents.PlaybackPaused += MediaComponents_PlaybackPaused;
                    _mediaComponents.PlaybackResumed += MediaComponents_PlaybackResumed;
                    _mediaComponents.PlaybackStopped += MediaComponents_PlaybackStopped;
                    CurrentCompositionLength = _mediaComponents.GetLengthInSeconds();
                    CurrentlyPlayingComposition = CurrentlySelectedComposition;
                    Duration = GetAudioDuration(CurrentlySelectedComposition.Filepath);
                    CurrentDuration = _mediaComponents.GetSeconds();

                }
                else
                {
                    if (_mediaComponents != null)
                    {
                        StopPlayback(null);
                        Thread.Sleep(100);
                    }
                }
            }
            if (_playbackState == PlaybackState.Stopped)
            {
                _mediaComponents = new MediaComponents(CurrentlySelectedComposition.Filepath, VolumeCurrent, _bands)
                {
                    PlaybackStopType = PlaybackStopTypes.PlaybackStoppedReachingEndOfFile
                };
                _mediaComponents.PlaybackPaused += MediaComponents_PlaybackPaused;
                _mediaComponents.PlaybackResumed += MediaComponents_PlaybackResumed;
                _mediaComponents.PlaybackStopped += MediaComponents_PlaybackStopped;
                CurrentCompositionLength = _mediaComponents.GetLengthInSeconds();
                CurrentlyPlayingComposition = CurrentlySelectedComposition;
                Duration = GetAudioDuration(CurrentlySelectedComposition.Filepath);
                CurrentDuration = _mediaComponents.GetSeconds();

            }
            PropertyChanged += OnPropertyChanged;
            _mediaComponents.TogglePlayPause(VolumeCurrent);
        }
    }
```

```
        private void OnPropertyChanged(object sender, PropertyChangedEventArgs
propertyChangedEventArgs)
        {
            _mediaComponents?.Equalizer?.Update();
        }

        private bool CanStartPlayback(object p)
        {
            if (CurrentlySelectedComposition != null)
            {
                return true;
            }
            return false;
        }

        private void StopPlayback(object p)
        {
            if (_mediaComponents != null)
            {
                _mediaComponents.PlaybackStopType = PlaybackStopTypes.PlaybackStoppedByUser;
                _mediaComponents.Stop();
            }
            PlayStop = false;
            CurrentDuration = TimeSpan.Zero;
        }

        private bool CanStopPlayback(object p)
        {
            return _playbackState == PlaybackState.Playing ||
                   _playbackState == PlaybackState.Paused;
        }

        private void ForwardToEnd(object p)
        {
            if (_mediaComponents != null)
            {
                _mediaComponents.PlaybackStopType                                    =
PlaybackStopTypes.PlaybackStoppedReachingEndOfFile;
                _mediaComponents.SetPosition(_mediaComponents.GetLengthInSeconds());
            }
        }
        private bool CanForwardToEnd(object p)
        {
            return _playbackState == PlaybackState.Playing;
        }

        private void Shuffle(object p)
        {
            {
                Playlist = Playlist.Shuffle();
                PlayStop = false;
            }
```

```
}
private bool CanShuffle(object p)
{
   return _playbackState != PlaybackState.Stopped;
}

private void CompositionControlMouseDown(object p)
{
   _mediaComponents?.Pause();
}

private void CompositionControlMouseUp(object p)
{
   if (_mediaComponents != null)
   {
      _mediaComponents.SetPosition(CurrentCompositionPosition);
      _mediaComponents.Play(PlaybackState.Paused, VolumeCurrent);
   }
}

private bool CanCompositionControlMouseDown(object p)
{
   return _playbackState == PlaybackState.Playing;
}

private bool CanCompositionControlMouseUp(object p)
{
   return _playbackState == PlaybackState.Paused;
}

private void VolumeControlValueChanged(object p)
{
   _mediaComponents?.SetVolume(VolumeCurrent);
}

private bool CanVolumeControlValueChanged(object p)
{
   return true;
}

public bool PlayStop
{
   get { return playStop; }
   set { playStop = value; OnPropertyChanged("playStop"); }
}

private float _savedVolumeLevel;
private void VolumeMute(object p)
{
   if (_mediaComponents != null)
   {
      if (_mediaComponents.GetVolume() == 0)
```

```
        {
            _mediaComponents.SetVolume(_savedVolumeLevel);
        }
        else
        {
            _savedVolumeLevel = _mediaComponents.GetVolume();
            _mediaComponents.SetVolume(0);
        }
    }
}

private bool CanVolumeMute(object p)
{
    return true;
}

private void Rewind10Sec(object p)
{
    _mediaComponents?.SetPosition(_currentCompositionPosition - 10);
}
private bool CanRewind10Sec(object p)
{
    return _playbackState == PlaybackState.Playing;
}
private void FastForward15Sec(object p)
{
    _mediaComponents?.SetPosition(_currentCompositionPosition + 15);
}
private bool CanFastForward15Sec(object p)
{
    return _playbackState == PlaybackState.Playing;
}

private bool CanToggleRepeatMode(object parameter)
{
    return true;
}

private void ToggleRepeatMode(object parameter)
{
    switch (RepeatMode)
    {
        case RepeatMode.Off:
            RepeatMode = RepeatMode.RepeatOne;
            break;
        case RepeatMode.RepeatOne:
            RepeatMode = RepeatMode.Off;
            break;
    }
}

private void SaveEqualizerPreset(object parameter)
```

```
    {
        SaveFileDialog saveFileDialog = new SaveFileDialog
        {
            Filter = "JSON files (*.json)|*.json",
            FilterIndex = 2,
            RestoreDirectory = true
        };

        if (saveFileDialog.ShowDialog() == true)
        {
            SaveEqualizerPreset(saveFileDialog.FileName);
        }
    }

    private void LoadEqualizerPreset(object parameter)
    {
        OpenFileDialog openFileDialog = new OpenFileDialog
        {
            Filter = "JSON files (*.json)|*.json",
            FilterIndex = 2,
            RestoreDirectory = true
        };

