МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«МОБІЛЬНИЙ ЗАСТОСУНОК НА KOTLIN З ДЕКЛАРАТИВНИМ ІНТЕРФЕЙСОМ ЗАСОБАМИ JETPACK COMPOSE»

Рівень «бакалавр»
Галузь знань 12 «Інформаційні технології»,
спеціальність 121 «Інженерія програмного забезпечення»
Виконав: студент 4 курсу
Лепетуха Євгеній Володимирович

Керівник: к.т.н., доцент
Задерейко Олександр Владиславович

Рецензент: к.т.н., доцент
Манаков Сергій Юрійович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»
Факультет кібербезпеки та інформаційних технологій
Кафедра інформаційних технологій
Галузь знань: 12 Інформаційні технології
Спеціальність: 121 Інженерія програмного забезпечення
Назва освітньої програми: Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інформаційних технологій
доцент Н.І. Логінова
_____ «____ »_____ 20__ року

З А В Д А Н Н Я
на кваліфікаційну (бакалаврську) роботу
здобувачу Лепетуха Євгенію Володимировичу

1. Тема роботи: «Мобільний застосунок на Kotlin з декларативним інтерфейсом засобами Jetpack Compose»
   Керівник роботи: к.т.н., доцент Задерейко Олександр Владиславович
   затверджені наказом НУ «ОЮА» від «02» квітня 2024 року № 809-06
2. Дата видачі завдання «15» вересня 2023 року
3. Строк подання здобувачем роботи «20» травня 2024 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та вибір засобів розробки	16.10.23 - 01.12.23	
2	Проєктування мобільного застосунку	04.12.23 - 26.01.24	
3	Розробка мобільного застосунку	29.01.24 - 15.03.24	
4	Тестування розробленого застосунку	18.03.24 - 26.04.24	
5	Оформлення пояснювальної записки	29.04.24 - 19.05.24	

Здобувач _____
(підпис) (прізвище та ініціали)

Керівник роботи _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Лепетуха Є.В. Мобільний застосунок на Kotlin з декларативним інтерфейсом засобами Jetpack Compose. – Кваліфікаційна робота бакалавра за спеціальністю 121 «Інженерія програмного забезпечення».

Кваліфікаційна робота викладена на 68 сторінках, містить 3 розділи, 27 ілюстрацій, 1 таблицю, 10 використаних джерел.

Об’єктом роботи є мобільний застосунок для обміну повідомленнями по Bluetooth.

Предмет роботи – розробка нативного мобільного застосунку на Kotlin для обміну повідомленнями по Bluetooth з використанням UI засобами Jetpack Compose.

Метою роботи є розробка мобільного Android-застосунку з можливістю обміну повідомленнями по Bluetooth.

Метод дослідження – прикладний.

У першому розділі виконано аналіз предметної області та постановка задачі. У другому розділі здійснено проєктування та розробка мобільного застосунку. У третьому розділі здійснюється тестування розробленого застосунку.

За результатами роботи зроблено висновки про отримані у роботі результати та потенціальні напрямки подальшого розвитку.

Ключові слова: Android, Kotlin, Android Studio, Jetpack Compose, MVVM, Dependency Injection, Bluetooth, чат.

ABSTRACT

Lepetukha E.V. A mobile application on Kotlin with a declarative interface using Jetpack Compose. – Bachelor's qualifying work on specialty 121 "Software engineering".

The qualification work is laid out on 68 pages, contains 3 chapters, 27 illustrations, 1 table, 10 used sources.

The object of the work is a mobile application for exchanging messages via Bluetooth.

The subject of the work is the development of a native mobile application on Kotlin for exchanging messages via Bluetooth using the UI using Jetpack Compose.

The goal of the work is the development of a mobile application with the possibility of exchanging messages via Bluetooth.

The research method is applied.

In the first section, the analysis of the subject area and the formulation of the problem were performed. In the second section, the design and development of the mobile application was carried out. In the third section, the developed application is tested.

Based on the results of the work, conclusions were made about the results obtained in the work and potential directions for further development.

Keywords: Android, Kotlin, Android Studio, Jetpack Compose, MVVM, Dependency Injection, Bluetooth, chat.

ЗМІСТ

ВСТУП

Мобільні пристрої, такі як смартфони, на сьогодні є невід'ємними аксесуарами переважної більшості членів суспільства. Необхідність постійно бути на зв'язку та здатність до обміну різного роду інформацією є звичайними вимогами на шляху до цифрової трансформації.

Але, за складних навколишніх умов сьогодення, нажаль, інколи можуть виникнути ситуації, коли звичні засоби мобільного зв'язку виявляються недосяжними з різних обставин. Прикладом таких тяжких обставин може бути пошук живих людей під завалами, де мобільний зв'язок відсутній взагалі, або неспроможний подолати наявні фізичні перешкоди.

Одним з можливих рішень цього актуального питання є використання інших технологій для бездротового обміну інформацією. Серед таких технологічних рішень, доречним виявляється бездротовий модуль Bluetooth, який є в наявності практично в кожному смартфоні чи планшеті та дозволяє встановлювати з'єднання двох пристроїв зі швидкістю до 24 Мбіт/с, та максимальною дистанцією до 100 м.

Розробка ж мобільних застосунків є окремою складовою сучасної галузі інформаційних технологій. Такі навики передбачать володіння однією або декількома платформами та мовами розробки: Kotlin та/чи Java для нативної Android-розробки, Swift для XCode під IoS, мультиплатформні засоби Flutter чи Kotlin Multiplatform. Для професійного застосування цих технологій необхідне знання особливостей апаратної платформи та глибоке вивчення документації.

Отже, справжній фахівець з мобільної розробки в змозі вирішувати актуальні, нестандартні та корисні задачі.

Саме таке завдання ставиться у даній кваліфікаційній роботі, а саме: створення нативного мобільного застосунку для платформи Android з використанням сучасних засобів розробки, як-от мова Kotlin та декларативний інтерфейс з Jetpack Compose. До того ж, обрана предметна область передбачає

використання спеціальної документацій стосовно програмування Bluetooth-модулів Android.

Об’єкт дослідження – мобільний застосунок для обміну повідомленнями по Bluetooth.

Предмет дослідження – розробка нативного мобільного застосунку на Kotlin для обміну повідомленнями по Bluetooth з використанням UI засобами Jetpack Compose.

Мета і завдання роботи – розробка мобільного Android-застосунку з можливістю обміну повідомленнями по Bluetooth.

Методика дослідження – прикладна.

Практична значущість результатів дослідження: можливість текстового чату через Bluetooth за умов відсутності інших засобів мобільного зв’язку.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Опис предметної області

Предметна область, яка розглядається у кваліфікаційній роботі – нативний мобільний застосунок для платформи Android [1] з функціями чату через Bluetooth з використанням мови програмування Kotlin [2] та фреймворку декларативного інтерфейсу Jetpack Compose [3] у середовищі розробки Android Studio [4].

Ця предметна область охоплює такі аспекти:

1) Bluetooth (від слів англ. Blue – синій і tooth – зуб), блютуз [1] – виробнича специфікація бездротових персональних мереж (Wireless personal area network, WPAN). Bluetooth забезпечує обмін інформацією між такими пристроями як персональні комп'ютери (настільні, кишенькові, ноутбуки), мобільні телефони та смартфони [5], інтернет-планшети, принтери, цифрові фотоапарати, миші, клавіатури, джойстики, навушники, гарнітури та акустичні системи на надійній, безкоштовній, повсюдно доступній радіочастоті для ближнього зв'язку. Bluetooth дозволяє цим пристроям повідомлятися, коли вони знаходяться один від одного в радіусі близько 100 м у старих версіях протоколу і до 150 м починаючи з версії Bluetooth 5. Дальність залежить від перешкод і завад, навіть в одному приміщенні.

2) Kotlin – статично типізована, об'єктно-орієнтована мова програмування, що працює поверх Java Virtual Machine і розробляється компанією JetBrains. Автори ставили за мету створити мову більш лаконічна і типобезпечна, ніж Java, і більш проста, ніж Scala. Наслідком спрощення порівняно зі Scala стали також швидша компіляція та краща підтримка мови у IDE. Мова повністю сумісна з Java, що дозволяє Java-розробникам поступово перейти до її використання; зокрема, мова також вбудовується Android, що дозволяє для існуючого Android-програми впроваджувати нові функції на Kotlin без переписування програми повністю.

3) Jetpack Compose – це сучасна декларативна структура інтерфейсу користувача з відкритим кодом на основі Kotlin для Android, розроблена Google.

Jetpack Compose підтримує версії Android 5.0 і новіші. Він використовує мову програмування Kotlin і забезпечує реактивну модель програмування, подібну до інших фреймворків інтерфейсу користувача, таких як Vue.js і React Native. Compose розроблено для бездоганної інтеграції з існуючими програмами та бібліотеками Android, дозволяючи розробникам поступово перетворювати свої програми на Compose. У Compose інтерфейс користувача визначається за допомогою функцій, які були анотовані за допомогою @Composable анотації, які відомі як комбіновані функції та визначають стан екрана. Анотація використовується компілятором Compose для створення шаблонного коду інтерфейсу користувача.

На рис. 1.1 побудована інтелект-карта (mind map) предметної області.

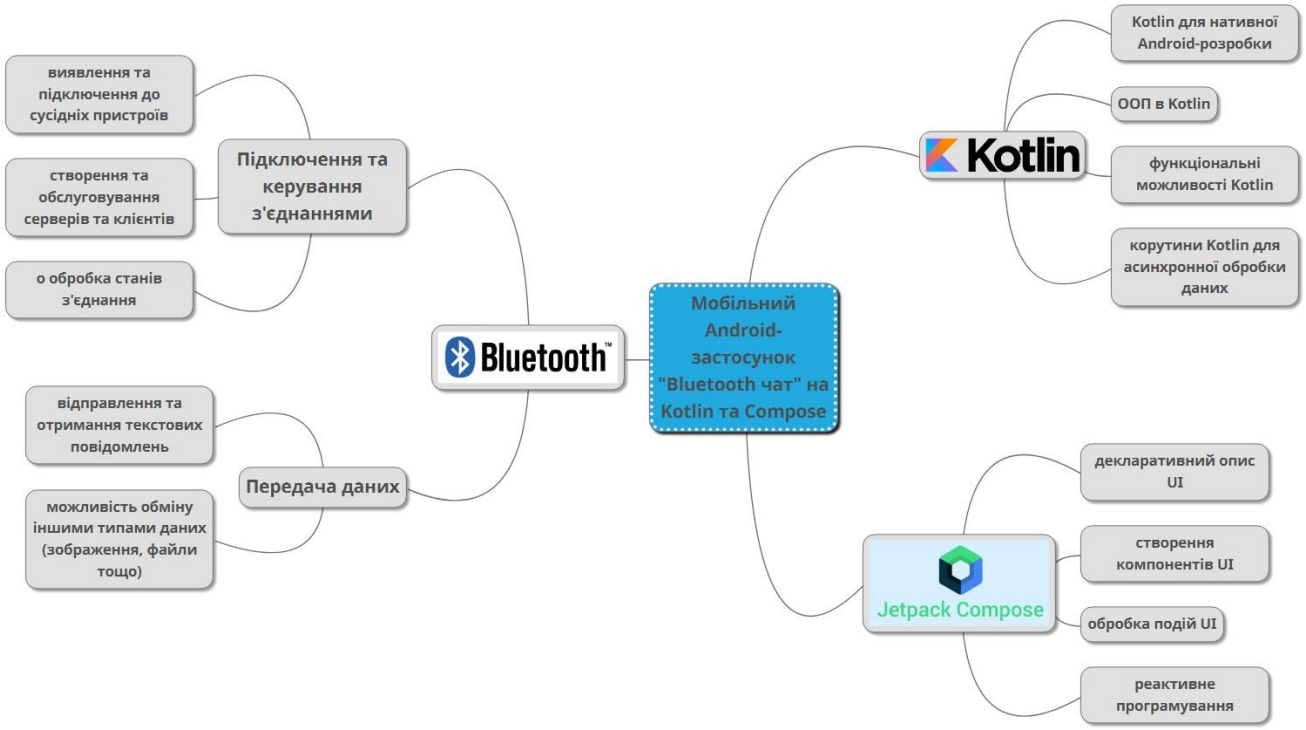


Рисунок 1.1 – Mind map предметної області

Додаткові аспекти, які можуть бути розглянуті:

– Шифрування даних: Забезпечення конфіденційності та цілісності даних, що передаються через Bluetooth;

– Багатокористувацький чат: підтримка одночасного чату з декількома користувачами;

– Виявлення та підключення до Bluetooth-пристроїв з низькою енергоємністю (BLE): Розширення можливостей чату для роботи з пристроями IoT та носимими пристроями.

– Графічний інтерфейс: розробка зручного та інтуїтивно зрозумілого інтерфейсу користувача для чату;

– Додаткові функції: додавання таких функцій, як голосові повідомлення, обмін файлами, групові чати тощо.

Важливо зазначити, що це загальний опис предметної області. Кваліфікаційна робота буде зосереджена на більш вузькому аспекті цієї області, а саме, на реалізації певного функціоналу Bluetooth-чату з використанням Kotlin та Compose з дотриманням принципів чистого коду та архітектури [дядя Роберт].

1.2 Огляд функціональних можливостей існуючих аналогів

Розглянемо застосунки з маркету Google Play за запитом «Bluetooth Chat».

1) Simple Bluetooth Chat. Simple Bluetooth Chat допоможе вам спілкуватися з найближчою людиною за допомогою технології Bluetooth. Працюватиме без інтернету. Це корисно для студентів у класі, у чаті з друзями в автобусі чи поїзді та у багатьох інших випадках. Простий Bluetooth-чат зараз розроблений тільки для текстових повідомлень. Вигляд інтерфейсу показаний на рис. 1.2.

2) Bluetooth Chat (автор: vapps2015). Застосунок чату Bluetooth дозволяє спілкуватися з друзями, членами сім'ї, які поряд з вами (в діапазоні Bluetooth) і встановили програму чату Bluetooth. Вам потрібно з'єднати два Android телефони за допомогою Bluetooth і почати Спілкуючись його безкоштовно і не потрібно підключення до Інтернету. Це безпечно, швидко та легко використовувати. Інтерфейсу застосунку показаний на рис. 1.3.

