Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська кваліфікаційна робота

на тему: Розробка програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку

Виконав: студент 4 курсу
групи 2ПІ-21б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Грінін А. В.
(прізвище та ініціали)

Керівник: к.т.н., доц. каф. ПЗ Романюк О. В.
(прізвище та ініціали)

Рецензент: к.т.н., доц. каф. ЗІ Войтович О. П.
(прізвище та ініціали)

Допущено до захисту

Зав. кафедри ПЗ
д.т.н., проф. О. Н. Романюк
«09» 06 2025 р.

ВНТУ – 2025

Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., професор
_____ О.Н. Романюк
«21» березня 2025 р.

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Грініну Андрію Вікторовичу

1. Тема роботи – розробка програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку.

Керівник роботи: Романюк Оксана Володимирівна, к.т.н., доц. каф. ПЗ, затверджені наказом вищого навчального закладу від 20 березня 2025 р. № 97.

2. Строк подання студентом роботи 2 червня 2025 р.

3. Вихідні дані до роботи: тип програми – Java-застосунок; модель розробки – ітеративно-інкрементна; вхідні дані – аудіофайл; вихідні дані – PCM-потік на аудіовихід, масиви спектральних даних, значення BPM, закладки; мова програмування – Java; середовище розробки – IntelliJ IDEA, бібліотеки JavaFX, TarsosDSP; дані для аналізу звукових параметрів – спектральні характеристики аудіосигналу, значення амплітуди для лівого і правого каналів, спектрограма, хвильова форма, фільтрований аудіосигнал.

4. Зміст текстової частини: вступ; аналіз сучасного стану аудіоплеєрів та постановка задач дослідження; розробка архітектури, методу та алгоритмів програмного продукту; розробка програмних модулів аудіоплеєра з розширеними можливостями аналізу звуку; тестування програми; висновки; список використаних джерел; додатки.

5. Перелік ілюстративного матеріалу: титульний слайд; актуальність теми; мета, об'єкт та предмет дослідження; задачі дослідження; наукова новизна та практична цінність отриманих результатів; порівняльний аналіз аналогів; архітектура програми; блок-схеми алгоритмів відображення рівня звуку за частотами та генерації хвильової форми; метод та блок-схема алгоритму визначення темпу; інтерфейс програми; модулі візуалізації параметрів звуку; демонстрація роботи програми; апробація та публікація результатів

бакалаврської кваліфікаційної роботи; фінальний слайд.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Романюк О.В., к.т.н., доцент кафедри ПЗ	25.03.25	30.05.25

7. Дата видачі завдання 24 березня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз сучасного стану аудіоплеєрів та аналогічних програмних продуктів	25.03.25 – 31.03.25	Виконано
2	Розробка архітектури програмного забезпечення	01.04.25 – 07.04.25	Виконано
3	Розробка методу визначення темпу	08.04.25 – 15.04.25	Виконано
4	Розробка алгоритмів аналізу звуку	16.04.25 – 23.04.25	Виконано
5	Аналіз та вибір засобів для реалізації програмного продуктку	24.04.25 – 30.04.25	Виконано
6	Розробка програмного забезпечення	01.05.25 – 07.05.25	Виконано
7	Тестування програмного застосунку	07.05.25 – 14.05.25	Виконано
8	Оформлення матеріалів до захисту БКР	15.05.25 – 30.05.25	Виконано

Студент _____ **Грінін А.В.**
(підпис) (прізвище та ініціали)

Керівник бакалаврської кваліфікаційної роботи _____ **Романюк О.В.**
(підпис) (прізвище та ініціали)

АНТОАЦІЯ

УДК 004.42

Грінін А.В. Розробка програмного забезпечення аудіоплеєра з розширеними можливостями аналізу звуку : бакалаврська квалафікаційна робота зі спеціальності 121 Інженерія програмного забезпечення, освітня програма – інженерія проргамного забезпечення. Вінниця : ВНТУ, 2025. 94 с.

На укр. Мові. Бібліогр. : 22 назв.; рис. : 23; табл. : 2.

У бакалаврській кваліфікаційній роботі обґрунтовано актуальність та практичну цінність розробки десктопного аудіоплеєра з розширеними можливостями аналізу звуку. Метою дослідження стало підвищення ефективності аудіоплеєра за рахунок розширення його функціональних можливостей, що дозволить здійснювати розширений аналіз звуку.

Запропоновано архітектуру аудіоплеєра, яка поєднує в одному середовищі програвання і аналіз, що дало можливіть розширити функціональні можливості аудіоплеєра. Удосконалено метод визначення темпу, який, на відміну від відомих, використовує методики короткочасної енергетичної оцінки та автокореляції з фільтрацією, що дало можливість забезпечити більш точну оцінку темпу музичного сигналу.

Програмне забезпечення розроблене в середовищі IntelliJ IDEA мовою програмування Java. Під час розробки було використано бібліотеки JavaFX для графічного представлення програми та TarsosDSP для функцій аналізу аудіо.

В результаті виконання бакалаврської кваліфікаційної роботи було розроблено програмний засіб для аудіоплеєра з розширеними можливостями аналізу звуку.

Ключові слова: аудіоплеєр, аналіз звуку, музика, темп.

ABSTRACT

Hrinin A.V. Development of audio player software with advanced sound analysis capabilities. Vinnytsia : VNTU, 2025. 94 p.

In ukrainian language. Bibliobrapher: 22 titles; fig. : 23; tabl. : 2.

The bachelor's qualification work substantiates the relevance and practical value of developing a desktop audio player with advanced sound analysis capabilities. The aim of the research was to increase the efficiency of the audio player by expanding its functionality, which will allow for advanced sound analysis.

An audio player architecture is proposed that combines playback and analysis in one environment, which made it possible to expand the functionality of the audio player. The tempo determination method has been improved, which, unlike the known ones, uses short-term energy estimation and autocorrelation with filtering techniques, which made it possible to provide a more accurate estimation of the tempo of a musical signal.

The software was developed in the IntelliJ IDEA environment in the Java programming language. During development, JavaFX libraries were used for the graphical representation of the program and TarsosDSP for audio analysis functions.

As a result of the bachelor's qualification work, a software tool for an audio player with advanced sound analysis capabilities was developed.

Keywords: audio player, sound analysis, music, tempo.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Сучасний етап розвитку цифрових технологій характеризується широким впровадженням інструментів інтелектуальної обробки мультимедійних даних, однак більшість існуючих аудіоплеєрів задовольняють лише базові потреби користувача. З одного боку, більшість аудіоплеєрів пропонують стандартний набір функцій – відтворення треків, елементарний еквалайзер та базову візуалізацію. Проте жоден із них не забезпечує точного спектрального аналізу з розкладкою по смугах частот у реальному часі, не дає можливості інтегрованої оцінки темпу музики та не підтримує гнучкі механізми зациклення й навігації по вибраним фрагментам.

З іншого боку, професійні цифрові аудіостанції – це надлишково складні рішення з високим порогом входження, що вимагають глибоких технічних знань та потужного обладнання. Вони не пристосовані до оперативного аналізу треків у режимі легкого прослуховування і не оптимізовані для швидкої корекції звучання на рівні користувача без спеціалізованої підготовки.

Таким чином, на ринку існує помітний розрив: ні прості програвачі, ні професійні платформи не поєднують у собі легкість використання із потужним інструментарієм для глибокого звукоаналізу. Саме це обумовлює актуальність розробки аудіоплеєра, що не тільки відтворює композиції, але й надає користувачеві можливість у реальному часі виконувати спектральний аналіз, коригувати частотні смуги за допомогою еквалайзера та автоматично визначати темп, забезпечуючи місток між звичайним плеєром і цифровою аудіостудією.

Зв’язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою роботи є підвищення ефективності аудіоплеєра за рахунок розширення його функціональних можливостей, що дозволить здійснювати розширений аналіз звуку.

Основні завдання дослідження:

- здійснити аналіз сучасного стану аудіоплеєрів та аналогічних програмних рішень;

- спроектувати архітектуру програмного продукту;

- розробити метод визначення темпу;

- розробити алгоритм відображення рівня звуку за частотами в реальному часі;

- розробити алгоритм подання композиції у вигляді хвильової форми;

- розробити інтерфейс програми;

- розробити модулі аналізу звуку;

- розробити модулі керування та візуалізації параметрів аудіо;

- виконати тестування програми;

- розробити інструкцію користувача.

Об’єкт дослідження – процес обробки та візуалізації аудіосигналів у програмному середовищі.

Предмет дослідження – методи та засоби обробки та візуалізації аудіосигналів.

Методи дослідження. У процесі досліджень використовувались: методи порівняння функціональних характеристик для аналізу аналогів; теорія алгоритмів для розробки та побудови блок-схем алгоритмів; методи цифрової обробки сигналів, метод швидкого перетворення Фур'є, метод віконної функції; комп’ютерне моделювання для аналізу та перевірки отриманих теоретичних положень.

Наукова новизна отриманих результатів.

1. Запропоновано нову архітектуру аудіоплеєра, яка поєднує в одному середовищі програвання і аналіз, що дало можливіть розширити функціональні можливості аудіоплеєра.

2. Удосконалено метод визначення темпу, який, на відміну від відомих, не використовує стандартну автокореляцію, а застосовує селективне виявлення onset-ів із динамічною класифікацією інтервалів та подальшою корекцією через

кандидатські значення, що дало можливість забезпечити більш точну оцінку темпу музичного сигналу.

Практична цінність отриманих результатів. Практична цінність отриманих результатів полягає в тому, що на основі отриманих в бакалаврській кваліфікаційній роботі теоретичних положень запропоновано алгоритми та розроблено програмний засіб аудіоплеєра з розширеними можливостями аналізу звуку.

Особистий внесок здобувача. Усі наукові результати, викладені у бакалаврській кваліфікаційній роботі, отримані автором особисто. У працях, опублікованих у співавторстві, автору належать результати аналізу значення розширеного аналізу звуку в сучасних аудіоплеєрах [1].

Апробація матеріалів бакалаврської кваліфікаційної роботи. Основні положення роботи доповідались та обговорювались на LIV Всеукраїнській науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (Вінниця, 2025).

Публікації. Основні результати досліджень опубліковано в 1 науковій праці.

Структура та обсяг бакалаврської кваліфікаційної роботи. Робота складаєтья зі вступу, чотирьох розділів, висновків, списку літератури, що містить 22 найменувань, 4 додатки. Робота містить 23 ілюстрацій, 2 таблиці. Загальний обсяг роботи складає 94 сторінки, основний зміст викладено на 51 сторінці друкованого тексту.

1 АНАЛІЗ СУЧАСНОГО СТАНУ АУДІОПЛЕЄРІВ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Загальна характеристика значення розширеного аналізу звуку в сучасних аудіоплеєрах

Сучасні програвачі аудіо зробили значний крок у розвитку, як і будь-які програмні технології у 21-му столітті. Наразі, вони мають градацію рішень від базових програвачів до комплексних інструментів для роботи з аудіо [2]. Проте, існує розрив між цими категоріями, який залишає потреби музикантів та ентузіастів поза увагою. Більшість аудіоплеєрів обмежується базовим функціоналом, якого недостатньо для задоволення потреб цієї групи користувачів, а використання складних професійних цифрових аудіостанцій у повсякденному використанні є незручним.

Таким чином, разом із зростанням запитів користувачів зростає цінність функцій аналізу звуку. Інтеграція розширених можливостей аналізу звуку в аудіоплеєри кардинально змінює підхід до взаємодії зі звуком, відкриваючи нові простори для творчості, навчання та професійного розвитку, оскільки історично функції аналізу та відтворення звуку були розмежовані між різними програмами, що породило цей розрив.

Перше з чим стикається користувач при роботі з аудіоплеєром – це навігація. Навіть така базова функціональність може бути покращена. Можливості швидкої навігації по композиції, такі як встановлення точок повторення або виділення фрагменту для зациклення, є надзвичайно важливими для різних цілей. Аби вловити всі тонкощі треку іноді потрібно прослухати деякі його фрагменти не один десяток разів, тому мати можливість зациклити конкретний фрагмент і не відволікатись на постійне клацання мишею для відтворення конкретного уривку є вкрай корисно. Така маленька дрібниця дасть можливість краще сфокусуватись на творчому процесі. Для прикладу можна уявити людину з гітарою, яка щоразу тягнеться до миші кожних 10-20 секунд

щоб зупинити, перемотати та відтворити з потрібного місця композицію яка вивчається – так не має бути.

Еквалайзер став майже невід'ємною частиною більшості сучасних аудіоплеєрів. Він пройшов шлях від простого інструменту регулювання частот до потужного засобу аналізу складу композиції. Для звичайного користувача, який не пов'язаний з музикою, цей інструмент може покращити досвід прослуховування треків за допомогою визначених пресетів для різних жанрів. Після виділення потрібних частот і послаблення непотрібних трек заграє новими фарбами. Для початківця-музиканта це незамінний інструмент в аналізі звучання різних частот композиції. Справа в тому, що здебільшого початківці не мають дорогого обладнання, яке б дозволило передати звучання в повній мірі. Тому можливість виділити певні частоти і почути деталі, які неможливо без дорогих девайсів, є незамінною при роботі з аудіо в домашніх умовах. Професійні музиканти також використовують цей інструмент для аналізу балансу інструментів, виявлення особливостей мастерингу.

Візуалізація звуку дозволяє сприймати музику не лише слухом, а й зором. Представлення у вигляді хвильової форми, рівнів гучності та спектрограми відкриває нові можливості аналізу. Раніше такі візуалізатори були доступні лише в спеціальних редакторах, тепер вони все частіше з'являються в нових аудіоплеєрах. Представлення амплітуди аудіо у вигляді хвильової форми дозволяє легко розглядати структуру композиції, знаходити кульмінаційні моменти і аналізувати динамічний діапазон. Таким чином можна досліджувати будову треків, що важливо для аранжування та компоновки музики. Спектрограма також є потужним інструментом візуального аналізу, що дозволяє побачити розподіл енергії за частотним спектром [3]. Досвідченні звукорежисери та продюсери можуть миттєво ідентифікувати проблемні частоти, виявити приховані елементи та оцінити якість мастерингу. Це дозволяє виявляти нюанси, які можуть бути непомітними на слух.

Інтеграція автоматичного визначення темпу в аудіоплеєр може кардинально змінити робочі процеси деяких професіоналів музичної галузі.

Зазвичай такий функціонал мають лише спеціалізовані програми з обмеженими можливостями відтворення, іноді вони навіть не мають базової навігації. Для діджеїв точна інформація про темп є основою успіху їх міксів, переходів [4]. Можливість швидко оцінити темп потрібних композицій без використання складних програм значно прискорить підготовку до виступів. Для ентузіастів це чудова можливість робити вправи на визначення темпу для розвитку слуху без необхідності пошуку інформації в інших ресурсах.

Таким чином, можливості розширеного аналізу звуку в аудіоплеєрі знаходить використання як для початківців так і професіоналів музичної галузі, як в навчанні, так і в професійній діяльності. Ключове значення має синергія різних функцій, що дозволяє значно ефективніше працювати з аудіоматеріалами. Можливість глибокого аналізу без необхідності використання кількох програм одночасно оптимізує робочі та навчальні процеси і стимулює творчий пошук [1].

1.2 Порівняльний аналіз аудіоплеєрів із підтримкою аналізу звуку

У сучасному сегменті аудіоплеєрів із розширеними можливостями аналізу звуку існує не багато рішень, які надають різні комбінації базових та просунутих функцій, що не дивно з огляду на історичне розмежовування функціоналу між плеєрами та професійними цифровими станціями. Серед аналогів можна зазначити такі продукти:

- VLC;
- foobar2000;
- MusicBee.

VLC давно зарекомендував себе як універсальний програвач мультимедіа з відкритим вихідним кодом, що підтримує майже всі існуючі аудіо- та відеоформати (див. рисунок 1.1) [5]. У його арсеналі є десятисмуговий графічний еквалайзер із базовими пресетами і кілька візуалізаторів, включно зі спектральним аналізатором, проте він не може в реальному часі відображати форму сигналу у вигляді хвилі або показувати рівень гучності окремих каналів, а вбудованого засобу аналізу темпу в його стандартному наборі немає.

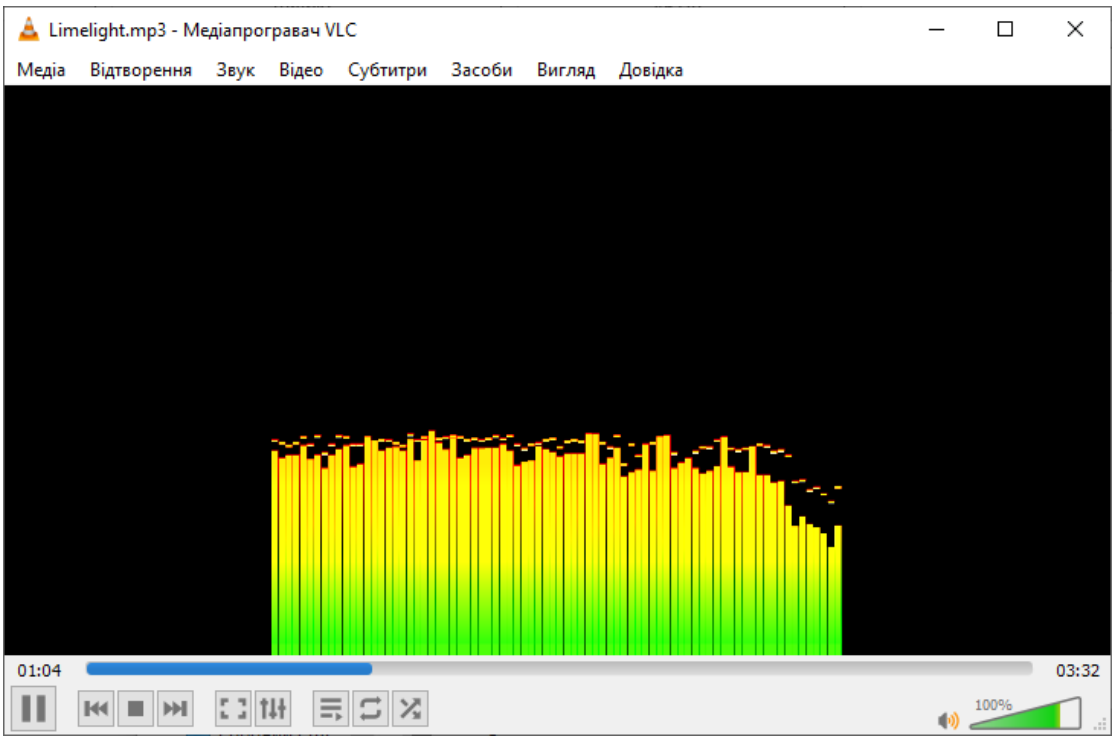


Рисунок 1.1 – Приклад роботи плеєра VLC

Foobar2000 вирізняється модульністю та підтримкою великої кількості сторонніх компонентів, які дозволяють додати еквалайзер із пресетами, спектральну візуалізацію та аналіз темпу через плагіни, проте для кінцевого користувача процес інсталяції й налаштування кожного з них вимагає певного досвіду та часу (див. рисунок 1.2) [6].

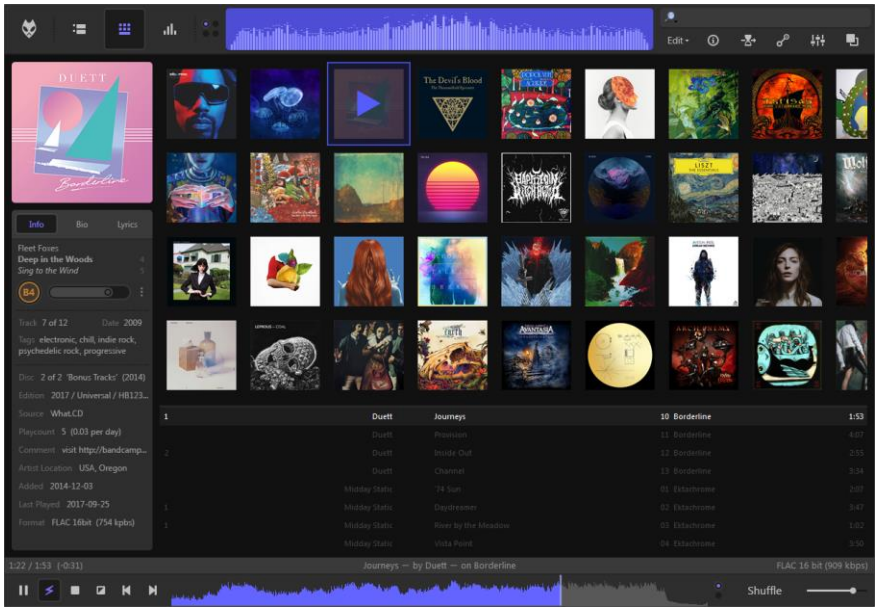


Рисунок 1.2 – Приклад роботи плеєра foobar2000

MusicBee, своєю чергою, пропонує відразу з коробки трисмуговий еквалайзер із пресетами та спектральні панелі, деякі з яких відображають розкладку за частотами, а також базовий інструмент для визначення темпу, але не дозволяє переглядати форму сигналу в реальному часі й не має гнучкого механізму закладок із циклічним програванням фрагментів (див. рисунок 1.3) [7].