        if (openFileDialog.ShowDialog() == true)
        {
            LoadEqualizerPreset(openFileDialog.FileName);
        }
    }

    public void SaveEqualizerPreset(string fileName)
    {
        string json = JsonConvert.SerializeObject(_bands);
        File.WriteAllText(fileName, json);
    }

    public void LoadEqualizerPreset(string fileName)
    {
        if (File.Exists(fileName))
        {
            string json = File.ReadAllText(fileName);
            _bands = JsonConvert.DeserializeObject<EqualizerBand[]>(json);
            CallChanges();
        }
        else
        {
            MessageBox.Show("Файл не знайдено: " + fileName, "Помилка",
MessageBoxButton.OK, MessageBoxImage.Error);
        }
    }

    private void CallChanges()
    {
```

```
            OnPropertyChanged("Band1");
            OnPropertyChanged("Band2");
            OnPropertyChanged("Band3");
            OnPropertyChanged("Band4");
            OnPropertyChanged("Band5");
            OnPropertyChanged("Band6");
            OnPropertyChanged("Band7");
            OnPropertyChanged("Band8");
            OnPropertyChanged("Bands");
        }

        private bool CanLoadEqualizerPreset(object parameter)
        {
            return true;
        }

        private bool CanSaveEqualizerPreset(object parameter)
        {
            return true;
        }

        [NotifyPropertyChangedInvocator]
        protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

ДОДАТОК В (ОБОВ'ЯЗКОВИЙ). ГРАФІЧНА ЧАСТИНА

ПРОГРАМНИЙ МОДУЛЬ ДЛЯ ВІДТВОРЕННЯ МУЗИЧНИХ ФАЙЛІВ

Виконав: студент 4 курсу, групи КН-22мс
спеціальності 122 – Комп'ютерні науки
(шифр і назва напряму підготовки, спеціальності)
Морозов Д.П.
(прізвище та ініціали)
Керівник: к.т.н., доцент каф. КН
Паночишин Ю.М.
(прізвище та ініціали)
«07» 06 2024 р.