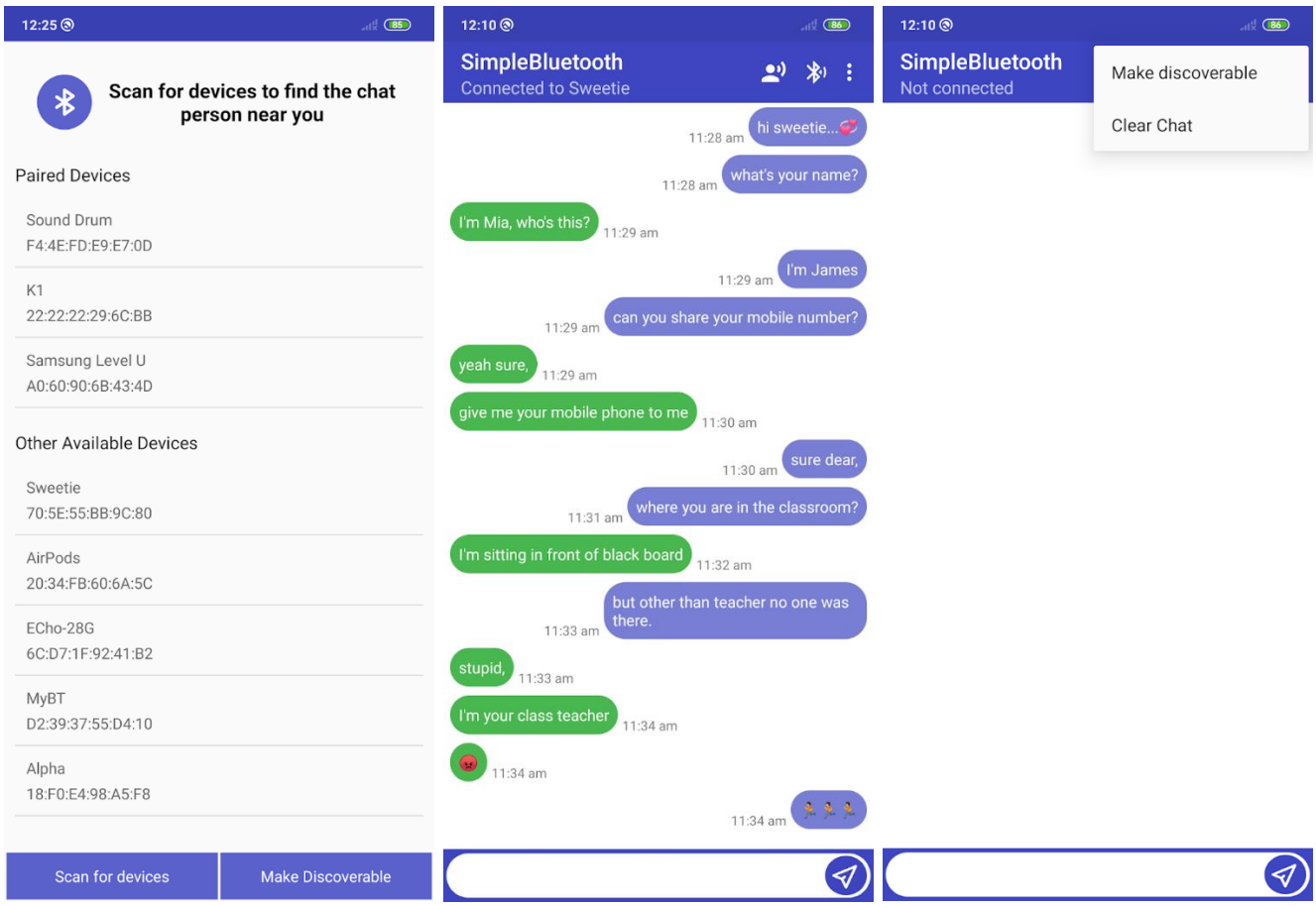


Рисунок 1.2 – Вигляд інтерфейсу застосунку Simple Bluetooth Chat

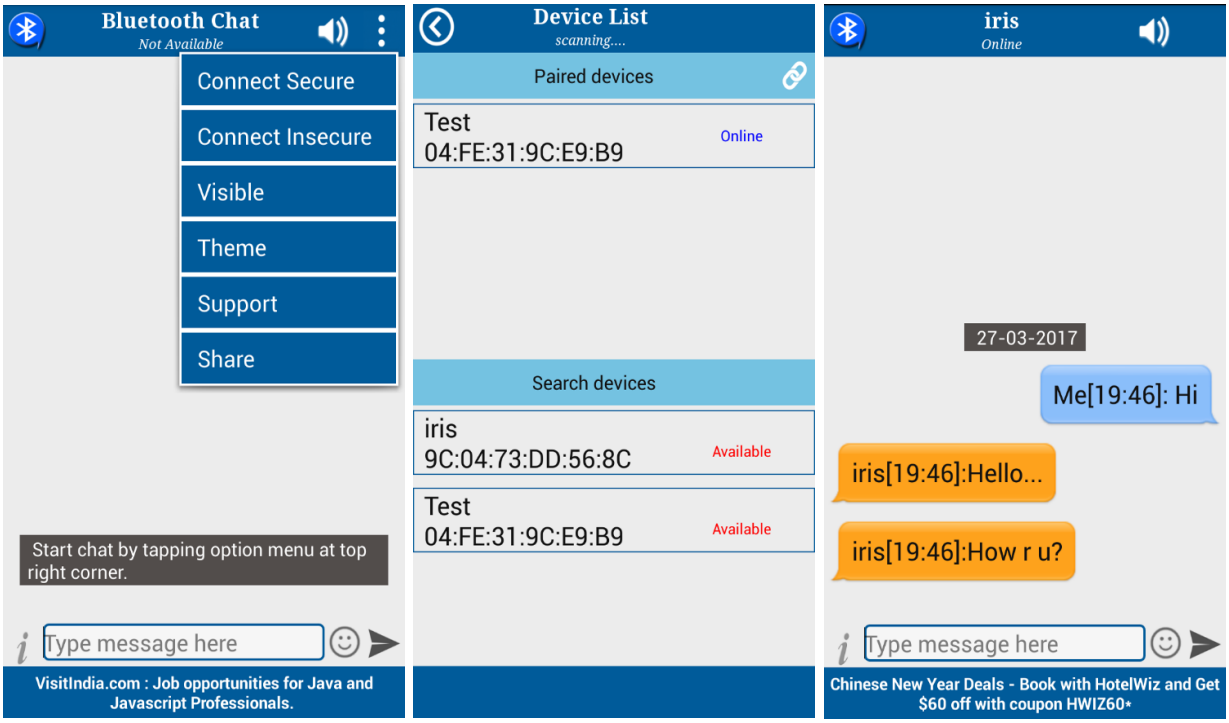


Рисунок 1.3 – Вигляд інтерфейсу застосунку Bluetooth Chat (автор: vapps2015)

3) Bluetooth Chatter – застосунок, призначений для надсилання повідомлень за допомогою блютуз. Надсилайте текстові та голосові повідомлення, а також будь-які файли та зображення. Основні можливості:

– запис голосових повідомлень;
– надсилання будь-яких файлів.

Вигляд застосунку Bluetooth Chatter показаний на рис. 1.4.

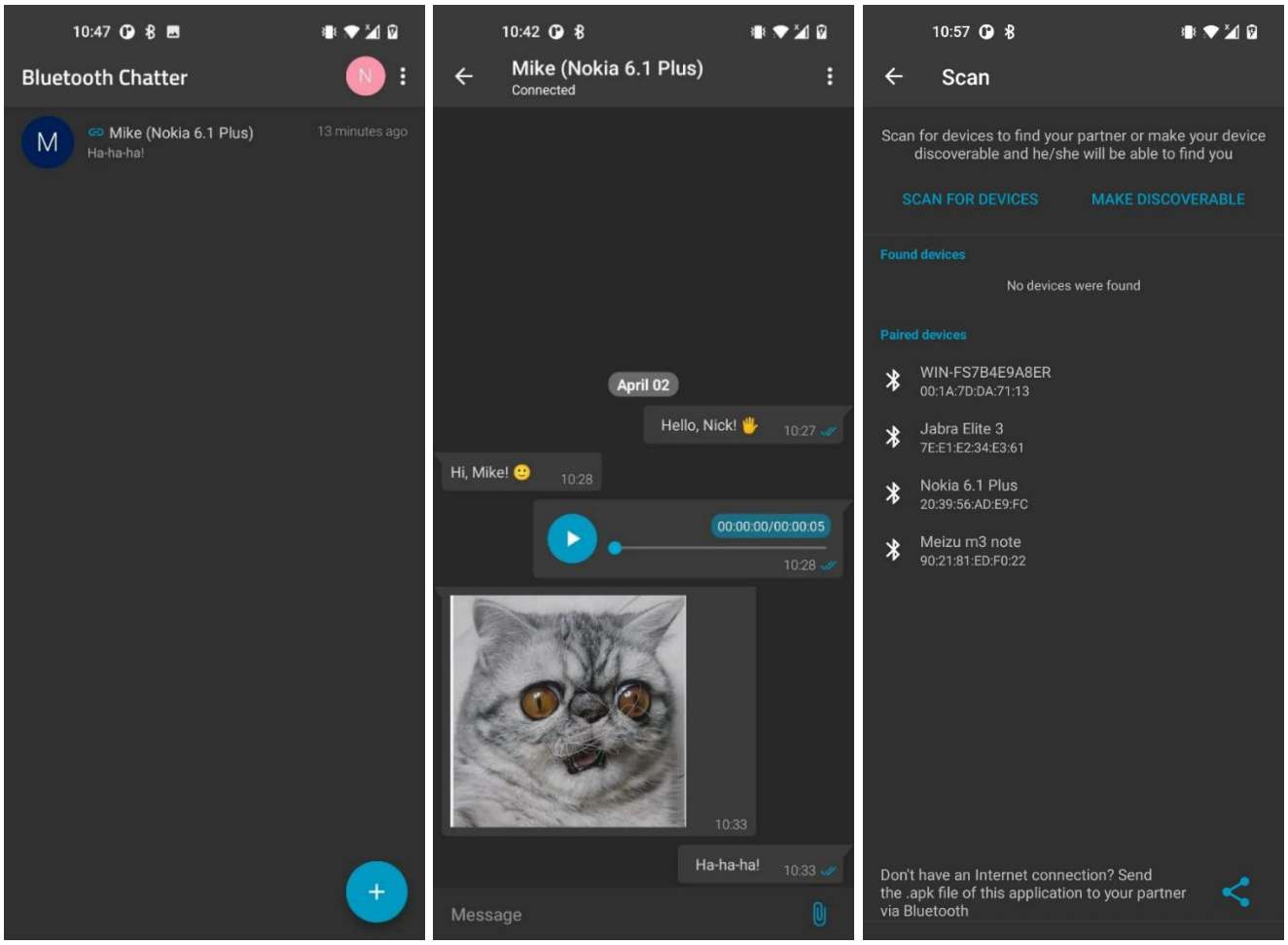


Рисунок 1.4 – Вигляд інтерфейсу застосунку Bluetooth Chatter

4) Gchat - Bluetooth Chat – чат або емулятор терміналу для Bluetooth. Це дозволяє Android гаджетам приєднатися до будь-яких віддалених Bluetooth пристроїв, що підтримують профіль послідовного порту і обмінюванися з ними інформацією. Можна приєднатися за допомогою пульта дистанційного гаджета як Bluetooth-master і як Bluetooth-slave. Це дає прості підходи до рішення з модулем

Bluetooth і відправити коди керування, ключі та інші дані. Вигляд інтерфейсу застосунку показаний на рис. 1.5.

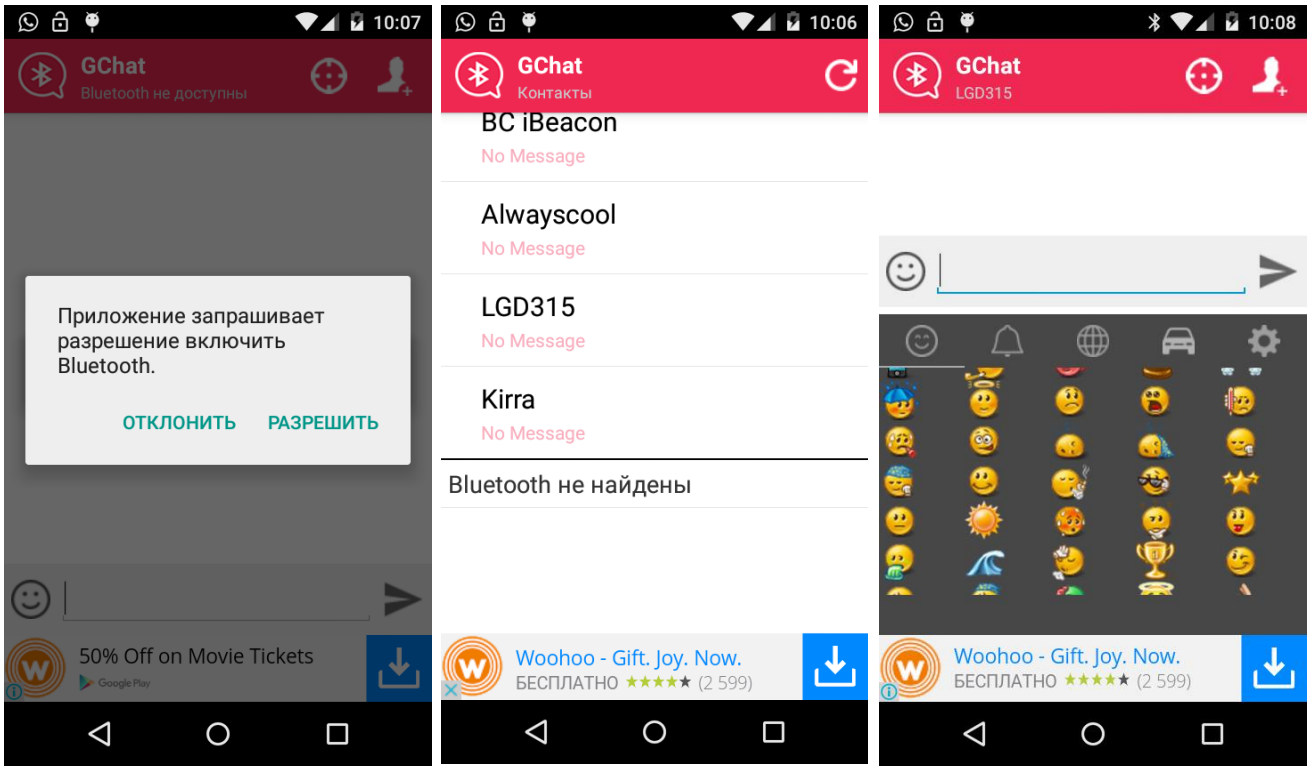


Рисунок 1.5 – Вигляд інтерфейсу застосунку Gchat - Bluetooth Chat

5) Bluetooth Chat (автор: Glodanif). Цей Bluetooth-чат допоможе вам надсилати короткі повідомлення та зображення за допомогою технології Bluetooth. Ви можете спілкуватися з друзями, якщо ви перебуваєте в діапазоні Bluetooth і не маєте доступу до Інтернету. Це буде корисно для студентів, якщо немає Wi-Fi у школі, для мандрівників, щоб спілкуватися між наметами у горах, та у багатьох інших випадках.