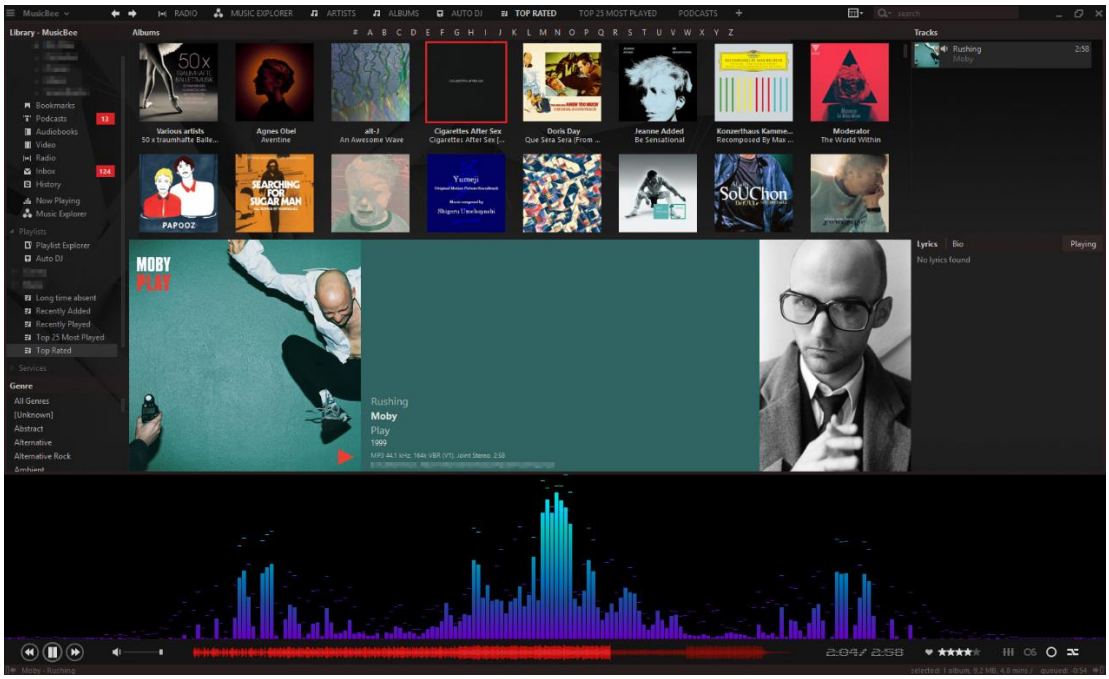


Рисунок 1.3 – Приклад роботи плеєра MusicBee

Для визначення оптимального рішення з розробки модуля аудіоплеєра із розширеними можливостями аналізу звуку було проведено порівняння існуючих аудіо систем. Результати порівняння аналогів наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика аналогів

Критерій	MusicBee	Foobar2000	VLC	Власна програма
1	2	3	4	5
Еквалайзер із пресетами	+	+	+	+

Продовження таблиці 1.1

1	2	3	4	5
Візуалізація хвильової форми	-	+	-	+
Індикатор гучності каналів	-	-	-	+
Спектральна візуалізація смуг частот	+	+	+	+
Спектрограма	-	-	-	+
Аналіз темпу	+	+	-	+
Система закладок	-	-	-	+
Підтримка плагінів	+	+	+	-
Загальна оцінка	4	5	3	7

Як видно з таблиці, класичні аудіоплеєри відзначаються високою стабільністю, однак їх функціонал у сфері аналізу звуку часто обмежений. Наприклад, MusicBee та foobar2000 вимагають використання додаткових плагінів для базової візуалізації, тоді як VLC орієнтований виключно на відтворення аудіо. Розроблювана програма інтегрує не лише стандартний функціонал відтворення, а й модулі аналізу звуку та систему закладок, що вирізняє її серед аналогів. Водночас відсутність підтримки плагінів у програмі виправдана прагненням забезпечити високу точність і продуктивність аналізу без ускладнення архітектури.

Отже, розробка такого аудіоплеєра має потенціал стати універсальним інструментом як для звичайних користувачів, так і для професіоналів аудіоіндустрії.

1.3 Аналіз методів розв’язання задачі

Для розв’язання задачі та реалізації програми аудіоплеєра з розширеними можливостями аналізу звуку важливо сформувати математичну основу, яка забезпечуватиме точне відтворення і обробку сигналу. Основою спектрального аналізу повинно стати дискретне перетворення Фур’є, яке дозволить переходити від часової області до частотної [8]. Обчислення спектра має здійснюватись формулою

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j\frac{2\pi}{N}kn}, \tag{1.1}$$

де X_k – значення комплексного спектрального компоненту, k – індекс спектрального біну, N – кількість семплів в одному вікні, n – відлік кожного аудіосигналу, x_n – семпл вхідного аудіосигналу, j – уявна одиниця.

Отримані результати дозволяють ефективно розбити звуковий сигнал на окремі смуги, що є основою для формування візуалізації спектрограми та візуалізатора частот. Для більш точного визначення характеру сигналу застосовується віконна функція, наприклад, вікно Геммінга, що описується формулою

$$\omega_n = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \tag{1.2}$$

де ω_n – коефіцієнт вікна для n-го зразка, N – загальна кількість зразків у вікні, n – індекс зразка [9]. За допомогою цього вікна зменшується ефект розмиття спектра та підвищується точність розрахунків, що є критично важливим при обчисленні амплітудних характеристик для подальшої візуалізації.

При реалізації еквалайзера необхідно оперувати біквадратними фільтрами Infinite Iimpulse Response (IIR) другого порядку, основна структура яких описується різницевим рівнянням

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} - a_1 y_{n-1} - a_2 y_{n-2}, \tag{1.3}$$

де y_n – вихідний семпл фільтра, x_n – вхідний семпл фільтра, b_0, b_1, b_2 – коефіцієнти підсилення вхідних семплів, a_1, a_2 – коефіцієнти зворотного зв'язку [10].

Візуалізація хвильової форми опиратиметься на безпосереднє виведення семплів x_n з формули 1.3 у часовій області з додатковим згладжуванням за допомогою ковзного середнього.

Для побудови спектрограми слід розбивати сигнал на накладні вікна і послідовно засосовувати метод швидкого перетворення Фур'є до кожного фрагмента. Амплітуда кожного спектрального біна перетворюється на інтенсивність кольору з координатами, в яких час відповідає порядковому номеру вікна, а частота – індексу біну.

Аналіз темпу базуватиметься на оцінці енергії сигналу в часовій області, що виражається через обчислення короткочасної енергії, що дасть змогу виявити періодичні ритмічні імпульси. Після чого виконується обчислення BPM=60/T, де BPM – це удари за хвилину, а T – середній інтервал між імпульсами.

Таким чином, методами розв'язання задачі для успішної реалізації аудіоплеєра з розширеними можливостями аналізу звуку є дискретне перетворення Фур'є, віконні функції, біквадратні IIR-фільтри. Ці методи будуть фундаментом для архітектури програми та забезпечать необхідний рівень точності.

1.4 Постановка задач дослідження

На підставі проведеного попереднього аналізу існуючих аудіоплеєрів, їхніх переваг та недоліків, а також методів аналізу звуку, було сформульовано такі завдання дослідження для розробки аудіоплеєра з розширеними можливостями аналізу звуку:

- виконати аналіз вхідних та вихідних даних;
- спроектувати архітектуру програмного забезпечення;
- розробити метод визначення темпу;
- розробити алгоритми відображення рівня звуку за частотами в реальному часі та подання композиції у вигляді хвильової форми;

- виконати варіантний аналіз та обґрунтування вибору засобів для реалізації програмного забезпечення;

- розробити інтерфейс програми;

- розробити модулі аналізу звуку;

- розробити модулі керування та візуалізації;

- провести тестування програмного забезпечення згідно поставлених задач;

- розробити інструкцію користувача.

Технічне завдання на розробку наведено в додатку А.

1.5 Висновки

У першому розділі було проведено аналіз сучасних аудіоплеєрів, таких як, MusicBee, foobar2000, VLC. Результат демонструє, що на ринку існує перелік якісних продуктів, здатних забезпечити базове відтворення аудіофайлів і навіть підтримувати деякі стандартні функції візуалізації. Проте більшість із них обмежені у можливостях інтегрованого аналізу звукового сигналу, що створює простір для розробки нового продукту з розширеним функціоналом.

Проаналізувавши переваги та недоліки існуючих програмних засобів, було сформульовано завдання для дослідження. Ці завдання включають аналіз вхідних та вихідних даних, проектування архітектури та розробку алгоритмів програми, виконання варіантного аналізу та обґрунтування вибору засобів для реалізації програмного забезпечення, розробку інтерфейсу, модулів аналізу звуку, керування та візуалізації, проведення тестування та розробку інструкції користувача. Таким чином було доведено актуальність розробки власного програмного рішення.

2 РОЗРОБКА АРХІТЕКТУРИ, МЕТОДУ ТА АЛГОРИТМІВ ПРОГРАМНОГО ПРОДУКТУ

2.1 Аналіз вхідних та вихідних даних системи

У процесі проектування аудіоплеєра з розширеними можливостями аналізу звуку на стадії моделювання необхідно чітко визначити характер і формат даних, що надходитимуть на вхід системи, а також форму результатів, які вона видаватиме користувачу й іншим компонентам. Вхідними даними слугуватимуть цифрові аудіосигнали із звукових файлів стандартних форматів WAV та MP3 [11]. Кожен файл передаватиметься до модуля попередньої обробки, де відбуватиметься декодування потоку даних, переводячи його в PCM-формат без втрат із визначеною оригіналом частотою дискретизації та бітовою глибиною 16 біт [12].

Далі коригований аудіосигнал подаватиметься на вхід програмного ядра для побудови спектрограми та фільтрації. Для FFT-алгоритму важливо, щоб дані надходили блоками фіксованого розміру, наприклад 1024 або 2048 семплів, із кроком перекриття 50 % для забезпечення плавності оновлення спектрограми. При виконанні еквалайзера набір коефіцієнтів фільтрів для кожної з дев’яти смуг прийматиметься як вхідні константи, які можуть бути обчислені на основі пресетів або задані вручну користувачем через інтерфейс налаштувань.

Механізм визначення темпу аналізує тимчасову послідовність семплів у часовій області. Це передбачає передавання короткочасних енергетичних значень із фіксованим кроком, що є вхідними даними. Затримка між оновленнями буде встановлена таким чином, щоб обчислення могло виконуватися в реальному часі без видимих затримок користувачеві.

Крім основного потоку аудіоданих, на систему також будуть надходити вхідні параметри контролю: команда відтворення або паузи, позиція перемотування та часові мітки для закладок. Ці сигнали реалізовуватимуться у вигляді структур даних іменованих подій, що передаються до відповідного сервісу управління станом плеєра. Завдяки синхронізації цих команд із часовим

індексом аудіопотоку стає можливим коректно реагувати на дії користувача та підтримувати стабільний стан відтворення.

Вихідні дані формуватимуться у двох основних напрямах. Перший – безпосередньо аудіосигнал, який після проходження через фільтри еквалайзера та інші обробні модулі повертається на аудіовихід системи у вигляді PCM-потоку або передається в звуковий драйвер платформи. Другою категорією є візуалізаційні дані, які генеруються для відображення хвильової форми, спектрограми, індикації рівнів лівого та правого каналу і змін параметрів еквалайзера. Ці результати подаватимуться у вигляді масивів числових значень або структурованих об’єктів даних, що містять часові та амплітудні координати для побудови графіків.

Отже, на етапі аналізу вхідних та вихідних даних сформовано чітке уявлення про структуру та формат інформації, що циркулює між модулями плеєра і інтерфейсом користувача, що є критично необхідним для подальшої розробки моделі та алгоритмів програмного продукту.

2.2 Розробка архітектури програмного продукту

Для кращого розуміння структури розроблюваного аудіоплеєра з розширеними можливостями аналізу звуку було спроектовано структурну схему, що зображена на рисунку 2.1.

З точки зору об’єктно-орієнтованої архітектури, програмний продукт побудовано за принципами модульності [13].

Клас Main – точка входу в додаток, що ініціалізує сцену.

MainController – відповідає за управління графічним інтерфейсом користувача. Він ініціалізує всі ключові компоненти:

- модуль відтворення AudioAPlayer;

- вікно еквалайзера EqualizerWindow;

- модулі візуалізації – WaveformView, VolumeView, SAView, SpectrogramView, VolumeIndicatorView;

- модуль аналізу музики MusicAnalyzer.

Користувацький інтерфейс обробляє взаємодію з елементами керування — кнопками відтворення, слайдером гучності, керуванням закладками та вмиканням режиму зациклення.

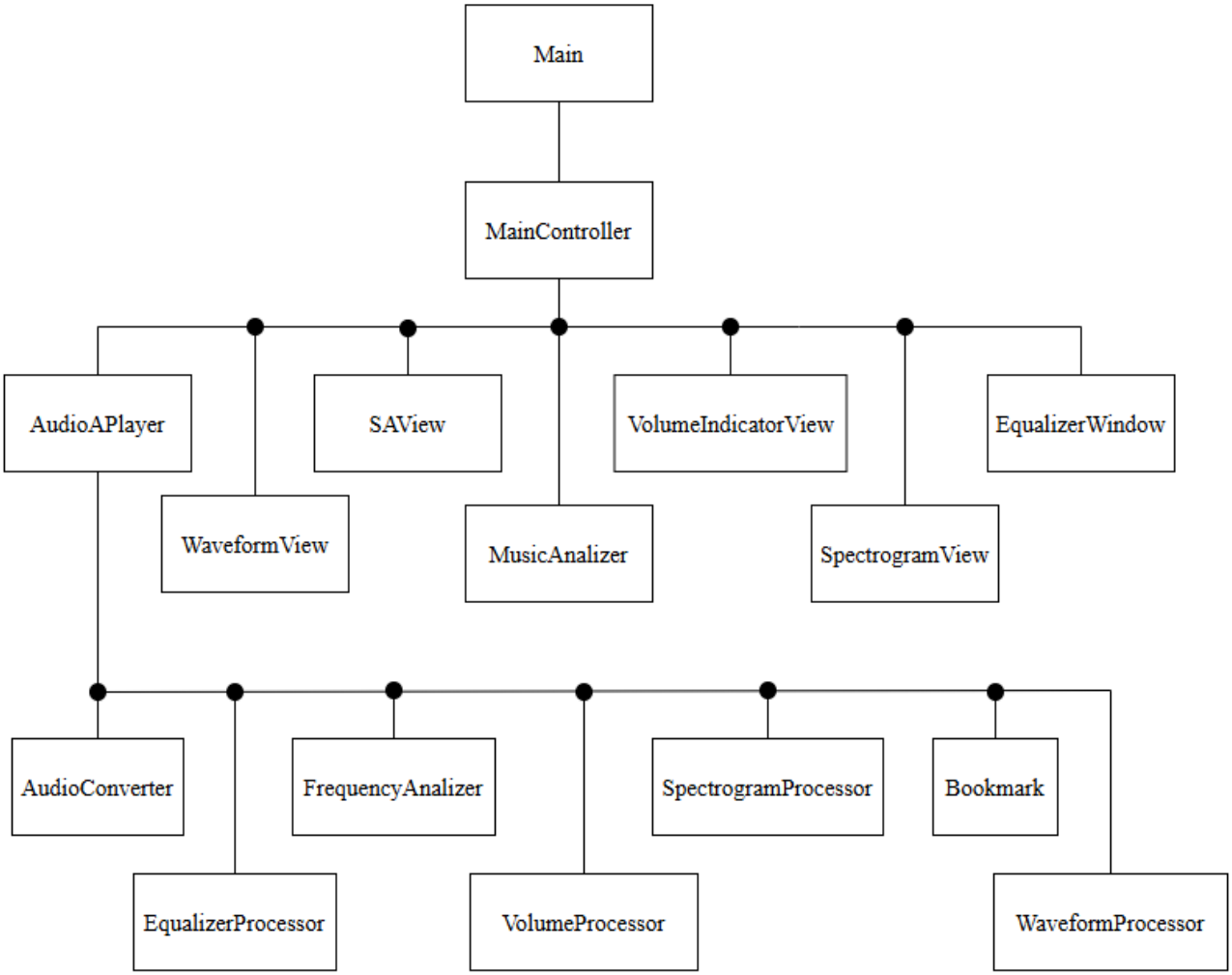


Рисунок 2.1 – Структурна схема програмного застосунку

AudioAPlayer – головний функціональний модуль аудіоплеєра. Він поєднує управління медіа, а саме – відкриття, конвертацію, відтворення, паузу, зупинку, перемотку, та генерацію хвильової форми, спектрограми, спектральний аналіз, керування гучністю та закладками. В рамках обробки сигналу він координує роботу таких класів:

- AudioConverter – модуль конвертації аудіофайлів у WAV-формат;

- EqualizerProcessor – аудіофільтр, який застосовується до сигналу у реальному часі. Реалізує обробку за 9 частотними смугами з можливістю

налаштування коефіцієнтів підсилення для кожної смуги, включно з пресетами;

- FrequencyAnalyzer – модуль спектрального аналізу на базі швидкого перетворення Фур'є. Обчислює поточні рівні кожної частотної смуги;

- SpectrogramProcessor – модуль відповідальний за обробку даних для побудови спектрограми;

- VolumeProcessor – модуль відповідальний за зміну гучності;

- Bookmark – структура даних для збереження часових міток, яка дозволяє реалізувати перемикання між ними та режим циклічного відтворення.

WaveformView – віджет для візуалізації звукової хвилі. Цей клас генерує графічне представлення аудіоданих, відображаючи як відтворену, так і невідтворену частини треку. Окрім цього, він підтримує інтерактивне керування позицією відтворення та відображення закладок користувача.

VolumeIndicatorView – модуль для візуального відображення рівня гучності по кожному каналу окремо.

SAView – візуальний компонент, який динамічно відображає амплітуди частотних смуг у вигляді кольорової гістограми.

SpectrogramView – візуальний компонент що відображає залежність спектральної щільності потужності сигналу від часу у вигляді графіка.

EqualizerWindow – окреме діалогове вікно з повзунками для керування еквалайзером. Забезпечує встановлення значень підсилення для кожної смуги, а також підтримує типові пресети.

Однією з ключових переваг даної архітектури є її гнучка модульна структура, що дозволяє інтегрувати додаткові підсистеми без істотного перепроектування. Усі взаємодії між модулями відбуваються через чітко визначені інтерфейси або слухачі, що спрощує розширення функціональності та забезпечує низький рівень зв’язності між компонентами.

Розробка структурної схеми була виконана у програмному забезпеченні Visio [14].

2.3 Розробка методу визначення темпу

Метод визначення темпу відрізняється від традиційних алгоритмів тим, що не покладається на стандартну автокореляцію, а використовує селективне виявлення початкових звукових подій onset-ів разом із динамічною класифікацією інтервалів і подальшою корекцією обчисленого темпу через аналіз кандидатських значень.

У першому етапі здійснюється аналіз аудіосигналу для ідентифікації моменти початку звукових подій onset-ів. Завдяки застосуванню селективного порогового фільтрування відбираються лише ті події, що мають високий рівень значущості, що дозволяє уникнути врахування випадкових або слабко виражених змін енергії сигналу. Такий підхід забезпечує достовірне виділення основних ритмічних маркерів, що є фундаментом для подальшого аналізу.

Блок-схема алгоритму реалізаціїметоду визначення темпу наведена на рисунку 2.2.

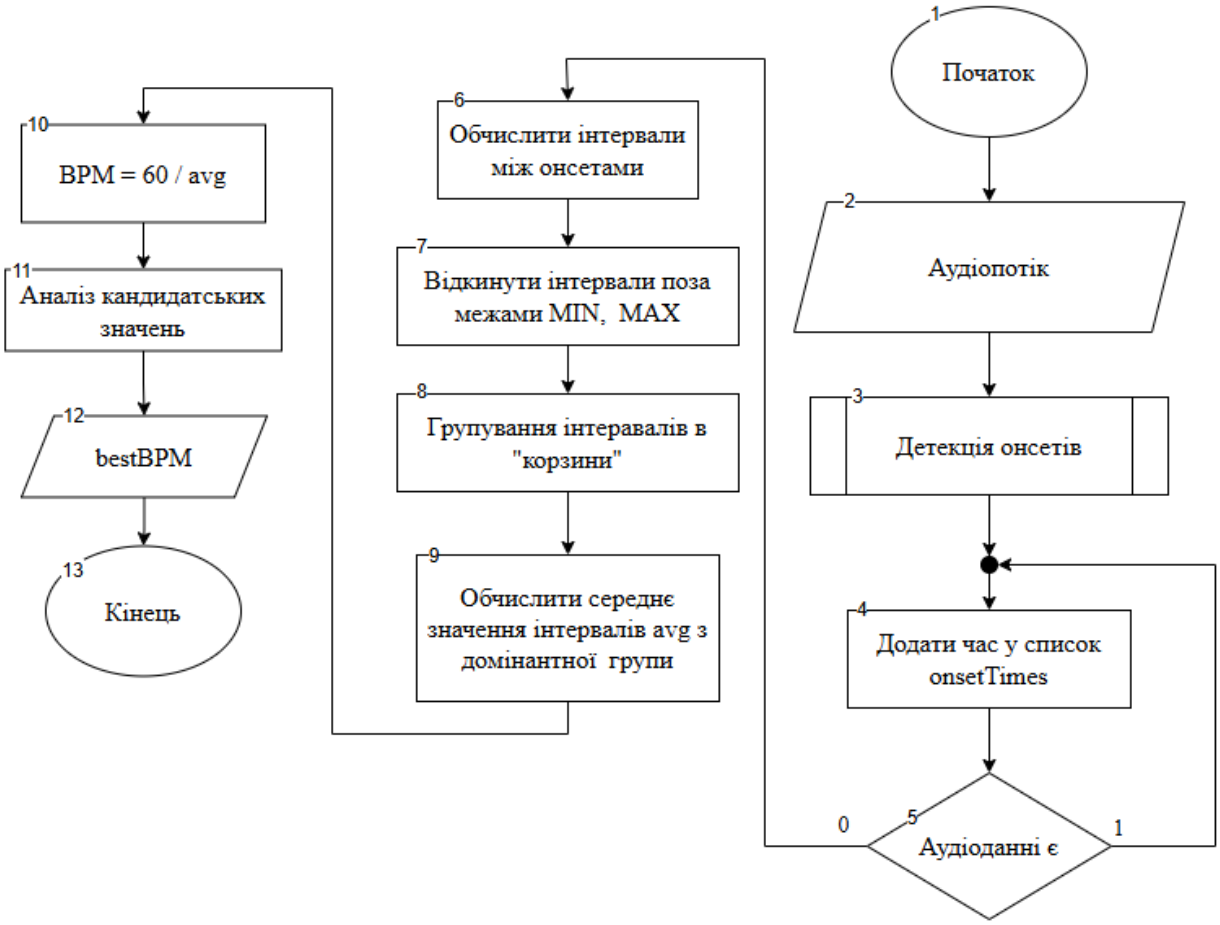


Рисунок 2.2 – Блок-схема алгоритму визначення темпу

Після виявлення onset-ів розглядаються часові інтервали між послідовними подіями. Алгоритм формує динамічні «корзини» для класифікації інтервалів, що дозволяє згрупувати схожі за значенням часові відрізки. Це групування є своєрідною формою фільтрації, яка усуває вплив випадкових вимірювань та шумових спрацьовувань, виділяючи домінантну групу інтервалів. Саме за допомогою цієї групи обчислюється середнє значення інтервалу, яке стає базою для розрахунку темпу.

Використовуючи базову формулу ділення 60-ти на середній інтервал, отримується початкове значення темпу. Однак, з метою підвищення точності, алгоритм проводить подальшу корекцію – обчислене значення перевіряється на відповідність заданому допустимому діапазону темпу. Для цього формується набір кандидатських значень, який включає кратні корекції, і остаточно обирається оптимальний показник, найбільш відповідний для аналізованого аудіосигналу.

