

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ ЗАКЛАД
«ЛУГАНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА»

Навчально-науковий інститут
математики та інформаційних технологій

Кафедра математики та інформатики

Хе Чжичжун

**АНАЛІЗ ДЕКЛАРАТИВНОГО ПІДХОДУ У ПРОГРАМУВАННІ КРОС-
ПЛАТФОРМОВИХ ДОДАТКІВ З ВИКОРИСТАННЯМ KOTLIN**

кваліфікаційна робота
здобувача вищої освіти другого (магістерського) рівня
за спеціальністю 122 Комп'ютерні науки

Особистий підпис _____ **Хе Чжичжун**

Науковий керівник _____ **Галина КОЗУБ,**
кандидат технічних наук,
доцент кафедри інформаційних
технологій та систем

В.о. завідувача кафедри _____ **Юрій КОЗУБ,**
доктор технічних наук,
професор кафедри
математики та інформатики

Полтава – 2025

АНОТАЦІЯ

Хе Чжичжун

Тема: Аналіз декларативного підходу у програмуванні крос-платформних додатків з використанням Kotlin

Спеціальність: 122 „Комп’ютерні науки”

Установа: ДЗ ЛНУ імені Тараса Шевченка, 2025р.

Кваліфікаційна робота містить: 82 стор., 18 рис., 40 джерел, додатки.

Об’єкт дослідження – мульти-платформний додаток „Arranger”

Предмет дослідження – технології розробки мульти-платформних додатків

Мета роботи – дослідити технології розробки мульти-платформних застосунків, розробити мульти-платформний застосунок „Arranger”

Результати роботи. У роботі проведено аналіз існуючих аналогів розробки крос-платформних та мульти-платформних додатків, а також докладно досліджено інструменти розробки Kotlin Multiplatform та Jetpack Compose. Описано бізнес-логіку та створено інтерфейс користувача під декілька платформ, що сприяє зменшенню вартості продукту та прискорення його розробки.

На базі отриманих результатів розроблено мульти-платформний додаток “Arranger” для операційних систем Android, Linux, Windows та macOS, який дозволяє створювати мелодії у форматі midi

Ключові слова: JETPACK COMPOSE, KOTLIN, MULTIPLATFORM, DECOMPOSE, MVIKOTLIN, ANDROID, DESKTOP

ANNOTATION

He Zhizhong

Theme: Analysis of the declarative approach in programming cross-platform applications using Kotlin

Specialty: 122 “ Computer Science ”

Institution: Luhansk Taras Shevchenko National University, 2025.

Qualification work contains: 82 pages, 18 figures, 40 sources.

The object of research is multiplatform application “Arranger”.

The subject of research – technologies for developing multiplatform applications.

The purpose of work – research technologies for developing multiplatform applications, develop a multiplatform application “Arranger”.

Results of work. The analysis of existing analogues of crossplatform and multiplatform application development is carried out, as well as the Development Tools Kotlin Multiplatform and Jetpack Compose are studied in detail. The business logic is described and a user interface is created for several platforms, which helps to reduce the cost of the product and speed up its development.

Conclusion. Based on the results obtained, a multi-platform application “Arranger” was developed for Android, Windows, Linux and macOS operating systems, which allows to create melodies in a format like midi.

Keywords: JETPACK COMPOSE, KOTLIN, MULTIPLATFORM, DECOMPOSE, MVIKOTLIN, ANDROID, DESKTOP

ЗМІСТ

	Стор.
ВСТУП.....	8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	11
1.1. Аналіз фреймворків кросплатформної розробки додатків.....	11
1.1.1. Фреймворк Xamarin.....	11
1.1.2. Фреймворк React Native	12
1.1.3. Фреймворк Flutter	13
1.1.4. Переваги та недоліки фреймворків.....	14
1.2. Фреймворк Kotlin Multiplatform	14
1.3. Декларативне програмування	17
1.4. Фреймворк Jetpack Compose	19
1.4.1. Принцип розділення завдань	19
1.4.2. Composable функції	20
1.4.3. Переваги Jetpack Compose.....	21
1.5. Патерн MVI.....	21
1.6. Шаблон Dependency Injection	25
1.7. Висновки до розділу 1	27
РОЗДІЛ 2. ЗАСОБИ ПРОЄКТУВАННЯ ДОДАТКУ	29
2.1. Мова програмування Kotlin	29
2.2. Бібліотека Kotlin Coroutines	30
2.3. Система збірки проєктів Gradle Kotlin DSL.....	31
2.4. Фреймворк Compose Multiplatform.....	32
2.5. Бібліотека Decompose.....	33
2.6. Фреймворк MVIKotlin.....	34
2.7. Бібліотека SQLDelight.....	35
2.8. Бібліотека Koin	37

2.9. Висновки до розділу 2	38
РОЗДІЛ 3. ПРОЄКТУВАННЯ ЗАСТОСУНКУ	39
3.1. Головне меню додатку - Main Menu	39
3.1.1. Розділ Music	40
3.1.2. Розділ Instruments	40
3.1.3. Розділ Settings	41
3.2. Компонент Add Music	42
3.3. Компонент Create Song	42
3.4. Компонент Record Sample	43
3.5. Компонент Import Song	43
3.6. Компонент Song Editor	43
3.7. Компонент Track Editor	44
3.8. Компонент Midi Converter	45
3.9. Компонент Player	45
3.10. Висновки до розділу 3	46
РОЗДІЛ 4. РОЗРОБКА ЗАСТОСУВАННЯ	47
4.1. Налаштування системи збірки проєктів	47
4.2. Розробка архітектури додатку	48
4.2.1. Common модуль	48
4.2.2. Платформні модулі	49
4.3. Створення компонентів екранів	50
4.4. Створення компонентів Store	52
4.5. Розробка формату мелодії	53
4.6. Створення бази даних SQLDelight	54
4.7. Створення сховища для налаштувань додатку	56
4.8. Налаштування відмінностей UI для різних платформ	56
4.9. Розробка системи відтворення звуків	57
4.10. Застосування Dependency Injection	58

4.11. Висновки до розділу 4	60
ВИСНОВКИ	61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	62
ДОДАТКИ	66
Додаток А. Файл build.gradle.kts (common модуль)	66
Додаток Б. Файл build.gradle.kts (android модуль).....	70
Додаток В. Файл build.gradle.kts (desktop модуль)	71
Додаток Г. Файл MusicScreen.kt	71
Додаток Д. Файл MusicComponent.kt	75
Додаток Е. Файл MusicComponentProvider.kt	76
Додаток Ж. Файл MusicStore.kt	77
Додаток З. Файл MusicStoreProvider.kt	78
Додаток И. Файл Song.sql	80
Додаток К. Файл PreferencesRepository.kt.....	81
Додаток Л. Файл AlertDialogExpected.kt (android модуль)	81
Додаток М. Файл AlertDialogExpected.kt (desktop модуль)	82

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

Coroutine	- співпрограма;
DI	- Dependency Injection;
Hot Reloading	- механізм зміни додатку під час його використання;
MVI	- Model-View-Intent;
UI	- User Interface;
UI-логіка	- логіка роботи інтерфейсу;
БД	- база даних;
Бізнес-логіка	- настраювані правила або алгоритми, які обробляють обмін інформацією між базою даних і інтерфейсом користувача.

ВСТУП

Враховуючи, що в світі щорічно з'являється 100 мільйонів стартапів- час виведення продукту на ринок став набагато важливіше вартості. Користувачі стають все вимогливішими, тому розробникам краще вибирати нативні рішення. Користувачі стають все вимогливішими, тому розробникам краще вибирати нативні рішення. Однак програмна реалізація окремих нативних додатків під декілька платформ сприяє збільшенню витрат на їх підтримку. Для вирішення цієї проблеми використовується фреймворк Kotlin Multiplatform, що дозволяє створювати універсальний код логіки додатку, який працює однаково на всіх платформах.

У поєднанні з UI фреймворком Compose Multiplatform, стає можливим написання єдиного коду логіки та інтерфейсу додатку для декількох платформ одночасно, що допомагає економити час та уникати значної кількості помилок, тому тема є актуальною.

Мета роботи – провести аналіз технології розробки крос-платформних та мульти-платформних застосунків, розробити мульти-платформний застосунок “Arranger”.

Для досягнення мети необхідно вирішити наступні завдання:

- проаналізувати фреймворки для мульти-платформної розробки;
- дослідити фреймворк Kotlin Multiplatform;
- дослідити фреймворк Compose Multiplatform;
- дослідити принципи декларативного програмування;
- розглянути засоби розробки мульти-платформних застосунків;
- спроектувати структуру застосунку;
- розробити архітектуру компонентів мульти-платформного застосунку;
- розробити формат мелодії застосунку;
- розробити мульти-платформне програмне застосування “Arranger”, який здатний створювати, редагувати та відтворювати мелодії у форматі на зразок midi.

Об’єктом дослідження є особливості Kotlin Multiplatform та Compose Multiplatform.

Предметом дослідження є використання фреймворків Kotlin Multiplatform та Compose Multiplatform для розробки мульти-платформних застосунків.

Наукова новизна:

- розроблено методику створення мульти-платформних застосунків;
- розроблено архітектуру компонентів у мульти-платформних застосунків;
- створено новий формат мелодій.

Практичне значення: прискорення розробки застосунків під різні платформи, зменшення кількості помилок, зменшення вартості розробки.

Особистий внесок:

- поєднано новітні інструменти мульти-платформної розробки у єдиному проєкті з використанням декларативного підходу для максимального перевикористання коду;
- створено новий формат мелодій;
- розроблено архітектуру компонентів у мульти-платформних додатках;
- розроблено застосунок “Arranger”, який дозволяє створювати та редагувати мелодії у новому форматі.

Структура і обсяг роботи. Робота складається з вступу, чотирьох розділів, висновку, списку використаних джерел та додатків. Обсяг роботи становить 82 сторінки, обсяг використаної літератури - 40 джерел.

Перший розділ містить аналіз існуючих аналогів розробки крос-платформних та мульти-платформних застосунків, опис фреймворку Kotlin Multiplatform, фреймворку Jetpack Compose та принципів декларативного програмування.

У другому розділі описано базові засоби проєктування додатків, а саме: мова програмування Kotlin, бібліотека Kotlin Coroutines, система збірки

проєктів Gradle Kotlin DSL. Представлено засоби проєктування мульти-платформних додатків: фреймворк Compose Multiplatform для розробки мульти-платформного інтерфейсу користувача, бібліотеку Decompose для створення UI-компонентів та налагодження навігації між ними, фреймворк MVIKotlin для опису логіки інтерфейсу користувача, бібліотеку SQLDelight для створення бази даних SQLite та бібліотеку Koin для налагодження Dependency Injection.

Третій розділ присвячено проєктуванню структури програмного застосунку “Arranger” який використовується для створення, редагування та відтворення музичних мелодій у власному форматі.

У четвертому розділі представлено ключові моменти створення додатку “Arranger”, налаштування системи збірки мульти-платформних проєктів, розробка архітектури застосунку, створення компонентів екранів, розробка формату мелодії, створення бази даних та сховища налаштувань програмного застосунку, опис відмінностей для різних платформ, розробка системи відтворення звуків та застосування Dependency Injection у додатку.

Додатки містять частини вихідного коду застосунку.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аналіз фреймворків кросплатформної розробки додатків

Крім значущості, якості та зручності використання, на успіх додатку також впливає швидкість розробки і впровадження нових функціональностей. Поліпшити цей показник можна, використовуючи кросплатформні інструменти.

Найвідомішими крос-платформними інструментами є: Xamarin, React Native, Flutter.

1.1.1. Фреймворк Xamarin

Xamarin – платформа від Microsoft для створення додатків під Android, iOS, Windows, Linux, macOS, watchOS та tvOS. Xamarin включає єдину загальну кодову базу C# і надає можливість тестувати додатки на декількох пристроях з використанням Xamarin Cloud.

Xamarin має два основних інструменти: Xamarin.Android, Xamarin.iOS і Xamarin.Forms. По частині кросплатформної розробки Xamarin пропонує використовувати єдиний API Xamarin.Essentials.

Xamarin.Android і Xamarin.iOS наділяють додаток тими ж можливостями і інтерфейсом, які є у нативних рішень. У випадку Xamarin.iOS програма компілюється безпосередньо в машинний код (АОТ-компіляція), тоді як в Xamarin.Android спочатку відбувається компіляція в байт-код, який потім інтерпретується віртуальною машиною (JIT-компіляція).

Для прискорення процесу написання коду, краще використовувати Xamarin.Forms – простіший інструмент, в якому майже всі елементи повністю сумісні з будь-якими платформами.

При роботі з ASP.NET на сервері дозволяє додатково заощадити певний обсяг роботи за рахунок використання загальних класів бізнес-логіки на сервері та в мобільному додатку.

Продуктивність Xamarin вважається близькою до нативної, але залежить від того, використовується Xamarin.Android, Xamarin.iOS або Xamarin.Forms. У Xamarin.Android та Xamarin.iOS хороша оптимізація завдяки активним компонентам, тоді як Xamarin.Forms заснований на 100% спільному використанні коду, що в цілому знижує його продуктивність в порівнянні з Xamarin.Android та Xamarin.iOS.

Фреймворк має сильне партнерське співтовариство, до якого відносяться корпорації Microsoft та IBM. На Xamarin зроблені такі додатки, як: Olo, The World Bank, Story та інші [7].

1.1.2. Фреймворк React Native

React Native – фреймворк для створення крос-платформних додатків на мові JavaScript. Дозволяє писати додатки для IOS, Android, Windows, Web, Windows Phone, VR, Android TV, macOS, tvOS.

React Native розроблений компанією Facebook, яка вкладає величезні ресурси у розвиток своїх технологій та створює навколо них цілу інфраструктуру і потужне IT-співтовариство.

Архітектура фреймворку описується принципом “Learn once, write anywhere,, який має на увазі застосування одного і того ж коду для різних платформ. Також в Native є функція Hot Reloading, що дозволяє додавати новий код і вносити правки прямо під час виконання, що є дуже корисно, для розробки інтерфейсу користувача.

Середовище поставляється з великим набором готових компонентів, однак вони не завжди адаптуються під різні платформи, що вимагає додаткових коригувань в коді. Завдяки великій підтримці спільноти також є багатий вибір сторонніх бібліотек.

Так як React Native націлений на результат, порівнянний з нативної розробкою, в гонитві за продуктивністю розробники найчастіше віддають перевагу саме цьому фреймворку. Native також дозволяє використовувати

кастомні модулі на мовах для нативної розробки, але їх необхідно писати окремо для кожної платформи.

React Native досить популярний, оскільки його вже застосовують технологічні гіганти. Серед них Instagram, Facebook, Walmart, Tesla, Pinterest, UberEats та інші. На цьому фреймворку написані UberEats, Facebook Groups і частково Instagram і Facebook [8].