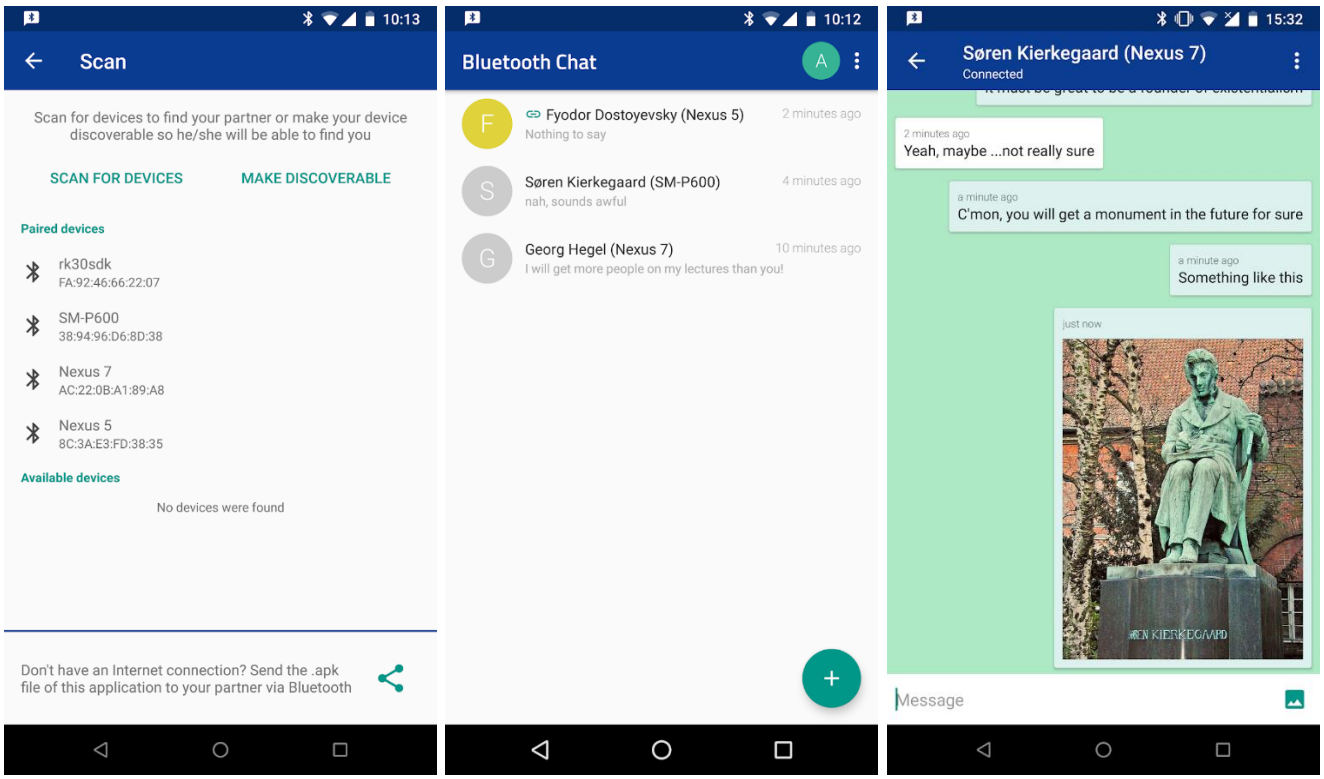


Рисунок 1.6 – Вигляд інтерфейсу застосунку Bluetooth Chat (автор: Glodanif)

Тепер проведемо SWOT-аналіз розглянутих вище конкурентних мобільних Android-застосунків з маркету Google Play. Результати приведені у табл. 1.1.

Таблиця 1.1 – SWOT-аналіз конкурентів з маркету Google Play

		Simple Bluetooth Chat	**Bluetooth Chat (автор: vapps2015)**	**Bluetooth Chatter**	**Gchat - Bluetooth Chat**	**Bluetooth Chat (автор: Glodanif)**
S		- Простота	- Можливість безпечного з'єднання; - Керування видимістю	- Передача файлів; - Наявність профілю користувача	- Робота з блютуз-модулями; - Видалення повідомлень	- Зручний інтерфейс
W		- Чати не зберігаються	- Чати не зберігаються	- Немає настройки інтерфесу	- Не помічаються прочитані повідомлення	- Передача тільки файлів зображень
O	Увімкнення Bluetooth у застосунку	–	–	–	+	–

Продовження табл. 1.1

		Simple Bluetooth Chat	**Bluetooth Chat (автор: vapps2015)**	**Bluetooth Chatter**	**Gchat - Bluetooth Chat**	**Bluetooth Chat (автор: Glodanif)**
	Список спарованих пристроїв	+	+	+	+	+
	Список інших доступних пристроїв	+	+	+	+	+
	Вказання часу відправлення повідомлення	+	+	+	–	+
	Смайлики у повідомленні	+	–	–	+	–
	Передавання файлів	–	–	+	–	+
	Збереження чатів	–	–	+	+	+

1.3 Визначення вимог до застосунку

З урахуванням проведеного вище аналізу, визначимо список функціональних вимог до мобільного застосунку, який реалізуватиме функції мобільного блютуз-чату:

– можливість увімкнути Bluetooth безпосередньо у застосунку;

– пошук спарованих Bluetooth-пристроїв;

– старт Bluetooth-сервера;

– підключення до наявного у списку пристрою;

– відправлення та отримання текстових повідомлень;

– автоматичне додавання часу відправлення до кожного повідомлення;

Функціональні можливості для користувача представимо у вигляді Use case діаграми на рис. 1.7.

Також, зазначимо системні вимоги:

– версія Android API пристроїв не нижче 26, для кращої сумісності;

– робота Bluetooth-модуля у звичайному режимі, тобто, не в режимі BLE (Bluetooth Low Energy) – який вважається автором недоцільним для використання через зменшену дальність зв’язку.

Рисунок 1.7 – Use case діаграма застосунку для актора Користувач

Нарешті, визначимо вимоги до користувацького інтерфейсу:

– інтерфейс користувача має бути зручним та інтуїтивно зрозумілим;

– використання сучасних дизайнів Material Design;

– чітка та лаконічна візуалізація інформації.

Отже, зазначені вимоги є підґрунтям для виконання подальших етапів: проєктування, розробки та тестування застосунку.

1.4 Вибір засобів розробки

Як випливає з назви теми кваліфікаційної роботи, мобільний застосунок буде створений мовою Kotlin. Але, як відомо, кожен мобільний Android-застосунок виконуються на пристрої у окремому екземплярі віртуальної машини Java. Вибір мови програмування Kotlin обумовлений наступними міркуваннями:

– мову Kotlin проголошено компанією Google основною для мобільної розробки у IDE Android Studio, на заміну Java;

– сучасність, компактність та виразність Kotlin, порівняно з Java;

– наявність у Kotlin ефективного механізму корутин (coroutines) для паралельного та асинхронного програмування;

– сумісність з кодом Java та підтримка Java-бібліотек;

– можливість використання декларативного способу створення інтерфейсу користувача (UI) засобами Jetpack Compose.

Декларативний підхід до створення UI (на відміну від статичного підходу з використанням XML-розмітки View-елементів) є сучасною тенденцією і активно впроваджується у інших засобах мобільної розробки, таких як Flutter та XCode.

Вибір платформи Android обумовлений більшою розповсюдженістю Android-пристроїв порівняно з такими від компанії Apple. До того ж, для створення IoS-застосунків розробник має використовувати MacBook чи десктоп iMac для роботи в IDE, та мобільний пристрій iPhone. Така конфігурація насьогодні не є доступною для будь-якого українського розробника.

У свою чергу, вибір Android Studio у якості середовища розробки обумовлений тим, що вона є офіційною IDE від Google, має детальну онлайн документацію [4], та постійно оновлюється та розвивається.

Також, розглядалися кросплатформні IDE: Flutter та Kotlin Multiplatform. Але, перша не є нативним середовищем розробки, що є важливим аргументом для роботи у предметній області Bluetooth-функціоналу. Друга ж – ще не достатньо розвинута, порівняно з Android Studio. На рис. 1.8 показані логотипи обраних засобів розробки.

Рисунок 1.8 – Логотипи обраних засобів розробки

1.5 Висновки до розділу 1

У першому розділі проведено огляд предметної області та здійснено аналіз наявних аналогів, на підставі якого сформовано перелік необхідних вимог для розробки мобільного застосунку Bluetooth-чат для платформи Android.

Також приведено аргументи для вибору платформи та засобів розробки.

Результати, отримані у розділі, будуть використані як вхідні дані для наступних розділів.

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА МОБІЛЬНОГО ЗАСТОСУНКУ

2.1 Архітектура мобільного застосунку

Bluetooth-з’єднання передбачає участь двох пристроїв, що однозначно вказує на застосування клієнт-серверної архітектури у проєктованому мобільному застосунку (рис. 2.1). Для обміну даними між двома пристроями, спочатку треба встановити з’єднання засобами Bluetooth Socket (по аналогії з сокетами TCP/IP). При цьому, будь-який з пристроїв може виступати у ролі як клієнта, так і сервера.

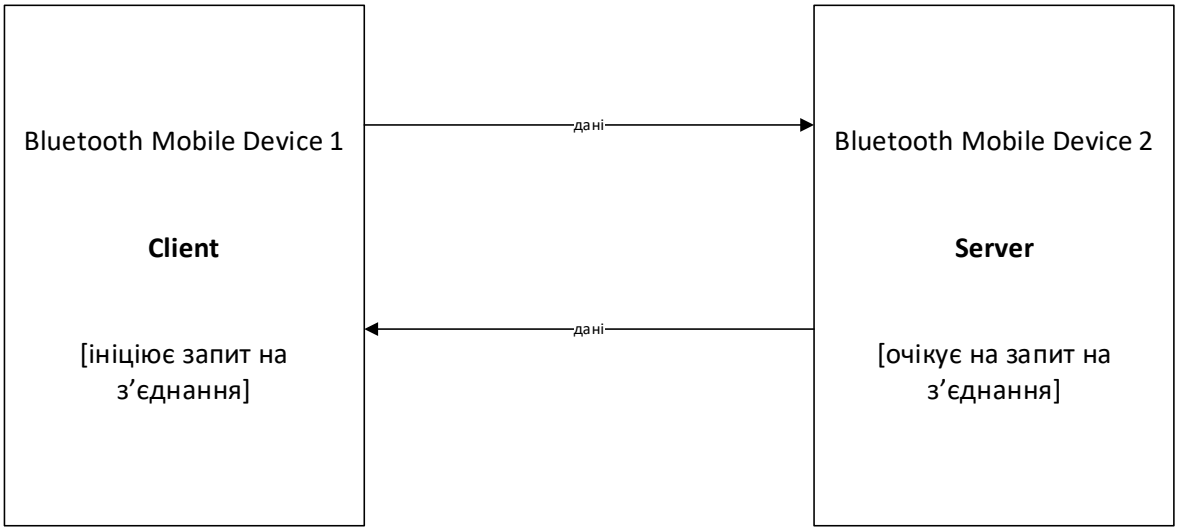


Рисунок 2.1 – Схема архітектури клієнт-сервер для двох Bluetooth-пристроїв

2.2 Моделювання програмної системи

Для моделювання та подальшої реалізації програмної системи мобільного застосунку обраний патерн MVVM [6], який широко застосовується у сучасних мобільних застосунках на Kotlin. Схема роботи шаблону MVVM (Model-View-ViewModel) показана на рис. 2.2.

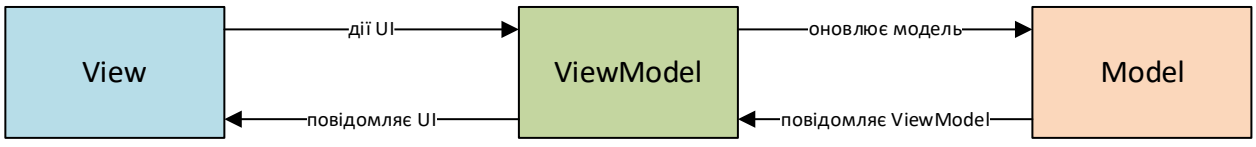


Рисунок 2.2 – Схема роботи шаблону MVVM

Даний патерн проєктування забезпечує відв’язування інтерфейсу користувача (View) від даних (Model), через посередника (ViewModel). При цьому, на відміну від схожих патернів MVC та MVP, ViewModel нічого не відомо про View. Це дозволяє безпечно змінювати інтерфейс застосунку. В загалом, патерн MVVM надає гнучкість і полегшує розробку та тестування Android-застосунків.

Застосування MVVM у якості шаблону проєктування часто передбачає необхідність впровадження залежностей між класами/екземплярами в рамках проєкту. Цьому відповідає патерн DI (англ. Dependency Injection, ін’єкція залежностей) [7]. Він знаходить застосування, коли якомусь класу або екземпляру класу необхідно використовувати екземпляр або окремі методи іншого класу. Звичайно, це можна реалізувати шляхом створення сутності другого класу у тілі першого. Але, це може призвести до ситуації, коли неконтрольовано створюються екземпляри класів, які мають функціонувати лише в єдиному екземплярі. Для запобігання цьому, можна передати такий об’єкт у конструктор класу-споживача. Також, при цьому треба створити файл-контейнер застосунку (AppContainer), в якому ін’єктовані залежності створюватимуться в єдиному екземплярі для всього застосунку. Описане являє собою сутність шаблону DI і може реалізовуватися як власноруч, так і з використанням спеціальних бібліотек, наприклад, Dagger Hilt [8, 9]. У роботі використовуватиметься ін’єкція залежностей за допомого цієї бібліотеки.

2.3 Розробка застосунку

Структура проєкту в Android Studio буде мати вигляд, як показано на рис. 2.3.

Як видно з рис. 2.3, структура проєкту складається з трьох рівнів:

– рівень даних (data.chat);
– рівень домену (domain.chat);
– рівень презентації (presentation).

Каталог **data.chat** відповідає рівню даних та містить такі файли:

– AndroidBluetoothController;
– BluetoothDataTransferService;
– BluetoothDeviceMapping.kt;
– BluetoothMessageMapper.kt;
– BluetoothStateReceiver;
– FoundDeviceReceiver.

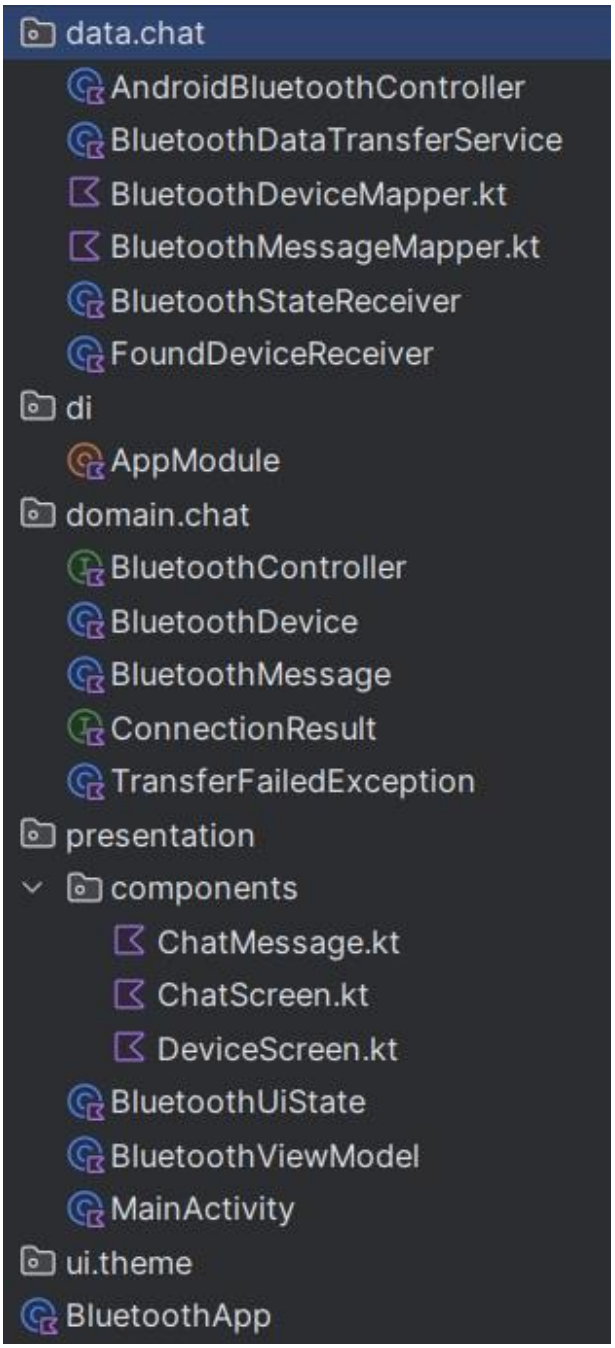


Рисунок 2.3 – Структура проєкту в Android Studio

Діаграма класів рівня даних показана на рис. 2.4. Розглянемо призначення його складових.