Таким чином, цей метод забезпечує точне визначення темпу музичного твору за допомогою підходу, що поєднує в собі ефективне відсівання шумових компонентів, адаптивну класифікацію часових інтервалів та фінальну корекцію результату, що дає змогу отримати високоточну оцінку.

2.4 Розробка алгоритмів аналізу звуку

Для розробки модуля візуалізації різних параметрів аудіо для подання композиції у вигляді звукової хвилі, а також відображення сили звуку в реальному часі необхідно розробити два алгоритми, згідно яких буде працювати програмний модуль.

Алгоритм відображення рівня звуку за частотами об’єднує два підмодулі: спектральний аналізатор FrequencyAnalyzer та візуалізатор SAView. Спочатку кожен аудіоблок моделюється як масив, до якого застосовується віконна функція Геммінга для згладження переходів. Потім виконується швидке перетворення Фур’є, і з результату береться масив амплітуд. Частотний діапазон розбивається на дев’ять смуг навколо центрованих частот від 62 до 16000 Гц, для кожної з яких

визначаються межі у бін-індексах. Усередині смуги обчислюються пікова та середня амплітуда, після чого результати масштабується за допомогою вагових коефіцієнтів та проходять через фільтр для згладжування швидких змін. Нормалізовані рівні передаються в SAView.

У SAView надходять цільові рівні смуг, які плавно підтягуються до поточних із фіксованим коефіцієнтом згладжування. Якщо нових даних немає довше за деякий поріг, застосовується пульсаційний ефект для створення анімованого фону. При кожному кадрі анімації перевіряється, чи змінилися рівні, і за потреби перерисовується: для кожної смуги малюється прямокутник висотою, пропорційною поточному рівню, з кольоровим градієнтом і підписом частоти. Така схема дозволяє користувачу бачити зміни енергетичного розподілу звуку по частотах в реальному часі. На рисунку 2.3 зображена блок-схема цього алгоритму.

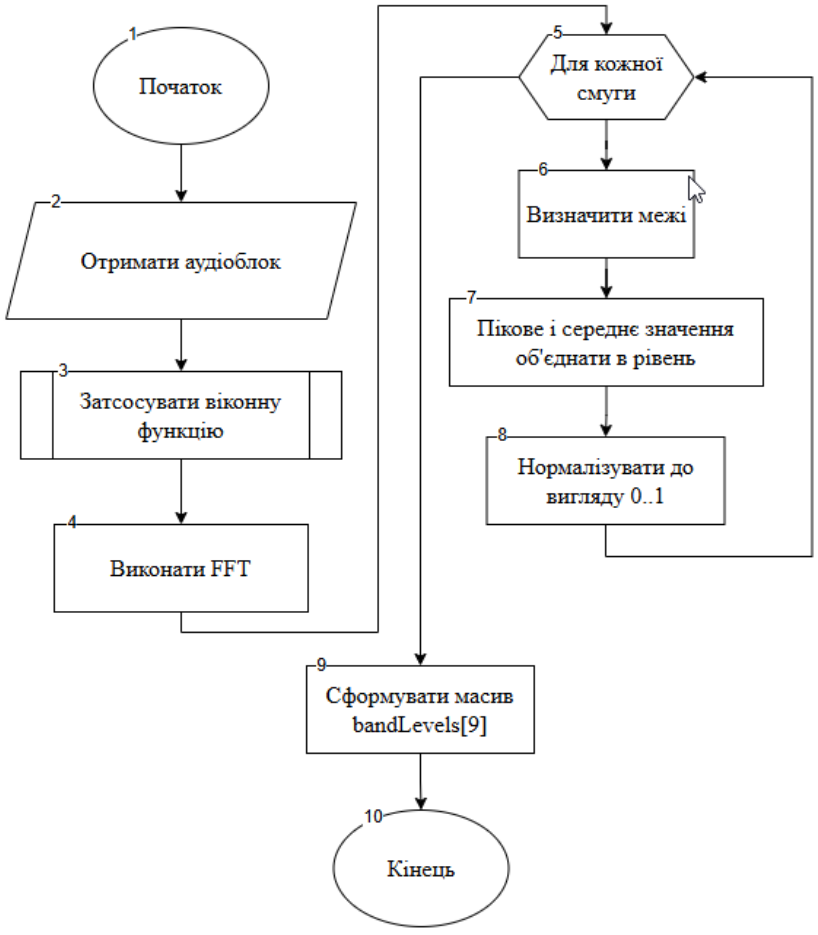


Рисунок 2.3 – Блок-схема алгоритму відображення рівня звуку за частотами

Алгоритм генерації хвильової форми починається з відкриття концентрованого джерела аудіоданих, представленого WAV-файлом, який може бути отриманий або шляхом конвертації вхідного аудіофайлу з іншого формату, або якщо аудіофайл уже знаходиться у WAV-форматі (див. рисунок 2.4). Основною метою цього алгоритму є розбиття аудіо потоку на менші сегменти, для кожного з яких обчислюється значення амплітуди, що дозволяє отримати детальний зразок динаміки сигналу протягом треку.

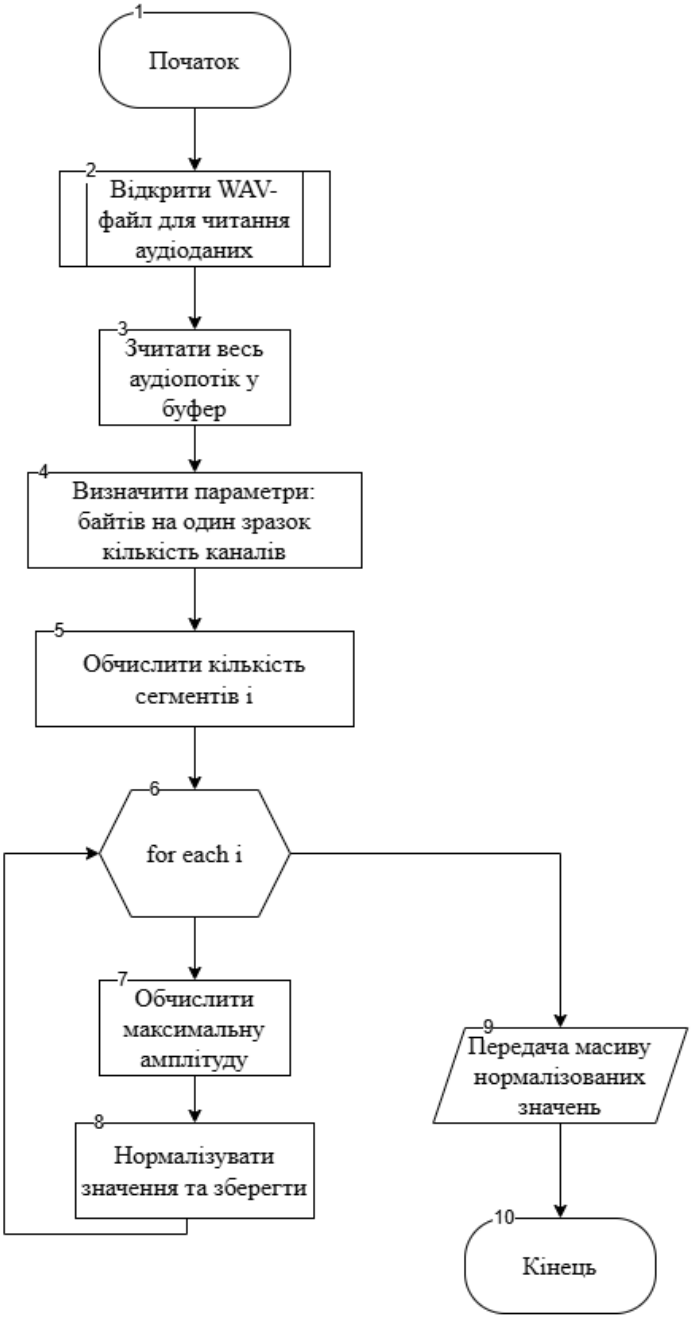


Рисунок 2.4 – Блок-схема алгоритму генерації хвильової форми

При запуску алгоритму весь аудіопотік зчитується у буфер у вигляді байтового масиву. Для коректної обробки даних визначається специфіка формату файлу: кількість байтів, що використовуються на один зразок, а також кількість каналів, що містить аудіо. Ці параметри є ключовими, оскільки вони впливають на спосіб розбиття потоку на вибірки. Далі визначається загальна кількість сегментів, які будуть обчислюватися згідно із заданою роздільною здатністю візуалізації. Визначення цього параметра дозволяє адаптувати деталізацію хвильової форми до розміру області відображення.

Після цього алгоритм послідовно проходить через усі визначені сегменти аудіо. Для кожного сегменту аналізуються значення амплітуди, що містяться в байтовому масиві, причому алгоритм перебирає кожен фрейм сегменту. Він обирає максимальне значення амплітуди, яке відображає пікове значення звукової хвилі у даній частині аудіо. Для досягнення узагальненого представлення динаміки сигналу це значення нормалізується до діапазону від 0 до 1, що дозволяє працювати з однаковим масштабом незалежно від початкових значень конкретного аудіофайлу. Отримані нормалізовані значення зберігаються у масиві, який стає базою для побудови графічної хвильової форми.

Після завершення розрахунків масив, що містить нормалізовані значення амплітуди для кожного сегменту, передається до віджетного компонента, відповідального за візуальне представлення. Цей компонент перетворює отриманий масив на графічне зображення, де кожен окремий елемент масиву відповідає за відображення відповідного вертикального стовпчика або лінії. Різниця в кольорах може підкреслювати частину треку, яка вже була прослухана, порівняно з тією, що ще не була відтворена.

Таким чином, алгоритм генерації хвильової форми не просто розбиває аудіо на окремі вибірки, а створює детальну карту амплітудного розподілу звукового сигналу. Це дозволяє користувачу отримати наочне представлення динаміки аудіо, що робить процес відтворення інтуїтивно зрозумілим і дозволяє взаємодіяти з аудіоданими у режимі реального часу.

Отже, було розроблено алгоритми відображення рівня звуку за частотами, та генерації хвильової форми та сформовано їх блок-схеми.

2.5 Висновки

У другому розділі було проведено аналіз вхідних та вихідних даних програми. Було виконано розробку архітектури та сформовано структурну схему програмного застосунку. Розроблено вдосконалений метод визначення темпу, який завдяки інтегрованій обробці вхідних часових характеристик сигналу та постійної корекції вихідного значення здатен досягати кращої точності. Також було детально розроблено алгоритми відображення рівня звуку за частотами в реальному часі та подання композиції у вигляді хвильової форми.

3 РОЗРОБКА ПРОГРАМНИХ МОДУЛІВ АУДІОПЛЕЄРА З РОЗШИРЕНИМИ МОЖЛИВОСТЯМИ АНАЛІЗУ ЗВУКУ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного забезпечення

При розробці інтегрованого аудіоплеєра, який забезпечує як базовий функціонал відтворення аудіо, так і розширений аналіз звуку, важливе значення має обґрунтований вибір платформ та інструментів розробки. Головними претендентами в якості мови програмування для реалізації цього програмного забезпечення є Java та C# [15, 16].

Обидві мови є потужними об’єктно-орієнтованими мовами програмування з широкими можливостями, проте при виборі технології для даної програми враховано кілька ключових чинників. Хоча C# має потужний набір бібліотек і тісно інтегрується із середовищем Windows, основна перевага Java полягає в її уніфікованості та незалежності від конкретної операційної системи. Серед переваг вибору Java для розробки слід виділити, що Java має величезну екосистему бібліотек і фреймворків, що дозволяє швидко інтегрувати різноманітні функціональні можливості, зокрема для аудіо і графічної обробки. Крім того, спільнота розробників регулярно підтримує і розширює існуючі рішення. Також завдяки багаторічному використанню Java у корпоративних додатках, ця мова демонструє високий рівень стабільності, що є критично важливим для розробки систем з обробкою в реальному часі, якою є розроблюваний аудіоплеєр.

У розробці графічного інтерфейсу аудіоплеєра також було проведено детальний аналіз доступних технологій. Серед можливих варіантів – традиційна бібліотека Swing, сучасна платформа JavaFX та безпосередня робота з OpenGL.

Swing, як одна з перших бібліотек для побудови графічних інтерфейсів у Java, має свою історію і забезпечує базовий набір компонентів. Однак вона поступається сучасним рішенням за рівнем інтерактивності, зручністю використання і підтримкою апаратного прискорення. Крім того, Swing не має

стандартного механізму для роботи з мультимедіа, що ускладнює інтеграцію аудіо та графічної візуалізації [17].

OpenGL – це потужний набір API для роботи з 3D-графікою, який забезпечує дуже високу продуктивність і низькорівневий контроль над візуальною частиною. Проте ця технологія є низькорівневою і потребує написання значущої кількості додаткового коду для організації інтерфейсу – робота з OpenGL помітно ускладнюється, якщо потрібно підтримувати стандартну побудову елементів управління, розташування компонентів і інтерактивність, що важливо для аудіоплеєра [18].

JavaFX, на відміну від Swing та OpenGL, поєднує переваги сучасного інтерфейсу із простотою розробки. JavaFX підтримує FXML для опису розмітки інтерфейсу, що робить процес дизайну простішим і дозволяє розділити логіку програми від її графічного відображення. Крім того, JavaFX пропонує систему анімації, ефектів та апаратного прискорення, що забезпечує плавну роботу навіть при високих вимогах до інтерактивності, наприклад, під час візуалізації хвильової форми аудіо. Таким чином, JavaFX обрано завдяки його сучасному дизайну, широким можливостям для розробки інтерактивної графіки та простоті інтеграції з іншими компонентами додатку [19].

Для роботи із аудіоданими та їх подальшої обробки розробникам доступні різні API. Для вибору засобу реалізації розроблюваної програми потрібно провести аналіз можливостей стандартного Java Sound API та альтернативного набору інструментів, до якого належить бібліотека TarsosDSP.

Java Sound API є стандартною частиною платформи Java і забезпечує базовий спектр функцій для зчитування, запису і конвертації аудіоданих. Завдяки своїй інтегрованості в Java Development Kit, Java Sound API дозволяє ефективно обробляти аудіопотоки, виконувати операції конвертації форматів і безперешкодно працювати з PCM-даними, що необхідні для подальшого аналізу та візуалізації. Його використання є достатнім для реалізації вимог розроблюваного аудіоплеєра, оскільки обчислювальне навантаження і швидкість виконання операцій перевіряються під час тестування [20].

Бібліотека TarsosDSP пропонує розширені можливості цифрової обробки сигналів, такі як більш складні алгоритми аналізу, виявлення тональних характеристик, ефективну обробку частотних спектрів тощо. Проте, ці додаткові можливості часто виходять за рамки основних потреб аудіоплеєра з розширеними можливостями аналізу звуку, що орієнтований переважно на стандартне відтворення, базову конвертацію і візуалізацію аудіоданих [21].

Аналіз цих інструментів показав, що TarsosDSP є оптимальним вибором для реалізації функціоналу аудіоплеєра з розширеними можливостями аналізу звуку. Воно забезпечує потрібну функціональність для зчитування, обробки та конвертації аудіоданих із мінімальними витратами ресурсів і є достатньо ефективним.

Таким чином для розробки модулів аудіоплеєра із розширеними можливостями аналізу звуку мовою програмування було обрано Java. Для реалізації графічного відображення – JavaFX, через ефективність роботи із мультимедіа. Для роботи зі звуком обрано TarsosDSP, оскільки вона володіє усім необхідним функціоналом.

3.2 Розробка інтерфейсу програмни

Графічний інтерфейс аудіоплеєра реалізовано за допомогою JavaFX та FXML, що дозволяє зручно розподілити візуальні компоненти за групами.

Основним кольором інтерфейсу обрано темно-сірий, а для елементів візуалізації даних – чорний та білий для найкращої контрастності. Допоміжними кольорами обрано червоний, оранжевий, жовтий, зелений, синій та фіолетовий, і щоб відокремити різні частоти різними кольорами та градієнтами при візуалізації спектрального аналізу в реальному часі.

При відкритті програми, користувача зустрічає головний екран, що також зображений на рисунку 3.1.

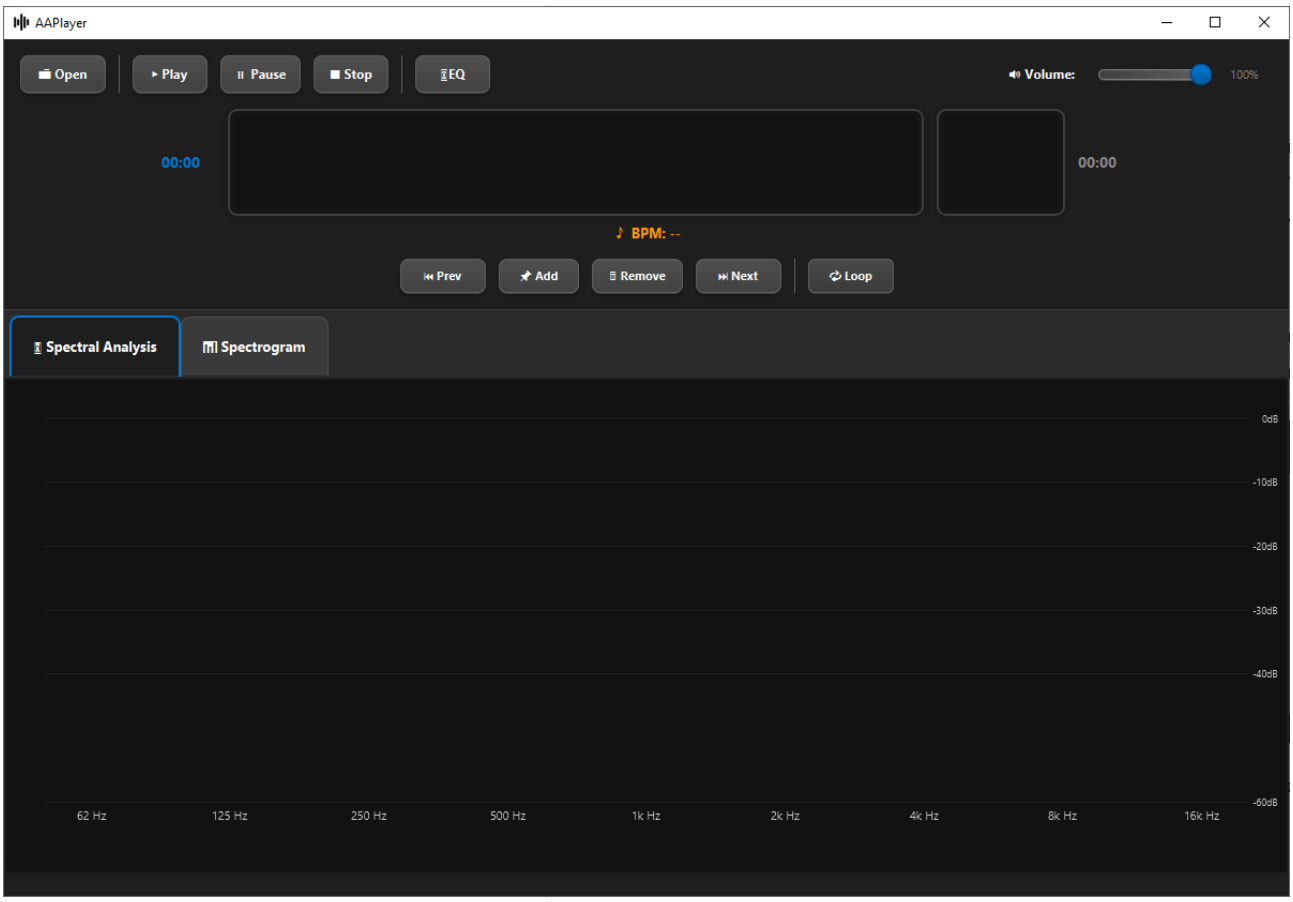


Рисунок 3.1 – Головний екран

На головному екрані розташовано все керування програмою: кнопки відкриття файлу, запуску, паузи, зупинки, еквалайзера, повзунок зміни гучності, а також кнопки переміщення, додавання і видалення закладок і перемикач зацикленості. Тут же розташовано дві панелі з можливістю перемикатись між візуалізацією рівня звуку за частотами та спектрограмою.

Після завантаження обраного аудіофайлу на головному екрані з’явиться відображення хвильової форми. Цей елемент також є елементом відображення прогресу програвання пісні, керування перемоткою та відображення закладок. Справа від відображення хвильової форми розташовані два індикатора гучності для лівого та правого каналу.

На рисунку 3.2 зображено вікно еквалайзера яке з’являється після натиснення відповідної кнопки.

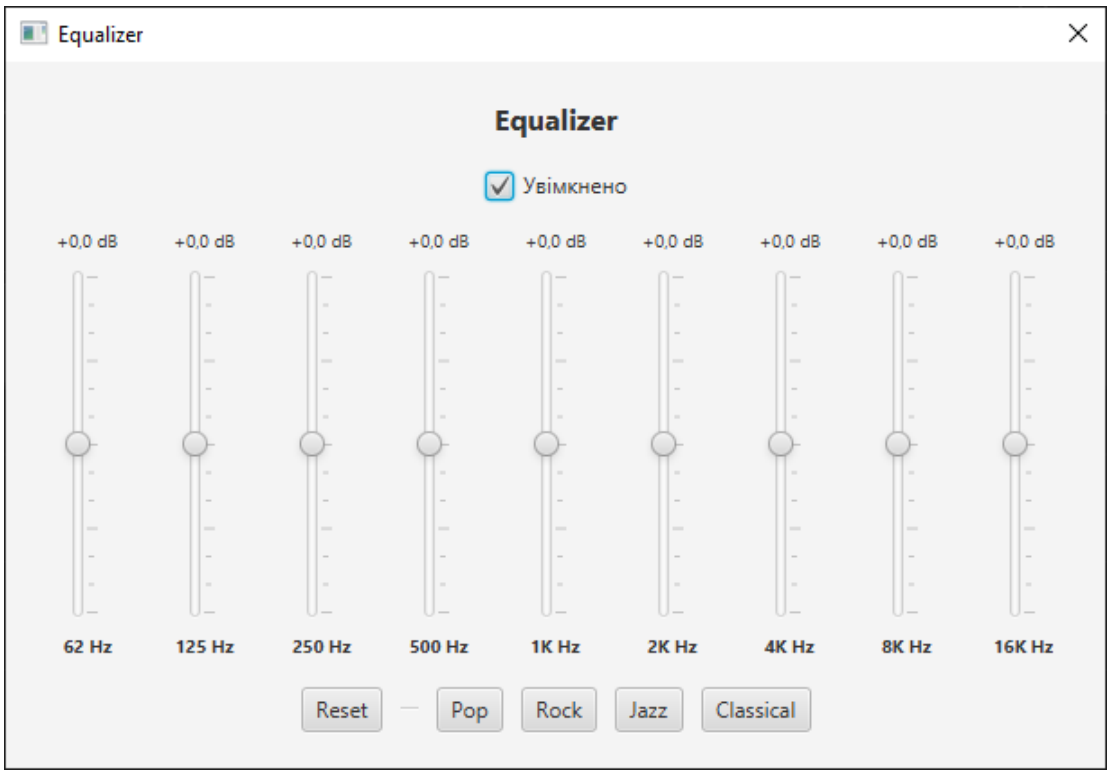


Рисунок 3.2 – Вікно еквалайзера

На цьому вікні розташовано чекбокс для швидкого увімкнення або вимкнення еквалайзера, повзунки підсилення контрольних частот та кнопки скидування і різних пресетів.