Вінниця ВНТУ – 2024 рік

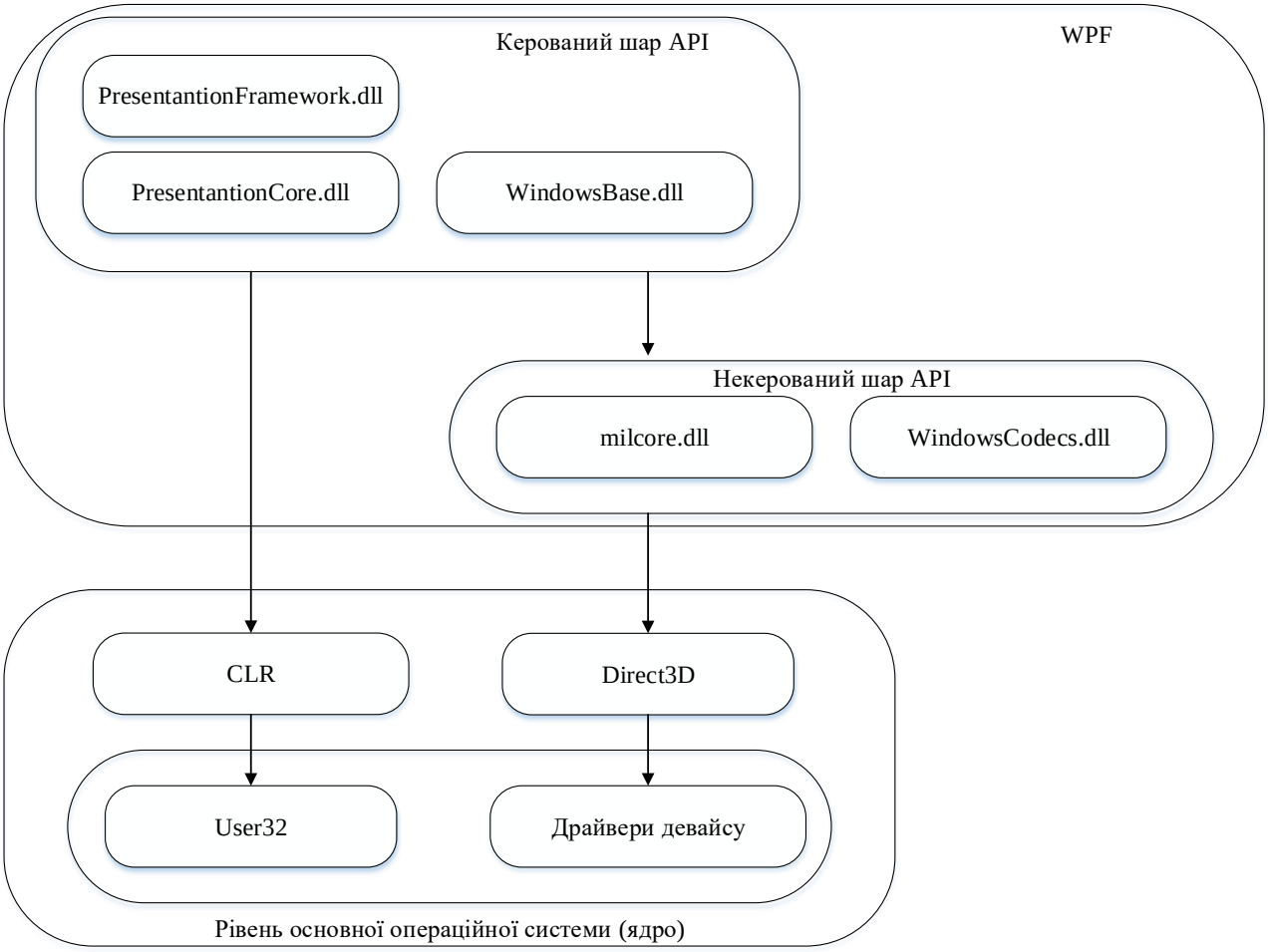

Рисунок В.1 – Схема основної архітектури WPF

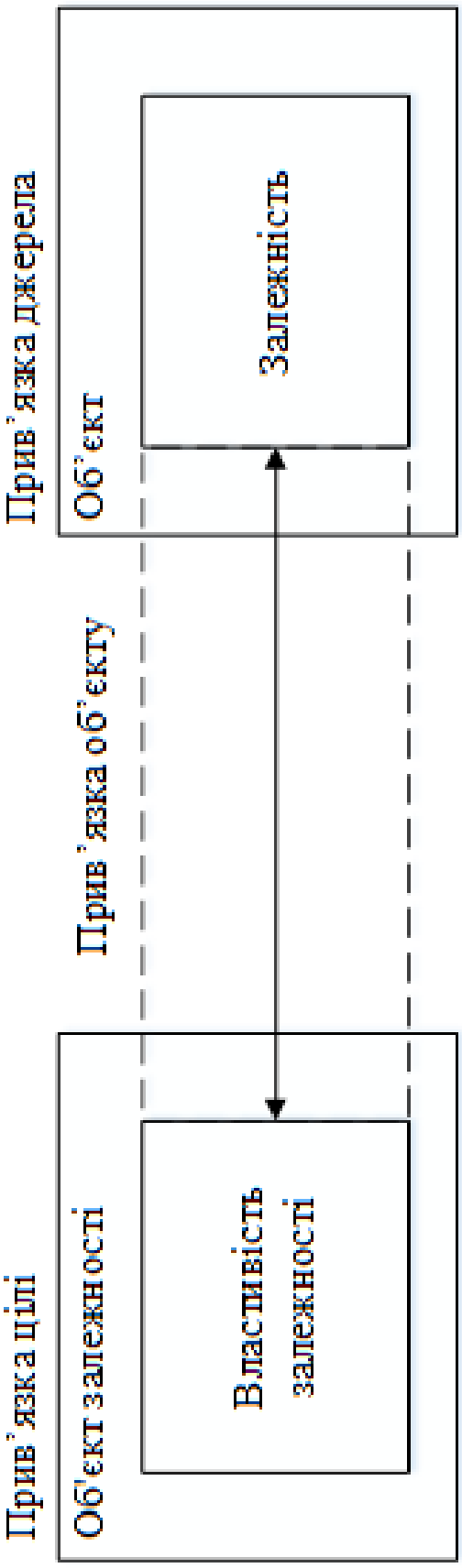

Рисунок В.2 - Механізм прив’язки даних

Рисунок В.3 - Архітектура .NET Framework

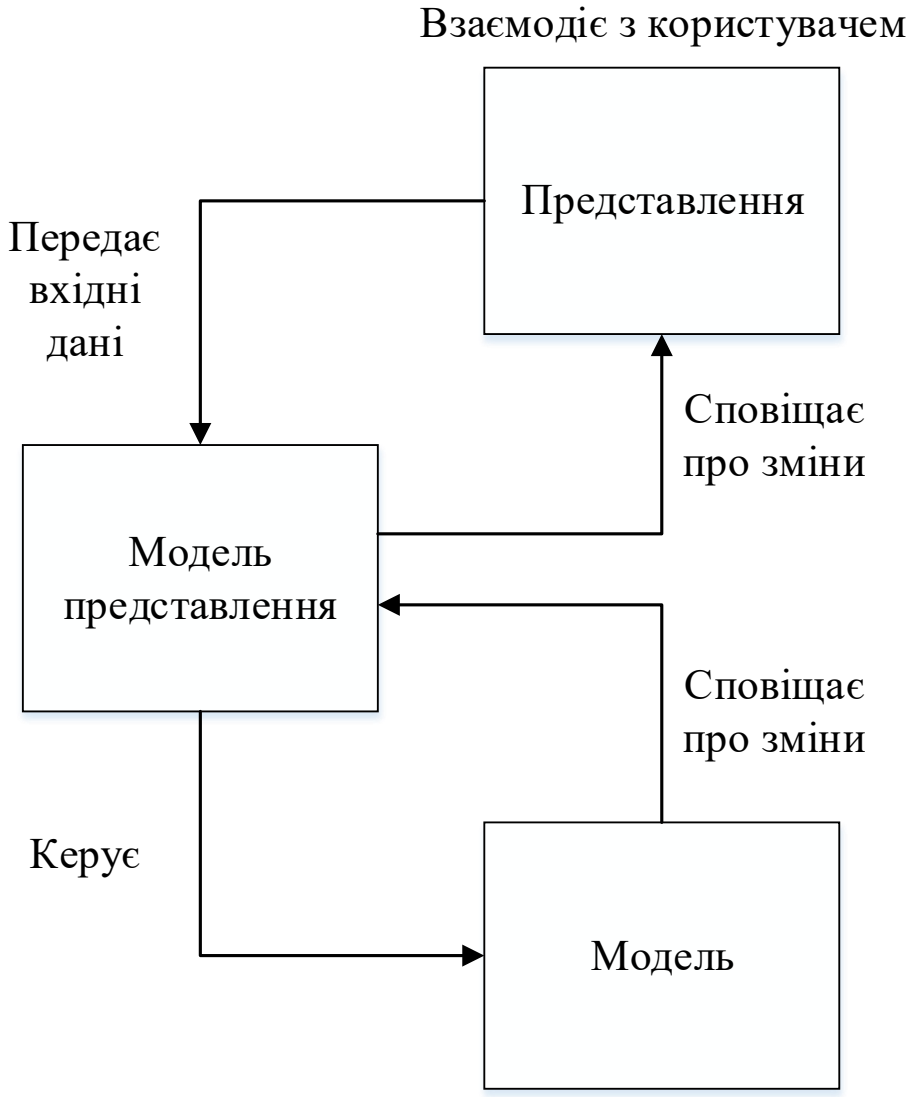

Рисунок В.4 - Схема роботи патерна MVVM

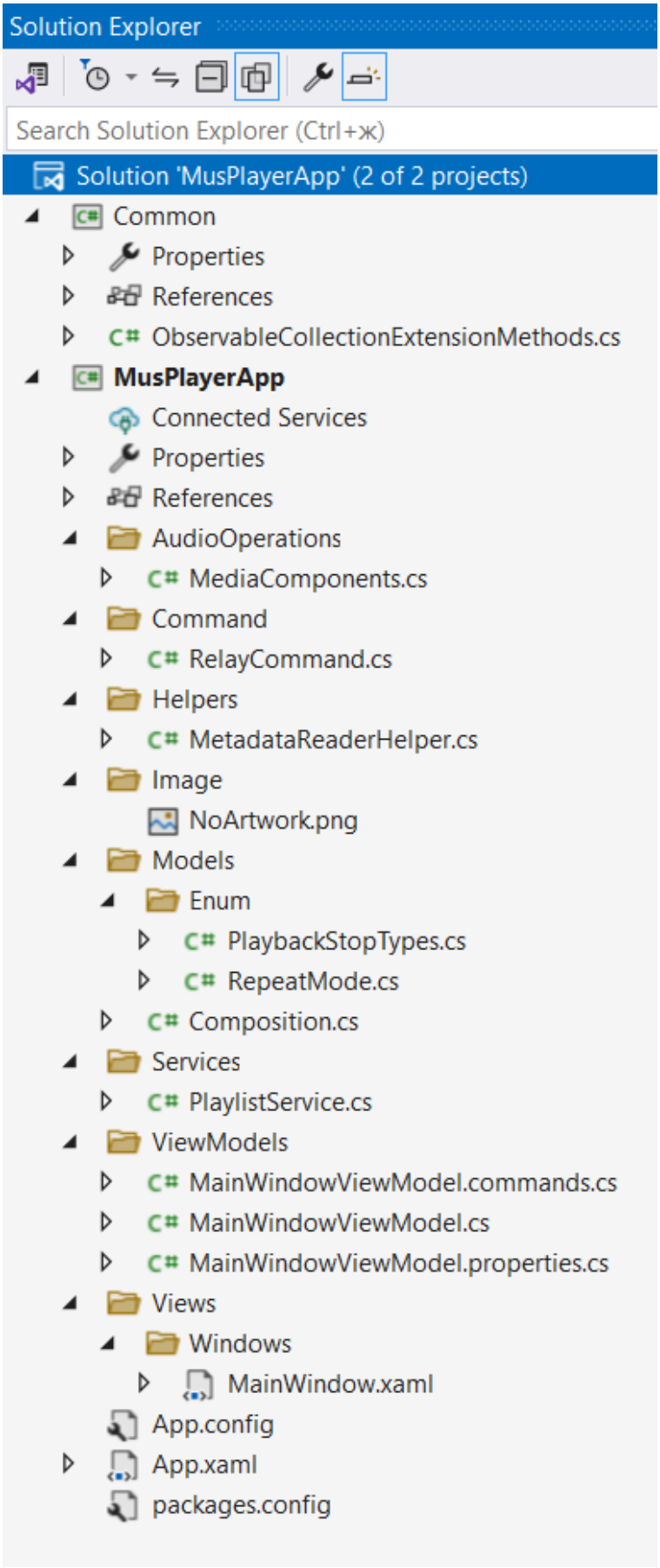

Рисунок В.5 - Загальна структура програмної реалізації модуля