Файл *AndroidBluetoothController* – це клас, який відповідає за всі операції, пов’язані з модулем блютуз мобільного пристроя. Його функціонал відображений на рис. 2.5 у вигляді інтелект-карти (mindmap).

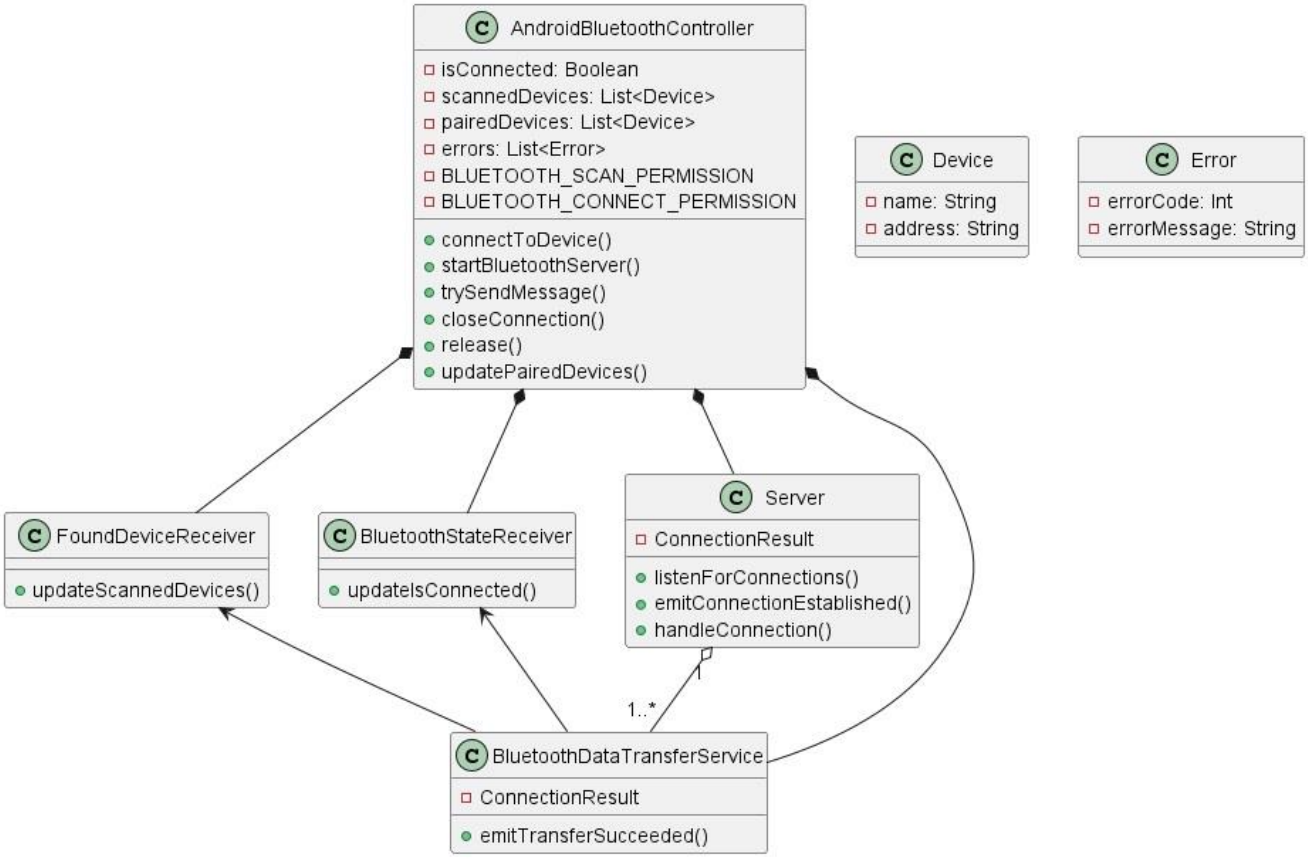


Рисунок 2.4 – Діаграма класів рівня даних

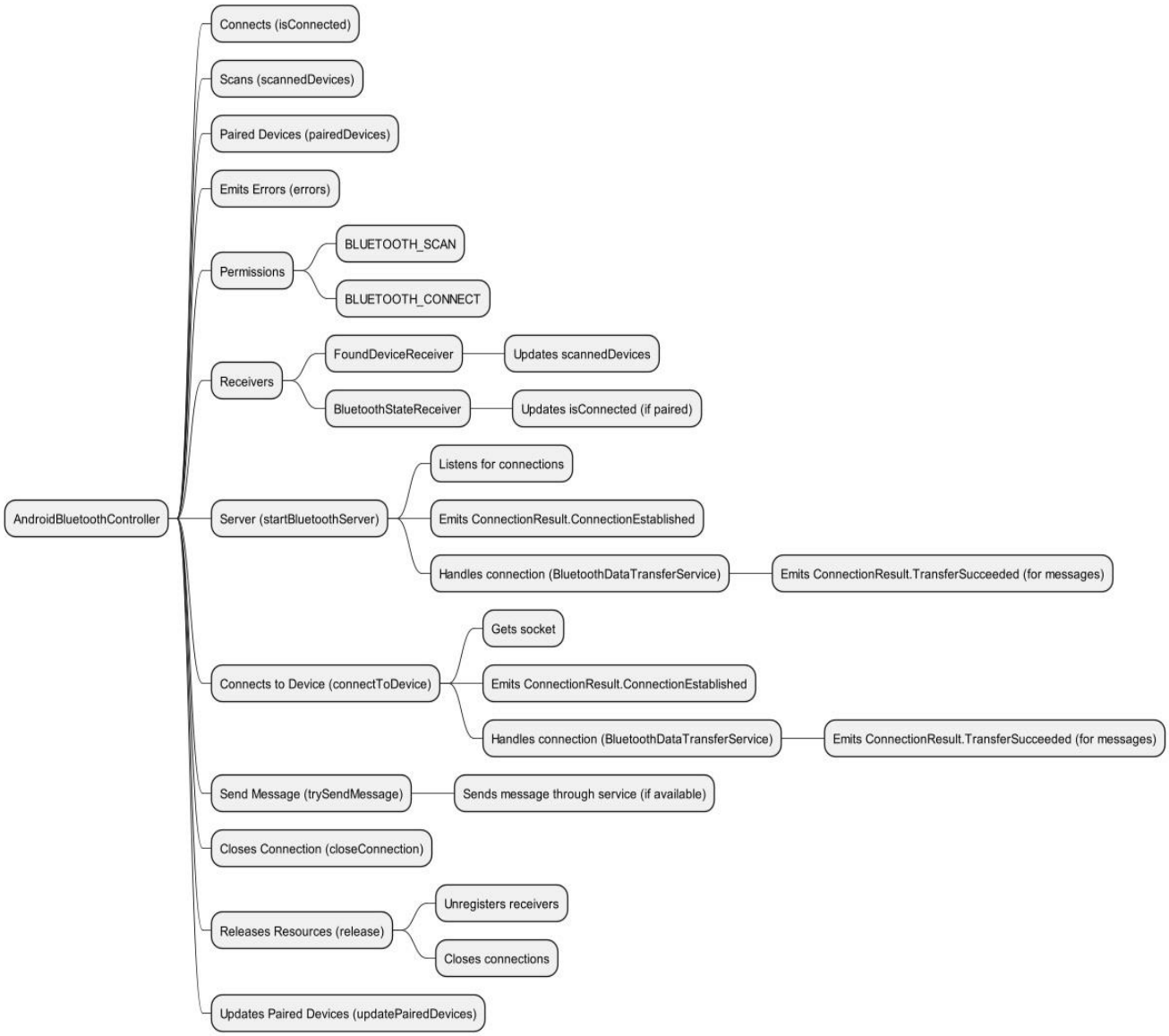


Рисунок 2.5 – Функціонал класу AndroidBluetoothController

Діаграма станів класу AndroidBluetoothController показана на рис. 2.6. Стани реалізуються у Kotlin-коді як MutableStateFlow [10], наприклад:

```
private val _isConnected = MutableStateFlow(false)
override val isConnected: StateFlow<Boolean>
```

Файл *BluetoothDataTransferService* – клас, який відповідає за приймання та відправлення повідомлень у чаті. Він реалізує дві функції:

```
fun listenForIncomingMessages(): Flow<BluetoothMessage>
```

та

```
suspend fun sendMessage(bytes: ByteArray): Boolean
```

Остання помічена як suspend, тому що буде запускатися з корутини.

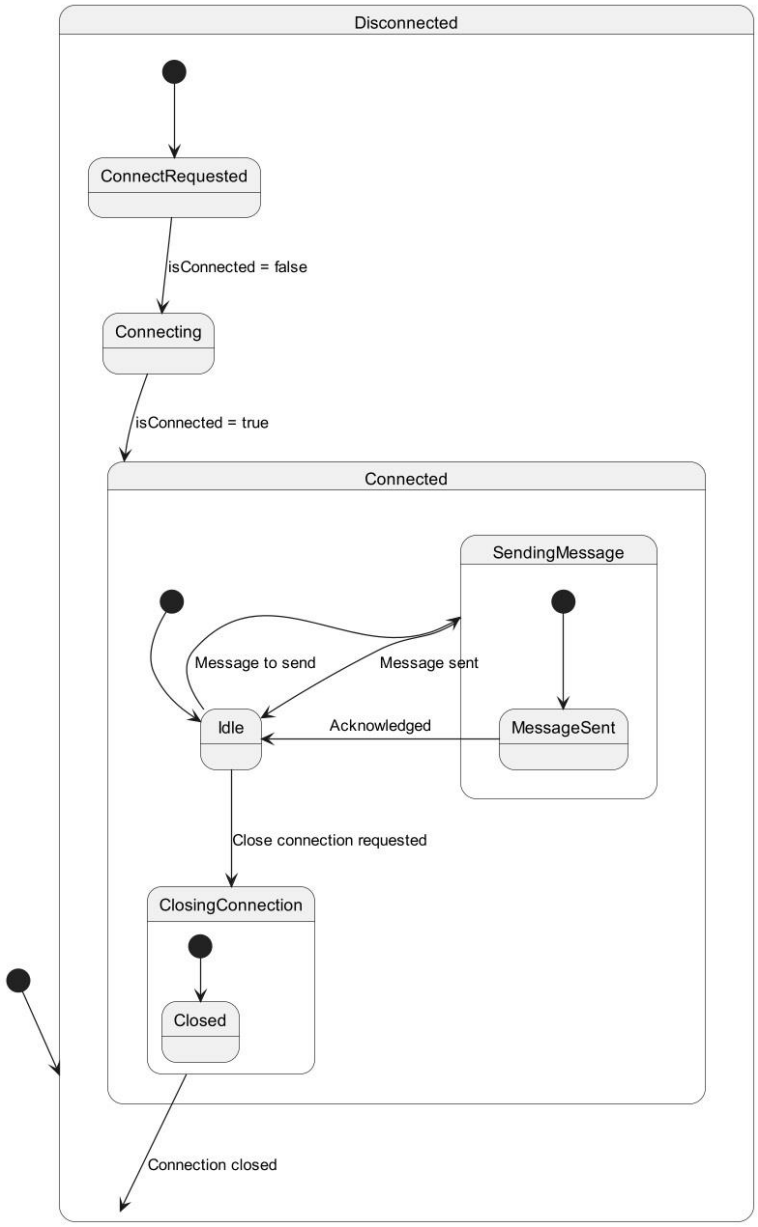


Рисунок 2.6 – Діаграма станів класу AndroidBluetoothController

Файл *BluetoothDeviceMapping.kt* містить функцію-маппер, яка закріплює за блютуз-пристроєм ім’я name та фізичну адресу address.

Файл BluetoothMessageMapper.kt містить функцію String.toBluetoothMessage(isFromLocalUser: Boolean), за допомогою якої текст кожного введеного повідомлення представляється у формі екземпляру класу BluetoothMessage, та функцію BluetoothMessage.toByteArray(), яка перетворює його на масив байтів для подальшого передавання через блютуз-сокет.

Файли *BluetoothStateReceiver.kt* та *FoundDeviceReceiver.kt* – містять приймачі широкомовних повідомлень (BroadCast Receivers []), які відстежують зміни у статусі блютуз-пристроїв та у списку знайдених пристроїв, відповідно.

Каталог **domain.chat** складається з файлів:

- BluetoothController.kt;
- BluetoothDevice.kt;
- BluetoothMessage.kt;
- ConnectionResult.kt;
- TransferFailedException.kt.

Файл *BluetoothController.kt* є інтерфейсом який реалізується через розглянутий вище клас AndroidBluetoothController.

Файл *BluetoothDevice* – data-клас для блютуз-пристроїв, містить name (ім’я пристрою) та address (фізична адреса).

Файл *BluetoothMessage* – data-клас, який задає формат об’єкта-повідомлення. Містить наступні поля:

```
val message: String,          // Текст повідомлення.
val senderName: String,       // Ім’я відправника.
val isFromLocalUser: Boolean  // Флаг власного повідомлення
```

Файл *ConnectionResult* задає тип для результату з’єднання у вигляді sealed interface. Варіанти результатів з’єднання:

- ConnectionEstablished – з’єднання встановлено;
- TransferSucceeded(message) – передавання успішне;
- Error(message) – помилка передавання повідомлення.

Файл *TransferFailedException* – містить клас для об’єкту помилки.

Каталог **presentation** містить:

- підкаталог ***components***, який містить Compose-функції що реалізують складові інтерфейсу застосунку:
  - ChatMessage.kt – повідомлення чату (рис. 2.7);
  - ChatScreen.kt – екран чату;
  - DeviceScreen.kt – екран пошуку та вибору пристроїв;
- Файл *BluetoothUiState* – data-клас, який містить перелік станів UI:

```
data class BluetoothUiState(
    val scannedDevices: List<BluetoothDevice> = emptyList(),
    val pairedDevices: List<BluetoothDevice> = emptyList(),
    val isConnected: Boolean = false,
    val isConnecting: Boolean = false,
    val errorMessage: String? = null,
    val messages: List<BluetoothMessage> = emptyList()
)
```

– Файл *BluetoothViewModel* являє собою єдину ViewModel застосунку;

– Файл *MainActivity* – головна активіті мобільного застосунку.

Рисунок 2.7 – Попередній вигляд повідомлення чату

Взаємозв’язок BluetoothViewModel з іншими компонентами застосунку показаний на діаграмі класів на рис. 2.8.

Діаграму станів BluetoothViewModel подано на рис. 2.9.

Діаграма діяльності (рис. 2.10) ілюструє алгоритміку функціонування BluetoothViewModel.

В свою чергу, файл (клас) MainActivity:

– ініціює запуск системної служби BluetoothManager, за допомогою якого створює bluetoothAdapter;

– оголошує Boolean-флаг isBluetoothEnabled, який показує чи ввімкнений модуль Bluetooth;

– оперує логікою системних дозволів на використання Bluetooth у застосунку та дозволяє ввімкнути модуль прямо у застосунку, якщо він не активний.

– в залежності від станів ViewModel, змінює поточний екран UI та задає його базовий вміст.