Основний акцент зроблено на інтерактивності, тому обробка подій миші на полотні для хвильової форми та даних з слайдера має бути плавною та реагувати без затримок. Це досягається за допомогою використання властивостей JavaFX і зворотних викликів, які в режимі реального часу оновлюють графічне представлення стану відтворення.

Розробка графічного інтерфейсу програмного застосунку була проведена у середовищі розробки IntelliJ IDEA з використанням JavaFX.

3.3 Розробка модулів аналізу звуку

Модулі аналізу звуку становлять серце аудіоплеєра з розширеними можливостями обробки, оскільки саме вони вирішують завдання перетворення сирих семплів у корисні цифрові дані, які лягатимуть в основу подальших операцій. Ключовими компонентами аналітичного конвеєра програмного

продукту є класи AudioConverter, EqualizerProcessor, SpectrogramProcessor, FrequencyAnalyzer, MusicAnalyzer та WaveformProcessor.

Модуль конвертації форматів аудіофайлів AudioConverter відповідає за забезпечення сумісності вхідних аудіофайлів незалежно від їх початкового формату, переводячи їх у WAV-формат, який є більш універсальним для подальшої обробки, аналізу та відображення даних. Функціональність модуля базується на використанні інтерфейсу TarsosDSP, що дозволяє зчитувати аудіодані, виконувати операції перетворення формату та запису файлів у стандартний формат PCM 16-bit.

Модуль конвертації працює наступним чином. Коли аудіофайл завантажується в систему, користувацький код передає шлях до цього файлу модулю конвертації. Спочатку відбувається перевірка розширення файлу: якщо файл уже має розширення .wav, модуль негайно повертає шлях до нього, оскільки додаток може без проблем працювати з WAV-сигналом. Якщо ж формат аудіофайлу відмінний від WAV, модуль намагається відкрити його за допомогою AudioSystem.getAudioInputStream. Після успішного зчитування визначаються параметри вхідного файлу, такі як частота дискретизації, кількість каналів і розрядність вибірки, що є критично необхідним для правильного перетворення формату.

Далі формується цільовий формат зі стандартними характеристиками – PCM_SIGNED, із тією ж частотою дискретизації, 16 бітами на зразок і відповідною кількістю каналів. Цей формат вибрано через його універсальність та сумісність із алгоритмами обробки аудіоданих, які використовуються в подальшій роботі додатку. В процесі конвертації створюється новий AudioInputStream, який забезпечує автоматичне перетворення аудіоданих із початкового формату у цільовий формат. Для унікального збереження отриманого WAV-файлу генерується ім’я за допомогою UUID, а створений файл записується у спеціально визначену тимчасову директорію, що була створена під час ініціалізації модуля.

Таким чином, подальша обробка аудіоданих здійснюється з використанням однакового, стандартизованого формату, що знижує ризик помилок під час аналізу сигналу та забезпечує більш точне відображення його характеристик. Дуже важливою частиною роботи модуля є також механізм очищення тимчасових файлів. Для цього реалізований метод, який перевіряє наявність файлів у тимчасовій директорії та видаляє їх, що дозволяє запобігти накопиченню непотрібних даних і знизити ризик витоку пам’яті.

Клас EqualizerProcessor реалізує обробник аудіосигналу, що вбудовується у конвеєр TarsosDSP. Після ініціалізації він створює для кожної з дев’яти центрованих частот від 62 Гц до 16 кГц вузькополосні фільтри BandPass із розрахованими межами смуг. У процесі обробки кожного аудіоблоку він копіює сирий буфер, послідовно застосовує до окремих копій фільтрацію для кожної смуги та накопичує відфільтрований сигнал, масштабуючи його відповідно до поточного коефіцієнта підсилення, який зберігається в лінійній шкалі. Після поєднання обробленого відгуку та оригінального сигналу відбувається обмеження значень, щоб уникнути перевантаження, і запис результату назад у буфер, який передається далі до виводу. Клас надає методи для встановлення та отримання підсилення окремих смуг, скидання всіх налаштувань до нейтрального рівня 0 дБ та увімкнення або вимкнення обробки еквалайзером. Пресети реалізовані як фіксовані набори значень дБ, що дозволяє одним викликом адаптувати звучання до типового музичного жанру.

Клас SpectrogramProcessor працює за принципом послідовного розбиття аудіопотоку на окремі кадри з використанням алгоритму швидкого перетворення Фур’є. При кожному виклику методу process клас отримує буфер аудіоданих, з якого формується комплексний вхідний масив для FFT з подвоєною довжиною. Перш ніж виконати перетворення, до даних застосовується віконна функція Хеммінга, що мінімізує спектральні «втрати» та артефакти.

Після проведення FFT модуль обчислює магнітуду кожної частотної компоненти та переводить значення в децибели. Отримані дані, що являють собою спектральну інформацію, зберігаються у списку кадрів spectrogramData, а

також фіксуються відповідні часові відмітки. Завдяки синхронізації доступу до цих списків забезпечується коректність накопичення даних навіть при паралельній обробці.

Крім основної обробки в режимі реального часу, клас SpectrogramProcessor пропонує механізм попереднього аналізу всього аудіофайлу через метод preAnalyzeFile. Цей підхід дозволяє обробити великий обсяг даних у пакетному режимі із заданим степенем перекриття для досягнення високої точності спектрального аналізу. Дані по 100 кадрів з пакету періодично відправляються слухачам через інтерфейс SpectrogramListener з метою оновлення графічних компонентів додатку.

Також модуль містить функцію очищення накопичених даних (clearData), що запобігає перевантаженню пам’яті та дозволяє проводити оновлення аналізу при завантаженні нового аудіофайлу. Останній крок роботи модуля – виклик методу processingFinished, який через JavaFX-потік оновлює інтерфейс користувача, передаючи накопичені спектральні дані та часові відмітки слухачам.

Таким чином, модуль спектрограми забезпечує повну обробку аудіосигналу від його розбиття і вкладення в FFT до конвертації у масштаб децибелів. Це дозволяє отримати детальну інформацію про енергетичний розподіл сигналу по частотних смугах, що є надзвичайно корисним для подальшої аналізу та візуалізації динаміки звуку в режимі реального часу.

Клас FrequencyAnalyzer спирається на алгоритм FFT для розкладу вхідного аудіосигналу на складові частоти. На початковій стадії, клас ініціалізує масиви для зберігання амплітудних значень, робочий буфер та віконну функцію Геммінга.

Подальший процес включає визначення смуг частот за допомогою фіксованих контрольних точок. Для кожної смуги обчислюються нижні та верхні межі та відповідні номери бінів FFT, що враховують частотне розрізнення, отримане як відношення частоти дискретизації до розміру буфера. Ці межі дозволяють точно ізолювати діапазони, у яких буде проводитись аналіз.

Всередині методу process аудіо блок спочатку копіюється у робочий буфер із застосуванням віконної функції. Далі здійснюється FFT, після чого обчислюються модулі комплексних чисел, що представляють амплітуду кожної частоти. Для підвищення точності кожної смуги сигнал аналізується за допомогою простих статистичних операцій: розраховується максимальне та середнє значення амплітуди в межах кожного бінового інтервалу, після чого ці значення комбінуються. До отриманого значення застосовується частотне зважування, яке дозволяє коректно відобразити внесок різних смуг, оскільки звукова енергія природно розподіляється не рівномірно по спектру.

Для плавності переходів між значеннями використовуються коефіцієнти атаки ATTACK_FACTOR та релізу RELEASE_FACTOR, що дозволяють згладити швидкі зміни рівня сигналу, забезпечуючи більш стабільне відображення. Результуючі значення нормалізуються до стандартного діапазону [0, 1] і оновлюються у внутрішньому масиві смуг. За умов валідності даних викликається метод зворотного зв’язку через інтерфейс EqualizerListener, який передає копію масиву рівнів для подальшого використання іншими компонентами додатку (наприклад, для візуалізації або налаштування еквалайзера).

Таким чином, модуль спектрального аналізу забезпечує точну і швидку оцінку енергетичного розподілу аудіосигналу по частотних смугах, що є надзвичайно важливим для подальших операцій як візуалізації, так і аудіообробки. Реалізація класу FrequencyAnalyzer дозволяє інтегрувати ці дані у різноманітні графічні компоненти, забезпечуючи інтуїтивне і детальне представлення характеру звуку в режимі реального часу.

Модуль реалізований у класі MusicAnalyzer при виклику методу analyzeFile(String path) створює та налаштовує AudioDispatcher з фіксованим розміром буфера та перекриттям. Одночасно встановлюється обробник ComplexOnsetDetector, що під час проходження кожного аудіоблоку виявляє моменти початку ударів і зберігає їхні часові мітки у внутрішній колекції. Усі операції виконуються в окремому фоновому потоці, що гарантує неперервну

роботу інтерфейсу без затримок. Після завершення зчитування звукового потоку зібрані мітки аналізуються: між кожними двома послідовними онсетами обчислюються інтервали, які потім фільтруються відповідно до заданих меж допустимого темпу. Отримані коректні інтервали агрегуються, і діленням 60-ти на середній інтервал у секундах визначається значення BPM. Остаточний результат передачі темпу здійснюється у JavaFX-потоці через виклик методів інтерфейсу слухача, що дозволяє одразу оновити відображення значення ритму в інтерфейсі користувача. Модуль також забезпечує коректне завершення аналізу та скидання внутрішніх структур даних для подальших запусків, що дозволяє ефективно працювати з будь-яким числом файлів без витоку ресурсів.

Клас WaveformProcessor є невід'ємною частиною аудіоплеєра, оскільки забезпечує попередню обробку аудіофайлів для отримання графічних даних, які згодом використовуються для відображення хвильової форми.

Процес роботи модуля починається з прийняття шляху до аудіофайлу та заданої кількості вибіркових значень (samples), що визначають роздільну здатність отриманої хвильової форми. Далі відбувається зчитування всього аудіоданого потоку у вигляді байтів. Визначаються параметри аудіофайлу, як формат, розрядність вибірки, кількість каналів, та зберігаються дані у тимчасовому буфері. Після завершення зчитування потік закривається для звільнення ресурсів.

Наступним етапом є розподіл отриманих даних на фрейми, що відповідають обраному числовому значенню samples. Для кожного блоку фреймів обчислюється максимальне абсолютне значення амплітуди серед усіх каналів.

Отриманий результуючий масив значень амплітуд, що нормалізовано до позитивного діапазону, повертається як фінальний результат. Ці дані можуть використовуватися для побудови графічної візуалізації форми хвилі на дисплеї аудіоплеєра.

Таким чином, за допомогою цих класів відбувається аналіз звуку у розроблюваному плеєрі.

3.4 Розробка модулів керування та візуалізації

Класу AudioAPlayer відповідає за комплексну обробку аудіоданих – від завантаження та конвертації файлів до потокового відтворення, аналізу та візуалізації сигналу.

При завантаженні аудіофайлу модуль спочатку забезпечує його сумісність із системою, використовуючи окрему підсистему конвертації, модуль AudioConverter, що перетворює вхідний файл у WAV-формат. Головним завданням далі є ініціалізація бібліотеки TarsosDSP через створення об’єкта AudioDispatcher, який відповідає за сегментацію аудіопотоку на блоки даних для реального часу.

Після цього цей клас організовує низку аудіопроцесорів для виконання спеціалізованих завдань, а саме: EqualizerProcessor, FrequencyAnalyzer, SpectrogramProcessor.

Також реалізовує підтримку управління відтворенням: функції play, pause, stop та seek забезпечують можливість точного перемикання між станами аудіоплеєра. Крім того, вбудований механізм роботи з закладками дозволяє додавати, видаляти та переміщатися між ними, що є зручним інструментом для аналізу та навігації по композиції.

Таким чином, модуль AudioAPlayer виступає універсальним ядром системи, яке не лише завантажує та відтворює аудіофайли, а й забезпечує розширений аналіз, адаптивну обробку даних.

Клас MainController відповідальний за ініціалізацію усіх основних компонентів додатку, таких як плеєр, візуалізатор хвильової форми, індикатор гучності, спектральний аналізатор через SAView, спектрограму SpectrogramView та еквалайзер. Модуль використовує можливості JavaFX для оновлення інтерфейсу у режимі реального часу, забезпечуючи інтуїтивне та динамічне відображення даних, що надходять від аудіоджерела.

Під час старту додатку, в методі initialize здійснюється створення екземплярів всіх ключових модулів:

- AudioAPlayer – ядро плеєра що відповідає за керування;

- WaveformView – компонент для візуалізації хвильової форми;
- VolumeIndicatorView – компонент для візуалізації рівня гучності;
- SpectrogramView – для відображення спектрограми аудіосигналу;
- SAView – компонент для візуалізації спектральних рівнів;
- MusicAnalyzer – для аналізуу темпу треків;
- EqualizerWindow - для відображення вікна еквалайзера.

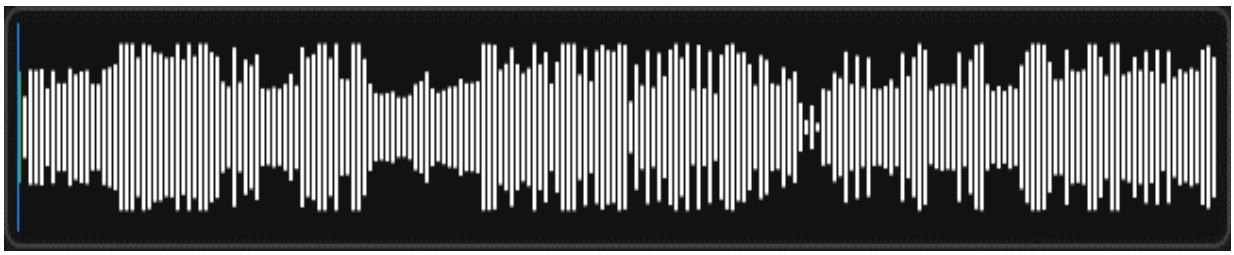
Крім того, MainController налаштовує інтерактивність інтерфейсу: кнопки відтворення, паузи, зупинки, відкриття файлів, а також елементи керування гучністю, переходи по закладках, циклічні відтворення та активацію еквалайзера.

Важливою особливістю класу є інтеграція численних слухачів, які відповідають за обробку вхідних аудіоданих. Завдяки цим механізмам, усі компоненти користувацького інтерфейсу постійно синхронізуються із поточним станом відтворення і аналізу.

Таким чином, клас MainController виступає центральним модулем, який інтегрує аудіопроцеси, графічну візуалізацію та взаємодію з користувачем, забезпечуючи комплексний, адаптивний та зручний інтерфейс для роботи з аудіо, що поєднує відтворення, аналіз і розширену візуалізацію звукових характеристик у реальному часі.

Клас WaveformView побудовано на базі JavaFX із використанням об'єкта Canvas для малювання, при цьому використовується GraphicsContext для виконання низки графічних операцій. Модуль динамічно адаптує свій розмір, реагуючи на зміни розмірів контейнера, що забезпечує гнучке розташування елементів навіть при зміні конфігурації інтерфейсу. Виконує такі основні функції: рендеринг форми хвилі, відображення прогресу відтворення, інтеракстивність, відображення закладок. На рисунку 3.3 зображено результат роботи візуалізатора хвильової форми.

Таким чином, WaveformView перетворює числові дані аудіозапису у зручне графічне представлення, що інтегрується із загальним інтерфейсом аудіоплеєра.



Риснунок 3.3 – Візуалізація хвильової форми за допомогою класу WaveformView

VolumeIndicatorView є важливим компонентом аудіоплеєра, оскільки забезпечує візуальне відображення поточної гучності аудіо сигналу для окремих каналів, що дозволяє користувачу оперативно контролювати аудіовихід.

Завдяки методові updateChannelVolumes, модуль отримує актуальні значення гучності для лівого та правого каналів, що зберігаються як цільові значення. Дані значення використовуються для анімаційного згладжування, що дозволяє уникнути раптових змін у відображенні.

Модуль використовує клас AnimationTimer для регулярного оновлення інтерфейсу. У кожному кадрі анімації застосовується smoothing-factor, що забезпечує поступове наближення відображуваних значень до цільових, створюючи ефект плавного переходу між рівнями.

За допомогою об'єкта Canvas і GraphicsContext модуль малює два окремих індикатори для кожного аудіоканалу. Лівий та правий канали відображаються як прямокутники, висота яких пропорційна поточному рівню гучності. Межі, розміри та позиціонування індикаторів розраховуються динамічно залежно від розміру контейнера, що робить модуль адаптивним до різних роздільних здатностей екрану.

Завдяки цим можливостям, VolumeIndicatorView надає інтуїтивно зрозумілий та динамічний зоровий зворотний зв’язок про рівень звуку, що сприяє кращому контролю за параметрами відтворення у реальному часі. Його роботу зображено на рисунку 3.4.

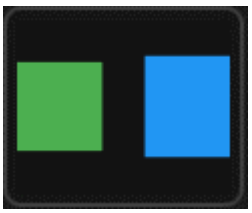


Рисунок 3.4 – Візуалізація рівнів звуку каналів за допомогою класу VolumeIndicatorView

SpectrogramView забезпечує обробку і графічне відображення спектральних даних аудіосигналу, що генеруються в режимі реального часу або в процесі попереднього аналізу аудіофайлу. Основна ідея полягає у відображенні спектрограми у вигляді матриці пікселів, де горизонтальна вісь відповідає часу, а вертикальна – частотному спектру. Візуалізація спектрограмиза допомогою цього класу зображена на рисунку 3.5.

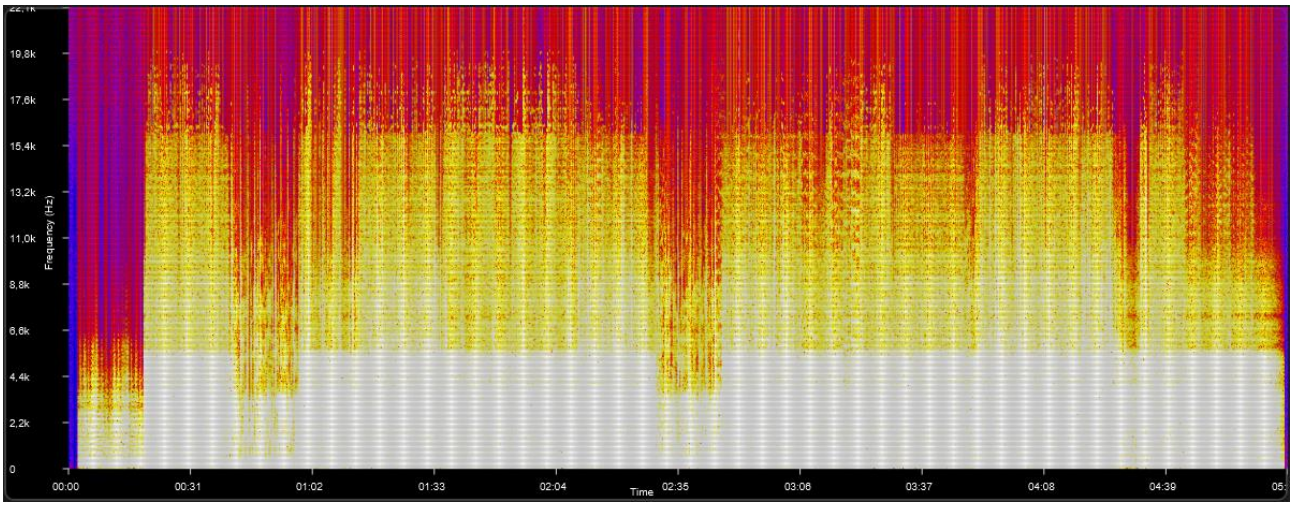


Рисунок 3.5 – Візуалізація спектрограми за допомогою класу SpectrogramView

За допомогою методу setSpectrogramData отримані масиви магнітуд, часові відмітки, а також параметри аудіо, такі як частота дискретизації і розмір FFT, записуються у відповідні змінні. Це дозволяє модулю отримувати актуальні дані для побудови спектрограми.

Для оптимізації продуктивності використовується метод drawSpectrogramOptimized, який зменшує кількість пікселів шляхом пропуску деяких кадрів. Значення магнітуди для кожного пікселя нормалізуються до

діапазону [0,1] з використанням заданих мінімальних і максимальних значень у децибелах. Далі, за допомогою функції getSpectrogramColor, нормалізована інтенсивність перетворюється у відповідний колір із градацією від холодни до теплих тонів.

Завдяки використанню потокових викликів через Platform.runLater, SpectrogramView забезпечує коректне оновлення інтерфейсу у JavaFX-поточці, що гарантує плавну анімацію та високу продуктивність навіть при роботі з великими обсягами даних. Таким чином, модуль SpectrogramView є потужним інструментом для візуального аналізу аудіосигналу, що дозволяє отримати детальне уявлення про розподіл енергії по частотних смугах у режимі реального часу.

SAView дозволяє користувачу спостерігати за динамічними змінами рівня частотних смуг у режимі реального часу. Візуалізацію реалізовану цим класом зображено на рисунку 3.6.