ДОДАТОК Г (ДОВІДНИКОВИЙ). ІНСТРУКЦІЯ КОРИСТУВАЧА

Щоб почати роботу з програмним модулем, необхідно запустити його, шляхом подвійного кліку на ярлик програми. Відкриється основне вікно програми, вигляд якого зображено на рисунку Г.1.

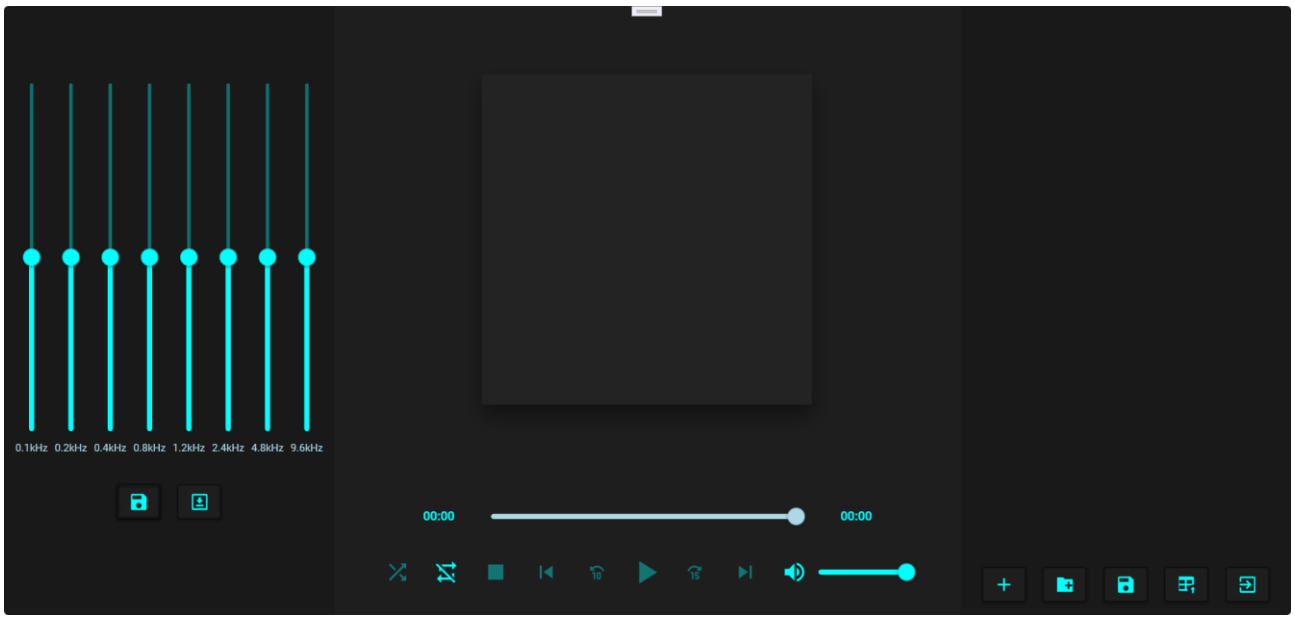

Рисунок Г.1 – Основне вікно програмного модулю

Для початку роботи з файлами, в меню справа оберіть одну з команд (рис Г.2).