1.1.3. Фреймворк Flutter

Flutter – безкоштовний кросплатформний фреймворк від Google з відкритим вихідним кодом для швидкої розробки додатків під Android, Linux, iOS, Windows, macOS, Web та Google Fuchsia. Фреймворк використовує об'єктно-орієнтовану мову програмування Dart, яка була розроблена Google.

Flutter використовує один і той же код для всіх платформ. На ньому легко створювати красиві інтерфейси. Фреймворк не підтримує написання різних стилів для різних ОС, оскільки автоматична адаптація для цих цілей не передбачена. Це пов'язано з тим, що замість нативних компонентів Flutter застосовує свій графічний движок. Однак він не відстає від React Native і також пропонує функцію Hot Reloading та великий набір готових віджетів. З Flutter можна випускати додатки для різних версій Android і iOS без додаткових складнощів: програми спокійно запускаються навіть на таких старих версіях, як Android Jelly Bean і iOS 8.

При інших рівних можна сказати, що Flutter перевершує конкурентів і демонструє найвищу продуктивність завдяки сучасній мові Dart і власному движку рендерингу.

Flutter включає сторонні SDK, API для 2D, анімації, власні віджети Material Design і надає можливість повторно використовувати існуючий код Java, Swift та Objective-C.

Flutter з'явився на ринку не так давно, але його популярність зростає за дуже короткий час. Додатки на ньому можна побачити у Alibaba, Hamilton Musical, Greentea, Google Ads [9].

1.1.4. Переваги та недоліки фреймворків

Перевагами кросплатформної розробки є висока швидкість та низька вартість розробки.

До недоліків відносяться:

- відносно низька продуктивність, у порівнянні з нативними додатками;
- займає більше місця у пам'яті;
- складна підтримка низкорівневих платформних функцій;
- нові платформні можливості з'являються пізніше ніж у нативних додатках;

Щоб вирішити ці недоліки, використовується фреймворк Kotlin Multiplatform.

1.2. Фреймворк Kotlin Multiplatform

Kotlin Multiplatform – технологія, яка дозволяє використовувати один раз написаний код на безлічі платформ одночасно. Для розробки використовується мова програмування Kotlin та середовище розробки IntelliJ IDEA або Android Studio. Системою збірки є Gradle, який підтримує синтаксис groovy і kotlin script (kts).

У Kotlin Multiplatform є поняття targets – цільові платформи. В даних блоках налаштовуються необхідні нам операційні системи, в нативний код яких і компілюється код на Kotlin [10].

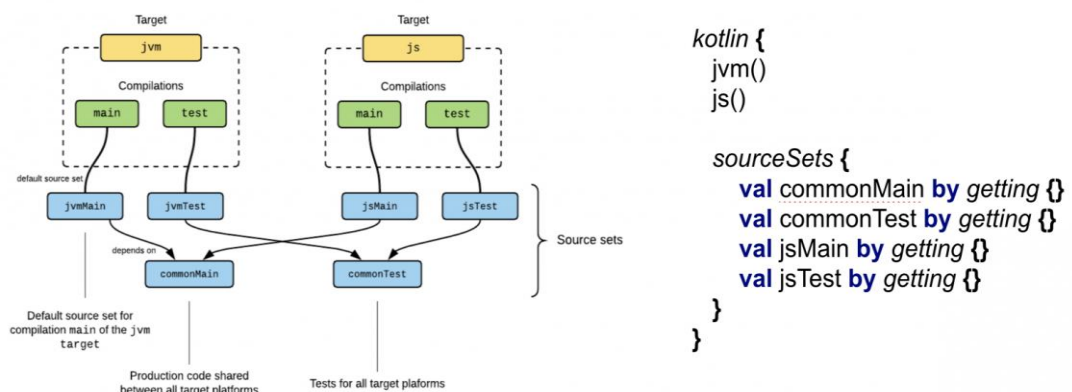


Рис. 1.1. Реалізація jvm та js таргетів [11]

Аналогічним чином реалізується підтримка інших платформ.

Source sets – це вихідні коди для платформ. Є загальний набір вихідних кодів і платформні (їх стільки, скільки в проєкті таргетів, за цим стежить IDE). Для реалізації цієї особливості використовується механізм expect-actual.



Рис. 1.2. Механізм expect-actual [11]

Expect-actual дозволяє із загального модуля звертатися до платформозалежного коду. Можна оголосити expect декларацію в Common модулі і реалізувати її в платформних модулях.

Приклад використання механізму для отримання дати на пристроях:

```
Common:
internal expect val timestamp: Long

Android/JVM:
internal actual val timestamp: Long
get() = java.lang.System.currentTimeMillis()

iOS:
internal actual val timestamp: Long
get() = platform.Foundation.NSDate().timeIntervalSince1970.toLong()
```

Як видно для платформних модулів доступні системні бібліотеки для Java та iOS пристроїв відповідно.

Структура мульти-платформних проєктів складається зі спільного модуля та модулів платформ.

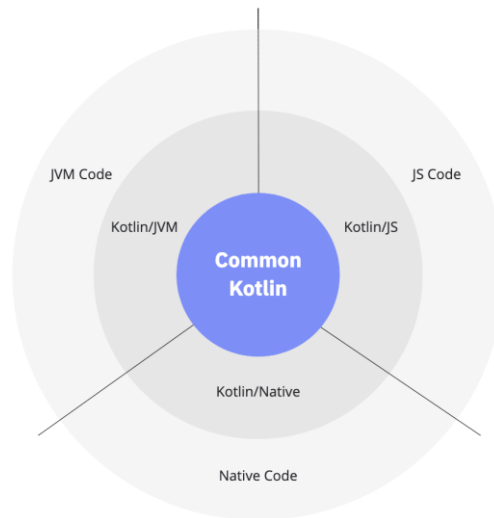


Рис. 1.3. Структура Kotlin Multiplatform

Common Kotlin - спільний код для всіх платформ. Він включає в себе основну логіку додатку, спільні бібліотеки та інструменти;

За допомогою мульти-платформних бібліотек Kotlin можна повторно використовувати мульти-платформну логіку в загальному і специфічному для платформи коді. Загальний код може покладатися на набір бібліотек, які охоплюють повсякденні завдання, такі як HTTP-запити, серіалізація та управління співпрограмами.

Для взаємодії з платформами використовуються версії Kotlin для конкретних платформ (Kotlin/JVM, Kotlin/JS, Kotlin/Native). Вони мають розширення мови Kotlin, а також бібліотеки та інструменти для конкретних платформ.

Переваги Kotlin Multiplatform:

- єдиний код, який можна в будь-який час дописувати і змінювати;
- час на додавання нового функціоналу для декількох платформ скоротився на 30-50%;
- поділ бізнес-логіки між платформами зі збереженням власного коду для кожного клієнтського інтерфейсу;

- помилки в разі виникнення з'являються відразу на обох платформах, що полегшує їх знаходження;
- будь-які зміни та коригування відбуваються одночасно на обох платформах.

Недоліки:

- фреймворк знаходиться на стадії розробки;
- не вистачає готових рішень;

1.3. Декларативне програмування

Декларативне програмування – це парадигма програмування, в якій програміст визначає, що має бути виконано програмою, не визначаючи, як це має бути реалізовано. Підхід фокусується на тому, чого необхідно досягти, замість того, щоб вказувати, як цього досягти. Це відрізняється від імперативної програми, яка має набір команд для вирішення певного набору проблем, описуючи кроки, необхідні для пошуку рішення. Декларативне програмування описує певний клас проблем з мовною реалізацією, що піклується про пошук рішення.

Декларативне програмування є домінуючою парадигмою великого і різноманітного набору областей, таких як: бази даних, створення шаблонів, управління конфігураціями, створення систем штучного інтелекту, автоматичного доказу теорем, розробки експертних систем та оболонок експертних систем, створення систем підтримки прийняття рішень, розробки систем обробки природної мови, побудови планів дій роботів та ін.

На практиці цей підхід передбачає надання специфічної для предметної області мови (domain-specific language, DSL) для вираження того, що хоче користувач, і захист її від низькорівневих конструкцій (циклів, умовних позначень, призначень), які матеріалізують бажаний кінцевий стан.

Порядок вираження операторів або реплікація оператора не мають жодного впливу на декларативне програмування. Декларативне

програмування можна додатково розділити на програмування обмежень, логічне програмування та програмування логіки обмежень. Програмістам в області декларативного програмування надаються інструменти, що дозволяють абстрагуватися від реалізації і допомогти в концентрації уваги на проблемі.

Переваги декларативного програмування:

1. **Читаємість та зручність використання:** DSL більш нагадує людську мову (наприклад, англійську), ніж до псевдокод, отже, краще читається і також легше засвоюється непрограммистами.
2. **Лаконічність:** більша частина шаблону абстрагується DSL, залишаючи менше рядків для виконання тієї ж роботи.
3. **Повторне використання:** простіше створювати код, який можна використовувати для різних цілей, що дуже складно при використанні імперативних конструкцій.
4. **Ідемпотентність:** можна працювати з кінцевими станами і дозволити програмі розібратися в цьому власноруч. Наприклад, за допомогою операції `upsert` можна або вставити рядок, якщо її там немає, або змінити її, якщо вона вже є, замість того, щоб писати код для вирішення обох випадків.
5. **Відновлення після помилок:** легко вказати конструкцію, яка зупиниться при першій помилці, замість того, щоб додавати прослуховувачі помилок для кожної можливої помилки.
6. **Довідкова прозорість:** хоча ця перевага, як правило, пов'язана з функціональним програмуванням, насправді вона застосовується до будь-якого підходу, який мінімізує ручну обробку стану та спирається на побічні ефекти.
7. **Комутативність:** можливість вираження кінцевого стану без необхідності вказувати фактичний порядок, в якому воно буде реалізовано [12; 13; 14].

Одним з прикладів використання декларативного програмування та DSL є UI фреймворк Jetpack Compose.

1.4. Фреймворк Jetpack Compose

Jetpack Compose – це фреймворк для створення інтерфейсу користувача для Android-додатків, та має адаптації під Desktop (Windows, Linux, macOS) та Web. Він спрощує і прискорює розробку UI, зменшує необхідну кількість коду для вирішення задач та має потужні інструменти і інтуїтивно зрозумілі API-інтерфейси Kotlin.

1.4.1. Принцип розділення завдань

Separation of concerns (розділення завдань) – це добре відомий принцип розробки програмного забезпечення. Корисно розглядати цей принцип в термінах Coupling (зв'язок) і Cohesion (цілісність).

Coupling – це залежність між блоками в різних модулях і відображення способів, якими частини одного модуля впливають на частини інших модулів.

Cohesion замість цього являє собою взаємозв'язок між одиницями всередині модуля і вказує, наскільки добре згруповані одиниці в модулі.

При написанні коду програмного забезпечення для якого потрібна постійна підтримка, необхідно мінімізувати зв'язаність і максимізувати його цілісність.

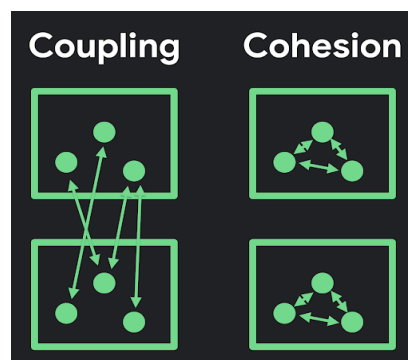


Рис. 1.4. Зв'язність та цілісність [15]

Коли є сильно пов'язані модулі, внесення змін до коду в одному місці означає необхідність внесення багатьох інших змін до інших модулів. Що ще гірше, зв'язок часто може бути неявним, так що несподівані речі ламаються через зміну, яка здається абсолютно не пов'язаною.

Separation of concerns полягає в тому, щоб згрупувати якомога більше зв'язаного коду разом, щоб код можна було легко підтримувати та масштабувати, коли додаток зростає.

1.4.2. *Composable функції*

Jetpack Compose заснований на функціональному підході до програмування. Для опису інтерфейсу користувача використовуються Composable функції.

Приклад Composable функції:

```
@Composable
fun App(appData: AppData) {
    val derivedData = compute(appData)
    Header()
    if (appData.isOwner) {
        EditButton()
    }
    Body {
        for (item in derivedData.items) {
            Item(item)
        }
    }
}
```

У цьому прикладі Composable функція отримує дані як параметри з класу appData та перетворює їх у елемент інтерфейсу користувача. Отже, при використанні будь-якого коду на Kotlin, обрані дані застосовують їх для опису ієрархії (наприклад: виклик Composable-функцій Header() і Body()).

Це означає, що при виклику інших Composable-функцій, вони відображають структуру нашого UI. Для використання надаються всі примітиви Kotlin, включаючи оператори if і цикли for для управління

структурою UI, щоб вирішувати більш складну логіку інтерфейсу користувача.

Також Composable-функції можуть приймати як параметр інші Composable-функції (наприклад: Body() приймає composable-лямбду у вигляді параметру), що означає ієрархію або структуру.

1.4.3. Переваги Jetpack Compose

Compose надає сучасний підхід до створення інтерфейсу користувача, який дозволяє ефективно розподіляти відповідальність в коді. Оскільки Composable функції дуже схожі на звичайні функції Kotlin, інструменти, можна використовувати ті ж самі інструменти для рефакторингу, що і для звичайного Kotlin-коду.

Переваги Jetpack Compose:

- Незалежність від конкретних версій цільової платформи.
- Вся робота з UI виконується за допомогою мови програмування Kotlin;
- Використання композиції замість спадкування. UI-компонент описується у вигляді функції з анотацією Composable, яка відповідають тільки за обмежений функціонал, тобто без зайвої логіки;
- Однонаправленість потоку даних;
- Зменшення кількості коду для UI-логіки.

Недоліки:

- Фреймворк знаходиться у стадії розробки;

1.5. Патерн MVI

Архітектурні шаблони – це схеми з набором правил, яким потрібно слідувати. Ці шаблони розвивалися через помилки, допущені при написанні коду протягом багатьох років.

Найбільш популярні архітектурні шаблони:

- Model View Controller (MVC)
- Model View Presenter (MVP)

- Model View ViewModel (MVVM)

Останнім доповненням до цих шаблонів є Model View Intent (MVI).

У MVI взаємодія з інтерфейсом користувача обробляється бізнес-логікою, що вносить зміни в стан, який впливає на відображення інтерфейсу користувача. Це призводить до односпрямованого і циклічного потоку даних.

За останній рік цей архітектурний шаблон привертає все більше уваги розробників. Він схожий на інші широко відомі шаблони, такі як MVP або MVVM, але він вводить дві нові концепції: Intent (намір) і State (стан).