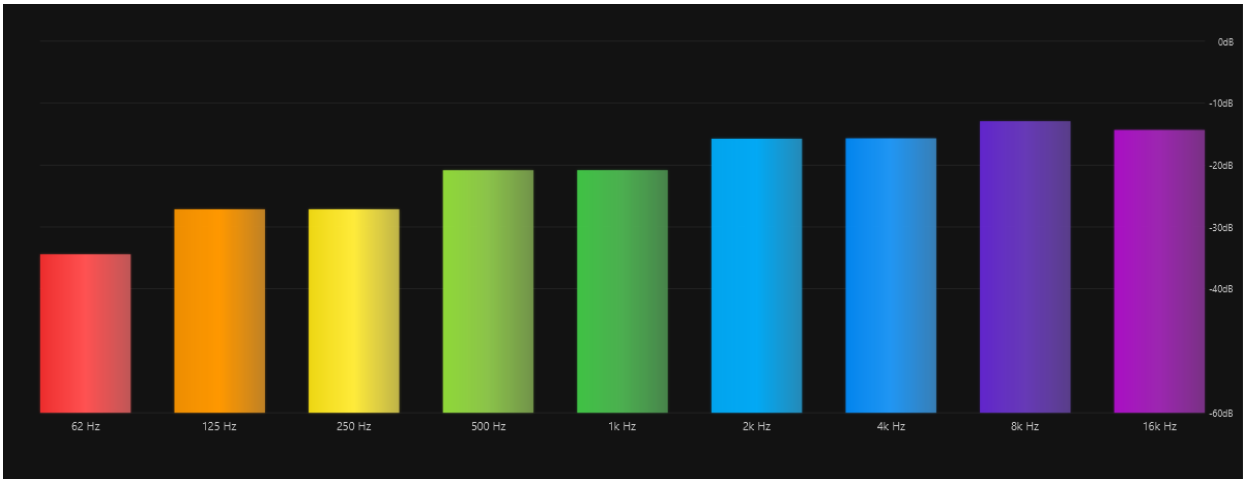


Рисунок 3.6 – Візуалізація спектрального аналізу за допомогою класу SAview

Компонент наслідується від Pane та використовує Canvas з GraphicsContext для малювання. Розміри полотна прив’язуються до розмірів контейнера, що дозволяє SAView автоматично підлаштовуватись під зміни в інтерфейсі.

Модуль розбиває спектральний сигнал на дев’ять частотних смуг, які відповідають контрольним точкам від 62 Hz до 16 kHz. Для кожної смуги

відображається вертикальний стовпчик із використанням кольорового градієнта, що формується за допомогою методу createGradient. Градієнти кешуються для покращення продуктивності при повторному малюванні.

На фоні відображення малюється сітка з позначеннями рівнів у децибелах. За допомогою функцій перетворення амплітуди в децибели і нормалізації до діапазону(MIN_DB – MAX_DB побудовано лінії сітки, а також нанесено текстові мітки з відповідними значеннями.

Використовуючи AnimationTimer, SAView періодично оновлює значення поточних рівнів частот, масив currentLevels, на основі отриманих з аналізатора значень, застосовуючи фактор згладжування SMOOTHING_FACTOR. У випадках, коли аудіо не відтворюється або даних немає, модуль поступово зменшує значення до нуля, що забезпечує природне згасання візуального відображення.

За допомогою внутрішнього прапорця isActivelyPlaying модуль розрізняє активний стан від стану «тиші». При відсутності активного аудіосигналу рівні смуг природним чином поступово знижуються до нуля.

SAView інтегрує JavaFX для створення зручного та інформативного інтерфейсу спектрального аналізу. Завдяки адаптивному малюванню, оптимізації через кешування градієнтів і плавним анімаціям, він забезпечує високоякісне візуальне представлення рівня звуку за частотами.

EqualizerWindow призначено для інтерактивного керування частотними смугами аудіосигналу, забезпечуючи користувачеві можливість тонко налаштовувати параметри звуку за допомогою графічного інтерфейсу.

Вікно еквалайзера створено без застосування FXML. За допомогою класів JavaFX, таких як Stage, Scene, VBox, HBox та Button, формується зручний інтерфейс для взаємодії з користувачем.

Для кожної з дев’яти частотних смуг модуль створює окремий вертикальний повзунок, який дозволяє встановлювати значення підсилення в діапазоні від -12 до +12 дБ. Поряд з повзунками розміщені лейбли із зазначенням поточного показника підсилення, який оновлюється в режимі реального часу.

Інтерфейс містить чекбокс для увімкнення або вимкнення еквалайзера, що дозволяє користувачеві швидко перемикатися між різними режимами роботи. Також передбачені кнопки для застосування різних пресетів та кнопка скидання налаштувань до початкових значень.

При зміні значення повзунка, оновлюється відповідний параметр у екземплярі EqualizerProcessor.

EqualizerWindow забезпечує користувача потужним інструментом для тонкого налаштування аудіосигналу. Інтуїтивно зрозумілий інтерфейс з інтерактивними повзунками, пресетами та можливістю увімкнення/вимкнення еквалайзера дозволяє швидко адаптувати звучання під конкретні потреби, що є важливою складовою загальної архітектури аудіоплеєра.

3.5 Висновки

У третьому розділі було обґрунтовано вибір технологій та інструментів, що використовувались для розробки програмних модулів та описано графічний інтерфейс програми. Виконавши аналіз програмного продукту та його задач мовою програмування було обрано Java. Для реалізації графічного інтерфейсу було обрано JavaFX. Для роботи з аудіофайлами було обрано бібліотеку TarsosDSP. Також було описано розробку основних програмних модулів.

4 ТЕСТУВАННЯ ПРОГРАМИ

4.1 Функціональне тестування

Тестування програмного забезпечення — це процес верифікації та валідації відповідності розробленого продукту заданим вимогам і специфікаціям. Це функціональне тестування проводитиметься вручну шляхом виконання заздалегідь підготовлених тест-кейсів, що охоплюють усі ключові функції аудіоплеєра: від базових команд відтворення до складних сценаріїв роботи еквалайзера, спектрального аналізу та закладок. Мета тестування полягає у виявленні помилок у логіці роботи, некоректної обробки за різних умов та перевірці відповідності результатів обробки аудіо заявленим критеріям точності та продуктивності [22].

Тест-кейс №1. Завантаження аудіофайлу.

Перед будь-якими наступними маніпуляціями варто засвідчитись що завантаження аудіофайлу працює коректно. Передумовою цього тест-кейсу є аудіофайл із відомими параметрами.

Кроки:

1. Натиснути «Open».
2. Обрати в провіднику заготовлений аудіофайл.
3. Натиснути «Spectrogram».

Очікуваним результатом такої перевірки буде побудова спектрограми, поява хвильової форми в контейнері, поява правильної тривалості пісні, а також відображення BPM. Як видно з рисунку 4.1 отримані результати збігаються з очікуваним.

Тест-кейс №2. Відтворення аудіо.

Перед тим, як тестувати більш складні функції або сценарії необхідно впевнитись, що програма здатна правильно відтворювати аудіо.

Кроки:

1. Натиснути «Open».
2. Обрати в провіднику заготовлений аудіофайл.

3. Натиснути «Play».

Рисунок 4.1 – Перевірка завантаження аудіофайлу

Очікуваним результатом є звичайне програвання завантаженої музики. На рисунку 4.2 із зеленої частини хвильової форми видно, що отриманий результат відповідає очікуваному.

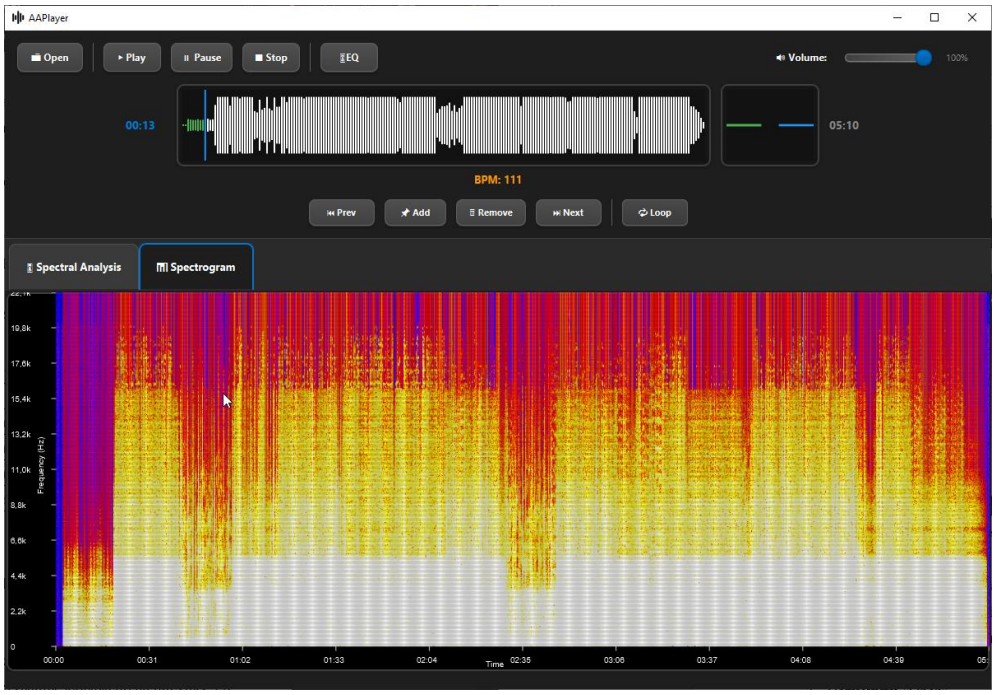


Рисунок 4.2 – Перевірка відтворення аудіо

Тест-кейс №3. Пауза, відновлення відтворення та стоп.

В цьому сценарії необхідно перевірити правильність роботи основних кнопок керування.

Кроки:

1. Натиснути «Open».
2. Обрати в провіднику заготовлений аудіофайл.
3. Натиснути «Play».
4. Зачекати, після чого натиснути «Pause».
5. Зачекати та натиснути «Play».
6. Натиснути «Stop».

Отриманий результат відповідає очікуваному. Отже, основні кнопки керування працюють без проблем.

Тест-кейс №4. Перемотування вперед та назад

Перевірка системи перемотування.

Кроки:

1. Натиснути «Open».
2. Обрати аудіофайл.
3. Натиснути «Play»
4. Перемістити слайдер позиції на 30 секунд.

Отриманий результат відповідає очікуваному. Отже, система перемотування працює коректно.

Тест-кейс №5. Створення, видалення закладок, переміщення між ними та зациклення.

Слід перевірити функціонал навігації закладок, а саме чи в правильному місці вони створюються, чи здійснюється перехід між ними, чи не виникає проблем для видалення закладок та чи працює зациклення коректно.

Передумови: аудіо програється.

Кроки:

1. Натиснути «Add» на 00:15.
2. Натиснути «Add» на 00:30.

3. Натиснути «Add» на 00:45.
4. Тричі з паузою натиснути «Prev».
5. Натиснути «Next».
6. Натиснути кнопку «Loop»
7. Після досягнення настопної закладки натиснути «Remove».

Очікуваний результат передбачає, що після трьох доданих закладок буде здійснено перевірку переходу між трьома такими точками в різні сторони. Далі перевірка зацикленості, де закладки можна порівняти з дужками в математиці, тобто програвання не має виходити за межі. Останнім було перевірено можливість видалити закладку. Отриманий результат відповідає очікуваному зображений на рисунку 4.3.

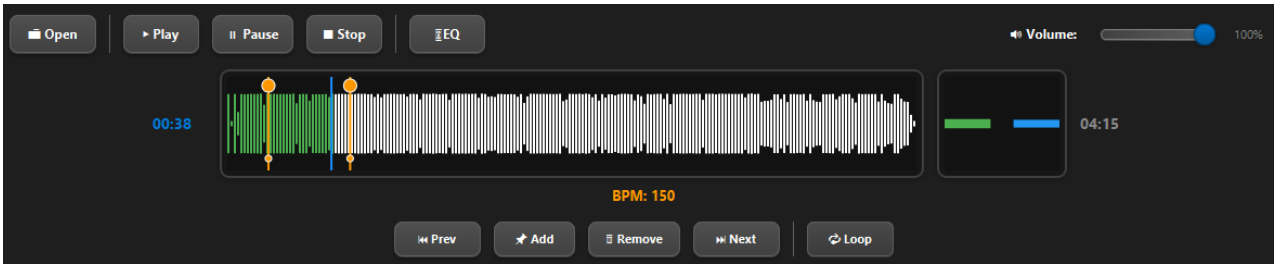


Рисунок 4.3 – Тестування функціональності закладок

Тест-кейс №6. Частотний аналіз в реальному часі.

Тестування спекрального аналізу за працездатність. Цей елемент є дуже важливим оскільки дає багато важливої інформації про трек в реальному часі.

Передумови: файл завантажено коректно.

Кроки:

1. Обрати активною панель «Spectral Analysis».
2. Натиснути кнопку «Play».
3. Натиснути кнопку «Pause».
4. Натиснути кнопку «Play».
5. Натиснути кнопку «Stop».

Після натискання «Play» на панелі мали з’явитись рівні різних частот. Після натискання паузи ці рівні мали повільно впасти. Таким же чином мали

поводитись після зупинення. Отриманий результат відповідає очікуваному і зображений на рисунку 4.4.

Рисунок 4.4 – Тестування спектрального аналізу

Тест-кейс №7. Еквалайзер

Передумови: аудіо відтворюється, відкрито вікно еквалайзера.

Кроки:

1. Вибрати пресет «Rock».
2. Натиснути «Reset».
3. Виставити повзунки 62 Гц та 125 Гц на максимум

При виборі пресету звучання має змінюватись, а після кнопки скидання повинно повертатись оригінальне. Після виставлення 62 та 125 Гц мають дуже добре виділятись барабани на бас, якщо вони є в пісні. Також зміну рівня звуку можна побачити на індикаторі звуку та спектрального аналізатора. Отриманий результат відповідає очікуваному і зображений на риснку 4.5.

Рисунок 4.5 – Тестування еквалайзера

Тест-кейс №8. Конвертація форматів

Багато функцій плеєра потребують формату WAV для коректної уніфікованої роботи, тому варто протестувати поведінку програми з різними форматами.

Кроки:

1. Відкрити файл test.wav.
2. Перевірити вміст папки для тимчасових файлів.
3. Відкрити файл test.mp3.
4. Перевірити вміст папки для тимчасових файлів.

Після перевірки папки тимачових файлів після першого відкриття файлу не було нічого знайдено. Але після завантаження файлу з розширенням .mp3 у папці з тимчасовими файлами з’явився .wav файл, а це означає що отриманий результат відповідає очікуваному. На рисунку 4.6 зображено папку з тимчасовим конвертованим файлом.

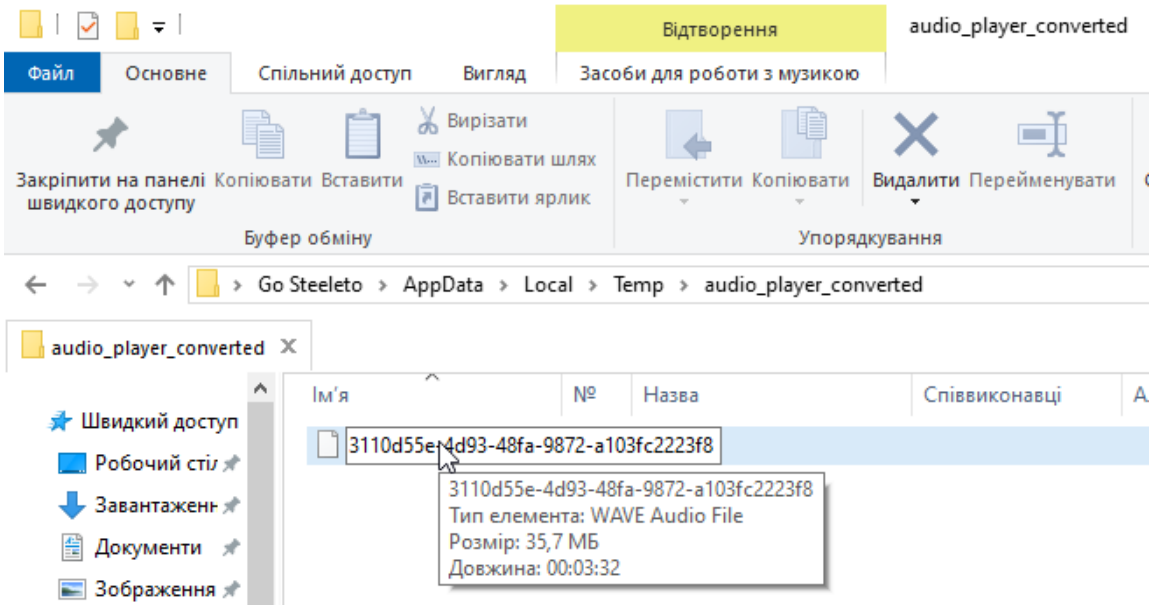


Рисунок 4.6 – Тимчасовий конвертований файл

Тест-кейс №9. Аналіз темпу.

Необхідно перевірити точність та надійність функції аналізу темпу.

Передумови: композиція з чітким темпом 140 BPM.

Кроки:

1. Завантажити композицію з чітким темпом 140 BPM.
2. Дочекатись результату.

Очікуваним результатом є значення BPM в межах 138-142, така точністю є допустимою. Отриманий результат відповідає очікуваному, як видно з рисунка 4.7.

Рисунок 4.7 – Результат тестування аналізу темпу

Таким чином було виконано функціональне тестування, результати свідчать про стабульну роботу програми.

4.2 Розробка інструкції користувача

Інструкція користувача передбачає визначення технічних вимог для запуску програмного продукту. Деталі щодо рекомендованої конфігурації розміщено в таблиці 4.1.

Таблиця 4.1 – Вимоги до апаратного забезпечення

Вимоги до конфігурування	Рекомендовано
Процесор	Intel Core i7
Оперативна пам'ять	4 ГБ RAM
Вільний простір на диску	300 МБ
Відеокарта	NVIDIA GeForce GT 1030 або аналогічна
Операційна система	Windows 10
Монітор	Роздільна здатність не менше 1920x1080 пікселів

Перед початком роботи слід переконатися в коректності установки Java Runtime Environment версії не нижче 11. Далі необхідно розпакувати архів із дистрибутивом у довільну папку та запустити файл AudioAPlayer.jar подвійним клацанням або через командний рядок командою java -jar AudioAPlayer.jar. Після успішного завантаження головного вікна програми користувач бачить простий інтерфейс із панеллю керування, де розташовані кнопки відтворення, паузи, стоп і повзунок позиції треку, а також індикатори гучності для лівого й правого каналу.

Щоб завантажити аудіофайл, потрібно натиснути на кнопку "Open" та в діалоговому вікні вибрати потрібний файл формату WAV чи MP3 (див. рисунок 4.8). Після вибору загальна тривалість композиції відобразяться у заголовку вікна. Для початку відтворення натискають кнопку "Play". Під час відтворення користувач може переміщувати повзунок прогресу, клікаючи по будь-якій точці шкали або перетягуючи маркер, що дозволяє точно перейти до бажаного

моменту (див. рисунок 4.9). Якщо необхідно припинити відтворення на певній позиції без скидання, достатньо натиснути "Pause", а відновити — натиснути "Play" знову. Завершити відтворення та повернутися на початок треку можна за допомогою кнопки "Stop".

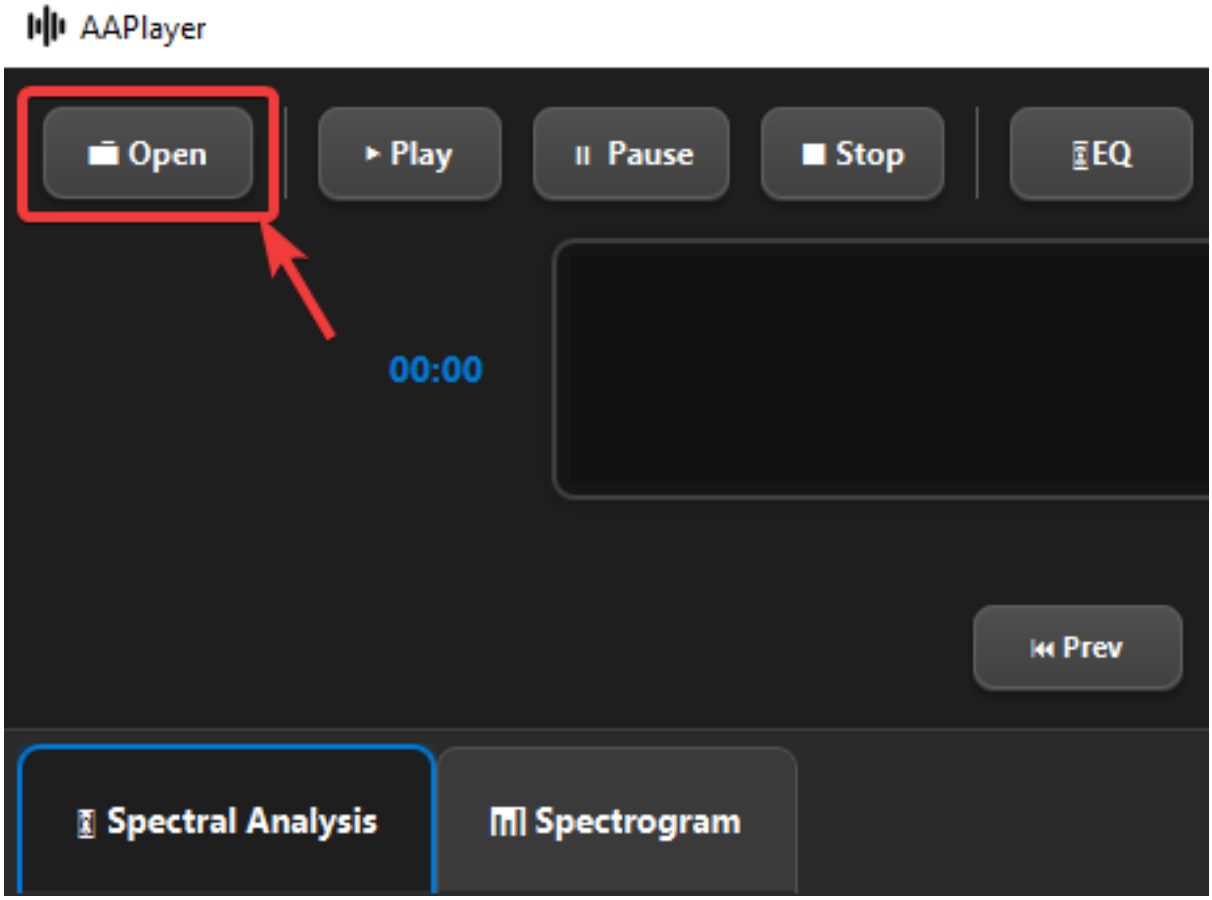


Рисунок 4.8 – Кнопка завантаження файлу до плеєра

Для налаштування еквалайзера слід відкрити окреме вікно, натиснувши на іконку "EQ" (див. рисунок 4.10). У вікні представлені повзунки для дев’яти частотних смуг і пункт вибору пресетів, серед яких є заготовлені налаштування "Rock", "Pop" та інші. Пересуваючи повзунки вгору чи вниз, користувач змінює рівень підсилення в обраній смузі. Після завершення регулювання не потрібно нічого натискати, щоб еквалайзер почав обробляти сигнал у режимі реального часу. Якщо користувач хоче повернутися до заводських налаштувань, передбачена кнопка "Reset".