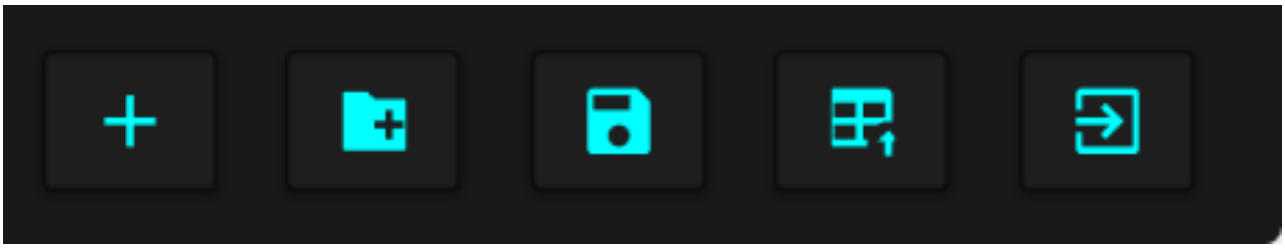

Рисунок Г.2 – Меню списку відтворення

Кнопка 1 – «Додати композицію до списку відтворення». Ця функція дозволяє обрати з файлової системи комп’ютера потрібний файл у відповідному форматі;

Кнопка 2 – «Додати папку до списку відтворення». Ця функція дозволяє обрати з файлової системи комп’ютера потрібну папку з аудіофайлами у відповідному форматі. З папки оберуться всі допустимі файли;

Кнопка 3 – «Зберегти плейлист». Ця функція потрібна для створення унікального списку відтворення користувача. Коли список відтворення буде заповнений різними музичними файлами, користувач має змогу зберегти порядок відтворення файлів в форматі .playlist;

Кнопка 4 – «Завантажити плейлист». Ця функція дозволяє обрати та завантажити плейлист з файлової системи комп’ютера;

Кнопка 5 – «Вийти». Ця функція закриває програму, що закінчує будь-яке відтворення аудіофайлу.

Оберемо кнопку 2, що відповідає за додавання папки до списку відтворення. Відкривається діалогове вікно файлової системи (рис. Г.3), в якому користувач обирає папку, яку він хоче додати. Аудіофайли з цієї папки будуть додані до списку відтворення (рис. Г.4).