Intent – це подія, відправлена у ViewModel через View для виконання певного завдання. Він може бути викликаний користувачем або іншими частинами додатку. В результаті цього на ViewModel встановлюється новий стан, який, в свою чергу, оновлює інтерфейс користувача. В архітектурі MVI View прослуховує стан. Кожен раз, коли стан змінюється, View отримує повідомлення.

intent(): функція, яка приймає вхідні дані від користувача (наприклад, події інтерфейсу користувача, такі як події click) і переводить в те, що буде передано як параметр функції model(). Це може бути простий рядок для установки значення моделі або більш складна структура даних, наприклад, об'єкт.

model(): функція, яка використовує вихідні дані з функції intent() як вхідні дані для роботи з моделлю. Результат роботи цієї функції-нова модель (зі зміненням станом). Функція model() - лише частина коду, відповідальна за створення нового об'єкта моделі, яка здійснює виклик бізнес-логіки додатку (будь-то Interactor, UseCase, Repository) і в результаті повертає новий об'єкт моделі.

view(): функція, яка отримує на вході модель від model() і просто відображає її. Зазвичай функція view() виглядає як view.render(model)

Для реалізації реактивного отримання оновлених даних, використовуються засоби Kotlin, такі як Flow, StateFlow та Channel.

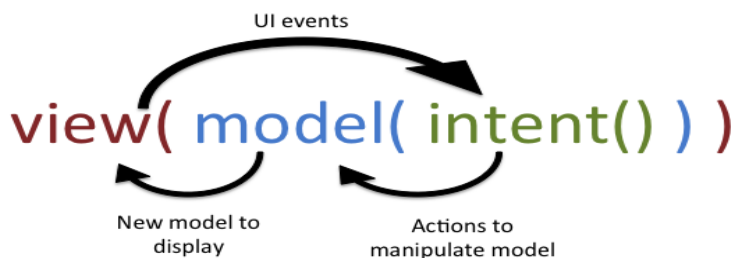


Рис. 1.5. Патерн Model-View-Intent [19]

Таким чином, View генерує Events за допомоги шаблону спостерігача, які знову передаються в функцію intent().

Модель як стан: інтерфейс користувача може мати різні стани - стан завантаження, стан даних, стан помилки, положення прокрутки користувача і т.д. У MVI моделі формалізуються як контейнер станів, чого не можна сказати про інші шаблони. Кожен раз створюється нова незмінна модель, яка потім спостерігається в поданні. Такий спосіб створення незмінною моделі забезпечує потокобезпечність.

Єдине джерело істини: кожен компонент може мати свій власний стан. В якості компонента маються на увазі View і Presenter / ViewModel. Збереження окремих станів на різних рівнях може призвести до конфліктів між станами. Тому, щоб уникнути подібних проблем, стан створюється на одному рівні (Presenter / ViewModel) і передається на інший рівень (View) в MVI. Єдиний спосіб змінити стан - запустити взаємодію з користувачем. Такий підхід обмежує будь-яку зміну стану системи тільки певним набором дій. Невизначена дія користувача не може привести до будь-яких небажаних змін в системі.

Відсутність зворотних викликів: зазвичай подання оновлюються за допомогою зворотних викликів після асинхронного виклику. Подання може бути відключено, коли асинхронний виклик призведе до результату. Тому, щоб уникнути збою або неправильної поведінки програми, необхідно перевірили, чи доступний екземпляр View і прикріплений він. Ці перевірки та зворотні виклики обробляються автоматично в MVI за допомогою реактивного програмування, що підтримує парадигму спостерігача.

Пошук збою: ви можете зіткнутися з ситуаціями, коли дуже складно відстежити і відтворити збій, навіть якщо є звіт про збій. Звіт про збій може містити трасування потоку коду, але не містить трасування потоку станів подання перед збоєм. У MVI відстеження збою стає простим за допомогою стану (Model). Будь-який розробник може відтворити збій за допомогою трасування стану і легко виправити його.

Повернення стану: хороший додаток повинен працювати без будь-яких збоїв навіть в непередбачуваних обставинах. Якщо орієнтація змінюється або процес активності переривається під час телефонного дзвінка, активність Android відтворюється. Відновлення стану було складним завданням у таких ситуаціях. Бібліотека MVI від Mosby підтримує поточний стан і відновлює його при повторному створенні подання.

Незалежні компоненти інтерфейсу користувача: Кожен архітектурний шаблон вчить нас, як компоненти повинні бути побудовані без будь-яких залежностей. View / Presenter не повинні бути пов'язані один з одним. Відповідальність View і Presenter полягає в тому, щоб просто відображати вміст і зіставляти дані для перегляду відповідно.

Мульти-платформний стандарт: реактивне програмування - це мульти-платформний стандарт. Реактивні коди, написані для Android (Kotlin), можуть бути використані на iOS (Swift) і навпаки, що спрощує перевірку коду.

Переваги MVI:

- Підтримка стану більше не є проблемою з цією архітектурою, оскільки вона фокусується в основному на станах.
- Оскільки MVS односпрямований, потік даних можна легко відстежувати і прогнозувати.
- Забезпечує потокобезопасність, так як об'єкти стану незмінні.
- Легко налагоджується, так як відомо стан об'єкта в момент виникнення помилки.

- Більш роз'єднаний, оскільки кожен компонент виконує свою власну відповідальність.
- Тестування програми також простіше, так як можна зіставити бізнес-логіку для кожного стану.

Недоліки MVI:

- Призводить до великої кількості шаблонного коду, так необхідно підтримувати стан для кожної дії користувача.
- Необхідно створювати безліч об'єктів для всіх станів. Це робить управління пам'яттю додатків занадто дорогим [19].

1.6. Шаблон Dependency Injection

Dependency Injection (впровадження залежностей) – це шаблон проєктування, при якому об'єкт отримує інші об'єкти, від яких він залежить. Як правило, приймаючий об'єкт називається клієнтом, а переданий об'єкт називається сервісом. Код, який передає сервіс клієнту, називається інжектором. Замість того, щоб клієнт вказував, який сервіс він буде використовувати, інжектор повідомляє клієнту, який сервіс використовувати. Впровадження відноситься до передачі залежності (сервісу) клієнту, який її використовує.

Сервіс стає частиною стану клієнта. Передача сервісу клієнту, замість дозволу клієнту створювати або знаходити сервіс, є основною вимогою шаблону.

Мета впровадження залежностей полягає в тому, щоб отримати поділ завдань на створення та використання об'єктів. Це може підвищити читабельність і повторне використання коду.

Впровадження залежностей є однією з форм більш широкої техніки інверсії контролю. Клієнт, який хоче запросити деякі сервіси, не повинен знати, як створювати ці сервіси. Замість цього клієнт делегує зовнішній код (інжектор). Клієнт не знає про інжектор. Інжектор передає клієнту сервіси, які

можуть існувати або бути створені самим інжектором. Потім клієнт користується отриманими сервісами.

Це означає, що клієнту не потрібно знати про інжектор, про те, як створювати сервіси, або навіть про те, які сервіси він фактично використовує. Клієнтові потрібно лише знати інтерфейси сервісів, оскільки вони визначають, як клієнт може використовувати сервіси. Це відокремлює відповідальність за використання від відповідальності за створення.

Впровадження залежностей вирішує наступні проблеми:

- Як клас може бути незалежним від того, як створюються об'єкти, від яких він залежить?
- Як можна вказати спосіб створення об'єктів в окремих файлах конфігурації?
- Як додаток може підтримувати різні конфігурації?

Створення об'єктів безпосередньо в класі прив'язує клас до певних реалізацій. Це ускладнює зміну екземпляра під час виконання, особливо в компільованих мовах, де зміна базових об'єктів може вимагати повторної компіляції вихідного коду.

Впровадження залежностей відокремлює створення залежностей клієнта від поведінки клієнта, що сприяє розвитку слабо пов'язаних програм і принципів інверсії залежностей і єдиної відповідальності. По суті, впровадження залежностей засноване на передачі параметрів методу.

Впровадження залежностей є прикладом більш загальної концепції інверсії управління.

Прикладом інверсії управління без впровадження залежностей є метод шаблону, де поліморфізм досягається за рахунок підкласів. Впровадження залежностей реалізує інверсію управління за допомогою композиції і часто аналогічно шаблону стратегії. Хоча шаблон стратегії призначений для залежностей, які взаємозамінні протягом усього терміну сервісу об'єкта, при впровадженні залежностей може використовуватися тільки один екземпляр

залежності. Це все ще забезпечує поліморфізм, але за рахунок делегування і композиції.

Впровадження залежностей безпосередньо контрастує з шаблоном пошуку служб, який дозволяє клієнтам знати про систему, яку вони використовують для пошуку залежностей [20].

Переваги шаблону:

- Не вимагає змін у поведінці коду, тому його можна застосовувати як рефакторинг. В результаті цього клієнти стають більш незалежними і над ними легше проводити модульне тестування в ізоляції з використанням макетів об'єкта, які імітують інші об'єкти, від яких залежить об'єкт, що тестується.
- Не вимагає від клієнта знань про конкретну реалізацію, яку йому потрібно використовувати. Це дозволяє ізолювати клієнта від впливу змін проєктування і дефектів, що сприяє повторному використанню, тестуванню і підтримці коду.
- Може використовуватися для перенесення деталей конфігурації системи в конфігураційні файли, що дозволяє системі змінювати конфігурацію без перекомпіляції. Окремі конфігурації можуть бути написані для різних ситуацій, що вимагають різних реалізацій компонентів.
- Впровадження залежностей сприяє паралельній і незалежній розробці.
- Знижує зв'язність між класом і його залежностями.

1.7. Висновки до розділу 1

У першому розділі проведено аналіз фреймворків крос-платформної розробки, описано їх переваги та недоліки. Досліджено фреймворк для розробки мульти-платформних додатків - Kotlin Multiplatform. Розглянуто парадигму декларативного програмування та досліджено фреймворк для створення інтерфейсу користувача – Jetpack Compose, як приклад

використання декларативного підходу. Описано переваги та недоліки декларативного програмування та фреймворку Jetpack Compose. Розглянуто патерн MVI, який найкраще підходить для опису логіки інтерфейсу користувача при використанні Jetpack Compose. Розглянуто шаблон проєктування Dependency Injection.

РОЗДІЛ 2

ЗАСОБИ ПРОЄКТУВАННЯ ДОДАТКУ

Для розробки мульти-платформних додатків використовуються інструменти та бібліотеки з підтримкою Kotlin Multiplatform, такі як: мова програмування Kotlin, Kotlin Coroutines, UI-фреймворк Jetpack Compose, патерн MVI, бібліотеки SQLDelight та Koin.

2.1. Мова програмування Kotlin

Kotlin – кроссплатформенна, статично типізована мова програмування загального призначення з підтримкою виводу типів. Kotlin розроблений для повної взаємодії з Java, де версія стандартної бібліотеки Kotlin для JVM залежить від бібліотеки класів Java, але тип взаємодії дозволяє зробити його синтаксис більш коротким і лаконічним. Kotlin в основному орієнтований на JVM, але також компілюється на JavaScript (наприклад, для інтерфейсних Web-додатків з використанням React) або власного коду (через LLVM), наприклад, для власних додатків iOS, що використовують бізнес-логіку спільно з додатками Android. Витрати на розробку мови несе JetBrains, в той час як Kotlin Foundation захищає торгову марку Kotlin.

7 травня 2019 року Google оголосила, що мова програмування Kotlin тепер є рекомендованою мовою для розробників додатків для Android. З моменту випуску Android Studio 3.0 в жовтні 2017 року, Kotlin був включений в якості альтернативи стандартному компілятору Java. Компілятор Android Kotlin за замовчуванням створює байт-код Java 8 (який запускається в будь-якій пізнішій JVM), але дозволяє програмісту вибирати для оптимізації Java 9 до 16, або надає додаткові функції, а також має підтримку взаємодії класів двонаправлених записів для JVM, введену в Java 16, яка вважається стабільною з версії Kotlin 1.5.

Підтримка Kotlin для компіляції безпосередньо в JavaScript (тобто класичний серверний інтерфейс) вважається розробниками стабільною з версії Kotlin 1.3, в той час як новий Kotlin / JS (IR) знаходиться у беті з версії 1.5.30. Нові оптимізовані реалізації Kotlin / JVM (IR) і Kotlin / JS (на основі IR) були представлені у версії 1.4. Kotlin / JVM (IR) вважається стабільним з версії 1.5. Kotlin / Native (наприклад, підтримка Apple silicon) вважається бета-версією з версії 1.3 [21].

2.2. Бібліотека Kotlin Coroutines

Асинхронне або неблокуюче програмування є важливою частиною розробки. При створенні серверних, настільних або мобільних додатків важливо забезпечити не тільки гнучкість з точки зору користувача, але і масштабованість, коли це необхідно. Паралельні обчислення дозволяють виконувати кілька завдань одночасно, а асинхронність дозволяє не блокувати основний хід програми під час виконання завдання, яка займає тривалий час.

У Kotlin підтримка асинхронності і паралельних обчислень втілена у вигляді корутин (coroutine). Корутина являє собою блок коду, який може виконуватися паралельно з іншим кодом. Базова функціональність, пов'язана з корутинами, зосереджена в бібліотеці `kotlin.coroutines` [22; 23].

Приклад створення корутини:

```
val scope = CoroutineScope(Dispatchers.Main)
scope.launch {
    print("Start")
    delay(5000)
    print("End")
}
```

Переваги використання корутин:

- Витрачають дуже малу частину ресурсів процесора, у порівнянні зі звичайними потоками;
- Працюють швидше звичайних потоків;
- Займають менше місця у коді.

2.3. Система збірки проєктів Gradle Kotlin DSL

Gradle – це інструмент автоматизації збірки для розробки багатомовного програмного забезпечення. Він контролює процес розробки в завданнях компіляції та упаковки проєктів для їх тестування, розгортання та публікації. Підтримувані мови включають Java (а також Kotlin, Groovy, Scala), C/C++ та JavaScript. Іншою, якщо не основною функцією Gradle, є збір статистичних даних про використання бібліотек програмного забезпечення по всьому світу.

Gradle ґрунтується на концепціях Apache Ant і Apache Maven, та представляє доменну мову на основі Groovy і Kotlin, на відміну від конфігурації проєкту на основі XML, використовуваної Maven. Gradle використовує орієнтований ациклічний граф для визначення порядку, в якому можуть виконуватися завдання, за допомогою забезпечення управління залежностями. Gradle працює на JVM.

Gradle був розроблений для складання проєктів, які можуть вирости до великих розмірів. Він працює на основі ряду завдань збірки, які можуть виконуватися послідовно або паралельно. Інкрементні збірки підтримуються шляхом визначення частин дерева збірки, які вже оновлені. Будь-яке завдання, що залежить тільки від цих частин, не вимагає повторного виконання. Він також підтримує кешування компонентів збірки. Вміє створювати Web-візуалізацію збірки, що називається скануванням збірки Gradle. Програмне забезпечення розширюється для нових функцій і мов програмування за допомогою підсистеми плагінів.

DSL Kotlin від Gradle надає альтернативний синтаксис традиційному DSL Groovy з поліпшеними можливостями редагування в підтримуваних IDE, з чудовою підтримкою контенту, рефакторингом, документацією і багатьом іншим [24; 25].