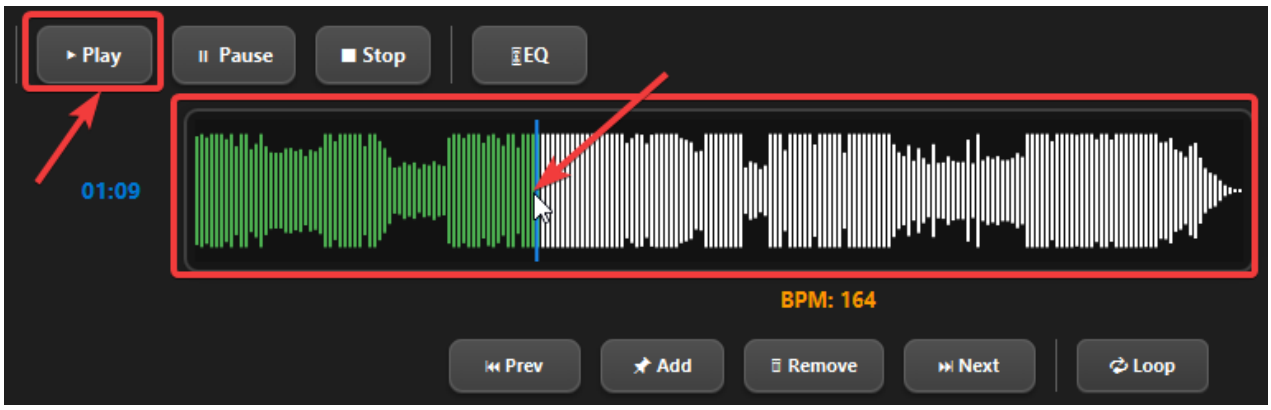


Рисунок 4.9 – Переміщення повзунка прогресу після початку відтворення

Робота з закладками здійснюється через відповідні елементи інтерфейсу на панелі керування. Щоб додати закладку, слід натиснути кнопку з піктограмою закладки у момент відтворення, який потрібно позначити як початок фрагмента. Повторне натискання у потрібній точці встановлює другу мітку. Створений інтервал відображається оранжевим маркером над прогрес-баром. Для активації циклічного відтворення закладеного фрагмента необхідно натиснути кнопку лупи. Вимкнення режиму лупу відбувається тим же натисканням.

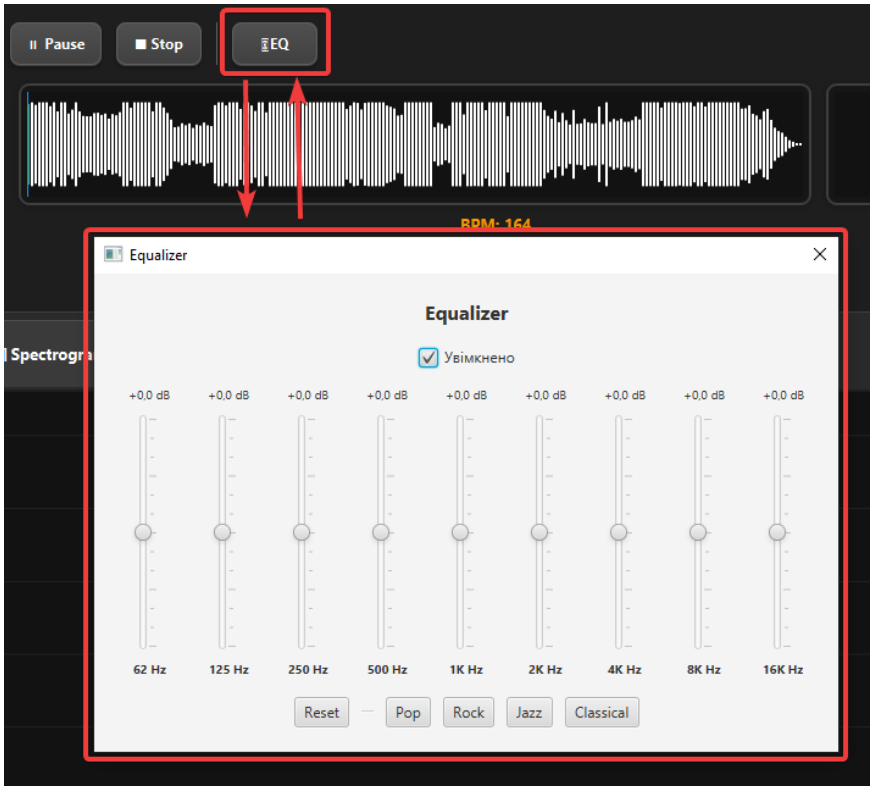


Рисунок 4.10 – Кнопка відкриття вікна еквалайзера

Аналітичні модулі працюють прозоро для користувача, але результати їхньої роботи можна спостерігати на спеціальних вкладках. У вкладці "Spectrogram" представлено кольорову карту частотних компонент, де по горизонтальній осі відкладено час, а по вертикальній – частоту у герцах.

4.3 Висновки

У четвертому розділі було проведено функціональне тестування програмного забезпечення аудіоплеєра з розширеними можливостями аналізу звуку. Було перевірено реакцію програмних засобів на помилкові ситуації, вхідні дані різних форматів. Отримані результати засвідчують стабільну роботу системи. Також було розроблено інструкцію користувача по початковій експлуатації програмного продукту.

ВИСНОВКИ

В ході бакалаврської кваліфікаційної роботи було розроблено програмний засіб аудіоплеєра з розширеними можливостями аналізу звуку. Для розробки використовувалося середовище програмування IntelliJ IDEA.

В першому розділі було проведено аналіз сучасного стану проблеми та значення розширеного аналізу звуку в сучасних аудіоплеєрах. Були розглянуті основні аналоги програмного продукту, такі як MusicBee, VLC, foobar2000, виявлені їхні недоліки та порівняно з власним розробленим продуктом. Також були проаналізовані методи вирішення задачі. На основі цього були сформульовані основні завдання дослідження.

В другому розділі було виконано аналіз вхідних та вихідних даних. Розроблено модульну архітектуру програми. Удосконалено метод визначення темпу з обробкою часових характеристик сигналу та корекцією вихідних значень для досягнення кращої точності і представлено його блок-схему. Розроблено алгоритми аналізу звуку, а саме відображення рівня звуку за частотами та генерації хвильової форми, та подано їх у вигляді блок-схем.

В третьому розділі під час аналізу технологій розробки було обґрунтовано вибір мови програмування Java, а також бібліотек JavaFX та TarsosDSP. Розроблено інтерфейс програми і модулі аналізу звуку. Після чого розроблено модулі керування та візуалізації.

В четвертому розділі було проведено функціональне тестування, для якого було сформовано теск-кейси. Тестування програми довело повну працездатність даного програмного продукту та відповідність поставленому технічному завданню. Також було розроблено інструкцію користувача.

Таким чином, було виконано всі поставлені задачі розробки програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Романюк О.В., Грінін А.В. Значення розширеного аналізу звуку в сучасних аудіоплеєрах – «LIV Всеукраїнська науково-технічна конференця факультету інформаційних технологій та комп'ютерної інженерії (2025)» Вінниця, 2025. [Електронний ресурс] – Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2025/paper/view/24138/19876 / (дата зверення: 18.05.2025).

2. Roads C. The Computer Music Tutorial. MIT Press, 2023. 1288 p.

3. Li F.F., Cox T.J. Digital Signal Processing in Audio and Acoustical Engineering. Boca Raton, FL: CRC Press, 2023. 242 p.

4. Owsinski B. The Mixing Engineer's Handbook.5th edition Boston : Cengage Learning, 2022. 352 p.

5. Офіційне завантаження медіапрогравача VLC [Електронний ресурс] – Режим доступу до ресурсу: https://www.videolan.org/vlc/index.uk.html (дата звернення: 18.05.2025)

6. Foobar2000 [Електронний ресурс] – Режим доступу до ресурсу: https://www.foobar2000.org/ (дата звернення: 18.05.2025)

7. MusicBee – The Ultimate Music Manager and Player [Електронний ресурс] – Режим доступу до ресурсу: https://getmusicbee.com/ (дата звернення: 18.05.2025).

8. Uncini A. Digital Audio Processing Fundamentals. Cham: Springer, 2022. 716 p.

9. Wang X., Cao X. Advanced Digital Signal Processing: From Theory to Applications. Beijing: Beihang Univ. Press, 2022. 250 p.

10. Zölzer U. Digital Audio Signal Processing. 3rd edition. Wiley, 2022. 416 p.

11. The Ultimate Guide to Audio File Formats: MP3, AAC, WAV, FLAC, and More [Електронний ресурс] – Режим доступу до ресурсу: https://primesound.org/audio-file-formats (дата звернення: 18.05.2025).

12. PCM vs Dolby Digital vs Passthrough – Key Differences [Електронний

ресурс] – Режим доступу до ресурсу: https://hometheateracademy.com/pcm-vs-dolby-digital-vs-passthrough/ (дата звернення: 18.05.2025).

13. Bass L., Clements P., Kazman R. Software Architecture in Practice, 4th edition. Boston : Addison- Wesley, 2022. 592 p.

14. Visio [Електронний ресурс] – Режим доступу до ресурсу: https://www.microsoft.com/uk-ua/microsoft-365/visio/flowchart-software (дата звернення: 18.05.2025).

15. Horstmann C. Core Java. Volume I – Fundamentals. 12th edition. Upper Saddle River, NJ : Pearson, 2022. 1100 p.

16. Price M.J. C# 10 and .NET 6 – Modern Cross- Platform Development, 6th Edition. Birmingham : Packt Publishing, 2021. 1050 p.

17. Introduction to Java Swing [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/introduction-to-java-swing/ (дата звернення: 22.05.2025).

18. OpenGL – The Industry's Foundation for High Performance Graphics [Електронний ресурс] – Режим доступу до ресурсу: https://www.opengl.org/archives/documentation/implementations/ (дата звернення: 22.05.2025).

19. JavaFX [Електронний ресурс] – Режим доступу до ресурсу: https://openjfx.io/ (дата звернення: 22.05.2025).

20. Java Sound API [Електронний ресурс] – Режим доступу до ресурсу: https://www.oracle.com/java/technologies/java-sound-api.html (дата звернення: 22.05.2025).

21. GitHub – JorenSin/TarsosDSP: A Real-Time Audio Processing Framework in Java [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/JorenSix/TarsosDSP (дата звернення: 22.05.2025).

22. Functional Testing : Definition, Types & Examples [Електронний ресурс] – Режим доступу до ресурсу: https://testsigma.com/guides/functional-testing/ (дата звернення: 22.05.2025).

ДОДАТКИ

Додаток А. Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
д.т.н., професор
_____ О.Н. Романюк
« 25 » березня 2025 року

Технічне завдання

на бакалаврську кваліфікаційну роботу «Розробка програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку» за спеціальністю

121 Інженерія програмного забезпечення

Керівник бакалаврської кваліфікаційної роботи:
к.т.н., доц. каф. О.В. Романюк
« 24 » березня 2025 року

Виконав:
студент гр.2ПІ-21б А.В. Грінін
« 24 » березня 2025 року

1. Найменування та галузь застосування

Бакалаврська кваліфікаційна робота: «Розробка програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку».

Галузь застосування – мультимедіа.

2. Підстава для розробки.

Підставою для виконання бакалаврської кваліфікаційної роботи (БКР) є індивідуальне завдання на БКР та наказ №97 від 20 березня 2025 р. ректора по ВНТУ про закріплення тем БКР.

3. Мета та призначення розробки.

Метою роботи є підвищення ефективності аудіоплеєра за рахунок розширення його функціональних можливостей, що дозволить здійснювати розширений аналіз звуку.

Призначення роботи – методи та засоби обробки та візуалізації аудіосигналів.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись БКР:

1. Roads C. The Computer Music Tutorial. MIT Press, 2023. 1288 p.

2. Li F.F., Cox T.J. Digital Signal Processing in Audio and Acoustical Engineering. Boca Raton, FL: CRC Press, 2023. 242 p.

3. Uncini A. Digital Audio Processing Fundamentals. Cham: Springer, 2022. 716 p.

5. Технічні вимоги

Тип програми – Java-застосунок; модель розробки – ітеративно-інкрементна; вхідні дані – аудіофайл; вихідні дані – PCM-потік на аудіовихід, масиви спектральних даних, значення BPM, закладки; мова програмування –

Java; середовище розробки – IntelliJ IDEA, бібліотеки JavaFX, TarsosDSP; дані для аналізу звукових параметрів – спектральні характеристики аудіосигналу, значення амплітуди для лівого і правого каналів, спектрограма, хвильова форма, фільтрований аудіосигнал.

6. Конструктивні вимоги

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

- пояснювальна записка до БКР;
- технічне завдання;
- лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

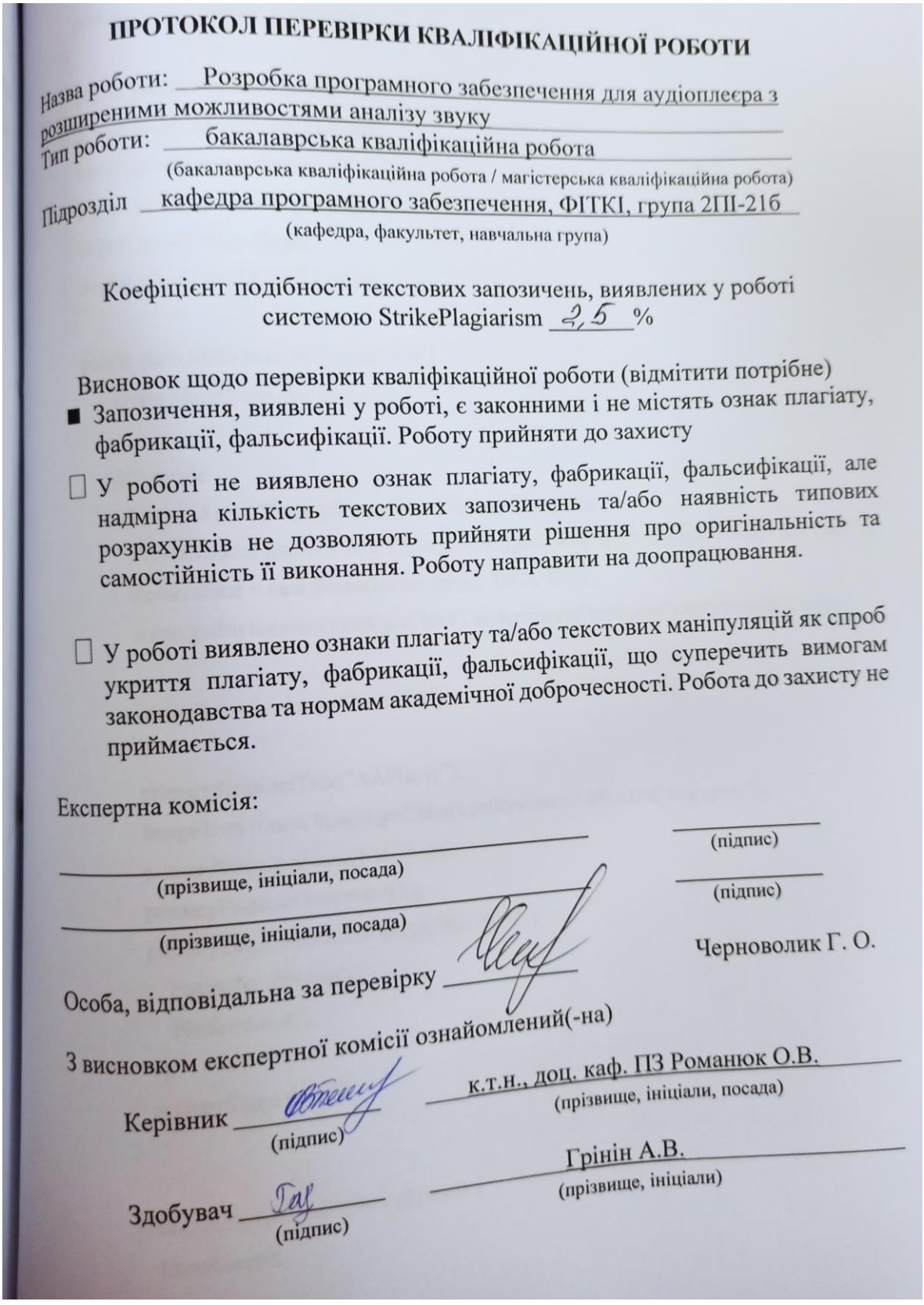
9. Стадії та етапи розробки:

№ з/п	Назва етапів бакалаврської кваліфікаційної роботи	Строк виконання етапів роботи
1	Аналіз сучасного стану аудіоплеєрів та аналогічних програмних продуктів	25.03.25 – 31.03.25
2	Розробка архітектури програмного забезпечення	01.04.25 – 07.04.25
3	Розробка методу визначення темпу	08.04.25 – 15.04.25
4	Розробка алгоритмів аналізу звуку	16.04.25 – 23.04.25
5	Аналіз та вибір засобів для реалізації програмного продуктку	24.04.25 – 30.04.25
6	Розробка програмного забезпечення	01.05.25 – 07.05.25
7	Тестування програмного застосунку	07.05.25 – 14.05.25
8	Оформлення матеріалів до захисту БКР	15.05.25 – 30.05.25

10. Порядок контролю та прийняття.

Виконання етапів бакалаврської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Захист бакалаврської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

Додаток Б. Протокол перевірки кваліфікаційної роботи на наявність текстових запозичень



ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Назва роботи: Розробка програмного забезпечення для аудіоплеєра з розширеними можливостями аналізу звуку

Тип роботи: бакалаврська кваліфікаційна робота

(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ кафедра програмного забезпечення, ФІТКІ, група 2ПІ-21б

(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism 2,5 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

■ Запозичення, виявлені у роботі, є законними і не містять ознак плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки плагіату та/або текстових маніпуляцій як спроб укриття плагіату, фабрикації, фальсифікації, що суперечить вимогам законодавства та нормам академічної доброчесності. Робота до захисту не приймається.

Експертна комісія:

(прізвище, ініціали, посада) (підпис)

(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку Черноволик Г. О.

З висновком експертної комісії ознайомлений(-на)

Керівник (підпис) к.т.н., доц. каф. ПЗ Романюк О.В. (прізвище, ініціали, посада)

Здобувач (підпис) Грінін А.В. (прізвище, ініціали)