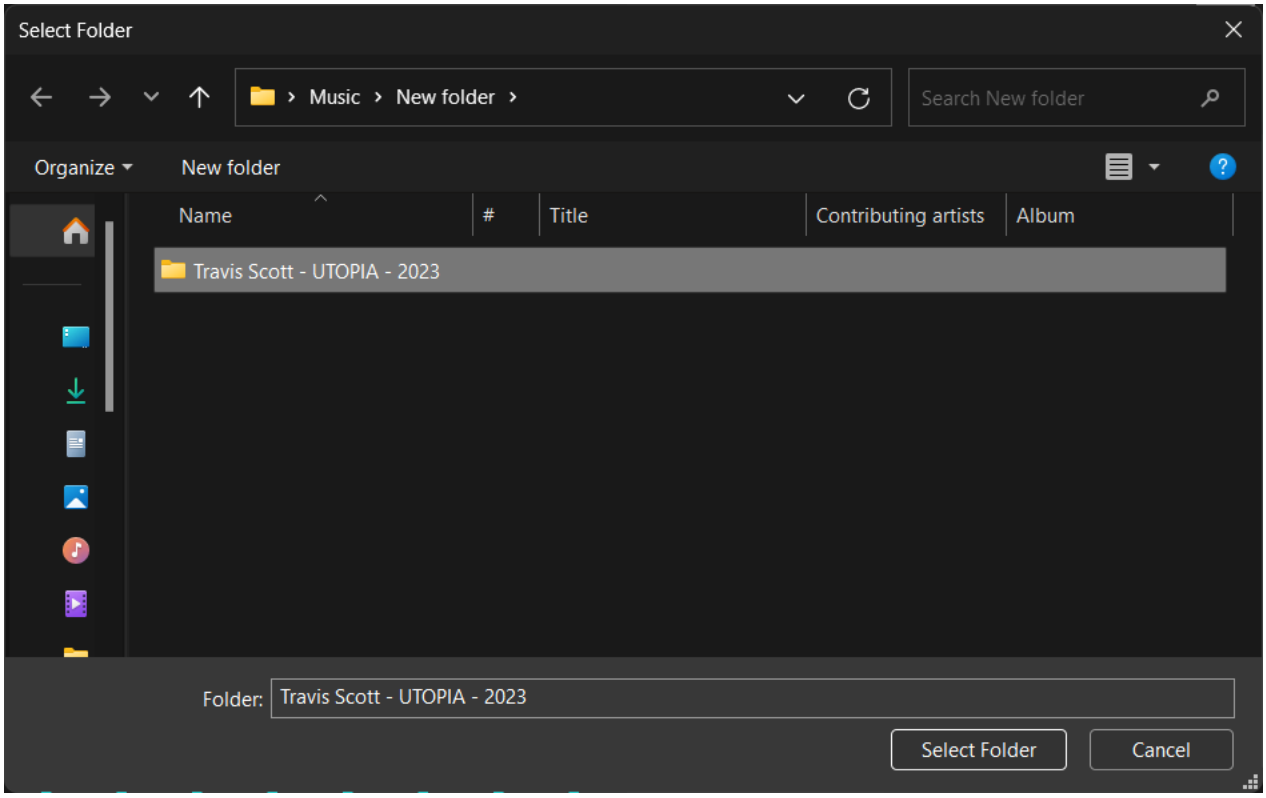

Рисунок Г.3 – Діалогове вікно файлової системи

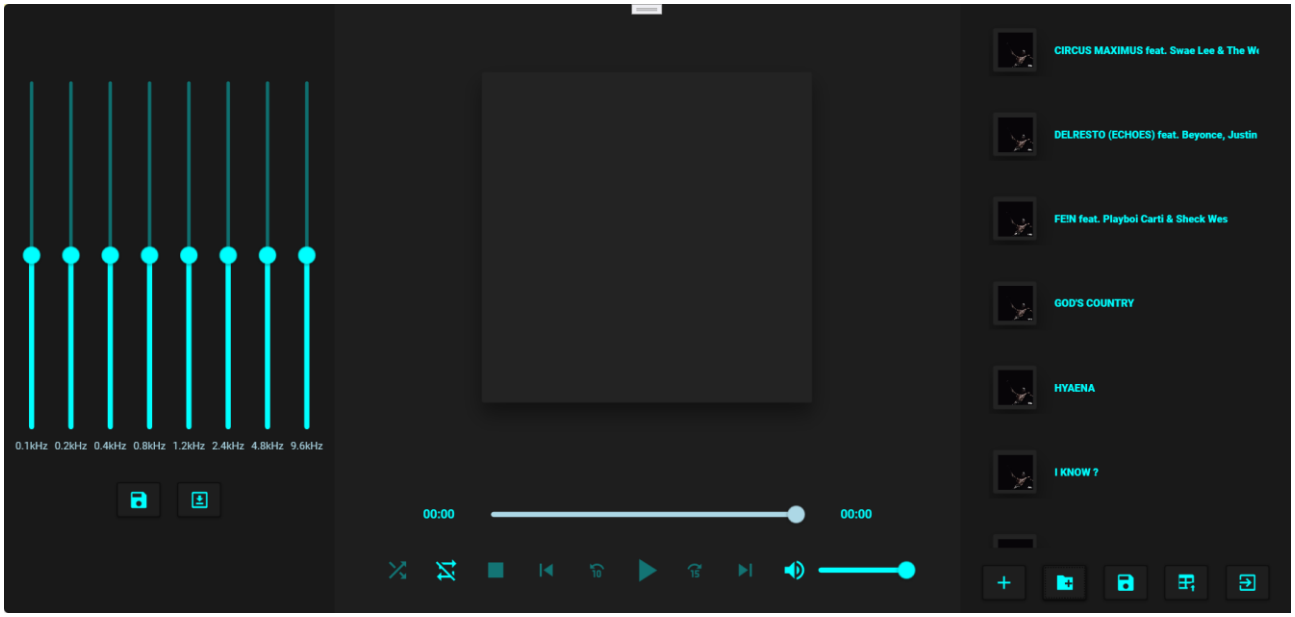

Рисунок Г.4 – Список відтворення з аудіофайлами у папці

Щоб почати відтворення файлу, потрібно обрати його зі списку відтворення, якщо перший файл, тоді вікно програвача набуде такого вигляду (рис Г.5):

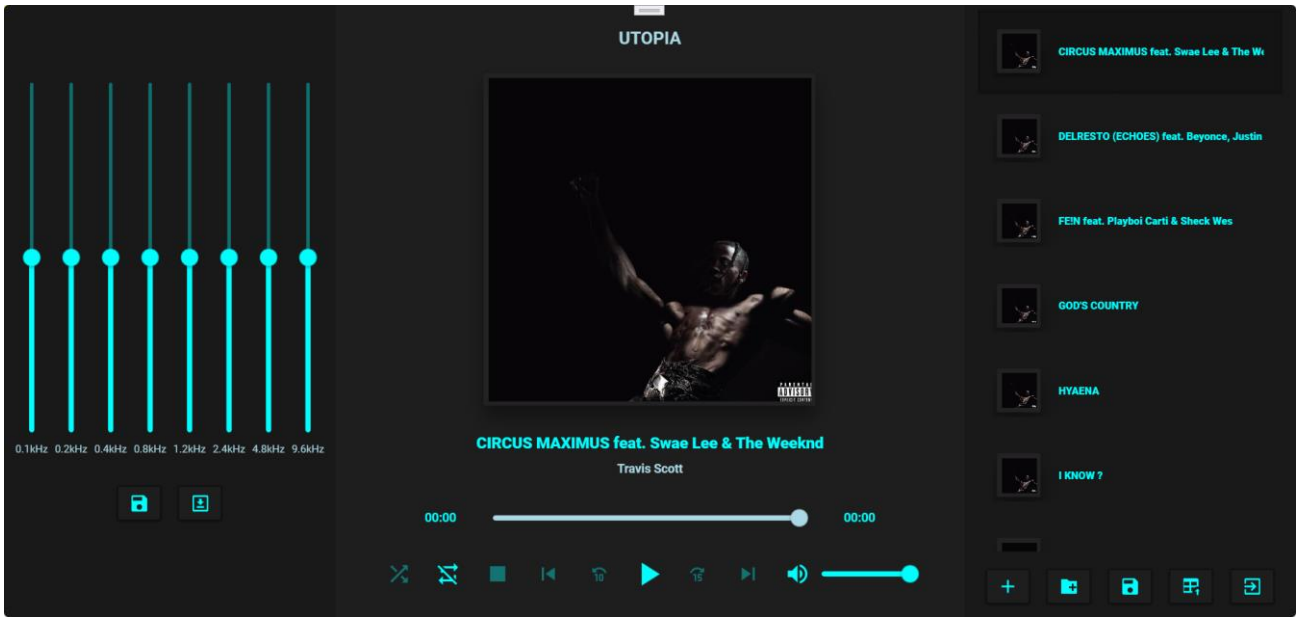

Рисунок Г.5 – Аудіофайл у вікні програвача

На вікні програвача користувач може побачити:

– Назву композиції;

– Назву альбому;

– Ім’я виконавця.