2.4. Фреймворк Compose Multiplatform

Compose Multiplatform – фреймворк для створення швидких і реактивних інтерфейсів для Android, Desktop та Web-додатків на Kotlin. Він базується на тому ж декларативному підході та API, що використовуються для сучасних додатків Android, для створення користувацьких інтерфейсів для настільних і браузерних додатків на базі Kotlin/JVM і Kotlin/JS.

Фреймворк складається з трьох частин:

- Jetpack Compose – фреймворк від Google для розробки інтерфейсів для Android-додатків;
- Compose for Desktop – адаптація Jetpack Compose для Desktop-додатків;
- Compose for Web – адаптація Jetpack Compose для Web-додатків.

Compose for Desktop націлений на JVM і використовує потужну нативну графічну бібліотеку Skia для підтримки високопродуктивного апаратно прискореного UI-рендеринга на основних десктопних платформах (macOS, Windows і Linux).

Composer for Web дозволяє створювати реактивні інтерфейси для Web-додатків на Kotlin, використовуючи концепти і API Jetpack Compose для опису стану, поведінки і логіки програми. Compose for Web пропонує різні способи оголошення користувацьких інтерфейсів в коді Kotlin. Це дозволяє вам повністю контролювати верстку Web-сайту за допомогою декларативного DOM API.

Раніше Compose for Desktop і Compose for Web використовували окремі набори артефактів. За допомогою механізмів, наданих Kotlin Multiplatform, у альфа-версії Compose Multiplatform частини фреймворку стали об'єднані під одним плагіном Gradle і групою артефактів. Це спростило і прискорило розробку інтерфейсів додатків завдяки використанню єдиного коду під декілька платформ одночасно [26; 27; 28].

2.5. Бібліотека Decompose

Decompose - це бібліотека, яка містить у собі набір мульти-платформних компонентів бізнес-логіки з урахуванням життєвого циклу (також відомі як BLoCs) [29] з функціями маршрутизації та підмикним UI (Android Views, Jetpack Compose, SwiftUI, JS React, тощо) [30].

Бібліотека підтримує наступні платформи:

- Android;
- JVM;
- JS;
- iosX64, iosArm64;
- tvosArm64, tvosX64;
- watchosArm32, watchosArm64, watchosX64;
- macosX64.

Decompose проводить чіткі межі між UI та кодом бізнес-логіки додатку, що дає наступні переваги:

- Краще розділення відповідальностей (Separation of Concerns);
- Підмикний UI для конкретної платформи (Compose, Swift UI, React і т. д.);
- Код бізнес-логіки можна протестувати за допомогою чистого мульти-платформного модульного тесту;
- Правильне впровадження залежностей (DI) і інверсія управління (IoC) за допомогою конструктору;
- Спільна логіка навігації;
- Компоненти, що враховують життєвий цикл;
- Компоненти у backstack не знищуються, вони продовжують працювати у фоновому режимі без інтерфейсу користувача
- Компоненти зберігають стан UI (використовується у Android);
- Збереження екземплярів при зміні конфігурації (використовується у Android) [31].

2.6. Фреймворк MVIKotlin

MVIKotlin – це фреймворк для Kotlin Multiplatform, який надає спосіб написання спільного коду з використанням шаблону MVI. Він також включає в себе потужні інструменти налагодження, такі як логування та “подорож у часі”.

MVIKotlin не приносить і не застосовує жодної конкретної архітектури. Його відповідальність описується наступним чином:

- Надає єдине джерело істини для стану;
- Забезпечує абстракцію для UI з ефективними оновленнями;
- Забезпечує прив’язку до життєвого циклу між входами та виходами.

Основою MVIKotlin є Store, що представляє собою модель від MVI, яка містить бізнес-логіку.

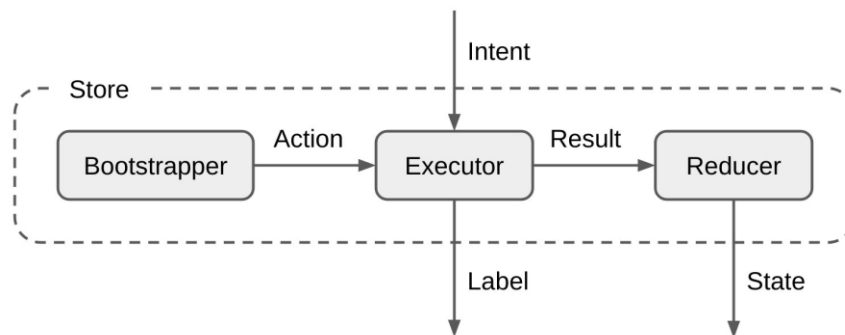


Рис. 2.1. Діаграма Store [33]

Store має такі компоненти:

- **Intent** – вхідна подія, яка приходить з UI;
- **Label** – одноразова вихідна подія, яка повертається до UI;
- **State** – стан, який відображається на UI;
- **Bootstrapper** – компонент, який виконує ініціалізацію *Store*, та при необхідності відправляє події до *Executor*;
- **Action** – події, які відправляє *Bootstrapper* до *Executor*;

- **Executor** – виконавець бізнес-логіки та усіх асинхронних операцій. Під час виконання або після завершення операцій, Executor може відправити *Label* до UI та *Result* до *Reducer*;
- **Result** – результат виконання операції, який передається до *Reducer*;
- **Reducer** – компонент, який приймає *Result* та змінює *State* [32; 33].

2.7. Бібліотека SQLDelight

У світі розробки програмного забезпечення бази даних часто використовуються для локального зберігання даних на клієнтських пристроях. Одним з варіантів роботи з базами даних в проєктах Kotlin Multiplatform є бібліотека SQLDelight. Вона генерує типобезпечні API-інтерфейси Kotlin з операторів SQL для різних реляційних баз даних. SQL Delight також надає мульти-платформну реалізацію драйвера SQLite .

SQLDelight перевіряє схему, інструкції та міграції під час компіляції та надає функції IDE, такі як автозаповнення та рефакторинг, що спрощують написання та підтримку SQL операцій [34; 35].

Для початку роботи з SQLDelight необхідно зробити наступні налаштування:

1. Додати classpath у gradle файл проєкту:

```
buildscript {
    repositories {
        google()
        mavenCentral()
    }
    dependencies {
        classpath 'com.squareup.sqldelight:gradle-plugin:1.5.0'
    }
}
```

2. Підключити плагін, додати блок налаштувань SQLDelight та додати залежності драйверів SQLDelight для різних платформ;

```
apply plugin: 'com.squareup.sqldelight'

sqldelight {
    Database {
        packageName = "com.example"
    }
}
```

```
kotlin {
    sourceSets.androidMain.dependencies {
        implementation "com.squareup.sqldelight:android-driver:1.5.0"
    }
    sourceSets.nativeMain.dependencies {
        implementation "com.squareup.sqldelight:native-driver:1.5.0"
    }
    sourceSets.jvmMain.dependencies {
        implementation "com.squareup.sqldelight:sqlite-driver:1.5.0"
    }
}
```

3. Помістити всі оператори SQL у файл типу .sq в розділі src/commonMain/sqldelight. Зазвичай перша інструкція у файлі SQL створює таблицю.

```
--src/commonMain/sqldelight/com/example/sqldelight/hockey/data/Player.sql
```

```
CREATE TABLE hockeyPlayer (
    player_number INTEGER NOT NULL,
    full_name TEXT NOT NULL
);
```

```
CREATE INDEX hockeyPlayer_full_name ON hockeyPlayer(full_name);
```

```
INSERT INTO hockeyPlayer (player_number, full_name)
VALUES (15, 'Ryan Getzlaf');
```

Оператори SQL всередині файлу .sq можуть бути позначені так, щоб для них була створена типобезпечна функція, доступна під час виконання.

```
selectAll:
SELECT *
FROM hockeyPlayer;
```

```
insert:
INSERT INTO hockeyPlayer(player_number, full_name)
VALUES (?, ?);
```

```
insertFullPlayerObject:
INSERT INTO hockeyPlayer(player_number, full_name)
VALUES ?;
```

Файли з позначеними операторами матимуть згенерований файл запитів, який відповідає імені файлу .sq.

Приклад використання згенерованого коду:

```
val database = Database(driver)

val playerQueries: PlayerQueries = database.playerQueries
println(playerQueries.selectAll().executeAsList())
playerQueries.insert(player_number = 10, full_name = "Corey Perry")
println(playerQueries.selectAll().executeAsList())
val player = HockeyPlayer(10, "Ronald McDonald")
playerQueries.insertFullPlayerObject(player)
```

2.8. Бібліотека Koin

Існує декілька бібліотек та фреймворків, які реалізують принцип інверсії залежностей, такі як: Dagger 2 [36], Katana [37], Kodein [38], Koin [39]. Серед них тільки Koin та Kodein підтримують Kotlin Multiplatform.

Koin - це невелика бібліотека для написання впровадження залежностей. Працює як Service Locator, використовує DSL та особливості мови Kotlin. Koin є мульти-платформною бібліотекою та використовується для налагодження зв'язку між модулями додатку [40; 41].

Для запуску бібліотеки необхідно викликати функцію `startKoin` та передати контекст зі списком модулів залежностей. Нижче наведено код ініціалізації Koin у Android:

```
class App : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@App)
            modules(firstModule, secondModule)
        }
    }
}
```

Koin має три види тимчасових областей:

- `single` (одиначний об'єкт) – створюється об'єкт, який зберігається протягом усього періоду роботи додатку (аналогічно синглтону);
- `factory` (фабрика об'єктів) – кожен раз створюється новий об'єкт, без збереження в контейнері (спільне використання неможливо);
- `scoped` (об'єкт в області) – створюється об'єкт, який зберігається в рамках періоду існування пов'язаної тимчасової області.

Приклад модуля Koin:

```
val myModule = module {
    single { Controller(get()) }
    factory { BusinessService() }
}
```

Існує декілька способів виконати ін'єкцію залежності:

- Додати інтерфейс `KoinComponent` до класу та викликати функцію `get()`;
- Викликати функцію `inject()` у компонентах Android (Activity, Fragment, Service);
- Зробити ін'єкцію компонента Koin через функцію `get()` та викликати у нього функцію `get()`.

Приклад ін'єкції залежності:

```
class TestClass: KoinComponent {  
    val otherClass: OtherClass by inject() }
```

2.9. Висновки до розділу 2

У другому розділі описано засоби проєктування мульти-платформних додатків, такі як: мова програмування Kotlin, бібліотека Kotlin Coroutines, система збірки проєктів Gradle Kotlin DSL, фреймворк Compose Multiplatform, бібліотека Decompose, фреймворк MVIKotlin, бібліотека SQLDelight та бібліотека Koin.

РОЗДІЛ 3

ПРОЄКТУВАННЯ ЗАСТОСУНКУ

Застосунок “Arranger” дозволяє створювати мелодії у форматі на зразок midi та проводити з ними різні дії: імпорт, експорт, видалення, редагування та відтворення. Також додаток має розділ для гри на музичних інструментах (music pad) та меню налаштувань.

Згідно поставленого завдання, було розроблено структуру додатку, яка складається з ієрархії компонентів.

Головним компонентом додатку є Root-компонент. Він містить у собі компонент головного меню (Main Menu) та компоненти допоміжних екранів (Add Music, Create Song, Record Song, Import Song, Song Editor, Track Editor, Midi Converter, Player).

Root компонент необхідний для ініціалізації інших компонентів, відображення тимчасових повідомлень та для налагодження навігації між головним меню та додатковими екранами додатку.

3.1. Головне меню додатку - Main Menu

Головне меню виконує роль навігатора між компонентами-розділами додатку та допоміжними екранами.

До розділів відносяться:

- Music - екран зі списком мелодій;
- Instruments - екран для гри на музичних інструментах.
- Settings - меню налаштувань.

Також у головному меню є кнопка, яка відкриває екран Add Song.



Рис. 3.1. Головне меню додатку

3.1.1. Розділ Music

Розділ Music використовується для роботи з мелодіями, плейлистами та семплами. Він складається з трьох частин:

- All Songs - список всіх мелодій додатку;
- Playlists - список плейлистів додатку;
- Samples - список семплів, які використовуються для створення нових мелодій.

Функції компоненту над елементами списку:

- пошук;
- сортування;
- перехід до редагування;
- можливість поділитися елементом списку через месенджер або соц. мережу;
- завантаження на пристрій у вигляді файлу;
- перегляд детальної інформації (назва, час створення та інше);
- видалення.

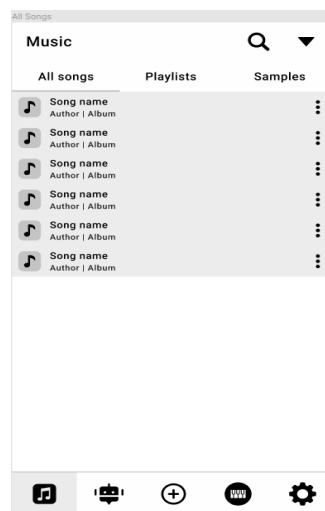


Рис. 3.2. Компонент Music

3.1.2. Розділ Instruments

Розділ Instruments використовується для гри на музичних інструментах за допомогою drum pad кнопок. Кожна кнопка відповідає своїй ноті, без урахування півтонів. Кожні сім кнопок утворюють октаву.

Складові розділа:

- Випадаюче меню для вибору інструмента;
- Випадаюче меню для вибору тональності;
- Drum pad кнопки інструменту.

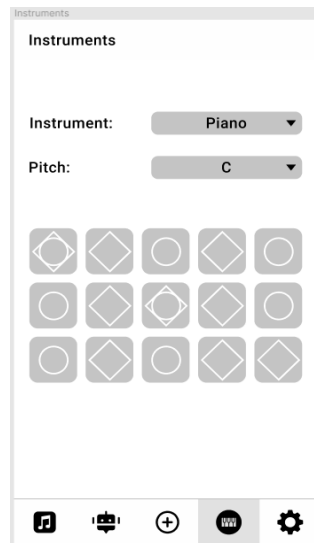


Рис. 3.3. Компонент Instruments

3.1.3. Розділ Settings

Розділ налаштувань додатку використовується для встановлення дозволів роботи з файловою системою та додатковими сервісами, для зміни зовнішнього вигляду та мови додатку, а також для виведення контактів зворотного зв'язку.

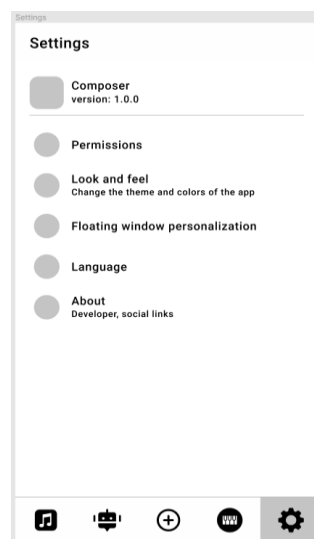


Рис. 3.4. Компонент Settings

3.2. Компонент Add Music

Компонент Add Music відповідає за створення та імпорт нових мелодій, плейлистів та семплів. Також є можливість конвертації midi мелодії у формат додатку та запис семплу у реальному часі.