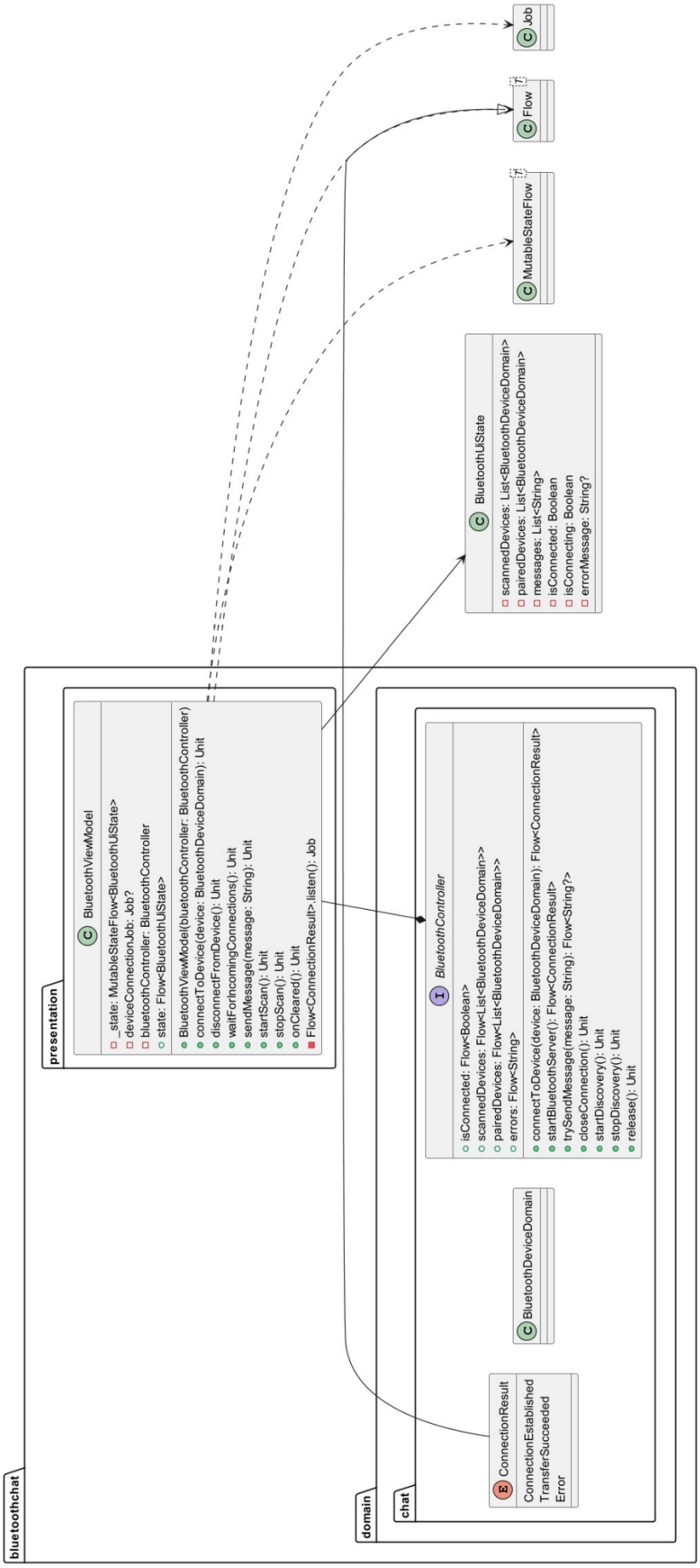


Рисунок 2.8 – Взаємозв'язок BluetoothViewModel з іншими компонентами

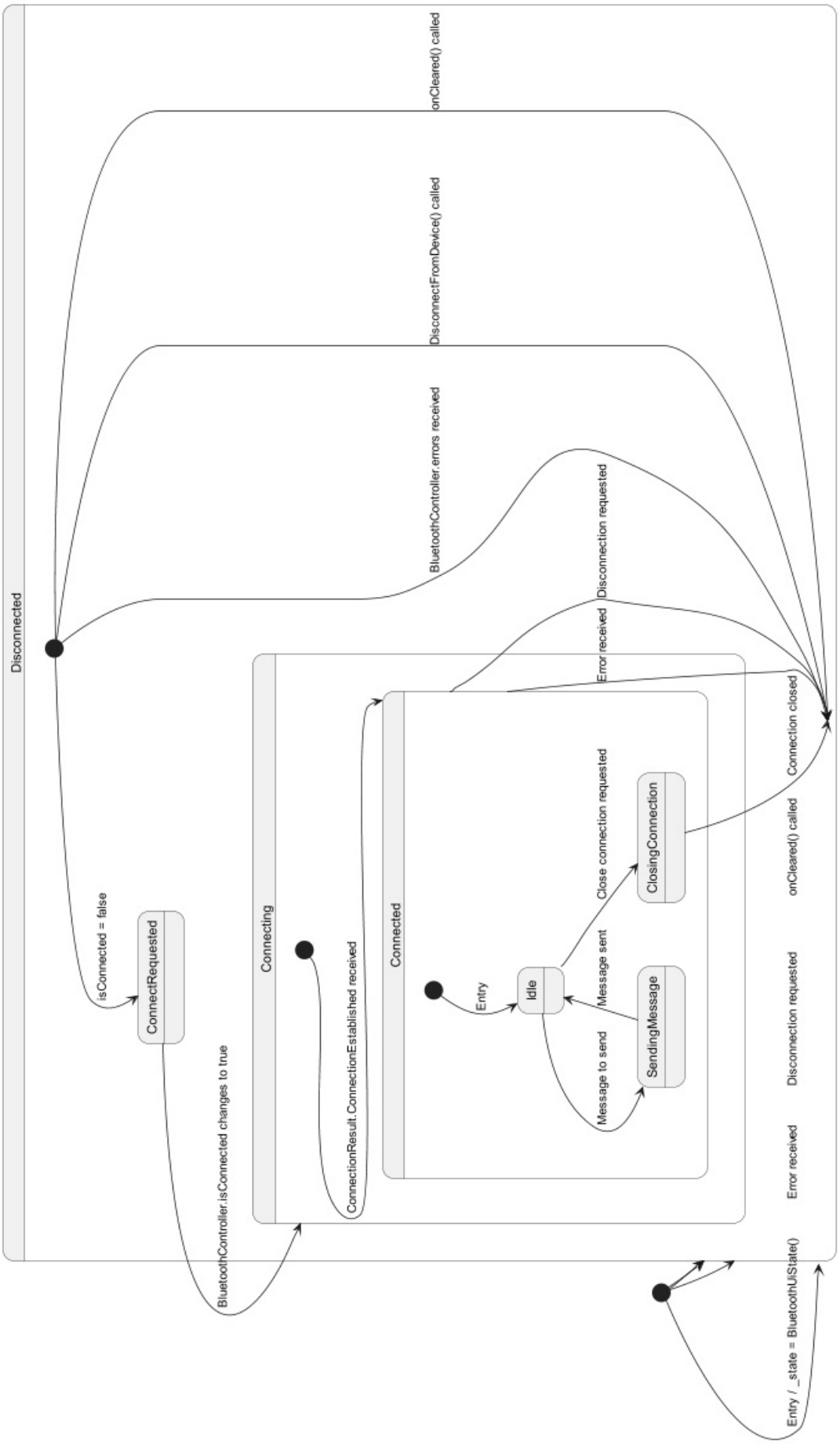
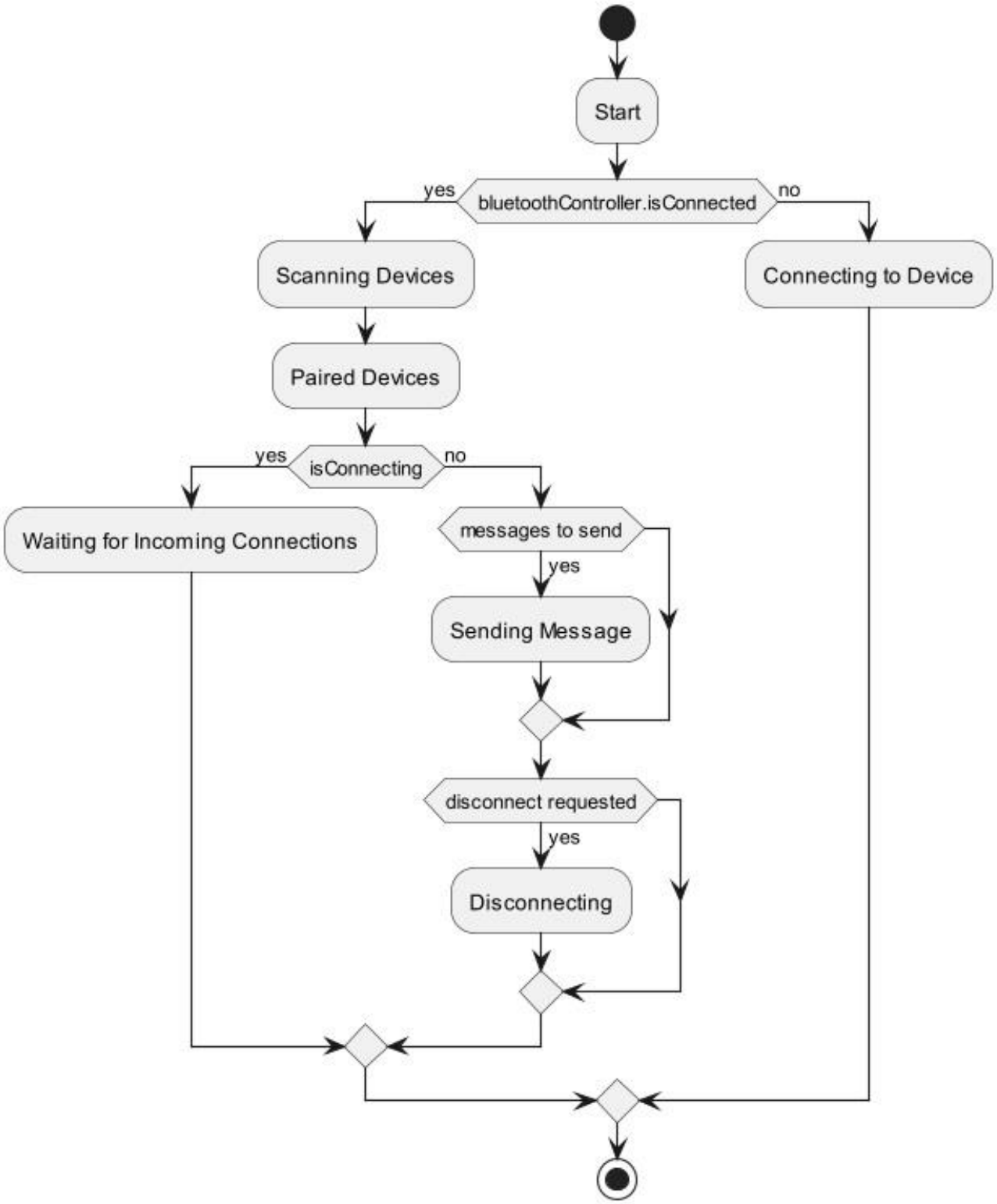


Рисунок 2.9 – Діаграма станів BluetoothViewModel



Діаграма діяльності BluetoothViewModel

Також, у структуру проєкту включений каталог **di** з файлом *AppModule*, та окремий файл *BluetoothApp*. Ці файли забезпечують ін’єкцію залежності: єдиний екземпляр класу (тобто, Singleton) bluetoothController впроваджується у клас

BluetoothViewModel. В даному застосунку це робиться з використанням відомої бібліотеки Hilt.

Окремо слід зазначити, що у файлі AndroidManifest.xml застосунку необхідно додати дозволи на використання у ньому модуля Bluetooth, що виглядає так:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.BLUETOOTH"
android:maxSdkVersion="30" />
    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
    <uses-permission android:name="android.permission.BLUETOOTH_SCAN"
        android:usesPermissionFlags="neverForLocation" />

    <uses-feature android:name="android.hardware.bluetooth" />
...
```

2.4 Висновки до розділу 2

У розділі описано архітектуру мобільного застосунку, яка на реалізовує модель клієнт-сервер для двох мобільних блютуз-пристроїв, при чому будь-який з них може виступати у ролі як сервера так і клієнта, що залежить від вибору користувачів.

При моделюванні програмної системи був вибраний шаблон MVVM, який є широко розповсюдженим для багатьох сучасних Android-застосунків, та пояснюється необхідність використання шаблону Dependency Injection.

Описано структуру застосунку. Процес розробки ілюструється за допомогою UML-діаграм.

Результати розділу виступають підґрунтям для тестування розробленого застосунку у наступному розділі.

3 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

3.1 Модульне тестування застосунку

Для модульного тестування будемо використовувати бібліотеку Junit.

Одним з основних класів застосунку є клас AndroidBluetoothController, тому складемо для нього наступні тести.

1) Тести startDiscovery та stopDiscovery:

```
@Test
fun `startDiscovery starts discovery and emits no errors when permission
granted`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_SCAN) }
returns PackageManager.PERMISSION_GRANTED
    val mockBluetoothAdapter = mockk<BluetoothAdapter>()
    val controller = AndroidBluetoothController(mockContext)

    controller.startDiscovery()

    verify { mockBluetoothAdapter.startDiscovery() }
    verify { mockContext.registerReceiver(any(), any()) }
    verify { controller._errors.emit(any()) wasNot called }
}

@Test
fun `startDiscovery does not start discovery and emits error when
permission denied`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_SCAN) }
returns PackageManager.PERMISSION_DENIED
    val controller = AndroidBluetoothController(mockContext)

    controller.startDiscovery()

    verify { mockBluetoothAdapter.startDiscovery() wasNot called }
    verify { mockContext.registerReceiver(any(), any()) wasNot called }
    verify { controller._errors.emit(any()) }
}

@Test
fun `stopDiscovery cancels discovery when permission granted`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_SCAN) }
returns PackageManager.PERMISSION_GRANTED
```

```
    val mockBluetoothAdapter = mockk<BluetoothAdapter>()
    val controller = AndroidBluetoothController(mockContext)

    controller.startDiscovery()
    controller.stopDiscovery()

    verify { mockBluetoothAdapter.cancelDiscovery() }
}

@Test
fun `stopDiscovery does nothing when permission denied`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_SCAN) }
returns PackageManager.PERMISSION_DENIED
    val controller = AndroidBluetoothController(mockContext)

    controller.stopDiscovery()

    verify { mockBluetoothAdapter.cancelDiscovery() wasNot called }
}
```

2) Тест startBluetoothServer:

```
@Test
fun `startBluetoothServer emits ConnectionEstablished when connection
established`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_CONNECT) }
returns PackageManager.PERMISSION_GRANTED
    val mockServerSocket = mockk<BluetoothServerSocket>()
    val mockClientSocket = mockk<BluetoothSocket>()
    val mockBluetoothAdapter = mockk<BluetoothAdapter>()
    every { mockBluetoothAdapter.listenUsingRfcommWithServiceRecord(any(),
any()) } returns mockServerSocket
    val controller = AndroidBluetoothController(mockContext)

    val flow = controller.startBluetoothServer()
    val collector = flow.test()

    collector.awaitSubscription()
    collector.assertValue(ConnectionState.ConnectionEstablished)
}

@Test
fun `startBluetoothServer emits Error when IOException occurs during
connection`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_CONNECT) }
returns PackageManager.PERMISSION_GRANTED
```