Додаток В. Лістинг програми

```
package com.example.audioaplayer;

import javafx.application.Application;
import javafx.application.Platform;
import javafx.scene.Scene;
import javafx.scene.image.Image;
import javafx.stage.Stage;
import javafx.fxml.FXMLLoader;

public class Main extends Application {
    private MainController controller;

    @Override
    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new FXMLLoader(getClass().getResource("MainView.fxml"));
        Scene scene = new Scene(loader.load(), 1200, 800);
        scene.getStylesheets().add(getClass().getResource("main.css").toExternalForm());

        controller = loader.getController();

        primaryStage.setTitle("AAPlayer");
        Image icon = new Image(getClass().getResourceAsStream("logo.png"));
        primaryStage.getIcons().add(icon);
        primaryStage.setScene(scene);
        primaryStage.setOnCloseRequest(event -> {
            controller.cleanup();
            Platform.exit();
        });
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
```

```
    }
}
package com.example.audioaplayer;

import be.tarsos.dsp.AudioDispatcher;
import be.tarsos.dsp.AudioEvent;
import be.tarsos.dsp.AudioProcessor;
import be.tarsos.dsp.io.jvm.AudioDispatcherFactory;
import be.tarsos.dsp.io.jvm.AudioPlayer;
import be.tarsos.dsp.io.jvm.JVMAudioInputStream;
import javafx.application.Platform;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.property.SimpleDoubleProperty;

import javax.sound.sampled.*;
import java.io.File;
import java.io.IOException;
import java.util.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicReference;

public class AudioAPlayer {

    private int BUFFER_SIZE;
    private static final int OVERLAP = 0;
    private static final int SAMPLE_RATE = 44100;
    private static final int WAVEFORM_RESOLUTION = 512;
    private static final double BOOKMARK_TIME_TOLERANCE = 0.5;
    private static final double SPECTRUM_INTERVAL = 0.03;
    private static final int SPECTRUM_BANDS = 128;
    private VolumeProcessor volumeProcessor;
```

```
    private static final long VOLUME_UPDATE_THRESHOLD = 100;


    private volatile long lastVolumeUpdateTime = 0;

    private AudioDispatcher dispatcher;
    private AudioFormat format;
    private final SimpleDoubleProperty currentTime = new SimpleDoubleProperty(0);
    private final BooleanProperty loopEnabled = new SimpleBooleanProperty(false);
    private double totalDuration = 0;
    private String currentFilePath;
    private String convertedFilePath;
    private float[] waveformData;
    private final float[] audioBuffer = new float[SPECTRUM_BANDS];
    private float leftChannelVolume = 0;
    private float rightChannelVolume = 0;
    private double currentVolume = 1.0;
    private final AtomicBoolean isPlaying = new AtomicBoolean(false);

    private final TreeSet<Bookmark> bookmarks = new TreeSet<>();

    private final List<AudioListener> listeners = new ArrayList<>();
    private final List<WaveformListener> waveformListeners = new ArrayList<>();
    private final List<ChannelVolumeListener> channelVolumeListeners = new
ArrayList<>();
    private final List<BookmarkListener> bookmarkListeners = new ArrayList<>();

    private final List<FrequencyAnalyzer.EqualizerListener> equalizerListeners = new
ArrayList<>();

    private ExecutorService audioThread = Executors.newSingleThreadExecutor();
    private final AtomicReference<Double> seekPosition = new AtomicReference<>(0.0);
    private final AtomicBoolean seekRequested = new AtomicBoolean(false);
```

```
        private Line.Info sourceLineInfo;
        private SourceDataLine line;
        private AudioInputStream audioInputStream;
        private Clip clip;

        private EqualizerProcessor equalizerProcessor;
        private SpectrogramProcessor spectrogramProcessor;
        private final List<SpectrogramProcessor.SpectrogramListener> spectrogramListeners =
new ArrayList<>();

        private double pausedPosition = 0;
        private double playbackOffset = 0;


        private volatile boolean manuallyPaused = false;
        private volatile boolean loopTransitionInProgress = false;


        private FrequencyAnalyzer frequencyAnalyzer;

        public void addSpectrogramListener(SpectrogramProcessor.SpectrogramListener listener)
{
            spectrogramListeners.add(listener);
        }

        public interface AudioListener {
            void onAudioDataAvailable(float[] audioData);
        }

        public interface WaveformListener {
            void onWaveformDataAvailable(float[] waveformData);
        }

        public interface ChannelVolumeListener {
            void onChannelVolumeChanged(float leftVolume, float rightVolume);
```

```
    }

    public interface BookmarkListener {
        void onBookmarksChanged(Collection<Bookmark> bookmarks);
    }

    public AudioAPlayer() {
        volumeProcessor = new VolumeProcessor();
    }
    public void setEqualizerProcessor(EqualizerProcessor processor) {
        this.equalizerProcessor = processor;
    }

    public void loadMedia(String uri) {
        if (dispatcher != null) {
            dispatcher.stop();
        }
        bookmarks.clear();
        if (spectrogramProcessor != null) {
            spectrogramProcessor.clearData();
        }
        notifyBookmarkListeners();


        currentFilePath = uri.startsWith("file:") ? uri.substring(5) : uri;
        currentFilePath = currentFilePath.replace("%20", " ");


        convertedFilePath = AudioConverter.convertToWav(currentFilePath);
        if (convertedFilePath == null) {
            System.err.println("Не вдалося конвертувати файл, використовуємо оригінальний");
            convertedFilePath = currentFilePath;
        }
```

```
            try {
                File audioFile = new File(convertedFilePath);
                audioInputStream = AudioSystem.getAudioInputStream(audioFile);
                format = audioInputStream.getFormat();
                totalDuration = audioFile.length() / (format.getSampleRate() *
format.getFrameSize());

                setupAudioDispatcher(audioFile);
                generateWaveformData();
                if (spectrogramProcessor != null) {
                    spectrogramProcessor.preAnalyzeFile(convertedFilePath);
                }

                System.out.println("Media loaded: " + convertedFilePath);
            } catch (Exception e) {
                System.err.println("Error loading media: " + e.getMessage());
                e.printStackTrace();
            }
        }

        private void setupAudioDispatcher(File audioFile) {
            try {
                if (dispatcher != null) {
                    dispatcher.stop();
                }

                if (format.getChannels() == 1)
                    BUFFER_SIZE = format.getFrameSize();
                else
                    BUFFER_SIZE = format.getFrameSize() - 2;
                dispatcher = AudioDispatcherFactory.fromFile(audioFile, BUFFER_SIZE,
OVERLAP);
                setupAudioProcessing();
            } catch (Exception e) {
                System.err.println("Error setting up audio dispatcher: " + e.getMessage());
```

```
            e.printStackTrace();
        }
    }

    private void setupAudioProcessing() {
        try {
            if (line == null || !line.isOpen()) {
                sourceLineInfo = new DataLine.Info(SourceDataLine.class, format);
                line = (SourceDataLine) AudioSystem.getLine(sourceLineInfo);
                line.open(format);
            }
            line.start();
            System.out.println("EqualizerProcessor in AudioAPlayer: " + equalizerProcessor);
            if (equalizerProcessor != null) {
                dispatcher.addAudioProcessor(equalizerProcessor);
                System.out.println("working");
            }
            if (volumeProcessor == null) {
                volumeProcessor = new VolumeProcessor();
                volumeProcessor.setVolume((float) currentVolume);
            }
            dispatcher.addAudioProcessor(volumeProcessor);
            AudioPlayer audioPlayer = new AudioPlayer(format);
            dispatcher.addAudioProcessor(audioPlayer);


            dispatcher.addAudioProcessor(new AudioProcessor() {
                @Override
                public boolean process(AudioEvent audioEvent) {
                    processAudioForChannelVolume(audioEvent.getFloatBuffer());
                    return true;
                }

                @Override
                public void processingFinished() {}
```

```
        });


        int fftBufferSize  = 1024;
        frequencyAnalyzer = new FrequencyAnalyzer(fftBufferSize, (int)
format.getSampleRate());
        frequencyAnalyzer.setEqualizerListener(bandLevels -> {

          if (!equalizerListeners.isEmpty()) {
            Platform.runLater(() -> {
              for (FrequencyAnalyzer.EqualizerListener listener : equalizerListeners) {
                listener.onBandLevelsChanged(bandLevels);
              }
            });
          }
        });
        dispatcher.addAudioProcessor(frequencyAnalyzer);

        int spectrogramFFTSize = 1024;
        spectrogramProcessor = new SpectrogramProcessor(spectrogramFFTSize, (int)
format.getSampleRate());
        spectrogramProcessor.addSpectrogramListener((spectrogramData, timeStamps,
sampleRate, fftSize) -> {
          for (SpectrogramProcessor.SpectrogramListener listener : spectrogramListeners) {
            listener.onSpectrogramDataAvailable(spectrogramData, timeStamps,
sampleRate, fftSize);
          }
        });
        dispatcher.addAudioProcessor(spectrogramProcessor);


        dispatcher.addAudioProcessor(new AudioProcessor() {
          @Override
          public boolean process(AudioEvent audioEvent) {
```

```
        double absoluteTime = playbackOffset + audioEvent.getTimeStamp();
        if (seekRequested.get()) {
            return false;
        }
        final double timeToReport = absoluteTime;
        Platform.runLater(() -> {
            currentTime.set(timeToReport);
            if (loopEnabled.get()) {
                checkLooping(timeToReport);
            }
        });
        return true;
    }

    @Override
    public void processingFinished() {
        Platform.runLater(() -> {
            if (!manuallyPaused && seekRequested.get() && isPlaying.get()) {
                seekRequested.set(false);
                try {
                    double seekPositionSeconds = seekPosition.get() * totalDuration / 100;
                    playbackOffset = seekPositionSeconds;
                    File audioFile = new File(convertedFilePath);
                    dispatcher = createDispatcherWithSkip(audioFile, seekPositionSeconds);
                    setupAudioProcessing();
                    if (isPlaying.get()) {
                        audioThread.submit(dispatcher);
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                    isPlaying.set(false);
                }
            } else if (!manuallyPaused && !loopEnabled.get()) {
                isPlaying.set(false);
                currentTime.set(0);
```

```
                }
            });
        }
    });
} catch (LineUnavailableException e) {
    System.err.println("Error setting up audio processing: " + e.getMessage());
    e.printStackTrace();
}
}


private void processAudioForChannelVolume(float[] buffer) {

    if (format.getChannels() == 1) {
        leftChannelVolume = calculateChannelVolume(buffer, 0, 1);
        rightChannelVolume = leftChannelVolume;
    } else if (format.getChannels() == 2) {
        leftChannelVolume = calculateChannelVolume(buffer, 0, 2);
        rightChannelVolume = calculateChannelVolume(buffer, 1, 2);
    }

    long now = System.currentTimeMillis();
    if (now - lastVolumeUpdateTime < VOLUME_UPDATE_THRESHOLD) {
        return;
    }
    lastVolumeUpdateTime = now;

    Platform.runLater(() -> {
        for (ChannelVolumeListener listener : channelVolumeListeners) {
            listener.onChannelVolumeChanged(leftChannelVolume, rightChannelVolume);
        }
    });
}

private float calculateChannelVolume(float[] buffer, int channelIndex, int totalChannels) {
```

```
        float maxVol = 0;
        for (int i = channelIndex; i < buffer.length; i += totalChannels) {
            float vol = Math.abs(buffer[i]);
            if (vol > maxVol) {
                maxVol = vol;
            }
        }
        return maxVol;
    }

    private void generateWaveformData() {
        try {
            new Thread(() -> {
                try {
                    waveformData                                                    =
WaveformProcessor.generateWaveformData(convertedFilePath, WAVEFORM_RESOLUTION);
                    Platform.runLater(() -> {
                        for (WaveformListener listener : waveformListeners) {
                            listener.onWaveformDataAvailable(waveformData);
                        }
                    });
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }).start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void play() {
        manuallyPaused = false;
        loopTransitionInProgress = false;
        if (convertedFilePath == null || convertedFilePath.isEmpty()) {
            System.err.println("Немає файлу для відтворення");
```

```
            return;
        }
        if (dispatcher == null || !isPlaying.get()) {
            try {
                double startTime = 0;
                if (pausedPosition > 0) {
                    startTime = pausedPosition;
                    pausedPosition = 0;
                    playbackOffset = startTime;
                } else if (seekRequested.get()) {
                    startTime = seekPosition.get() * totalDuration / 100;
                    playbackOffset = startTime;
                    seekRequested.set(false);
                }
                File audioFile = new File(convertedFilePath);
                if (startTime > 0) {
                    dispatcher = createDispatcherWithSkip(audioFile, startTime);
                } else {
                    dispatcher = AudioDispatcherFactory.fromFile(audioFile, BUFFER_SIZE,
OVERLAP);
                }
                setupAudioProcessing();
                if (line != null && !line.isActive()) {
                    line.start();
                }
                isPlaying.set(true);
                audioThread.submit(dispatcher);
            } catch (Exception e) {
                System.err.println("Помилка при відтворенні: " + e.getMessage());
                e.printStackTrace();
                isPlaying.set(false);
            }
        }
    }
```

```
public void pause() {
    if (isPlaying.get()) {
        pausedPosition = getCurrentTime();
        manuallyPaused = true;
        loopTransitionInProgress = false;
        seekRequested.set(false);
        isPlaying.set(false);
        if (dispatcher != null) {
            dispatcher.stop();
        }
        if (line != null && line.isActive()) {
            line.stop();
        }
    }
}

public void stop() {
    manuallyPaused = false;
    loopTransitionInProgress = false;
    seekRequested.set(false);
    isPlaying.set(false);
    if (dispatcher != null) {
        dispatcher.stop();
        dispatcher = null;
    }
    if (line != null) {
        line.stop();
        line.flush();
    }
    currentTime.set(0);
    seekPosition.set(0.0);
    pausedPosition = 0;
    playbackOffset = 0;
    Platform.runLater(() -> {
        for (WaveformListener listener : waveformListeners) {
```

```
            if (waveformData != null) {
                listener.onWaveformDataAvailable(waveformData);
            }
        }
    });
}

public void seek(double percent) {
    if (percent < 0)
        percent = 0;
    if (percent > 100)
        percent = 100;
    seekPosition.set(percent);
    seekRequested.set(true);
    if (dispatcher != null) {
        dispatcher.stop();
    }
    boolean wasPlaying = isPlaying.get();
    isPlaying.set(false);
    if (wasPlaying) {
        new Thread(() -> {
            try {
                Thread.sleep(100);
                Platform.runLater(() -> {
                    if (!manuallyPaused) {
                        play();
                    }
                });
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

```
    public void setVolume(double volume) {
        this.currentVolume = volume / 100.0;
        if (volumeProcessor != null) {
            volumeProcessor.setVolume((float) currentVolume);
        }
    }

    private AudioDispatcher createDispatcherWithSkip(File audioFile, double skipSeconds) throws Exception {
        AudioInputStream originalAIS = AudioSystem.getAudioInputStream(audioFile);
        AudioFormat baseFormat = originalAIS.getFormat();
        long bytesToSkip = (long) (skipSeconds * baseFormat.getSampleRate() * baseFormat.getFrameSize());
        originalAIS.close();
        AudioInputStream ais = AudioSystem.getAudioInputStream(audioFile);
        long totalSkipped = 0;
        while (totalSkipped < bytesToSkip) {
            long skipped = ais.skip(bytesToSkip - totalSkipped);
            if (skipped <= 0)
                break;
            totalSkipped += skipped;
        }
        if (totalSkipped < bytesToSkip) {
            System.err.println("Не вдалося пропустити всі байти: " + totalSkipped + " з " + bytesToSkip);
        }
        return new AudioDispatcher(new JVMAudioInputStream(ais), BUFFER_SIZE, OVERLAP);
    }


    public void addBookmarkAtCurrentPosition() {
        double time = getCurrentTime();
        Bookmark bookmark = new Bookmark(time);
        bookmarks.add(bookmark);
```

```
            notifyBookmarkListeners();
        }

        public boolean removeBookmarkAtCurrentPosition() {
            double time = getCurrentTime();
            Bookmark nearestBookmark = findNearestBookmark(time);
            if (nearestBookmark != null && Math.abs(nearestBookmark.getTimePosition() - time)
<= BOOKMARK_TIME_TOLERANCE) {
                bookmarks.remove(nearestBookmark);
                notifyBookmarkListeners();
                return true;
            }
            return false;
        }

        public void seekToNextBookmark() {
            if (bookmarks.isEmpty()) return;
            double time = getCurrentTime();
            Bookmark next = getNextBookmark(time);
            if (next != null) {
                double seekPositionPercent = (next.getTimePosition() / totalDuration) * 100;
                seek(seekPositionPercent);
            }
        }

        public void seekToPreviousBookmark() {
            if (bookmarks.isEmpty()) return;
            double time = getCurrentTime();
            Bookmark prev = getPreviousBookmark(time);
            if (prev != null) {
                double seekPositionPercent = (prev.getTimePosition() / totalDuration) * 100;
                seek(seekPositionPercent);
            } else {
                seek(0);
            }
```

```
    }


    private void checkLooping(double currentTimeInSeconds) {
      if (!loopEnabled.get() || bookmarks.isEmpty() || loopTransitionInProgress) {
        return;
      }
      Bookmark next = getNextBookmark(currentTimeInSeconds);
      if (next == null) {
        Bookmark last = getLastBookmark();
        if (last != null && currentTimeInSeconds >= last.getTimePosition() -
BOOKMARK_TIME_TOLERANCE) {
          Bookmark first = getFirstBookmark();
          if (first != null) {
            double seekPositionPercent = (first.getTimePosition() / totalDuration) * 100;
            loopTransitionInProgress = true;
            seek(seekPositionPercent);
          }
        }
        return;
      }
      if (currentTimeInSeconds >= next.getTimePosition() -
BOOKMARK_TIME_TOLERANCE) {
        Bookmark prev = getPreviousBookmark(next.getTimePosition());
        double seekPositionSeconds = (prev != null) ? prev.getTimePosition() : 0;
        double seekPositionPercent = (seekPositionSeconds / totalDuration) * 100;
        loopTransitionInProgress = true;
        seek(seekPositionPercent);
      }
    }

    private Bookmark findNearestBookmark(double time) {
      if (bookmarks.isEmpty())
        return null;
      Bookmark floor = bookmarks.floor(new Bookmark(time));
```

```
        Bookmark ceiling = bookmarks.ceiling(new Bookmark(time));
        if (floor == null)
            return ceiling;
        if (ceiling == null)
            return floor;
        return (time - floor.getTimePosition() < ceiling.getTimePosition() - time) ? floor :
ceiling;
    }

    private Bookmark getFirstBookmark() {
        return !bookmarks.isEmpty() ? bookmarks.first() : null;
    }

    private Bookmark getLastBookmark() {
        return !bookmarks.isEmpty() ? bookmarks.last() : null;
    }

    private Bookmark getNextBookmark(double currentTime) {
        Bookmark dummy = new Bookmark(currentTime);
        return bookmarks.higher(dummy);
    }

    private Bookmark getPreviousBookmark(double time) {
        Bookmark dummy = new Bookmark(time);
        return bookmarks.lower(dummy);
    }


    public void addAudioListener(AudioListener listener) {
        listeners.add(listener);
    }

    public void addWaveformListener(WaveformListener listener) {
        waveformListeners.add(listener);
        if (waveformData != null) {
```

```
            listener.onWaveformDataAvailable(waveformData);
        }
    }

    public void addChannelVolumeListener(ChannelVolumeListener listener) {
        channelVolumeListeners.add(listener);
    }

    public void addBookmarkListener(BookmarkListener listener) {
        bookmarkListeners.add(listener);
        listener.onBookmarksChanged(bookmarks);
    }


    public void addEqualizerListener(FrequencyAnalyzer.EqualizerListener listener) {
        equalizerListeners.add(listener);
    }

    private void notifyBookmarkListeners() {
        for (BookmarkListener listener : bookmarkListeners) {
            listener.onBookmarksChanged(bookmarks);
        }
    }


    public double getCurrentTime() {
        return currentTime.get();
    }

    public SimpleDoubleProperty currentTimeProperty() {
        return currentTime;
    }

    public double getTotalDuration() {
        return totalDuration;
```

```
}

public float[] getAudioBuffer() {
    return audioBuffer;
}

public float[] getWaveformData() {
    return waveformData;
}

public String getConvertedFilePath() {
    return convertedFilePath;
}

public float getLeftChannelVolume() {
    return leftChannelVolume;
}

public float getRightChannelVolume() {
    return rightChannelVolume;
}

public Collection<Bookmark> getBookmarks() {
    return Collections.unmodifiableCollection(bookmarks);
}

public boolean isLoopEnabled() {
    return loopEnabled.get();
}

public void setLoopEnabled(boolean enabled) {
    loopEnabled.set(enabled);
}

public BooleanProperty loopEnabledProperty() {
```

```
        return loopEnabled;
    }

    public Object getMediaPlayer() {
        return dispatcher;
    }

    public void cleanup() {
        if (dispatcher != null) {
            dispatcher.stop();
            dispatcher = null;
        }
        if (line != null) {
            line.stop();
            line.close();
            line = null;
        }
        audioThread.shutdownNow();
        AudioConverter.cleanupTempFiles();
    }
}
package com.example.audioaplayer;

import javax.sound.sampled.*;
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.UUID;

public class AudioConverter {

    private static final String TEMP_DIR = System.getProperty("java.io.tmpdir");
    private static final Path CONVERTER_DIR = Paths.get(TEMP_DIR,
"audio_player_converted");
```

```
        static {
            try {
                if (!Files.exists(CONVERTER_DIR)) {
                    Files.createDirectories(CONVERTER_DIR);
                }
            } catch (IOException e) {
                System.err.println("Не вдалося створити тимчасову директорію: " +
e.getMessage());
            }
        }


        public static String convertToWav(String inputFilePath) {
            try {

                if (inputFilePath.toLowerCase().endsWith(".wav")) {
                    return inputFilePath;
                }
                File inputFile = new File(inputFilePath);
                if (!inputFile.exists()) {
                    System.err.println("Вхідний файл не знайдено: " + inputFilePath);
                    return null;
                }
                AudioInputStream                audioInputStream                =
AudioSystem.getAudioInputStream(inputFile);
                AudioFormat baseFormat = audioInputStream.getFormat();
                AudioFormat targetFormat = new AudioFormat(
                        AudioFormat.Encoding.PCM_SIGNED,
                        baseFormat.getSampleRate(),
                        16,
                        baseFormat.getChannels(),
                        baseFormat.getChannels() * 2,
                        baseFormat.getSampleRate(),
```

```
                    false
                );
                AudioInputStream convertedStream = AudioSystem.getAudioInputStream(targetFormat, audioInputStream);
                String outputFileName = UUID.randomUUID().toString() + ".wav";
                Path outputPath = CONVERTER_DIR.resolve(outputFileName);
                File outputFile = outputPath.toFile();
                AudioSystem.write(convertedStream, AudioFileFormat.Type.WAVE, outputFile);
                convertedStream.close();
                audioInputStream.close();
                System.out.println("Файл успішно конвертовано: " + outputFile.getAbsolutePath());
                return outputFile.getAbsolutePath();

            } catch (UnsupportedAudioFileException e) {
                System.err.println("Непідтримуваний формат аудіо: " + e.getMessage());
            } catch (IOException e) {
                System.err.println("Помилка введення/виведення: " + e.getMessage());
            } catch (Exception e) {
                System.err.println("Помилка конвертації: " + e.getMessage());
                e.printStackTrace();
            }
            return null;
        }


        public static void cleanupTempFiles() {
            try {
                if (Files.exists(CONVERTER_DIR)) {
                    Files.list(CONVERTER_DIR).forEach(file -> {
                        try {
                            Files.delete(file);
                        } catch (IOException e) {
                            System.err.println("Не вдалося видалити тимчасовий файл: " + file);
                        }
```