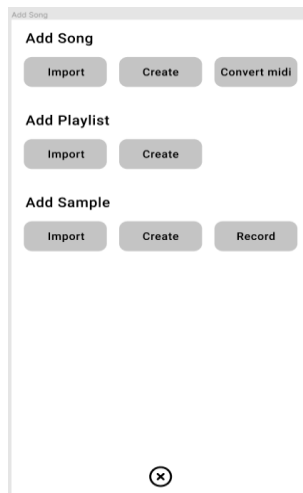


Рис. 3.5. Компонент Add Song

3.3. Компонент Create Song

Компонент Add Song відповідає за створення нової пустої мелодії. Для цього необхідно ввести назву мелодії та необов'язкові дані о композиторі, альбомі та перекладачі.

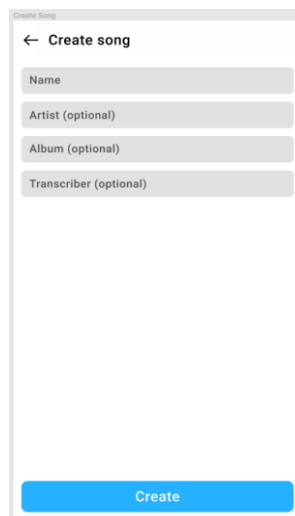


Рис. 3.6. Компонент створення нової пустої мелодії

3.4. Компонент Record Sample

Компонент Record Sample використовується для запису семплів.

Функції компоненту:

- зміна інструменту та музичного тону;
- старт, пауза та збереження запису;
- запис послідовності натиску кнопок інструменту.

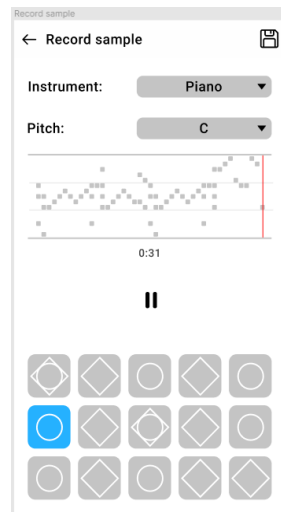


Рис. 3.7. Компонент Record Sample

3.5. Компонент Import Song

Компонент Import Song виконує функцію імпорту мелодій. Він використовує стандартні засоби операційної системи для вибору одного чи декількох файлів мелодії. Після вибору файлів, мелодії обробляються в компоненті та зберігаються в базі даних.

3.6. Компонент Song Editor

Компонент Song Editor використовується для роботи з треками мелодій.

Функції компоненту:

- створення та видалення треків;
- використання семплів для створення нових треків;
- редагування інформації о мелодії;
- редагування налаштувань треків (інструмент, музичний тон та BPM);
- прослуховування мелодії та збереження змін;

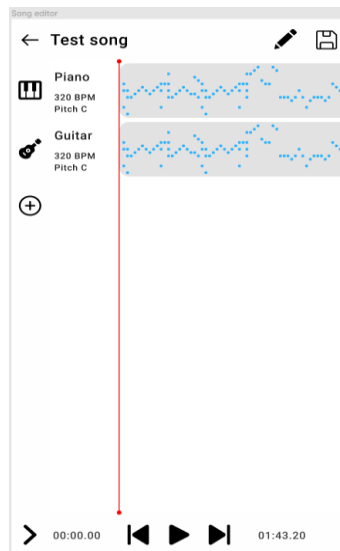


Рис. 3.8. Компонент Song Editor

3.7. Компонент Track Editor

Компонент Track Editor використовується для редагування треків мелодії.

Функції компоненту:

- переключення між треками;
- прослуховування та історія змін треку;
- додавання та видалення нот;
- копія, вирізання, видалення акордів;
- режим прив'язки до найближчого біту.

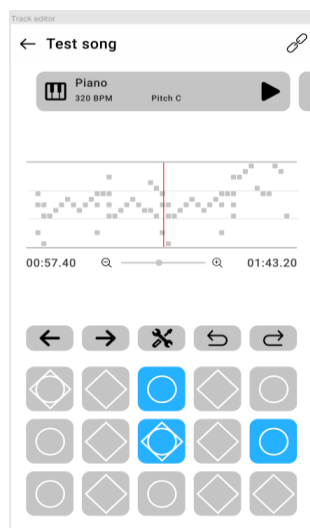


Рис. 3.9. Компонент Track Editor

3.8. Компонент Midi Converter

Компонент Midi Converter використовується для створення нових мелодій з існуючих мелодій у форматі midi файлів.

Функції компоненту:

- зміна BPM;
- переключення шарів midi мелодії;
- додавання та видалення треків інструментів на шарах midi мелодії;
- збереження семплів з треків;
- прослуховування та збереження треків інструментів у нову мелодію додатку.

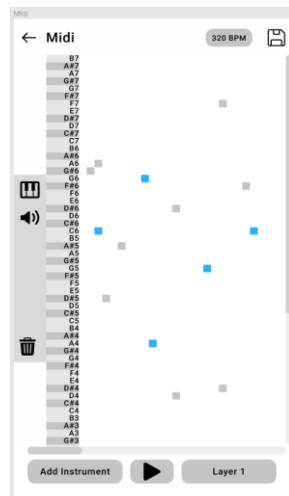


Рис. 3.10. Компонент Midi Converter

3.9. Компонент Player

Компонент Player використовується для прослуховування існуючих мелодій додатку.

Функції компоненту:

- відтворення мелодії та пауза;
- переключення мелодії на наступну чи попередню у списку;
- перехід до редагування мелодії;
- режим циклічного прослуховування мелодії або списку мелодій;

- відтворення мелодій у випадковій послідовності.

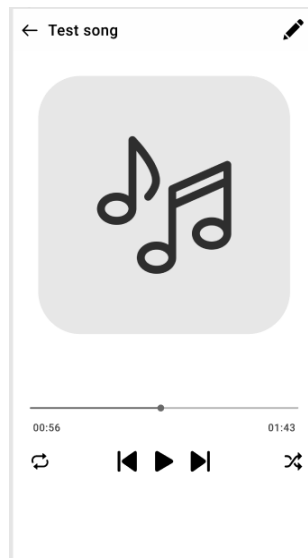


Рис. 3.11. Компонент Player

3.10. Висновки до розділу 3

У третьому розділі представлено результати проектування додатку “Arranger”, який можна використовувати для відтворення, створення та редагування музичних мелодій у власному форматі.

РОЗДІЛ 4

РОЗРОБКА ЗАСТОСУВАННЯ

4.1. Налаштування системи збірки проєктів

Для роботи з мульти-платформними модулями, проведено налаштування системи збору проєктів Gradle.

Список модулів додатку та репозиторії залежностей описані у файлі `settings.gradle.kts` в корені проєкту:

```
pluginManagement {
    repositories {
        google()
        jcenter()
        gradlePluginPortal()
        mavenCentral()
    }
}
rootProject.name = "Arranger"

include(":android")
include(":desktop")
include(":common")
```

Налаштування Gradle описані у файлі `gradle.properties`:

```
kotlin.code.style=official
android.useAndroidX=true

kotlin.native.enableDependencyPropagation=false
kotlin.mpp.stability.nowarn=true

org.gradle.jvmargs=-Xms512M -Xmx4g -XX:MaxPermSize=1024m -
XX:MaxMetaspaceSize=1g -Dkotlin.daemon.jvm.options="-Xmx1g"
```

Налаштування Gradle проєкту виконано у файлі `build.gradle.kts` у корені проєкту:

```
buildscript {
    dependencies {
        classpath("org.jetbrains.compose:compose-gradle-plugin:1.0.0-beta1")
    }
}
```

```

        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.31")
        classpath("com.android.tools.build:gradle:7.1.0-beta02")
        classpath("com.squareup.sqldelight:gradle-plugin:1.5.0")
    }
}
group = "com.zhukovartemvl.Arranger"
version = "0.0.1"

```

Налаштування спільного модуля додатку виконано у файлі build.gradle.kts модуля common (Додаток А).

Налаштування Gradle android додатку виконані у файлі build.gradle.kts модуля android (Додаток Б).

Налаштування Gradle desktop додатку виконані у файлі build.gradle.kts модуля desktop (Додаток В).

4.2. Розробка архітектури додатку

Архітектура проєкту поділена на дві частини:

- common модуль - містить у собі основну логіку додатку, а також платформні реалізації компонентів;
- платформні модулі - виконують ініціалізацію та запуск додатку на певній платформі.

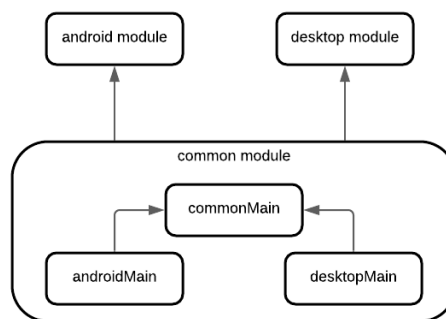


Рис. 4.1. Модульна структура додатку

4.2.1. Common модуль

Модуль спільної логіки складається з 5 частин:

- core пакет - містить класи типів даних додатку;

- data пакет - реалізує роботу з базою даних та надає класи-репозиторії для доступу до даних з feature пакету;
- feature пакет - містить компоненти-екрани додатку з бізнес-логікою та UI частиною, а також описує навігацію між компонентами;
- di пакет - виконує налаштування dependency injection у додатку;
- база даних SQL Delight.

4.2.2. Платформні модулі

Платформні модулі виконують ініціалізацію головного компонента додатку зі спільного модуля, вмикають DI та запускають додаток.

Ініціалізація android додатку виглядає наступним чином:

- Вмикається dependency injection у класі App:

```

        androidContext(this@App)
class App : Application() {
    override fun onCreate() {
        super.onCreate()
        initKoin {
            androidContext(this@App)
        }
    }
}

```

- Створюється головний компонент та використовується при створенні головного вікна додатку:

```

class MainActivity : AppCompatActivity() {
    @ExperimentalDecomposeApi
    @ExperimentalComposeUiApi
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val component = RootComponentProvider(
            componentContext = defaultComponentContext(),
            storeFactory = DefaultStoreFactory()
        )
        setContent {
            RootScreen(component)
        }
    }
}

```

Ініціалізація desktop додатку виконується у main функції додатку:

```

fun main() {
    initKoin()
    val lifecycle = LifecycleRegistry()
    val root = RootComponentProvider(
        componentContext = DefaultComponentContext(lifecycle),
        storeFactory = DefaultStoreFactory()
    )
    application {
        LifecycleController(lifecycle, rememberWindowState())
        Window(onCloseRequest = ::exitApplication, state = windowState, title =
"Arranger") {
            RootScreen(component = root)
        }}}

```

4.3. Створення компонентів екранів

Кожен екран додатку складається з файлу `Screen`, інтерфейсу `Component`, класу `ComponentProvider`, та за необхідністю, інтерфейсу `Store` та його реалізацією `StoreProvider`.

Файл `Screen` містить у собі декілька `Composable` функцій. Головною з них є функція `Screen`, яка приймає об'єкт типу `Component` та додаткові параметри (Додаток Г):

```

@Composable
fun TestScreen(component: TestComponent, typeOfSort: TypeOfSort) {
    val model by component.model.subscribeAsState()
    LazyColumn {
        itemsIndexed(items = model.songs) { index, song ->
            Song(song = song)
        }}}

```

Навігація між екранами виконується через заміну дочірнього компоненту у `Composable` функції `Children`:

```

Children(
    routerState = component.routerState,
    animation = crossfade()
) {
    when (val child = it.instance) {

```

```

        is MusicComponent.Child.AllMusic -> AllMusicScreen(
            component = child.component,
            typeOfSort = model.currentTypeOfSort,
            searchFilter = searchFilter
        )
        is MusicComponent.Child.AllPlaylists ->
AllPlaylistsScreen(child.component)
        is MusicComponent.Child.AllSamples -> AllSamplesScreen(child.component)
    }}

```

Інші Composable функції у файлі Screen необхідні для створення кастомних View для інтерфейсу користувача, що використовуються у Composable функції Screen.

```

@Composable
private fun Song(song: SongShortInfoDto) {
    Card(modifier = Modifier.fillMaxWidth().height(48.dp)) {
        Row(
            modifier = Modifier.fillMaxWidth(),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Box(modifier = Modifier
                .size(40.dp)
                .padding(4.dp)
                .background(color = Color.Gray, shape = RoundedCornerShape(8.dp)))
            Column(modifier = Modifier.weight(1f)) {
                Text(text = song.name)
                Text(text = song.artist + " - " + song.length)
            }
        }
    }
}

```

У інтерфейсі Component описується контракт взаємодії Composable функції Screen та бізнес-логіки компонента (Додаток Д):

```

interface TestComponent {
    val model: Value<TestStore.State>

    fun navPlaySong(songId: Long)
    fun navEditSong(songId: Long)

    fun sendIntent(intent: TestStore.Intent)
}

```

Реалізація контракту `Component` виконується у класі `ComponentProvider` (Додаток Е):

```
class TestComponentProvider(
    componentContext: ComponentContext,
    storeFactory: StoreFactory,
    private val navPlaySong: (songId: Long) -> Unit,
    private val navEditSong: (songId: Long) -> Unit
) : ComponentContext by componentContext, AllMusicComponent, KoinComponent {
    private val store = instanceKeeper.getStore {
        AllMusicStoreProvider(
            storeFactory = storeFactory,
            songRepository = get()
        ).provide()
    }

    override val model: Value<AllMusicStore.State> = store.asValue()

    override fun navPlaySong(songId: Long) {
        navPlaySong(songId = songId)
    }

    override fun navEditSong(songId: Long) {
        navEditSong(songId = songId)
    }

    override fun sendIntent(intent: AllMusicStore.Intent) {
        store.accept(intent = intent)
    }
}
```

4.4. Створення компонентів Store

Опис контракту бізнес-логіки компонентів реалізується у інтерфейсах `Store` (Додаток Ж):

```
interface TestStore : Store<Intent, State, Nothing> {
    //Опис подій
    sealed class Intent {
        data class TabChanged(val newTab: Tab) : Intent()
    }

    //Опис стану
```

```

interface TestStore : Store<Intent, State, Nothing> {
    //Опис подій
    sealed class Intent {
        data class TabChanged(val newTab: Tab) : Intent()
    }

    //Опис стану

    data class State(
        val selectedTab: Tab = Tab.AllMusic
    )

    //Допоміжні сутності стану
    enum class Tab {
        AllMusic, AllPlaylists, AllSamples
    }
}

```

Реалізація контрактів Store виконується у класах StoreProvider (Додаток 3).

4.5. Розробка формату мелодії

Формат мелодії додатку складається з трьох класів: Song, Track та Chord.