Натиснувши на виконання, аудіофайл почне відтворюватись, почне рухатись лінія часу треку, та з’явиться тривалість файлу (рис Г.6).

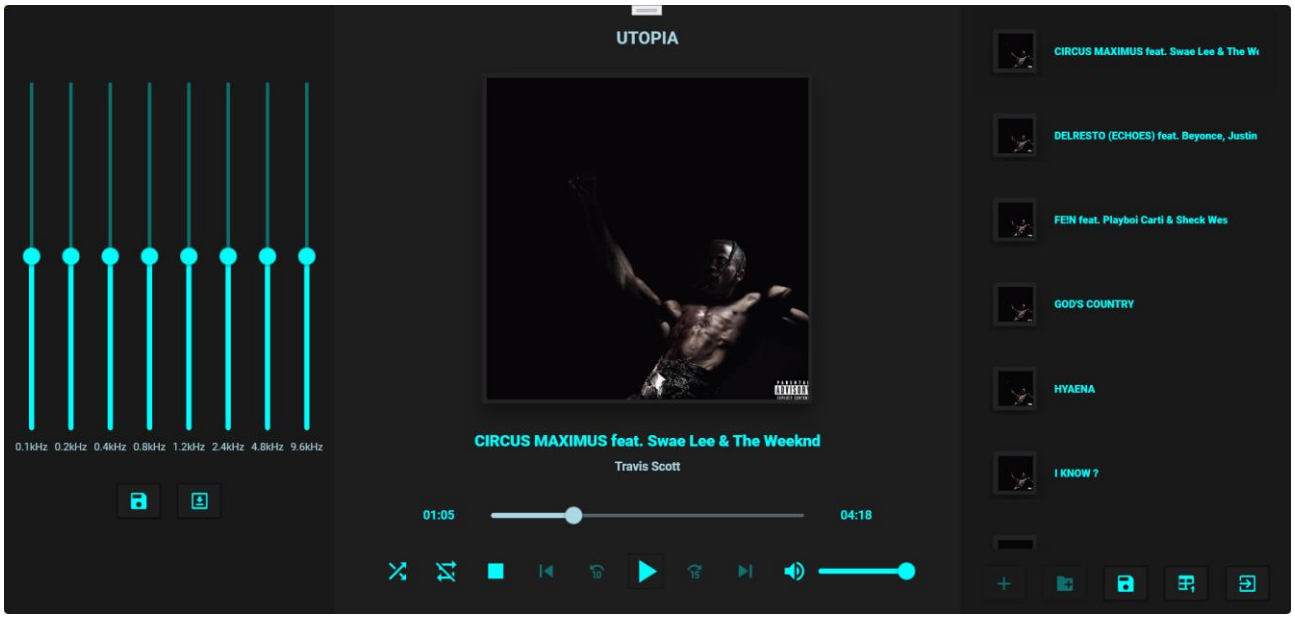

Рисунок Г.6 – Відтворення аудіофайлу у вікні програвача

Користувач може обрати шляхи взаємодії з файлами, обираючи функціональні команди на панелі програвача (рис. Г.7).

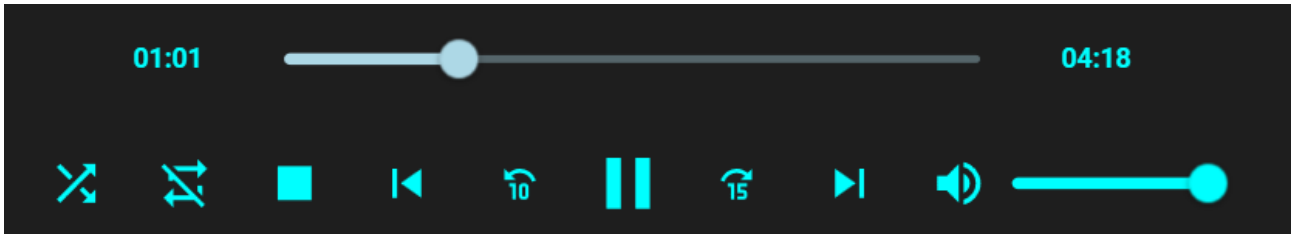

Рисунок Г.7 – Функціональні команди на панелі програвача

Команда 1 – Зміна таймлайну трека, шляхом перетягування по лінії;

Команда 2 – Перемішування списку відтворення;

Команда 3 – Повторення аудіофайлу;

Команда 4 – Повна зупинка аудіофайлу;

Команда 5 – Попередній трек;

Команда 6 – Переміщення таймлайну на 10 секунд назад;

Команда 7 – Пауза/Відтворення;

Команда 8 - Переміщення таймлайну на 15 секунд вперед;

Команда 9 - Наступний трек;

Команда 10 – Зміна гучності/Повне вимкнення гучності.

Перейдемо до меню зліва, де розташовується еквалайзер, що служить для індивідуального налаштування звуку під потреби користувача. Рухаючи повзунки, що відповідають за підсилення або послаблення певної частоти звуку, можна добитись унікального звучання, подібні зміни відображено на рисунку Г.8.