```
                    });
                }
            } catch (IOException e) {
                System.err.println("Помилка при очищенні тимчасових файлів: " +
e.getMessage());
            }
        }
    }
    package com.example.audioaplayer;

    public class Bookmark implements Comparable<Bookmark> {
        private final double timePosition;
        public Bookmark(double timePosition) {
            this.timePosition = timePosition;
        }
        public double getTimePosition() {
            return timePosition;
        }
        @Override
        public int compareTo(Bookmark other) {
            return Double.compare(this.timePosition, other.timePosition);
        }
        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            Bookmark bookmark = (Bookmark) obj;
            return Double.compare(bookmark.timePosition, timePosition) == 0;
        }
        @Override
        public int hashCode() {
            return Double.hashCode(timePosition);
        }}
```

Додаток Г. Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АУДІОПЛЕЄРА З РОЗШИРЕНИМИ МОЖЛИВОСТЯМИ АНАЛІЗУ ЗВУКУ

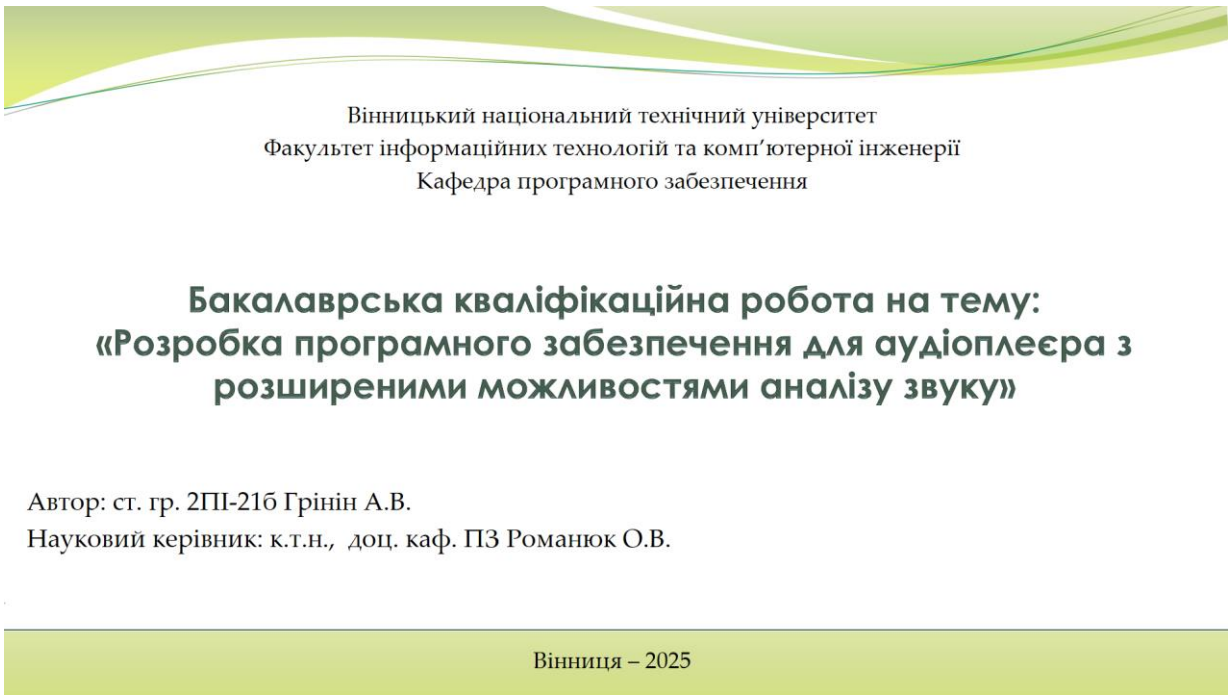


Рисунок Г.1 – Титульний слайд

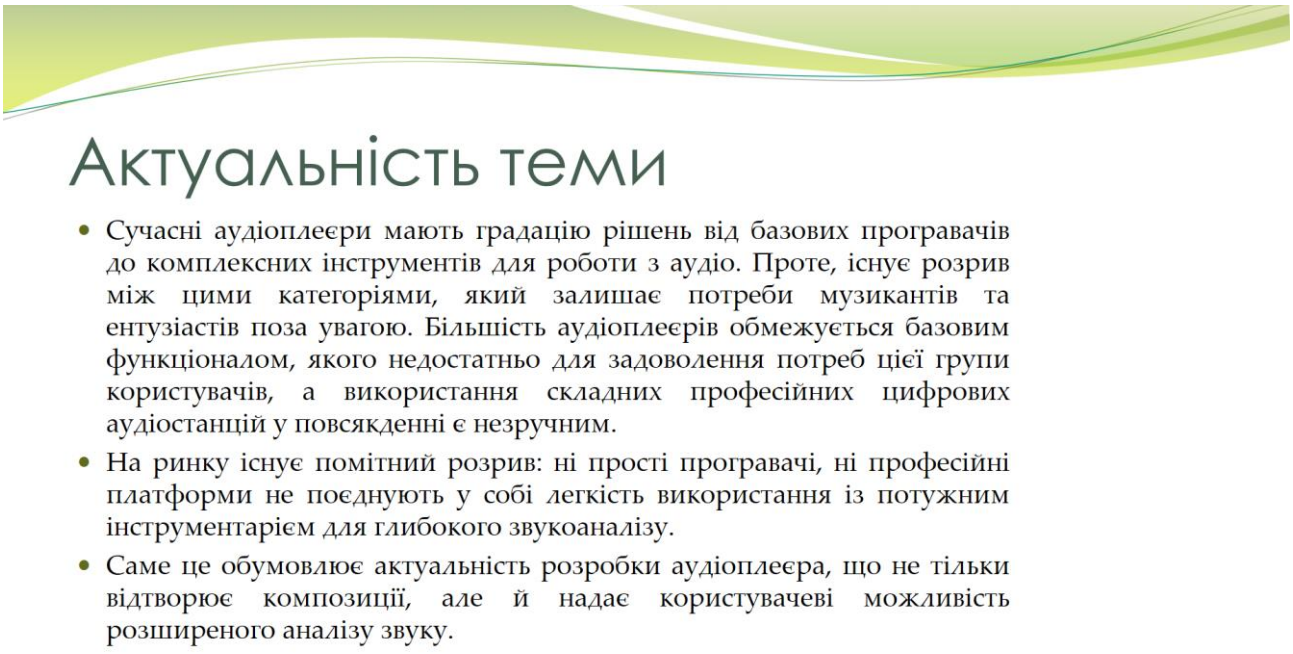


Рисунок Г.2 – Актуальність теми

Рисунок Г.3 – Мета, об'єкт та предмет дослідження

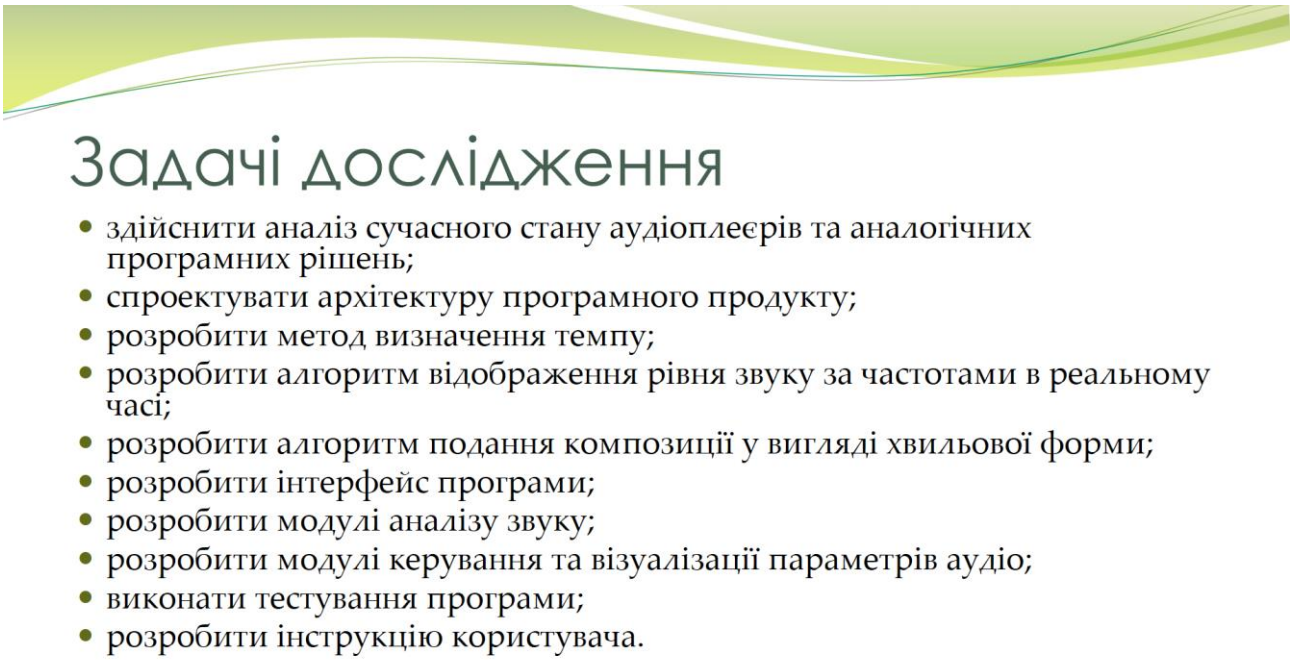


Рисунок Г.4 – Задачі дослідження

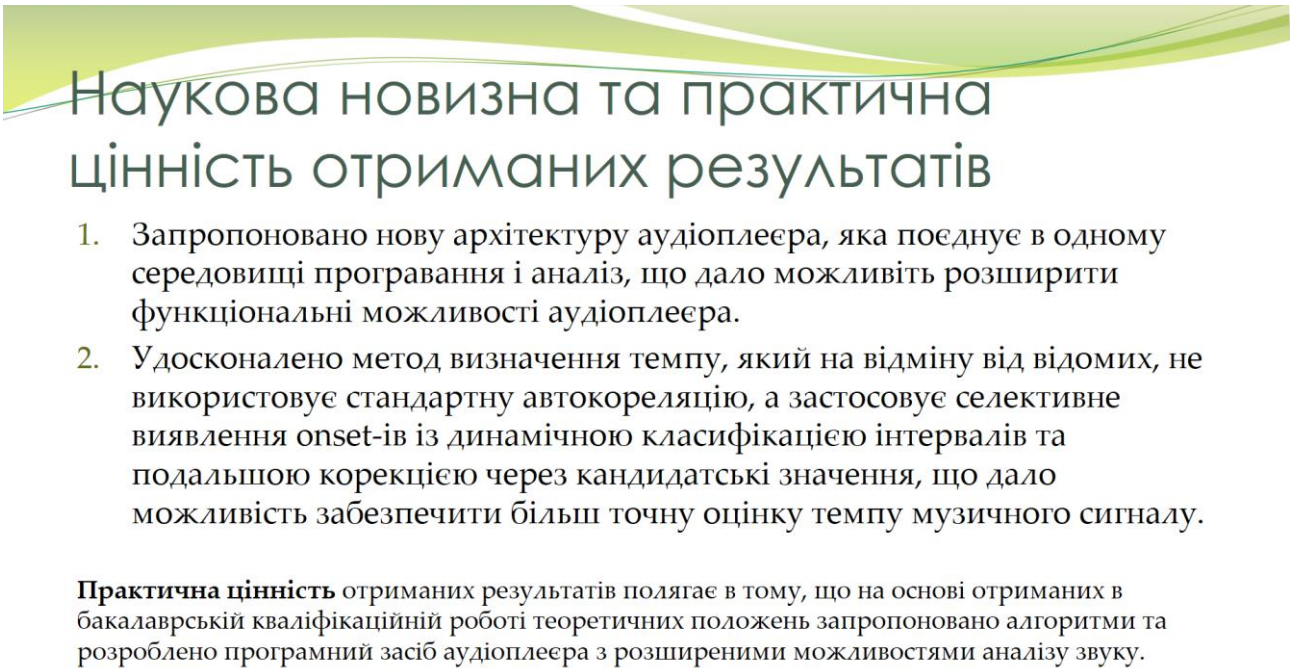


Рисунок Г.5 – Наукова новизна та практична цінність отриманих результатів

Критерій	MusicBee	Foobar2000	VLC	Власна програма
Еквалайзер із пресетами	+	+	+	+
Візуалізація хвильової форми	-	+	-	+
Індикатор гучності каналів	-	-	-	+
Спектральна візуалізація смуг частот	+	+	+	+
Спектрограма	-	-	-	+
Аналіз темпу	+	+	-	+
Система закладок	-	-	-	+
Підтримка плагінів	+	+	+	-
Загальна оцінка	4	5	3	7

Рисунок Г.6 – Порівняльний аналіз аналогів

Рисунок Г.7 – Архітектура програми

Рисунок Г.8 – Блок-схеми алгоритмів відображення рівня звуку за частотами та генерації хвильової форми

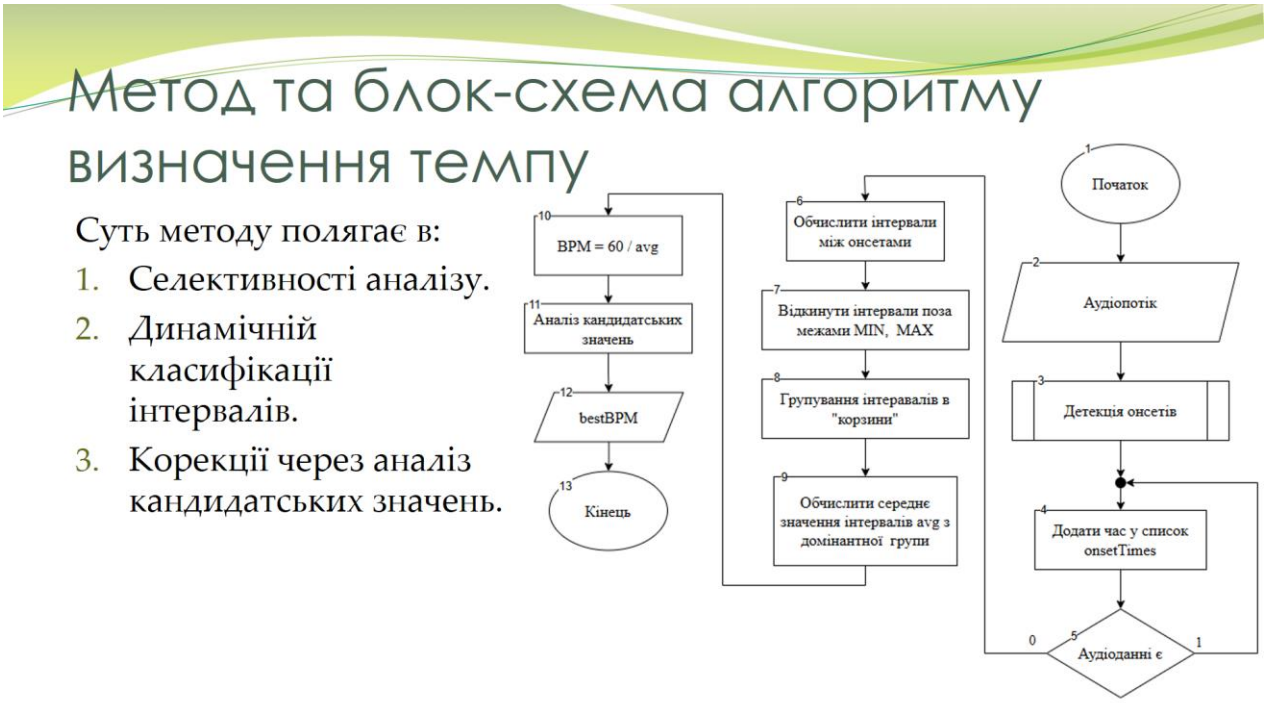


Рисунок Г.9 – Метод та блок-схема алгоритму визначення темпу

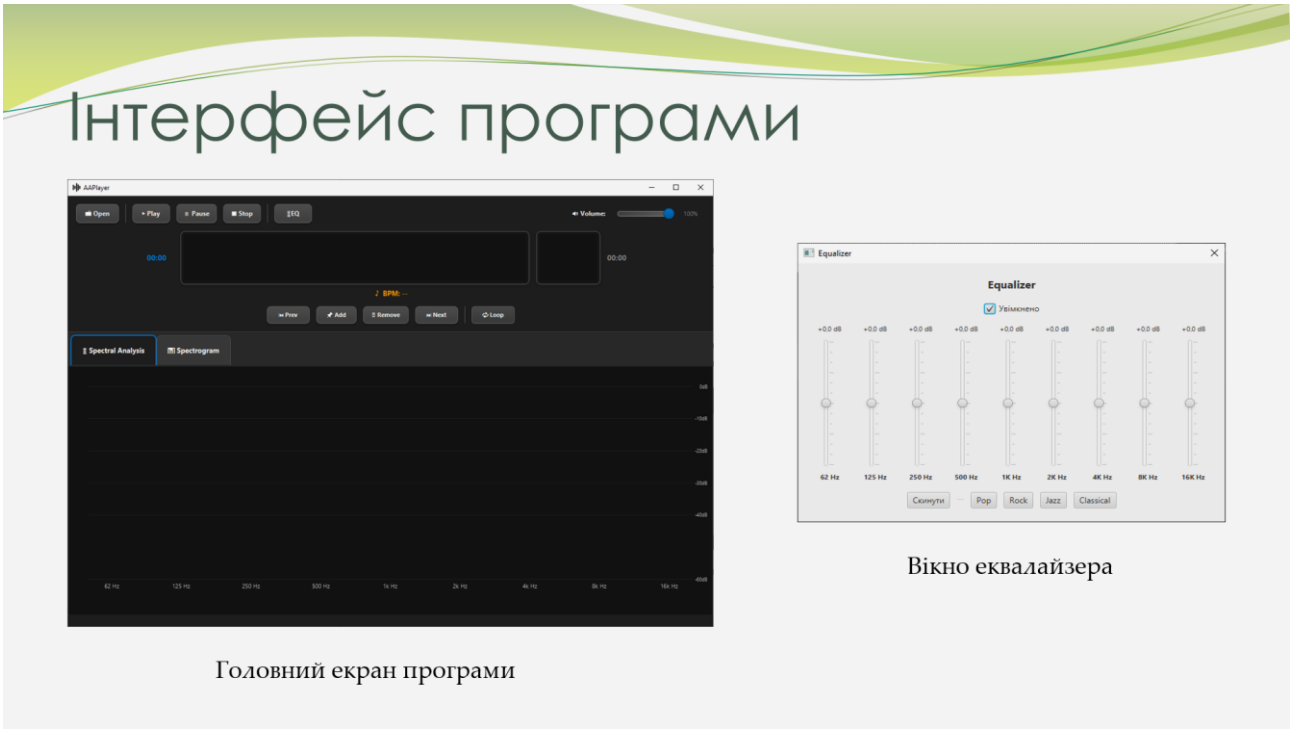


Рисунок Г.10 – Інтерфейс програми

Рисунок Г.11 – Модулі візуалізації параметрів звуку

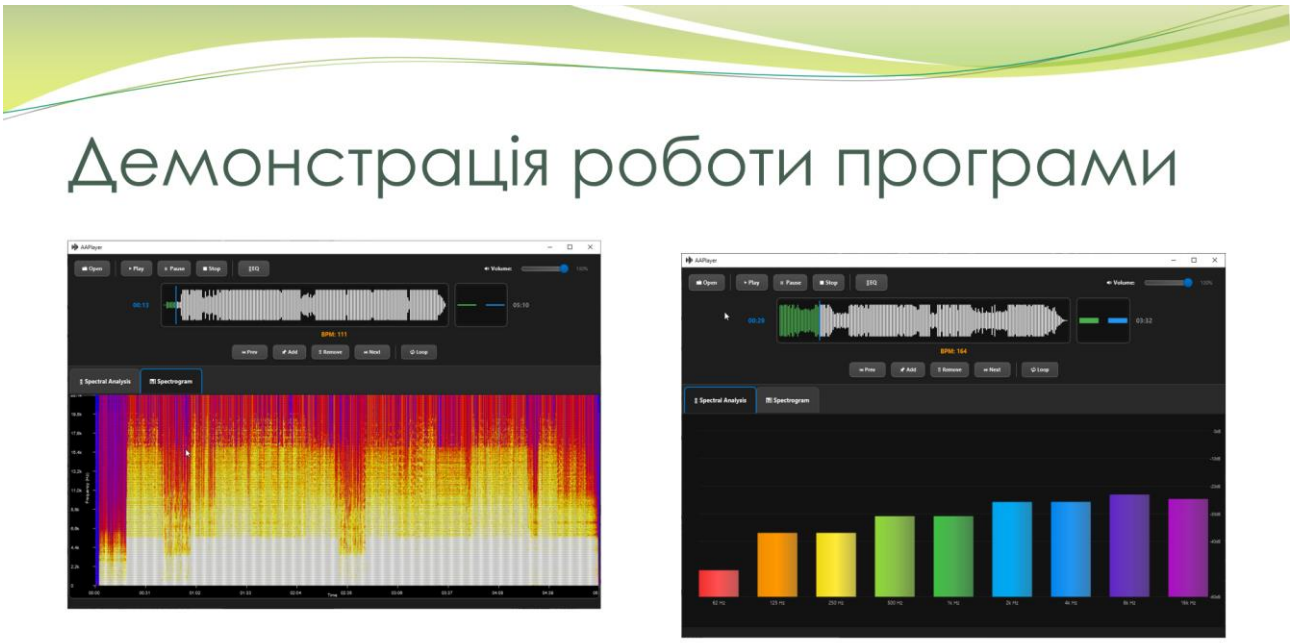


Рисунок Г.12 – Демонстрація роботи програми

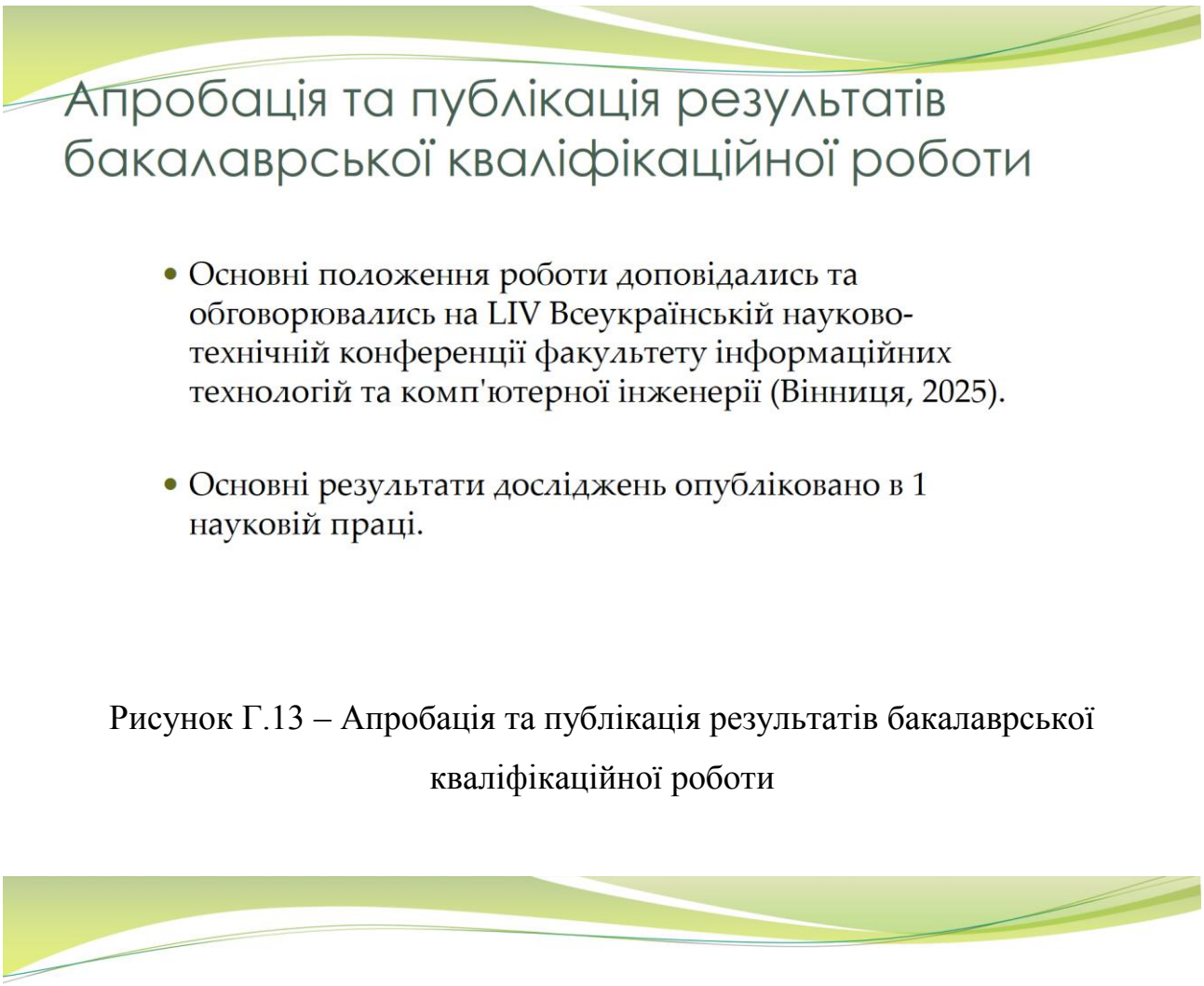


Рисунок Г.13 – Апробація та публікація результатів бакалаврської кваліфікаційної роботи

Рисунок Г.14 – Фінальний слайд