Клас Song містить у собі ідентифікатор, назву, ім'я композитора, назву альбому, ім'я переписувача, довжину мелодії у мілісекундах, час створення та редагування та список треків.

```

data class SongDto(
    val id: Long,
    val name: String,
    val artist: String,
    val album: String,
    val transcriber: String,
    val length: Long,
    val creationDate: Instant,
    val editDate: Instant,
    val tracks: List<TrackDto>
)

```

Клас `Track` описує сутність треку. Він має ідентифікатор, інструмент, тон мелодії, BPM, гучність та список акордів.

```
data class TrackDto(
    val id: Int,
    val instrument: Instrument,
    val pitch: Int,
    val bpm: Int,
    val volume: Int,
    val chords: List<ChordDto>
)
```

Клас `Chord` описує сутність акорду та містить час на звуковій доріжці у мілісекундах та список ідентифікаторів нот.

```
data class ChordDto(
    val time: Long,
    val notes: List<Int>
)
```

4.6. Створення бази даних `SQLDelight`

У базі даних додатку зберігаються мелодії, плейлисти та семпли. Створення таблиць та методів роботи з ними виконується у SQL файлах (Додаток II):

```
CREATE TABLE song (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    artist TEXT NOT NULL,
    album TEXT NOT NULL,
    transcriber TEXT NOT NULL,
    length INTEGER NOT NULL,
    creationDate TEXT NOT NULL,
    editDate TEXT NOT NULL,
    tracks TEXT AS List<TrackEntity> NOT NULL
);

selectAll:
SELECT id, name, artist, length FROM song;

selectById:
SELECT * FROM song WHERE id = ?;
```

Параметри для створення бази даних описуються у `build.gradle.kts` файлі `common` модуля:

```
sqlDelight {
    database("ComposerDatabase") {
        packageName = "com.zhukovartemvl.common.db"
        sourceFolders = listOf("sqlDelight")
        version = 1
    }
}
```

Для створення об'єкту бази даних використовується клас `DatabaseFactory`:

```
class DatabaseFactory {
    fun createDatabase(driver: SqlDriver): ComposerDatabase {
        return ComposerDatabase(
            driver = driver,
            songAdapter = Song.Adapter(tracksAdapter = tracksAdapter)
        )
    }
}
```

Складні типи даних обробляються у адаптерах:

```
@OptIn(ExperimentalSerializationApi::class)
private val tracksAdapter = object : ColumnAdapter<List<TrackEntity>, String>
{
    override fun decode(databaseValue: String): List<TrackEntity> {
        return Json.decodeFromString(databaseValue)
    }
    override fun encode(value: List<TrackEntity>): String {
        return Json.encodeToString(value)
    }
}
```

Для роботи з таблицями бази даних у компонентах додатку, використовуються репозиторії:

```
class SongRepository(private val database: ComposerDatabase) {
    suspend fun getSongById(id: Long): SongDto? {
        return database.songQueries
            .selectById(id)
            .executeAsOneOrNull()?.transform()
    }
}
```

```
suspend fun getAllSongs(): List<SongShortInfoDto> {
    return database.songQueries.selectAll { id, name, artist, length ->
        SongShortInfoDto(id, name, artist, length)
    }.executeAsList()
}
```

4.7. Створення сховища для налаштувань додатку

Для збереження налаштувань додатку використовується `PreferencesRepository`, який записує примітивні дані у форматі ключ-значення до платформного сховища (Додаток К):

```
class PreferencesRepository(private val settings: Settings) {
    fun saveSongsTypeOfSort(typeOfSort: TypeOfSort) {
        settings.putString(key = SongsTypeOfSortKey, value = typeOfSort.key)
    }
    fun getSongsTypeOfSort(): TypeOfSort {
        return TypeOfSort.getByKey(key = settings.getString(key =
SongsTypeOfSortKey))
    }
    companion object {
        private const val SongsTypeOfSortKey = "songsTypeOfSort"
    }
}
```

4.8. Налаштування відмінностей UI для різних платформ

Мульти-платформний інтерфейс користувача зазвичай має деякі відмінності між платформами. Для застосування платформних компонентів у спільному модулі, використовується механізм `expect-actual`:

- Описується контракт очікуваного компоненту:

```
@Composable
expect fun AlertDialogExpected(
    modifier: Modifier,
    contentColor: Color,
    backgroundColor: Color,
    shape: Shape,
    properties: ComposerDialogProperties,
    onDismissRequest: () -> Unit,
    title: @Composable () -> Unit,
```



```

        text: @Composable () -> Unit,
        buttons: @Composable () -> Unit
    )

```

- Створюється реалізація компоненту для android платформи (Додаток Л);
- Створюється реалізація компоненту для desktop платформи (Додаток М).

4.9. Розробка системи відтворення звуків

Для відтворення звуків створено спеціальну утиліту. Вона виконує завантаження звуків з ресурсів додатку та відтворює з наданням необхідного музичного тону. Звуки зберігаються в ресурсах спільного модуля у вигляді MP3 файлів.

Контракт утиліти описано у інтерфейсі SoundMachine:

```

interface SoundMachine {
    suspend fun loadSounds(soundsPaths: List<String>): List<Sound>
    suspend fun playSound(sound: Sound, pitch: Double = 1.0)
}

```

Реалізація SoundMachine для android:

```

class SoundMachineImpl : SoundMachine, KoinComponent {
    override suspend fun loadSounds(soundsPaths: List<String>): List<Sound> {
        val sounds = arrayListOf<Sound>()
        withAndroidContext(context = get()) {
            soundsPaths.forEach { filePath ->
                try {
                    sounds.add(resourcesVfs[filePath].readSound())
                } catch (e: Exception) { }
            }
        }
        return sounds
    }

    override suspend fun playSound(sound: Sound, pitch: Double) {
        withAndroidContext(context = get()) {
            sound.play(params = PlaybackParameters(pitch = pitch, bufferTime =
0.0.seconds))}
    }
}

```

Реалізація SoundMachine для desktop:

```

class SoundMachineImpl : SoundMachine {
    override suspend fun loadSounds(soundsPaths: List<String>): List<Sound> {

```

```

val sounds = arrayListOf<Sound>()
soundsPaths.forEach { filePath ->
    try {
        sounds.add(resourcesVfs[filePath].readSound())
    } catch (e: Exception) {}
}
return sounds
}

override suspend fun playSound(sound: Sound, pitch: Double) {
    sound.play(params = PlaybackParameters(
        pitch = pitch,
        bufferTime = 0.0.seconds))}

```

4.10. Застосування Dependency Injection

Для налагодження доступу до репозиторіїв з компонентів бізнес-логіки додатку, використовується впровадження залежностей. Логіка впровадження залежностей описується у файлах-модулях Koin.

`DatabaseModule` описує створення об'єкту бази даних для кожної платформи окремо:

- Контракт у common модулі:

```
expect val databaseModule: Module
```

- Реалізація для android:

```

actual val databaseModule = module {
    single { createDatabase(context = get()) }
}

private fun createDatabase(context: Context): ComposerDatabase {
    val driver = AndroidSqliteDriver(ComposerDatabase.Schema, context,
    "composer_database.db")
    return DatabaseFactory().createDatabase(driver)
}

```

- Реалізація для desktop:

```

actual val databaseModule = module {
    single { createDatabase() }
}

private fun createDatabase(): ComposerDatabase {
    val databasePath = File(System.getProperty("user.home") + File.separator +
    "Arranger" + File.separator + "Database", "ComposerDatabase.db")
}

```

```

        databasePath.parentFile.mkdirs()
        val driver = JdbcSqliteDriver(url =
"jdbc:sqlite:${databasePath.absolutePath}").also {
            if (!databasePath.exists()) ComposerDatabase.Schema.create(it)
        }
        return DatabaseFactory().createDatabase(driver)
    }

```

PreferencesModule створює об'єкт Settings, який використовується для роботи з платформними сховищами налаштувань додатку:

```

val preferencesModule = module {
    factory { Settings() }
}

```

RepositoryModule реалізує створення об'єктів репозиторіїв:

```

val repositoryModule = module {
    factory<SongRepository>()
    factory<PreferencesRepository>()
}

```

SoundMashineModule створює об'єкт утиліти для відтворення звуків:

- Контракт у common модулі:

```
expect val soundMachineModule: Module
```

- Реалізація для android:

```

actual val soundMachineModule = module {
    factory { SoundMachineImpl() as SoundMachine }
}

```

Реалізація для desktop:

```

actual val soundMachineModule = module {
    factory { SoundMachineImpl() as SoundMachine }
}

```

Всі модулі впровадження залежностей ініціюються у функції initKoin:

```

fun initKoin(appDeclaration: KoinAppDeclaration = {}): KoinApplication {
    return startKoin {
        appDeclaration()
        modules(
            databaseModule,
            preferencesModule,
            repositoryModule,
            soundMachineModule
        )
    }
}

```

```

un initKoin(appDeclaration: KoinAppDeclaration = {}): KoinApplication {
    return startKoin {
        appDeclaration()
        modules(
            databaseModule,
            preferencesModule,
            repositoryModule,
            soundMachineModule
        )
    }
}

```

4.11. Висновки до розділу 4

У четвертому розділі описано ключові моменти створення додатку “Arranger”, а саме: налаштування системи збірки мульти-платформних проєктів, розробка архітектури додатку, створення компонентів екранів, розробка формату мелодії, створення бази даних та сховища налаштувань додатку, опис відмінностей для різних платформ, розробка системи відтворення звуків та застосування Dependency Injection у додатку.

ВИСНОВКИ

У магістерській роботі проведено дослідження фреймворків розробки мульти-платформних застосунків Kotlin Multiplatform та Compose Multiplatform.

Удосконалено професійні вміння та навички з розробки мульти-платформних застосунків, розширено і систематизовано знання, придбано практичний досвід.

Проведені дослідження дозволяють узагальнити методологію розробки мульти-платформних застосунків на мові програмування Kotlin та використання декларативного підходу для розробки інтерфейсів користувача. Описано архітектурний патерн MVI, який найкраще підходить під декларативний стиль мульти-платформного фреймворку Compose Multiplatform, а також інші засоби проєктування мульти-платформних застосунків, таких як: бібліотека SQLDelight для налагодження роботи з базами даних SQLite, бібліотека Decompose та фреймворк MVIKotlin.

Спроектowana структура застосунку та створена архітектура компонентів у мульти-платформних застосунках.

Розроблено новий формат мелодій з підтримкою декількох треків та різних музичних інструментів.

Розроблений застосунок дозволяє створювати музичні мелодії у новому форматі, а також проводити з ними різні маніпуляції: імпорт, експорт, видалення та редагування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Android MVI-Reactive Architecture Pattern. URL: <https://abhiappmobiledeveloper.medium.com/android-mvi-reactive-architecture-pattern-74e5f1300a87> (дата звернення: 25.12.2024);
2. Build better apps faster with Jetpack Compose. URL: <https://developer.android.com/jetpack/compose> (дата звернення: 25.12.2024);
3. Compose Multiplatform. URL: <https://www.jetbrains.com/ru-ru/lp/compose-mp/> (дата звернення: 25.12.2024);
4. Compose Multiplatform Goes Alpha, Unifying Desktop, Web, and Android UIs. URL: <https://blog.jetbrains.com/kotlin/2024/08/compose-multiplatform-goes-alpha/> (дата звернення: 25.12.2024);
5. Configure SQLDelight for data storage. URL: <https://kotlinlang.org/docs/kmm-configure-sqldelight-for-data-storage.html> (дата звернення: 25.12.2024);
6. Coroutines. URL: <https://kotlinlang.org/docs/coroutines-overview.html> (дата звернення: 25.12.2024);
7. Dagger. URL: <https://dagger.dev/dev-guide/> (дата звернення: 25.12.2024);
8. Declarative Programming: Is It A Real Thing?. URL: <https://www.toptal.com/software/declarative-programming> (дата звернення: 25.12.2024);
9. Declarative Programming. URL: <https://www.techopedia.com/definition/18763/declarative-programming> (дата звернення: 25.12.2024);
10. Decompose Overview. URL: <https://arkivanov.github.io/Decompose/> (дата звернення: 25.12.2024);
11. Decompose. URL: <https://github.com/arkivanov/Decompose> (дата звернення: 25.12.2024);

- 12.Dependency injection. URL:
https://en.wikipedia.org/wiki/Dependency_injection (дата звернення: 25.12.2024);
- 13.Flutter. URL: <https://flutter.dev/> (дата звернення: 25.12.2024);
- 14.Flutter: A Better Navigation Using BLoC. URL: <https://medium.com/capital-one-tech/flutter-a-better-navigation-using-bloc-733a73edc4da> (дата звернення: 25.12.2024);
- 15.Gradle Kotlin DSL Primer. URL:
https://docs.gradle.org/current/userguide/kotlin_dsl.html (дата звернення: 25.12.2024);
- 16.Gradle. URL: <https://en.wikipedia.org/wiki/Gradle> (дата звернення: 25.12.2024);
- 17.Katana. URL: <https://github.com/rewe-digital/katana> (дата звернення: 25.12.2024);
- 18.Kodein. URL: <https://kodein.org/> (дата звернення: 25.12.2024);
- 19.Koin - a smart Kotlin injection library to keep you focused on your app, not on your tools. URL: <https://insert-koin.io/> (дата звернення: 25.12.2024);
- 20.Koin — бібліотека для внедрения зависимостей, написанная на чистом Kotlin. URL: <https://habr.com/ru/company/otus/blog/530024/> (дата звернення: 25.12.2024).
- 21.Koin. URL: <https://github.com/InsertKoinIO/koin> (дата звернення: 25.12.2024);
- 22.Kotlin (programming language). URL:
[https://en.wikipedia.org/wiki/Kotlin_\(programming_language\)](https://en.wikipedia.org/wiki/Kotlin_(programming_language)) (дата звернення: 25.12.2024);
- 23.MVIKotlin Overview. URL: <https://arkivanov.github.io/MVIKotlin/> (дата звернення: 25.12.2024);
- 24.MVIKotlin. URL: <https://github.com/arkivanov/MVIKotlin> (дата звернення: 25.12.2024);