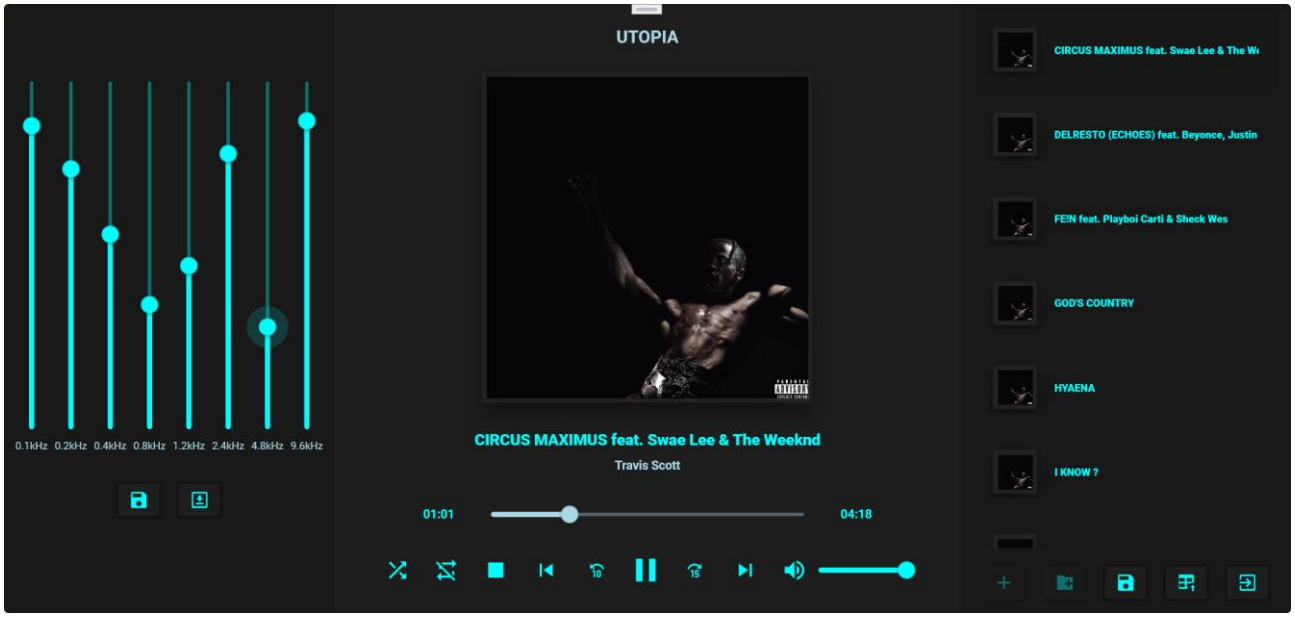

Рисунок Г.8 – Зміна частот в еквалайзері

Також, функції в меню еквалайзера (рис. Г.9) дозволяють зберегти пресет, який можна буде використовувати в подальших прослуховуваннях, покажемо цей процес на рисунку Г.10.

Рисунок Г.9 – Функціональні команди меню еквалайзера

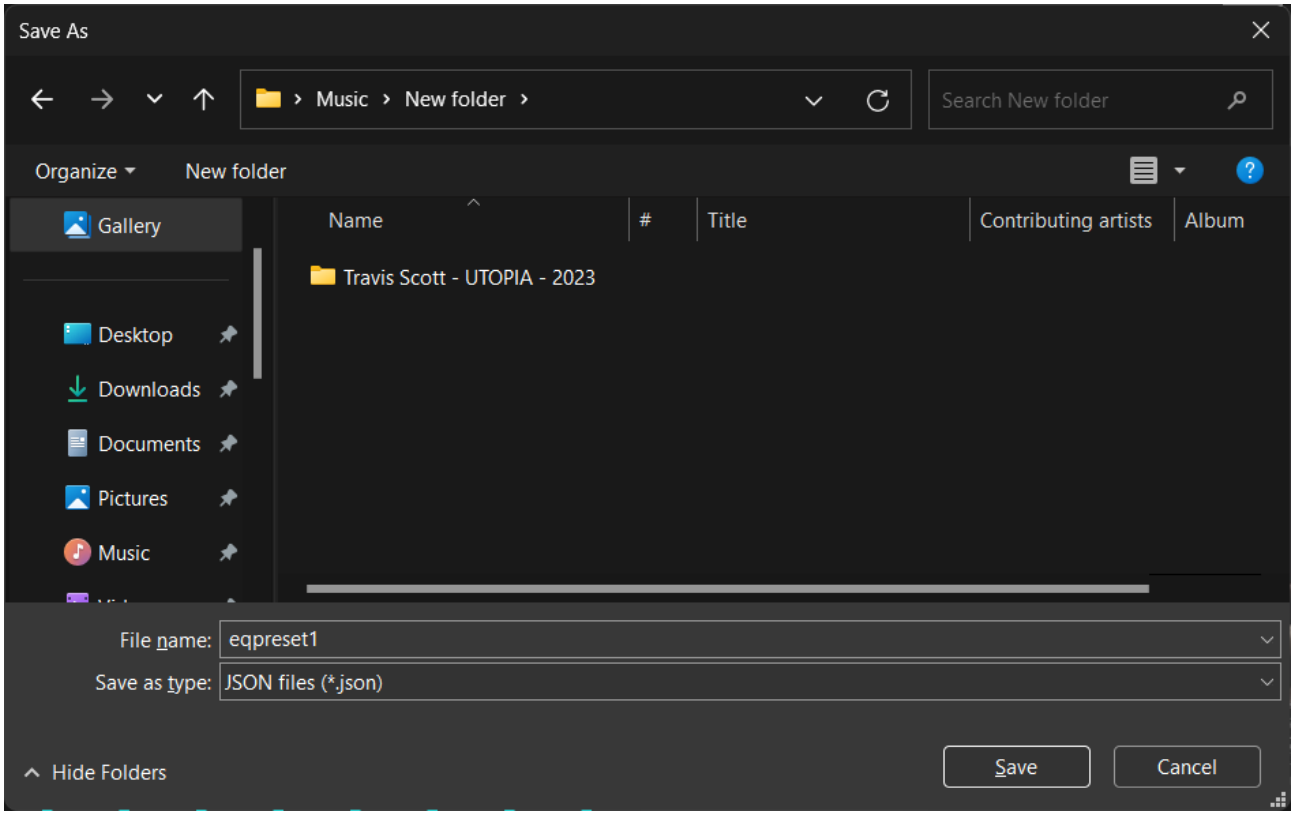

Рисунок Г.10 – Збереження пресету еквалайзера

Перша функція меню відповідає за збереження пресету, друга ж дозволяє користувачу завантажити пресет еквалайзера для того ж файлу, або будь-яких інших.