```
    val mockServerSocket = mockk<BluetoothServerSocket>()
    val mockBluetoothAdapter = mockk<BluetoothAdapter>()
    every { mockBluetoothAdapter.listenUsingRfcommWithServiceRecord(any(),
any()) } returns mockServerSocket
    every { mockServerSocket.accept() } throws IOException()
    val controller = AndroidBluetoothController(mockContext)

    val flow = controller.startBluetoothServer()
    val collector = flow.test()

    collector.awaitSubscription()
    collector.assertValue(ConnectionState.Error(reason = "Connection was
interrupted"))
}
```

3) Тест connectToDevice:

```
@Test
fun `connectToDevice emits ConnectionEstablished when connection
successful`() {
    val mockContext = mockk<Context>()
    every {
mockContext.checkSelfPermission(Manifest.permission.BLUETOOTH_CONNECT) }
returns PackageManager.PERMISSION_GRANTED
    val mockDevice = mockk<BluetoothDevice>()
    val mockSocket = mockk<BluetoothSocket>()
    val mockBluetoothAdapter = mockk<BluetoothAdapter>()
    every { mockBluetoothAdapter.getRemoteDevice(any()) } returns
mockDevice
    every { mockDevice.createRfcommSocketToServiceRecord(any()) } returns
mockSocket
    val controller = AndroidBluetoothController
```

Наступним важливим класом застосунку є клас BluetoothViewModel.

1) Тестування оновлень стану на основі scannedDevices і pairedDevices:

```
@Test
fun `state updates scannedDevices when bluetoothController emits new
devices`() {
    val testViewModel = createViewModel()
    val testDevices = listOf(BluetoothDeviceDomain("name", "address"))

    val flow = flowOf(testDevices)
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        // Inject a mock flow for scanned devices
        testViewModel.bluetoothController.scannedDevices = flow

        // Emit new devices
        flow.emit(testDevices)
```

```
        // Assert that state reflects the new devices

assertThat(testViewModel.state.value.scannedDevices).isEqualTo(testDevices
)
    }
}

@Test
fun `state updates pairedDevices when bluetoothController emits new
devices`() {
    val testViewModel = createViewModel()
    val testDevices = listOf(BluetoothDeviceDomain("name", "address"))

    val flow = flowOf(testDevices)
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        // Inject a mock flow for paired devices
        testViewModel.bluetoothController.pairedDevices = flow

        // Emit new devices
        flow.emit(testDevices)

        // Assert that state reflects the new devices

assertThat(testViewModel.state.value.pairedDevices).isEqualTo(testDevices)
    }
}
```

2) Тестування оновлень стану підключення:

```
@Test
fun `state updates isConnected to true on successful connection`() {
    val testViewModel = createViewModel()
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        // Mock successful connection result
        val connectionResult = mockk<Flow<ConnectionResult>> {
            coEvery { collect } throws CompletionStateException
            coEvery { onEach { it ->
it(ConnectionResult.ConnectionEstablished) } } returns Unit
        }
        testViewModel.deviceConnectionJob = connectionResult.listen()

        // Assert state reflects connection
        assertThat(testViewModel.state.value.isConnected).isTrue()
    }
}

@Test
```

```
fun `state updates isConnected to false on connection error`() {
    val testViewModel = createViewModel()
    val errorMessage = "Connection failed"
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        // Mock connection error
        val connectionResult = mockk<Flow<ConnectionResult>> {
            coEvery { collect } throws CompletionStateException
            coEvery { onEach { it ->
it(ConnectionResult.Error(errorMessage)) } } returns Unit
        }
        testViewModel.deviceConnectionJob = connectionResult.listen()

        // Assert state reflects error
        assertThat(testViewModel.state.value.isConnected).isFalse()

assertThat(testViewModel.state.value.errorMessage).isEqualTo(errorMessage)
    }
}
```

3) Тестування надсилання повідомлень:

```
@Test
fun `state updates messages on successful message sending`() {
    val testViewModel = createViewModel()
    val testMessage = "Test message"
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        val sentMessage = mockk<BluetoothMessage>()
        val mockController = mockk<BluetoothController> {
            coEvery { trySendMessage(testMessage) } returns sentMessage
        }
        testViewModel.bluetoothController = mockController

        testViewModel.sendMessage(testMessage)

        // Assert state reflects sent message

assertThat(testViewModel.state.value.messages.last()).isEqualTo(sentMessag
e)
    }
}
```

4) Тестування обробки помилок:

```
@Test
fun `state updates errorMessage on bluetoothController error`() {
    val testViewModel = createViewModel()
```

```
    val errorMessage = "Bluetooth error"
    val testCoroutineRule = TestCoroutineRule()

    testCoroutineRule.runBlockingTest {
        val errorFlow = flowOf(Throwable(errorMessage))
        val mockController = mockk<BluetoothController> {
            coEvery { errors } returns errorFlow
        }
        testViewModel.bluetoothController = mockController

        // Trigger error emission
        testCoroutineRule.advanceTimeBy(500) // Simulate error collection