- 25.React Native. URL: <https://reactnative.dev/> (дата звернення: 25.12.2024);
- 26.SQLDelight Overview. URL: <https://cashapp.github.io/sqldelight/> (дата звернення: 25.12.2024);
- 27.State and Jetpack Compose. URL: <https://developer.android.com/jetpack/compose/state> (дата звернення: 25.12.2024);
- 28.Under the hood of Jetpack Compose — part 2 of 2. URL: <https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd> (дата звернення: 25.12.2024);
- 29.Understanding Jetpack Compose — part 1 of 2. URL: <https://medium.com/androiddevelopers/understanding-jetpack-compose-part-1-of-2-ca316fe39050> (дата звернення: 25.12.2024);
- 30.What is “donut-hole skipping” in Jetpack Compose?. URL: <https://www.jetpackcompose.app/articles/donut-hole-skipping-in-jetpack-compose> (дата звернення: 25.12.2024);
- 31.Xamarin documentation. URL: <https://docs.microsoft.com/en-us/xamarin/> (дата звернення: 25.12.2024);
- 32.Декларативное программирование. URL: https://ru.wikipedia.org/wiki/Декларативное_программирование (дата звернення: 25.12.2024);
- 33.Жуков А. В., Козуб Г.О. Розробка мобільного додатку за допомогою мови програмування Kotlin та сервісу Firebase. *Тенденції та перспективи розвитку науки і освіти в умовах глобалізації* : 2019 рік : зб. наук.праць Міжнар. наук.-практ. інтернет-конф. Переяслав, 2019.-Вип №52. С. 273–275;
- 34.Жуков А. В., Козуб Г.О. Застосування фреймворку Jetpack Compose у багатомодульному Android-додатку. Вітчизняна наука на зламі епох: проблеми та перспективи розвитку : 2021 рік : зб. наук.праць Всеукраїнська наук. інтернет-конф. Переяслав, 2021.-Вип №67;

- 35.Как Kotlin Multiplatform помогает сократить время разработки приложений. URL: <https://habr.com/ru/post/525888/> (дата звернення: 25.12.2024);
- 36.Козуб, Г., Козуб Ю. , Могильний Г., Жуков А. «Розробка мобільного Android-додатку з застосуванням принципів Clean Architecture». *Вісник Східноукраїнського національного університету імені Володимира Даля*, вип. 5 (269), Вересень 2021, с. 5-10, doi:10.33216/1998-7927-2021-269-5-5-10.
- 37.Корутины. URL: <https://metanit.com/kotlin/tutorial/8.1.php> (дата звернення: 25.12.2024);
- 38.Обзор кросс-платформенных фреймворков мобильной разработки. URL: <https://habr.com/ru/post/421227/> (дата звернення: 25.12.2024);
- 39.Почему мы выбрали Kotlin одним из целевых языков компании. Часть 2: Kotlin Multiplatform. URL: <https://habr.com/ru/company/domclick/blog/499820/> (дата звернення: 25.12.2024);
- 40.Сравнение решений для кроссплатформенной разработки: PhoneGap, Xamarin, Flutter, React Native. URL: <https://smartum.pro/ru/blog-ru/sravneniye-resheniy-krossplatformennoy-razrabotki-phonegap-xamarin-flutter-react-native/> (дата звернення: 25.12.2024);

ДОДАТКИ

Додаток А. Файл `build.gradle.kts` (common модуль)

```
import org.jetbrains.compose.compose

plugins {
    kotlin("multiplatform")
    id("org.jetbrains.compose")
    id("com.android.library")
    id("kotlin-parcelize")
    kotlin("plugin.serialization") version "1.5.10"
    id("com.squareup.sqldelight")
}

group = "com.zhukovartemvl"
version = "0.0.1"

repositories {
    google()
    maven("https://dl.bintray.com/korlibs/korlibs")
}

sqlDelight {
    database("ComposerDatabase") {
        packageName = "com.zhukovartemvl.common.db"
        sourceFolders = listOf("sqlDelight")
        version = 1
    }
}

kotlin {
    android()
    jvm("desktop") {
        compilations.all {
            kotlinOptions.jvmTarget = "11"
        }
    }
    sourceSets {
        val commonMain by getting {
            dependencies {
                api(compose.runtime)
                api(compose.foundation)
                api(compose.material)

                // Decompose
                api("com.arkivanov.decompose:decompose:0.4.0")
                api("com.arkivanov.essenty:lifecycle:0.2.2")
                api("com.arkivanov.essenty:parcelable:0.2.2")
                api("com.arkivanov.essenty:state-keeper:0.2.2")
                api("com.arkivanov.essenty:instance-keeper:0.2.2")
                api("com.arkivanov.essenty:back-pressed:0.2.2")

                // MVIKotlin
                api("com.arkivanov.mvikotlin:mvikotlin:3.0.0-alpha02")
                implementation("com.arkivanov.mvikotlin:rx:3.0.0-alpha02")
                api("com.arkivanov.mvikotlin:mvikotlin-main:3.0.0-alpha02")

                // Coroutines
                implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2")
            }
        }
    }
}
```

```

// DI
api("io.insert-koin:koin-core:3.1.2")

// HTTP
implementation("io.ktor:ktor-client-core:1.6.2")
implementation("io.ktor:ktor-client-json:1.6.2")
implementation("io.ktor:ktor-client-auth:1.6.2")
implementation("io.ktor:ktor-client-serialization:1.6.2")

// Database - SQLDelight
implementation("com.squareup.sqldelight:coroutines-extensions:1.5.0")

// Serializer      implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.2.2")

// Preferences
implementation("com.russhwolf:multiplatform-settings-no-arg:0.8.1")

// KotlinX DateTime
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

// Logging
implementation("io.github.aakira:napier:1.5.0")

// File
implementation("com.squareup.okio:okio-multiplatform:3.0.0-alpha.9")

// Audio
implementation("com.soywiz.korlibs.korau:korau:2.2.0")

// IO
implementation("com.soywiz.korlibs.korio:korio:2.2.0")

// Date Time
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

// Logger
api("io.github.aakira:napier:1.5.0")
}
}
val commonTest by getting
val androidMain by getting {
    dependencies {
        api("androidx.appcompat:appcompat:1.3.1")
        api("androidx.core:core-ktx:1.6.0")
        api("androidx.activity:activity-compose:1.3.1")
        implementation("com.squareup.sqldelight:android-driver:1.5.0")
    }
}
val androidTest by getting
val desktopMain by getting {
    dependencies {
        api(compose.desktop.common)
        implementation("com.squareup.sqldelight:sqlite-driver:1.5.0")
    }
}
val desktopTest by getting
}
}

android {
    compileSdk = 31
}

DI
api("io.insert-koin:koin-core:3.1.2")

// HTTP
implementation("io.ktor:ktor-client-core:1.6.2")
implementation("io.ktor:ktor-client-json:1.6.2")

```

```

        implementation("io.ktor:ktor-client-auth:1.6.2")
        implementation("io.ktor:ktor-client-serialization:1.6.2")

        // Database - SQLDelight
implementation("com.squareup.sqldelight:coroutines-extensions:1.5.0")

        // Serializer      implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.2.2")

        // Preferences
implementation("com.russhwolf:multiplatform-settings-no-arg:0.8.1")

        // KotlinX DateTime
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

        // Logging
        implementation("io.github.aakira:napier:1.5.0")

        // File
implementation("com.squareup.okio:okio-multiplatform:3.0.0-alpha.9")

        // Audio
implementation("com.soywiz.korlibs.korau:korau:2.2.0")

        // IO
implementation("com.soywiz.korlibs.korio:korio:2.2.0")

        // Date Time
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

        // Logger
        api("io.github.aakira:napier:1.5.0")
    }
}
val commonTest by getting
val androidMain by getting {
    dependencies {
        api("androidx.appcompat:appcompat:1.3.1")
        api("androidx.core:core-ktx:1.6.0")
        api("androidx.activity:activity-compose:1.3.1")
        implementation("com.squareup.sqldelight:android-driver:1.5.0")
    }
}
val androidTest by getting
val desktopMain by getting {
    dependencies {
        api(compose.desktop.common)
implementation("com.squareup.sqldelight:sqlite-driver:1.5.0")
    }
}
val desktopTest by getting
}
}

android {
    compileSdk = 31
    sourceSets["main"].manifest.srcFile("src/androidMain/AndroidManifest.xml")
    sourceSets["main"].assets.srcDirs(
        File("${project.projectDir}/src/commonMain/resources"),
        File("${project.projectDir}/src/androidMain/resources"),
        File("${project.projectDir}/src/main/resources"),
    )
implementation("io.ktor:ktor-client-auth:1.6.2")
        implementation("io.ktor:ktor-client-serialization:1.6.2")

```

```

        // Database - SQLDelight
implementation("com.squareup.sqldelight:coroutines-extensions:1.5.0")

        // Serializer      implementation("org.jetbrains.kotlinx:kotlinx-
serialization-json:1.2.2")

        // Preferences
implementation("com.russhwolf:multiplatform-settings-no-arg:0.8.1")

        // KotlinX DateTime
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

        // Logging
implementation("io.github.aakira:napier:1.5.0")

        // File
implementation("com.squareup.okio:okio-multiplatform:3.0.0-alpha.9")

        // Audio
implementation("com.soywiz.korlibs.korau:korau:2.2.0")

        // IO
implementation("com.soywiz.korlibs.korio:korio:2.2.0")

        // Date Time
implementation("org.jetbrains.kotlinx:kotlinx-datetime:0.2.1")

        // Logger
api("io.github.aakira:napier:1.5.0")
    }
}
val commonTest by getting
val androidMain by getting {
    dependencies {
        api("androidx.appcompat:appcompat:1.3.1")
        api("androidx.core:core-ktx:1.6.0")
        api("androidx.activity:activity-compose:1.3.1")
        implementation("com.squareup.sqldelight:android-driver:1.5.0")
    }
}
val androidTest by getting
val desktopMain by getting {
    dependencies {
        api(compose.desktop.common)
implementation("com.squareup.sqldelight:sqlite-driver:1.5.0")
    }
}
val desktopTest by getting
}
}

android {
    compileSdk = 31
    sourceSets["main"].manifest.srcFile("src/androidMain/AndroidManifest.xml")
    sourceSets["main"].assets.srcDirs(
        File("${project.projectDir}/src/commonMain/resources"),
        File("${project.projectDir}/src/androidMain/resources"),
        File("${project.projectDir}/src/main/resources"),
    )
    defaultConfig {
        minSdk = 24
        targetSdk = 31
// Database - SQLDelight
    }
}

```

Додаток Б. Файл **build.gradle.kts** (android модуль)

```
plugins {  
    id("org.jetbrains.compose")  
    id("com.android.application")  
    kotlin("android")  
}  
group = "com.zhukovartemvl"  
version = "0.0.1"  
dependencies {  
    implementation(project(":common"))  
}  
  
android {  
    compileSdk = 31  
    defaultConfig {  
        applicationId = "com.zhukovartemvl.Arranger"  
        minSdk = 24  
        targetSdk = 31  
        versionCode = 1  
        versionName = "0.0.1-alpha"  
    }  
    compileOptions {  
        sourceCompatibility = JavaVersion.VERSION_1_8  
        targetCompatibility = JavaVersion.VERSION_1_8  
    }  
    kotlinOptions {  
        jvmTarget = "1.8"  
    }  
}
```

Додаток В. Файл build.gradle.kts (desktop модуль)

```
import org.jetbrains.compose.compose
import org.jetbrains.compose.desktop.application.dsl.TargetFormat

plugins {
    kotlin("multiplatform")
    id("org.jetbrains.compose")
}

group = "com.zhukovartemv1"
version = "0.0.1"

kotlin {
    jvm {
        compilations.all {
            kotlinOptions.jvmTarget = "11"
        }
    }
    sourceSets {
        val jvmMain by getting {
            dependencies {
                implementation(project(":common"))
            }
        }
    }
}

compose.desktop {
    application {
        mainClass = "MainKt"
        nativeDistributions {
            targetFormats(TargetFormat.Dmg, TargetFormat.Msi, TargetFormat.Deb)
            packageName = "jvm"
            packageVersion = "1.0.0"
        }
    }
}
```

Додаток Г. Файл MusicScreen.kt

```
@OptIn(ExperimentalDecomposeApi::class)
@Composable
fun MusicScreen(component: MusicComponent) {
    val model by component.model.subscribeAsState()

    Scaffold(topBar = {
        Column {
            when (val screenState = model.screenState) {
                MusicStore.MusicScreenState.Default -> {
                    DefaultTopBar(
                        onSearchClicked = {
                            component.sendIntent(MusicStore.Intent.SearchClick) },
                        typeOfSorts = when (model.selectedTab) {
                            MusicStore.Tab.AllMusic -> songsTypeOfSort
                            MusicStore.Tab.AllPlaylists -> playlistsTypeOfSort
                            MusicStore.Tab.AllSamples -> samplesTypeOfSort
                        },
                    )
                }
            }
        }
    })
}
```

```

        selectedSort = model.currentTypeOfSort,
        onSortChanged = { typeOfSort -> component.sendIntent(intent =
MusicStore.Intent.TypeOfSortChanged(typeOfSort = typeOfSort)) }
    )
}
is MusicStore.MusicScreenState.Search -> {
    SearchTopBar(
        searchFilter = screenState.searchFilter,
        onSearchFilterChanged = { searchFilter ->
component.sendIntent(intent =
MusicStore.Intent.SearchFilterChanged(searchFilter)) },
        onSearchCancelClicked = { component.sendIntent(intent =
MusicStore.Intent.SearchCancelClick) }
    )
}
is MusicStore.MusicScreenState.ItemsSelection -> {
    ItemsSelectionTopBar(
        selectedTab = model.selectedTab,
        itemsSelected = screenState.selectedItems,
        onSelectionCancelClicked = {},
        onAddToPlaylistClicked = {},
        onDeleteFromPlaylistClicked = {},
        onDeleteClicked = {},
        onSelectAllClicked = {}
    )
}
}
NavTopTabs(
    selectedTab = model.selectedTab,
    enabled = model.screenState == MusicStore.MusicScreenState.Default,
    onTabClicked = { tab ->
component.sendIntent(MusicStore.Intent.TabChanged(newTab = tab)) }
)
}
)) {
    val screenState = model.screenState
    val searchFilter = if (screenState is MusicStore.MusicScreenState.Search)
{
        screenState.searchFilter
    } else {
        null
    }
    Children(
        routerState = component.routerState,
        animation = crossfade()
    ) {
        when (val child = it.instance) {
            is MusicComponent.Child.AllMusic -> AllMusicScreen(
                component = child.component,
                typeOfSort = model.currentTypeOfSort,
                searchFilter = searchFilter
            )
            is MusicComponent.Child.AllPlaylists ->
AllPlaylistsScreen(child.component)

```



```

        is MusicComponent.Child.AllSamples ->
AllSamplesScreen(child.component)
    }
}
}
}

@Composable
private fun DefaultTopBar(
    onSearchClicked: () -> Unit,
    typeOfSort: List<TypeOfSort>,
    selectedSort: TypeOfSort,
    onSortChanged: (typeOfSort: TypeOfSort) -> Unit
) {
    val sortMenuOpened = remember { mutableStateOf(false) }
    TopAppBar {
        Text(
            modifier = Modifier.weight(1f),
            text = "Music"
        )
        IconButton(onClick = onSearchClicked) {
            Icon(imageVector = Icons.Default.Search, contentDescription = null)
        }
        IconButton(onClick = { sortMenuOpened.value = true }) {
            Icon(imageVector = Icons.Default.ArrowDropDown, contentDescription =
null)
        }
        ComposerDropdownMenu(
            expanded = sortMenuOpened.value,
            onDismissRequest = { sortMenuOpened.value = false }
        ) {
            typeOfSorts.forEach { typeOfSort: TypeOfSort ->
                val selectedType = typeOfSort == selectedSort
                ComposerDropdownMenuItem(
                    onClick = {
                        if (!selectedType) {
                            onSortChanged(typeOfSort)
                            sortMenuOpened.value = false
                        }
                    }
                ) {
                    Text(
                        text = typeOfSort.title,
                        fontWeight = if (selectedType) FontWeight.Bold else
FontWeight.Normal
                    )
                }
            }
        }
    }
}

@Composable
private fun SearchTopBar(