        // Assert state reflects error

assertThat(testViewModel.state.value.errorMessage).isEqualTo(errorMessage)
    }
}
```

В результаті виконання всіх приведених вище тестів помилок не виявлено.

3.2 Функціональне тестування застосунку

Даний вид тестування буде проведено для наступної конфігурації з двох розповсюджених сучасних Android-смартфонів:

- Xiaomi Redmi Note 13 (ОС Android 13), «пристрій А»;
- Xiaomi Redmi Note 13 Pro (ОС Android 14), «пристрій Б».

Обидва пристрої обладнані модулем Bluetooth. Перевірка працездатності застосунку передбачає почергове використання ролей клієнта та сервера для обох пристроїв.

Відстань між пристроями становить 10 м, сигнал Bluetooth проходить через 3 стіни.

Встановлення мобільного застосунку проводилося автоматично засобами Android Studio, при запуску проєкту.

На початку тестування пристрій А обрав роль сервера, а пристрій Б, відповідно, клієнта.

На обох пристроях запускається розроблений застосунок «BluetoothChat». На пристрої А модуль блютуз не був увімкнутий, тому на його екрані спочатку з’являється запит на ввімкнення модуля у застосунку (рис. 3.1).

Рисунок 3.1 – Запит на ввімкнення блютуз-модуля у застосунку

Далі на обох пристроях відображається головне вікно застосунку, яке містить список спарованих та сканованих пристроїв. Та три кнопки: «Start scan», «Stop scan» та «Start server». При першому використанні конфігурації двох пристроїв необхідно виконати парування обох пристроїв засобами ОС Android

через меню Налаштування. Після цього обидва пристрої будуть «бачити» один одного у списку «Paired devices» (рис. 3.2).

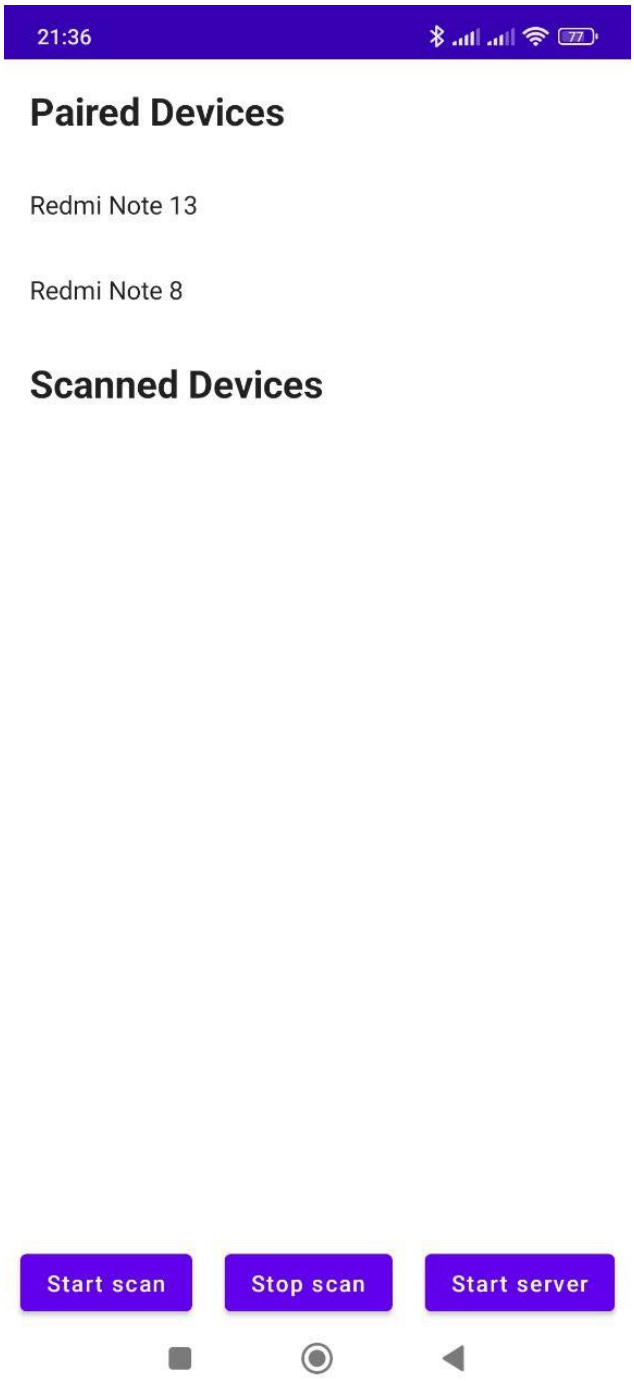


Рисунок 3.2 – Пристрій Б бачить пристрій А у списку Paired devices

Далі, на пристрої А натискається кнопка «Start server», після чого на пристрої Б натискається назва спарованого пристрою А. Після натискання цих

кнопок на екранах обох пристроїв з’являється текст «Connecting…», що сигналізує про процес з’єднання пристроїв (рис. 3.3).

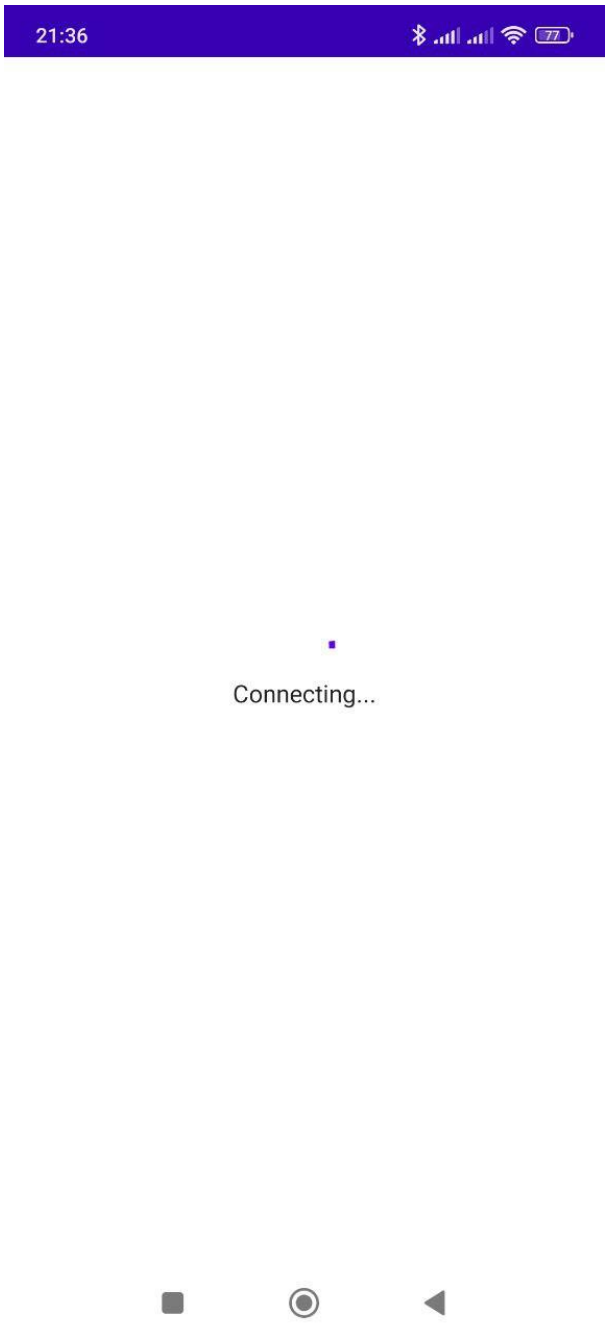


Рисунок 3.3 – З’єднання спарованих пристроїв

Після успішного з’єднання, на екранах обох пристроїв з’являється вікно чату (рис. 3.4). Для виходу з чату (та переривання з’єднання) можна у будь-який час натиснути кнопку «X» біля напису «Messages» у верхній частині екрану.

Рисунок 3.4 – Вікно чату

Далі, будь-який з пристроїв, у даному випадку це Б, починає вводити текст першого повідомлення (рис. 3.5). Після чого натискає кнопку надсилання праворуч.

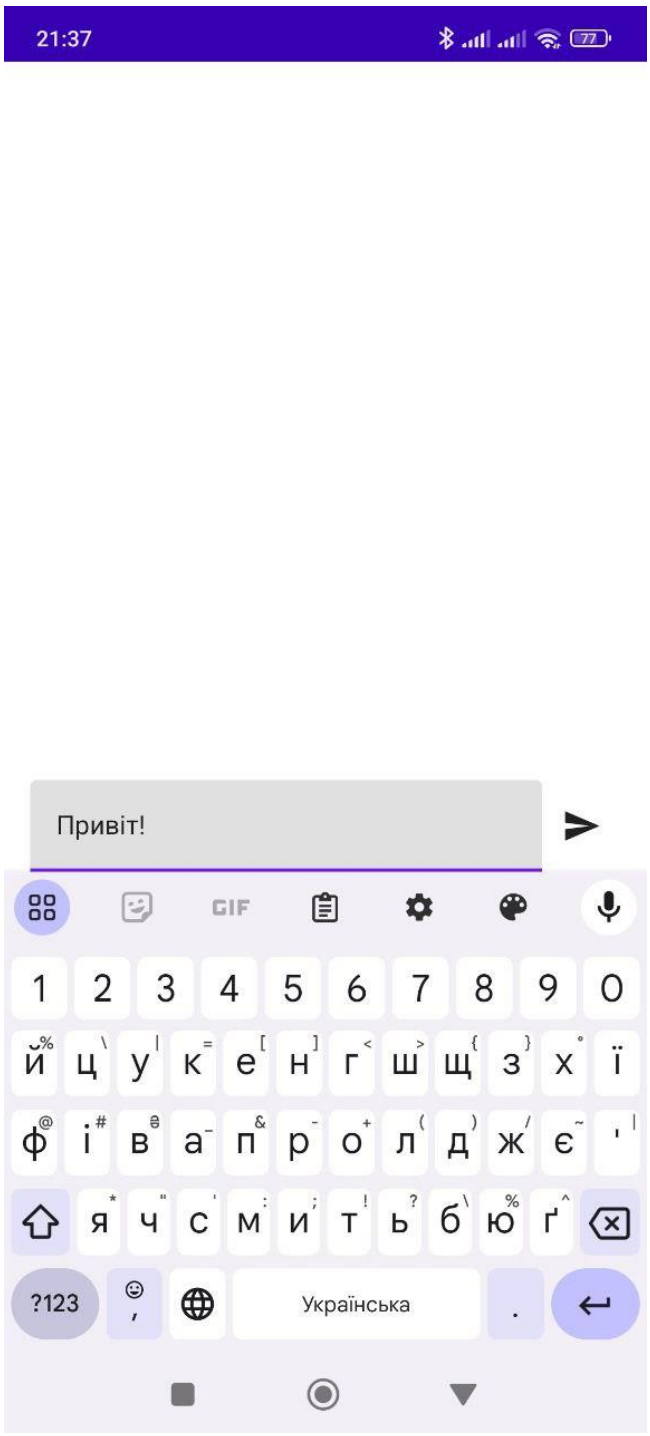


Рисунок 3.5 – Введення тексту повідомлення

Відправник (пристрій Б) бачить на екрані своє повідомлення. До тексту додається назва (ім’я) його пристрою та час надсилання (рис. 3.6).

Рисунок 3.6 – Відображення власного відправленого повідомлення

Далі, пристрій А отримує це повідомлення на надсилає відповідь (рис. 3.7). Власні повідомлення мають червоний фон, а повідомлення з другого пристрою – жовтий. Також, гострі кути повідомлень співбесідників мають різну позицію.

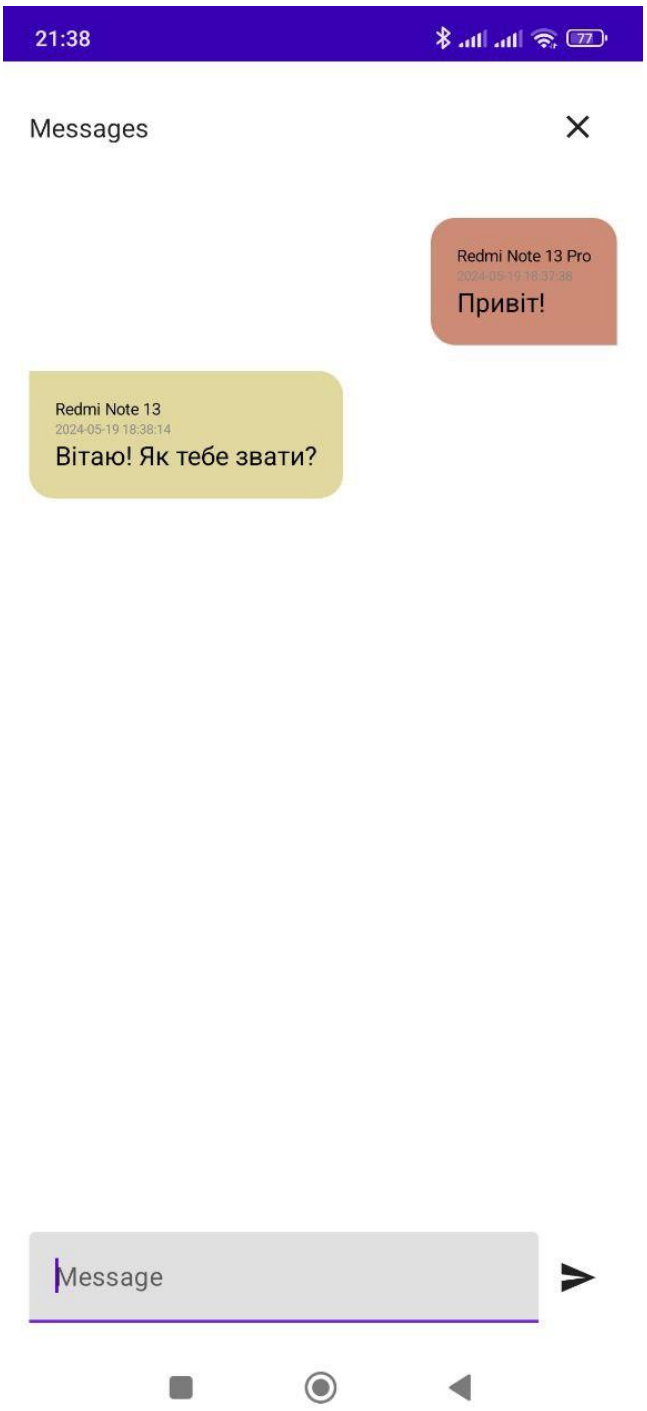


Рисунок 3.7 – Надходження повідомлення-відповіді

Спілкування у чаті продовжується, екран повністю заповнюється повідомленнями (рис. 3.8).

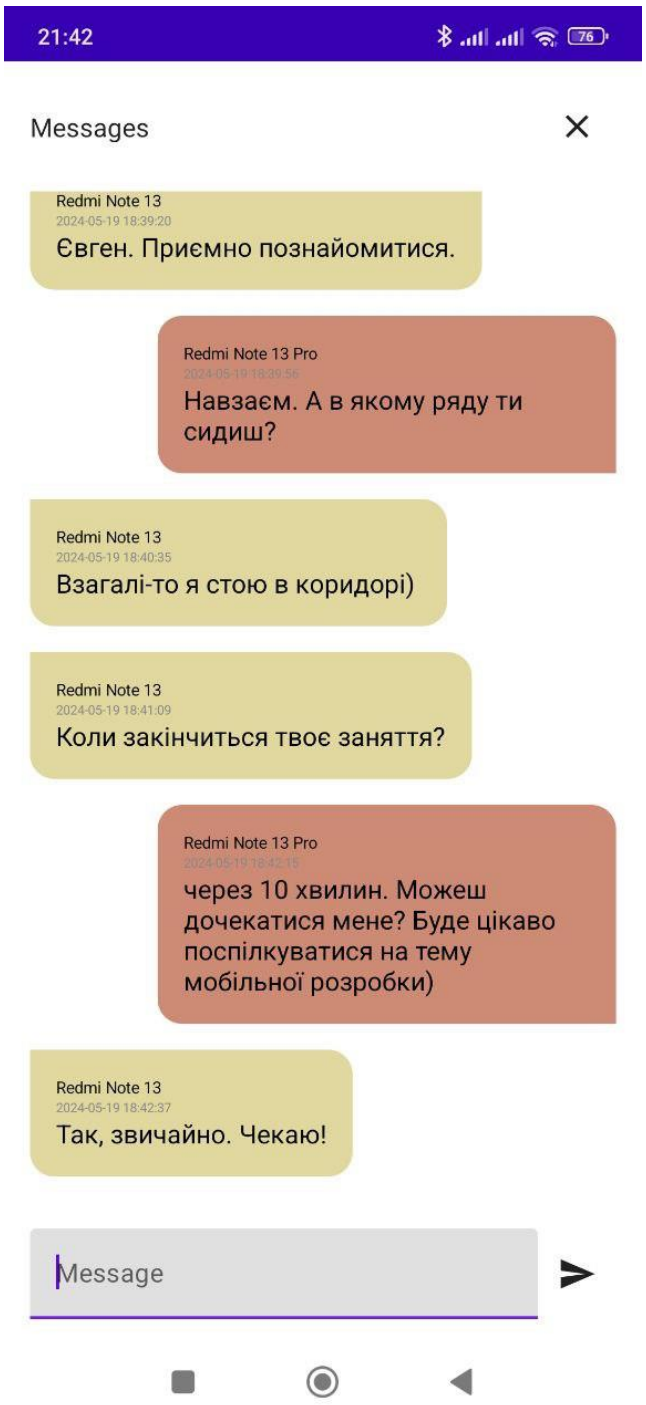


Рисунок 3.8 – Продовження спілкування

При зростанні числа повідомлень, у вікні чату працює вертикальне прокручування (scrolling), як це видно з рис. 3.9.

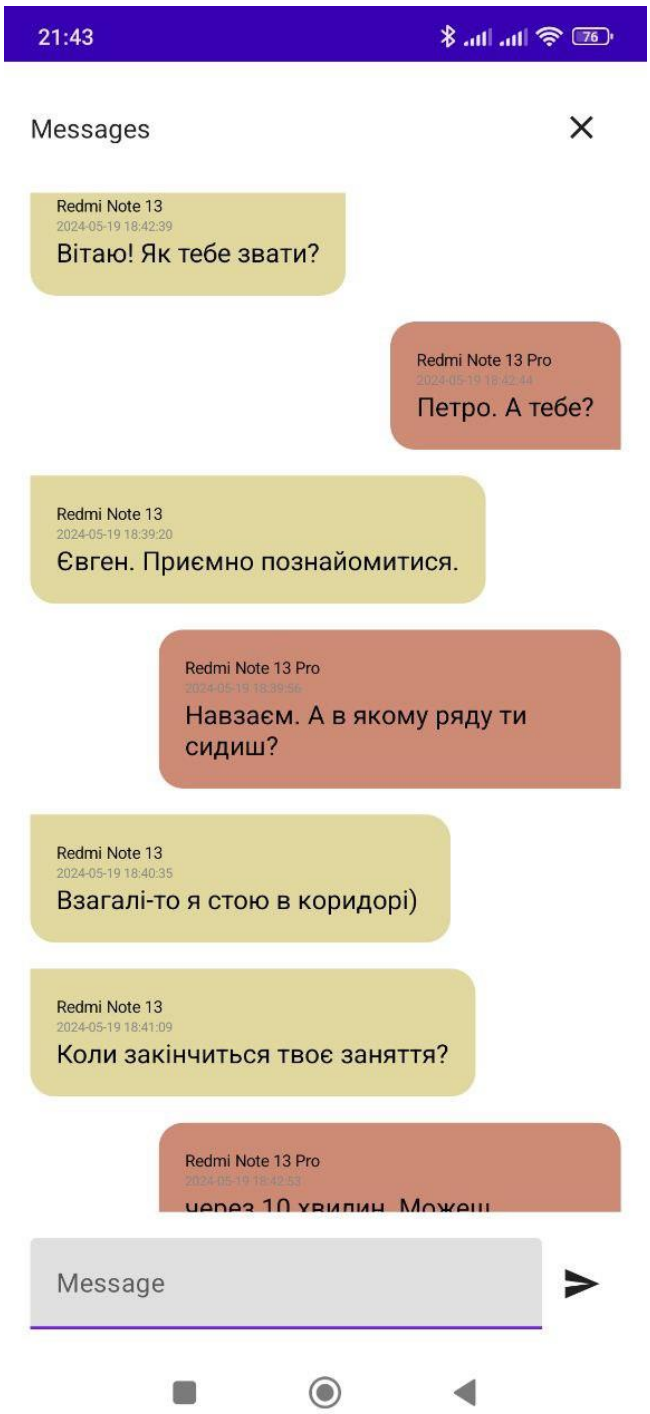


Рисунок 3.9 – Вертикальне прокручування у вікні чату

При натисканні на будь-якому пристрої кнопки «X» відбувається розривання з’єднання, звільнення ресурсів та повернення на початковий екран застосунку на обох пристроях (рис. 3.10).

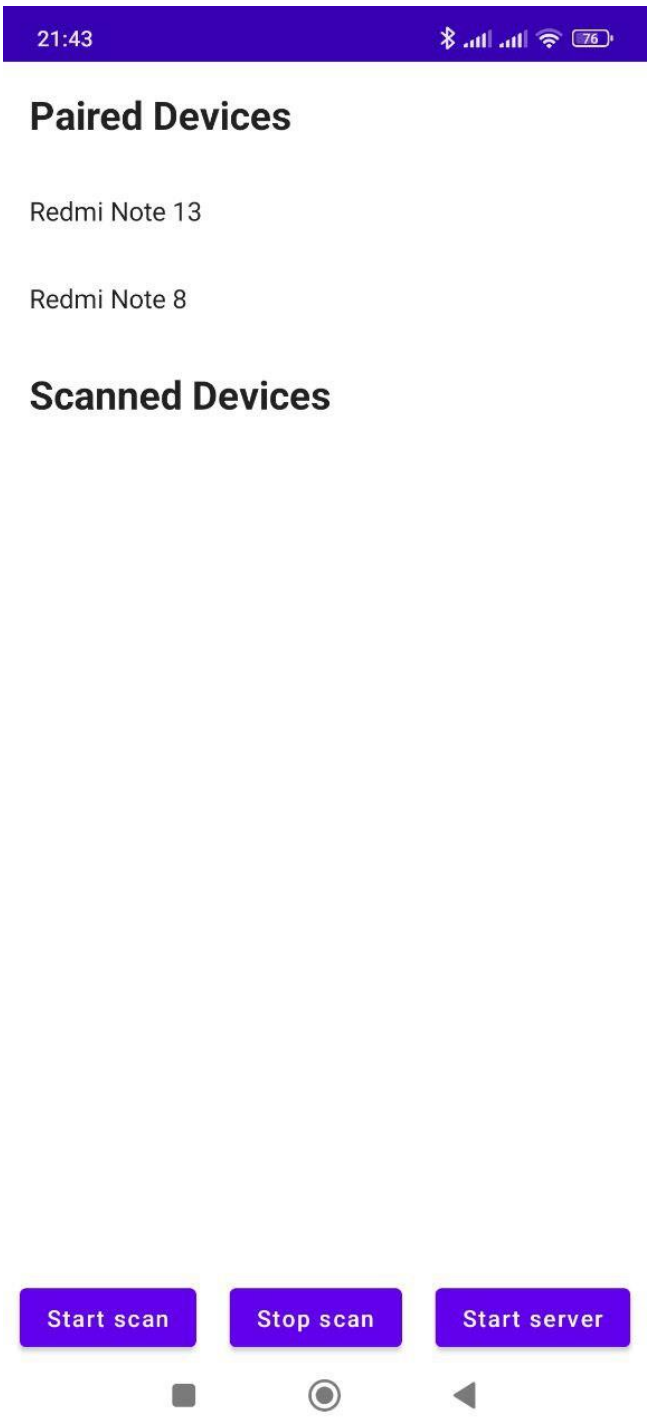


Рисунок 3.10 – Повернення на початковий екран після закриття вікна чату

Після завершення спілкування, пристрої міняються ролями, та повторюють описані вище кроки.

За результатами функціонального тестування для даної конфігурації пристроїв застосунок показав свою повну працездатність та відповідність вимогам.

3.3 Висновки до розділу 3

У розділі проведено модульне та функціональне тестування розробленого мобільного застосунку.

Описаний вигляд інтерфейсу застосунку на кожному етапі функціонального тестування.

Всі тести виявили працездатність застосунку та відповідність до сформульованих у розділі 1 вимог.

ВИСНОВКИ

У кваліфікаційній роботі бакалавра здійснено розробку мобільного Android-застосунку на мові програмування Kotlin з декларативним підходом до створення користувацького інтерфейсу засобами Jetpack Compose.

Під час виконання роботи були виконані всі поставлені завдання, а саме:

– аналіз предметної області та вибір засобів розробки;

– проєктування мобільного застосунку;

– розробка мобільного застосунку;

– тестування розробленого застосунку.

Виконаний аналіз наявних у Google-маркеті аналогів для предметної області «мобільний Bluetooth-чат для платформи Android». Сформульовані вимоги до такого мобільного застосунку. Обґрунтовано вибір платформи та засобів розробки, серед яких:

– нативне середовище розробки Android-застосунків Android Studio;

– сучасна мова програмування Kotlin, що визнана Google як основна для мобільної розробки;

– декларативний підхід до створення UI, з використанням Composable-функцій.

У застосунку використана типова для Bluetooth архітектура Client-Server.

У ході моделювання обрано актуальні для Android-застосунків архітектурні шаблони MVVM та Dependency Injection. Описано трирівневу структуру проєкту, призначення та функціонал його складових, для чого приведена низка UML-діаграм.

Після розробки застосунку було виконано його модульне та функціональне тестування, в ході якого показано повний цикл станів інтерфейсу користувача. Результати тестування показали повну працездатність розробленого мобільного застосунку відповідно до сформульованих вимог.

Подальший розвиток проєкту дозволить реалізувати більше цікавих та корисних для користувачів функціональних можливостей та особливостей інтерфейсу, як-от:

– збереження історії чатів у локальній базі даних SQLite;
– можливість створення користувацьких профілів;
– автентифікація та авторизація різних користувачів;
– удосконалення інтерфейсу;
– робота застосунку у фоні;
– звуковий сигнал при отриманні повідомлення;
– публікація у маркеті Google Play.

За результатами проведеного у роботі аналізу, такий функціонал буде спроможним вивести застосунок на перше місце серед наявних аналогів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Android Platform Architecture. URL: https://developer.android.com/guide/platform/
2. Мова програмування Kotlin. URL: https://kotlinlang.org
3. Jetpack Compose UI App Development Toolkit. URL: https://developer.android.com/develop/ui/compose
4. Android Studio IDE. URL: https://developer.android.com/studio
5. Документація по роботі з модулем Bluetooth в Android. URL: https://developer.android.com/develop/connectivity/bluetooth/
6. Використання шаблону MVVM у розробці мобільних Android-застосунків з Jetpack Compose. URL: https://developer.android.com/topic/libraries/architecture/viewmodel
7. Dependency injection in Android. URL: https://developer.android.com/training/dependency-injection
8. Dependency injection with Hilt. URL: https://developer.android.com/training/dependency-injection/hilt-android
9. Бібліотека Hilt для ін’єкції залежностей у Android-застосунках. URL: https://dagger.dev/hilt/
10. Compose State vs StateFlow: State Management in Jetpack Compose. URL: https://medium.com/@husayn.fakher/compose-state-vs-stateflow-state-management-in-jetpack-compose-c99740732023

ДОДАТКИ

Додаток А. Програмний код застосунку

AndroidBluetoothController.kt