```

```

searchFilter: String,
onSearchFilterChanged: (searchFilter: String) -> Unit,
onSearchCancelClicked: () -> Unit
) {
    TopAppBar {
        TextField(
            modifier = Modifier.fillMaxWidth().padding(4.dp),
            value = searchFilter,
            onValueChange = onSearchFilterChanged,
            leadingIcon = {
                Icon(imageVector = Icons.Default.Search, contentDescription = null)
            },
            trailingIcon = {
                IconButton(onClick = onSearchCancelClicked) {
                    Icon(imageVector = Icons.Default.Close, contentDescription = null)
                }
            }
        )
    }
}

@Composable
private fun ItemsSelectionTopBar(
    selectedTab: MusicStore.Tab,
    itemsSelected: Int,
    onSelectionCancelClicked: () -> Unit,
    onAddToPlaylistClicked: () -> Unit,
    onDeleteFromPlaylistClicked: () -> Unit,
    onDeleteClicked: () -> Unit,
    onSelectAllClicked: () -> Unit
) {
    TopAppBar {
        Row {
            IconButton(onClick = onSelectionCancelClicked) {
                Icon(imageVector = Icons.Default.Close, contentDescription = null)
            }
            Text(
                modifier = Modifier.weight(1f),
                text = "$itemsSelected selected"
            )
        }
        val dropdownOpened = remember { mutableStateOf(false) }
        IconButton(onClick = { dropdownOpened.value = true }) {
            Icon(imageVector = Icons.Default.MoreVert, contentDescription = null)
        }
        ComposerDropdownMenu(
            expanded = dropdownOpened.value,
            onDismissRequest = { dropdownOpened.value = false },
        ) {
            if (selectedTab != MusicStore.Tab.AllSamples) {
                ComposerDropdownMenuItem(onClick = onAddToPlaylistClicked) {
                    Text(text = "Add to playlist")
                }
            }
            if (selectedTab != MusicStore.Tab.AllPlaylists) {
                ComposerDropdownMenuItem(onClick = onDeleteFromPlaylistClicked) {
                    Text(text = "Delete from playlist")
                }
            }
        }
    }
}

```


Додаток Е. Файл MusicComponentProvider.kt

```
class MusicComponentProvider private constructor(
    componentContext: ComponentContext,
    storeFactory: StoreFactory,
    private val allMusic: (ComponentContext) -> AllMusicComponent,
    private val allPlaylists: (ComponentContext) -> AllPlaylistsComponent,
    private val allSamples: (ComponentContext) -> AllSamplesComponent,
) : ComponentContext by componentContext, MusicComponent, KoinComponent {

    constructor(
        componentContext: ComponentContext,
        storeFactory: StoreFactory,
        navEditSong: (songId: Long) -> Unit,
        navPlaySong: (songId: Long) -> Unit
    ) : this(
        componentContext = componentContext,
        storeFactory = storeFactory,
        allMusic = { childContext -> AllMusicComponentProvider(childContext,
            storeFactory, navEditSong, navPlaySong) },
        allPlaylists = { childContext ->
            AllPlaylistsComponentProvider(childContext, storeFactory) },
        allSamples = { childContext -> AllSamplesComponentProvider(childContext,
            storeFactory) }
    )

    private val router: Router<Config, MusicComponent.Child> =
        router(
            initialConfiguration = Config.AllMusic,
            handleBackButton = true,
            childFactory = ::createChild
        )

    override val routerState: Value<RouterState<*, MusicComponent.Child>> =
        router.state

    private val store = instanceKeeper.getStore {
        MusicStoreProvider(
            storeFactory = storeFactory,
            preferencesRepository = get(),
            onTabChanged = { tab -> onTabChanged(tab) }
        ).provide()
    }

    override val model: Value<MusicStore.State> = store.asValue()

    override fun sendIntent(intent: MusicStore.Intent) {
        store.accept(intent)
    }

    private fun createChild(
        config: Config,
        componentContext: ComponentContext
    ): MusicComponent.Child =
        when (config) {
            Config.AllMusic ->
                MusicComponent.Child.AllMusic(allMusic(componentContext))
            Config.AllPlaylists ->
                MusicComponent.Child.AllPlaylists(allPlaylists(componentContext))
            Config.AllSamples ->
                MusicComponent.Child.AllSamples(allSamples(componentContext))
        }

    private fun onTabChanged(tab: MusicStore.Tab): Unit =
        when (tab) {
```

```

        MusicStore.Tab.AllMusic -> router.navigateSingleTop(config = {
Config.AllMusic })
        MusicStore.Tab.AllPlaylists -> router.navigateSingleTop(config = {
Config.AllPlaylists })
        MusicStore.Tab.AllSamples -> router.navigateSingleTop(config = {
Config.AllSamples })
    }

    private sealed class Config : Parcelable {
        @Parcelize
        object AllMusic : Config()

        @Parcelize
        object AllPlaylists : Config()

        @Parcelize    object AllSamples : Config()
    }
}

```

Додаток Ж. Файл MusicStore.kt

```

interface MusicStore : Store<MusicStore.Intent, MusicStore.State, Nothing> {

    sealed class Intent {
        data class TabChanged(val newTab: Tab) : Intent()

        object SearchClick : Intent()
        data class TypeOfSortChanged(val typeOfSort: TypeOfSort) : Intent()
        data class SearchFilterChanged(val searchFilter: String) : Intent()
        object SearchCancelClick : Intent()

        data class SelectedItemsChanged(val selectedItems: Int) : Intent()
        object RemoveSelection : Intent()
    }

    data class State(
        val selectedTab: Tab = Tab.AllMusic,
        val currentTypeOfSort: TypeOfSort = TypeOfSort.Ascending,
        val screenState: MusicScreenState = MusicScreenState.Default
    )

    enum class Tab {
        AllMusic, AllPlaylists, AllSamples
    }

    sealed class MusicScreenState {
        object Default : MusicScreenState()
        data class Search(val searchFilter: String = "") : MusicScreenState()
        data class ItemsSelection(val selectedItems: Int = 1) :
MusicScreenState()
    }
}

```

Додаток 3. Файл MusicStoreProvider.kt

```

class MusicStoreProvider(
    private val storeFactory: StoreFactory,
    private val preferencesRepository: PreferencesRepository,
    private val onTabChanged: (MusicStore.Tab) -> Unit
) {

    fun provide(): MusicStore =
        object : MusicStore, Store<MusicStore.Intent, MusicStore.State, Nothing>
        by storeFactory.create(
            name = ClassTag,
            initialState = MusicStore.State(),
            bootstrapper = SimpleBootstrapper(Unit),
            executorFactory = ::ExecutorImpl,
            reducer = ReducerImpl
        ) {}

    private sealed class Result {
        data class TabChanged(val tab: MusicStore.Tab) : Result()
        data class TypeOfSortChanged(val typeOfSort: TypeOfSort) : Result()
        data class ScreenStateChanged(val screenState:
        MusicStore.MusicScreenState) : Result()
    }

    private object ReducerImpl : Reducer<MusicStore.State, Result> {
        override fun MusicStore.State.reduce(result: Result): MusicStore.State =
            when (result) {
                is Result.TabChanged -> copy(selectedTab = result.tab)
                is Result.TypeOfSortChanged -> copy(currentTypeOfSort =
                result.typeOfSort)
                is Result.ScreenStateChanged -> copy(screenState =
                result.screenState)
            }
    }

    private inner class ExecutorImpl : CoroutineExecutor<MusicStore.Intent,
    Unit, MusicStore.State, Result, Nothing>() {

        override fun executeAction(action: Unit, getState: () ->
        MusicStore.State) {
            dispatch(Result.TypeOfSortChanged(typeOfSort =
            preferencesRepository.getSongsTypeOfSort()))
        }

        override fun executeIntent(intent: MusicStore.Intent, getState: () ->
        MusicStore.State) {
            when (intent) {
                is MusicStore.Intent.TabChanged -> changeTab(newTab = intent.newTab)

                MusicStore.Intent.SearchClick ->
                dispatch(Result.ScreenStateChanged(screenState =
                MusicStore.MusicScreenState.Search(searchFilter = "")))
                is MusicStore.Intent.SearchFilterChanged -> {
                    dispatch(Result.ScreenStateChanged(screenState =
                    MusicStore.MusicScreenState.Search(searchFilter = intent.searchFilter)))
                }
                MusicStore.Intent.SearchCancelClick ->
                dispatch(Result.ScreenStateChanged(screenState =
                MusicStore.MusicScreenState.Default))
            }
        }
    }
}

```

```

        is MusicStore.Intent.TypeOfSortChanged ->
changeTypeOfSort(newTypeOfSort = intent.typeOfSort, currentTab =
getState().selectedTab)

        is MusicStore.Intent.SelectedItemsChanged -> TODO()
        MusicStore.Intent.RemoveSelection -> TODO()
    }
}

private fun changeTab(newTab: MusicStore.Tab) {
    val newTypeOfSort = when (newTab) {
        MusicStore.Tab.AllMusic -> preferencesRepository.getSongsTypeOfSort()
        MusicStore.Tab.AllPlaylists ->
preferencesRepository.getPlaylistsTypeOfSort()
        MusicStore.Tab.AllSamples ->
preferencesRepository.getSamplesTypeOfSort()
    }
    dispatch(Result.TypeOfSortChanged(typeOfSort = newTypeOfSort))
    dispatch(Result.TabChanged(tab = newTab))
    onTabChanged(newTab)
}

private fun changeTypeOfSort(newTypeOfSort: TypeOfSort, currentTab:
MusicStore.Tab) {
    when (currentTab) {
        MusicStore.Tab.AllMusic ->
preferencesRepository.saveSongsTypeOfSort(newTypeOfSort)
        MusicStore.Tab.AllPlaylists ->
preferencesRepository.savePlaylistsTypeOfSort(newTypeOfSort)
        MusicStore.Tab.AllSamples ->
preferencesRepository.saveSamplesTypeOfSort(newTypeOfSort)
    }
    dispatch(Result.TypeOfSortChanged(typeOfSort = newTypeOfSort))
}
}
}

```

Додаток И. Файл Song.sql

```

CREATE TABLE song (
  id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  artist TEXT NOT NULL,
  album TEXT NOT NULL,
  transcriber TEXT NOT NULL,
  youtubeLink TEXT NOT NULL,
  length INTEGER NOT NULL,
  tags TEXT AS List<String> NOT NULL,
  creationDate TEXT NOT NULL,
  editDate TEXT NOT NULL,
  tracks TEXT AS List<TrackEntity> NOT NULL
);

selectAll:
SELECT id, name, artist, length FROM song;

selectById:
SELECT * FROM song WHERE id = ?;

selectShortInfoById:
SELECT id, name, artist, length FROM song WHERE id = ?;

insertSong:
INSERT INTO song(name, artist, album, transcriber, youtubeLink, length, tags,
creationDate, editDate, tracks) VALUES ?;

updateSong:
UPDATE song SET name = ?, artist = ?, album = ?, transcriber = ?, youtubeLink
= ?, length = ?, tags = ?, editDate = ?, tracks = ? WHERE id = ?;

deleteSongById:
DELETE FROM song WHERE id = ?;

deleteSongsByIds:
DELETE FROM song WHERE id IN ?;

```


Додаток К. Файл PreferencesRepository.kt

```
class PreferencesRepository(private val settings: Settings) {
    //Songs type of sort
    fun saveSongsTypeOfSort(typeOfSort: TypeOfSort) {
        settings.putString(key = SongsTypeOfSortKey, value = typeOfSort.key)
    }
    fun getSongsTypeOfSort(): TypeOfSort {
        return TypeOfSort.getByKey(
            key = settings.getString(key = SongsTypeOfSortKey)
        )
    }

    //Playlists type of sort
    fun savePlaylistsTypeOfSort(typeOfSort: TypeOfSort) {
        settings.putString(
            key = PlaylistsTypeOfSortKey,
            value = typeOfSort.key
        )
    }
    fun getPlaylistsTypeOfSort(): TypeOfSort {
        return TypeOfSort.getByKey(
            key = settings.getString(key = PlaylistsTypeOfSortKey)
        )
    }

    //Samples type of sort
    fun saveSamplesTypeOfSort(typeOfSort: TypeOfSort) {
        settings.putString(key = SamplesTypeOfSortKey, value = typeOfSort.key)
    }
    fun getSamplesTypeOfSort(): TypeOfSort {
        return TypeOfSort.getByKey(
            key = settings.getString(key = SamplesTypeOfSortKey)
        )
    }
    companion object {
        private const val SongsTypeOfSortKey = "songsTypeOfSort"
        private const val PlaylistsTypeOfSortKey = "playlistsTypeOfSort"
        private const val SamplesTypeOfSortKey = "samplesTypeOfSort"
    }
}
```

Додаток Л. Файл AlertDialogExpected.kt (android модуль)

```

@Composable
actual fun AlertDialogExpected(
    modifier: Modifier,
    contentColor: Color,
    backgroundColor: Color,
    shape: Shape,
    properties: ComposerDialogProperties,
    onDismissRequest: () -> Unit,
    title: @Composable () -> Unit,
    text: @Composable () -> Unit,
    buttons: @Composable () -> Unit
) {
    AlertDialog(
        modifier = modifier,
        contentColor = contentColor,
        backgroundColor = backgroundColor,
        shape = shape,
        properties = DialogProperties(
            dismissOnBackPress = properties.dismissOnBackPress,
            dismissOnClickOutside = properties.dismissOnClickOutside
        ),
        onDismissRequest = onDismissRequest,
        title = title,
        text = text,
        buttons = buttons
    )
}

```

Додаток М. Файл AlertDialogExpected.kt (desktop модуль)

```

@OptIn(ExperimentalMaterialApi::class)
@Composable
actual fun AlertDialogExpected(
    modifier: Modifier,
    contentColor: Color,
    backgroundColor: Color,
    shape: Shape,
    properties: ComposerDialogProperties,
    onDismissRequest: () -> Unit,
    title: @Composable () -> Unit,
    text: @Composable () -> Unit,
    buttons: @Composable () -> Unit
) {
    AlertDialog(
        modifier = modifier,
        contentColor = contentColor,
        backgroundColor = backgroundColor,
        shape = shape,
        dialogProvider = PopupAlertDialogProvider,
        onDismissRequest = onDismissRequest,
        title = title,
        text = text,
        buttons = buttons
    )
}

```