```
package com.vkr.bluetoothchat.data.chat

import android.Manifest
import android.annotation.SuppressLint
import android.bluetooth.BluetoothAdapter
import android.bluetooth.BluetoothDevice
import android.bluetooth.BluetoothManager
import android.bluetooth.BluetoothServerSocket
import android.bluetooth.BluetoothSocket
import android.content.Context
import android.content.IntentFilter
import android.content.pm.PackageManager
import com.vkr.bluetoothchat.domain.chat.BluetoothController
import com.vkr.bluetoothchat.domain.chat.BluetoothDeviceDomain
import com.vkr.bluetoothchat.domain.chat.BluetoothMessage
import com.vkr.bluetoothchat.domain.chat.ConnectionResult
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import java.io.IOException
import java.util.*

@SuppressLint("MissingPermission")
class AndroidBluetoothController(
    private val context: Context
): BluetoothController {
    private val bluetoothManager by lazy {
        context.getSystemService(BluetoothManager::class.java)
    }
    private val bluetoothAdapter by lazy {
        bluetoothManager?.adapter
    }
    private var dataTransferService: BluetoothDataTransferService? = null
    private val _isConnected = MutableStateFlow(false)
    override val isConnected: StateFlow<Boolean>
        get() = _isConnected.asStateFlow()
    private val _scannedDevices =
MutableStateFlow<List<BluetoothDeviceDomain>>(emptyList())
    override val scannedDevices: StateFlow<List<BluetoothDeviceDomain>>
        get() = _scannedDevices.asStateFlow()
    private val _pairedDevices =
MutableStateFlow<List<BluetoothDeviceDomain>>(emptyList())
    override val pairedDevices: StateFlow<List<BluetoothDeviceDomain>>
        get() = _pairedDevices.asStateFlow()
    private val _errors = MutableSharedFlow<String>()
    override val errors: SharedFlow<String>
        get() = _errors.asSharedFlow()
```

```
    private val foundDeviceReceiver = FoundDeviceReceiver { device ->
        _scannedDevices.update { devices ->
            val newDevice = device.toBluetoothDeviceDomain()
            if(newDevice in devices) devices else devices + newDevice
        }
    }
    private val bluetoothStateReceiver = BluetoothStateReceiver { isConnected,
bluetoothDevice ->
        if(bluetoothAdapter?.bondedDevices?.contains(bluetoothDevice) == true) {
            _isConnected.update { isConnected }
        } else {
            CoroutineScope(Dispatchers.IO).launch {
                _errors.emit("Can't connect to a non-paired device.")
            }
        }
    }
    private var currentServerSocket: BluetoothServerSocket? = null
    private var currentClientSocket: BluetoothSocket? = null
    init {
        updatePairedDevices()
        context.registerReceiver(
            bluetoothStateReceiver,
            IntentFilter().apply {
                addAction(BluetoothAdapter.ACTION_CONNECTION_STATE_CHANGED)
                addAction(BluetoothDevice.ACTION_ACL_CONNECTED)
                addAction(BluetoothDevice.ACTION_ACL_DISCONNECTED)
            }
        )
    }
    override fun startDiscovery() {
        if(!hasPermission(Manifest.permission.BLUETOOTH_SCAN)) {
            return
        }
        context.registerReceiver(
            foundDeviceReceiver,
            IntentFilter(BluetoothDevice.ACTION_FOUND)
        )
        updatePairedDevices()
        bluetoothAdapter?.startDiscovery()
    }
    override fun stopDiscovery() {
        if(!hasPermission(Manifest.permission.BLUETOOTH_SCAN)) {
            return
        }
        bluetoothAdapter?.cancelDiscovery()
    }
    override fun startBluetoothServer(): Flow<ConnectionResult> {
        return flow {
            if(!hasPermission(Manifest.permission.BLUETOOTH_CONNECT)) {
                throw SecurityException("No BLUETOOTH_CONNECT permission")
            }
            currentServerSocket =
bluetoothAdapter?.listenUsingRfcommWithServiceRecord(
                "chat_service",
                UUID.fromString(SERVICE_UUID)
            )
            var shouldLoop = true
            while(shouldLoop) {
```

```
                currentClientSocket = try {
                    currentServerSocket?.accept()
                } catch(e: IOException) {
                    shouldLoop = false
                    null
                }
                emit(ConnectionResult.ConnectionEstablished)
                currentClientSocket?.let {
                    currentServerSocket?.close()
                    val service = BluetoothDataTransferService(it)
                    dataTransferService = service
                    emitAll(
                        service
                            .listenForIncomingMessages()
                            .map {
                                ConnectionResult.TransferSucceeded(it)
                            }
                    )
                }
            }
        }.onCompletion {
            closeConnection()
        }.flowOn(Dispatchers.IO)
    }
    override fun connectToDevice(device: BluetoothDeviceDomain):
Flow<ConnectionResult> {
        return flow {
            if(!hasPermission(Manifest.permission.BLUETOOTH_CONNECT)) {
                throw SecurityException("No BLUETOOTH_CONNECT permission")
            }
            currentClientSocket = bluetoothAdapter
                ?.getRemoteDevice(device.address)
                ?.createRfcommSocketToServiceRecord(
                    UUID.fromString(SERVICE_UUID)
                )
            stopDiscovery()
            currentClientSocket?.let { socket ->
                try {
                    socket.connect()
                    emit(ConnectionResult.ConnectionEstablished)
                    BluetoothDataTransferService(socket).also {
                        dataTransferService = it
                        emitAll(
                            it.listenForIncomingMessages()
                                .map { ConnectionResult.TransferSucceeded(it) }
                        )
                    }
                } catch(e: IOException) {
                    socket.close()
                    currentClientSocket = null
                    emit(ConnectionResult.Error("Connection was interrupted"))
                }
            }
        }.onCompletion {
            closeConnection()
        }.flowOn(Dispatchers.IO)
    }
    override suspend fun trySendMessage(message: String): BluetoothMessage? {
```

```
        if(!hasPermission(Manifest.permission.BLUETOOTH_CONNECT)) {
            return null
        }
        if(dataTransferService == null) {
            return null
        }
        val bluetoothMessage = BluetoothMessage(
            message = message,
            senderName = bluetoothAdapter?.name ?: "Unknown name",
            isFromLocalUser = true
        )
        dataTransferService?.sendMessage(bluetoothMessage.toByteArray())
        return bluetoothMessage
    }
    override fun closeConnection() {
        currentClientSocket?.close()
        currentServerSocket?.close()
        currentClientSocket = null
        currentServerSocket = null
    }
    override fun release() {
        context.unregisterReceiver(foundDeviceReceiver)
        context.unregisterReceiver(bluetoothStateReceiver)
        closeConnection()
    }

    private fun updatePairedDevices() {
        if(!hasPermission(Manifest.permission.BLUETOOTH_CONNECT)) {
            return
        }
        bluetoothAdapter
            ?.bondedDevices
            ?.map { it.toBluetoothDeviceDomain() }
            ?.also { devices ->
                _pairedDevices.update { devices }
            }
    }
    private fun hasPermission(permission: String): Boolean {
        return context.checkSelfPermission(permission) ==
PackageManager.PERMISSION_GRANTED
    }
    companion object {
        const val SERVICE_UUID = "27b7d1da-08c7-4505-a6d1-2459987e5e2d"
    }
}
```

BluetoothDataTransferService.kt

```
package com.vkr.bluetoothchat.data.chat

import android.bluetooth.BluetoothSocket
import com.vkr.bluetoothchat.domain.chat.BluetoothMessage
import com.vkr.bluetoothchat.domain.chat.TransferFailedException
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.flow
import kotlinx.coroutines.flow.flowOn
import kotlinx.coroutines.withContext
```

```
import java.io.IOException

class BluetoothDataTransferService(
    private val socket: BluetoothSocket
) {
    fun listenForIncomingMessages(): Flow<BluetoothMessage> {
        return flow {
            if(!socket.isConnected) {
                return@flow
            }
            val buffer = ByteArray(1024)
            while(true) {
                val byteCount = try {
                    socket.inputStream.read(buffer)
                } catch(e: IOException) {
                    throw TransferFailedException()
                }
                emit(
                    buffer.decodeToString(
                        endIndex = byteCount
                    ).toBluetoothMessage(
                        isFromLocalUser = false
                    )
                )
            }
        }.flowOn(Dispatchers.IO)
    }
    suspend fun sendMessage(bytes: ByteArray): Boolean {
        return withContext(Dispatchers.IO) {
            try {
                socket.outputStream.write(bytes)
            } catch(e: IOException) {
                e.printStackTrace()
                return@withContext false
            }
            true
        }
    }
}
```

BluetoothDeviceMapper.kt

```
package com.vkr.bluetoothchat.data.chat

import android.annotation.SuppressLint
import android.bluetooth.BluetoothDevice
import com.vkr.bluetoothchat.domain.chat.BluetoothDeviceDomain

@SuppressLint("MissingPermission")
fun BluetoothDevice.toBluetoothDeviceDomain(): BluetoothDeviceDomain {
    return BluetoothDeviceDomain(
        name = name,
        address = address
    )
}
```

BluetoothMessageMapper.kt

```
package com.vkr.bluetoothchat.data.chat

import com.vkr.bluetoothchat.domain.chat.BluetoothMessage

fun String.toBluetoothMessage(isFromLocalUser: Boolean): BluetoothMessage {
    val name = substringBeforeLast("#")
    val message = substringAfter("#")
    return BluetoothMessage(
        message = message,
        senderName = name,
        isFromLocalUser = isFromLocalUser
    )
}

fun BluetoothMessage.toByteArray(): ByteArray {
    return "$senderName#$message".encodeToByteArray()
}
```

BluetoothStateReceiver.kt

```
package com.vkr.bluetoothchat.data.chat

import android.bluetooth.BluetoothDevice
import android.content.BroadcastReceiver
import android.content.Context
import android.content.Intent
import android.os.Build

class BluetoothStateReceiver(
    private val onStateChanged: (isConnected: Boolean, BluetoothDevice) -> Unit
): BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        val device = if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU) {
            intent?.getParcelableExtra(
                BluetoothDevice.EXTRA_DEVICE,
                BluetoothDevice::class.java
            )
        } else {
            intent?.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
        }
        when(intent?.action) {
            BluetoothDevice.ACTION_ACL_CONNECTED -> {
                onStateChanged(true, device ?: return)
            }
            BluetoothDevice.ACTION_ACL_DISCONNECTED -> {
                onStateChanged(false, device ?: return)
            }
        }
    }
}
```

FoundDeviceReceiver.kt

```
package com.vkr.bluetoothchat.data.chat
import android.bluetooth.BluetoothDevice
import android.content.BroadcastReceiver
```

```
import android.content.Context
import android.content.Intent
import android.os.Build
class FoundDeviceReceiver(
    private val onDeviceFound: (BluetoothDevice) -> Unit
): BroadcastReceiver() {

    override fun onReceive(context: Context?, intent: Intent?) {
        when(intent?.action) {
            BluetoothDevice.ACTION_FOUND -> {
                val device = if (Build.VERSION.SDK_INT >=
Build.VERSION_CODES.TIRAMISU) {
                    intent.getParcelableExtra(
                        BluetoothDevice.EXTRA_DEVICE,
                        BluetoothDevice::class.java
                    )
                } else {
                    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
                }
                device?.let(onDeviceFound)
            }
        }
    }
}
```

AppModule.kt

```
package com.vkr.bluetoothchat.di
import android.content.Context
import com.vkr.bluetoothchat.data.chat.AndroidBluetoothController
import com.vkr.bluetoothchat.domain.chat.BluetoothController
import dagger.Module
import dagger.Provides
import dagger.hilt.InstallIn
import dagger.hilt.android.qualifiers.ApplicationContext
import dagger.hilt.components.SingletonComponent
import javax.inject.Singleton

@Module
@InstallIn(SingletonComponent::class)
object AppModule {
    @Provides
    @Singleton
    fun provideBluetoothController(@ApplicationContext context: Context):
BluetoothController {
        return AndroidBluetoothController(context)
    }
}
```

BluetoothController.kt

```
package com.vkr.bluetoothchat.domain.chat
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.SharedFlow
import kotlinx.coroutines.flow.StateFlow
interface BluetoothController {
    val isConnected: StateFlow<Boolean>
    val scannedDevices: StateFlow<List<BluetoothDevice>>
```

```
    val pairedDevices: StateFlow<List<BluetoothDevice>>
    val errors: SharedFlow<String>
    fun startDiscovery()
    fun stopDiscovery()
    fun startBluetoothServer(): Flow<ConnectionResult>
    fun connectToDevice(device: BluetoothDevice): Flow<ConnectionResult>
    suspend fun trySendMessage(message: String): BluetoothMessage?
    fun closeConnection()
    fun release()
}
```

BluetoothDevice.kt

```
package com.vkr.bluetoothchat.domain.chat

typealias BluetoothDeviceDomain = BluetoothDevice

data class BluetoothDevice(
    val name: String?,
    val address: String
)
```

BluetoothMessage.kt

```
package com.vkr.bluetoothchat.domain.chat

data class BluetoothMessage(
    val message: String,
    val senderName: String,
    val isFromLocalUser: Boolean
)
```

ConnectionResult.kt

```
package com.vkr.bluetoothchat.domain.chat

sealed interface ConnectionResult {
    object ConnectionEstablished: ConnectionResult
    data class TransferSucceeded(val message: BluetoothMessage): ConnectionResult
    data class Error(val message: String): ConnectionResult
}
```

TransferFailedException.kt

```
package com.vkr.bluetoothchat.domain.chat
import java.io.IOException
class TransferFailedException: IOException("Reading incoming data failed")
```

ChatMessage.kt

```
package com.vkr.bluetoothchat.presentation.components
import androidx.compose.foundation.background
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.widthIn
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
```

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.draw.clip
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.tooling.preview.Preview
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.vkr.bluetoothchat.domain.chat.BluetoothMessage
import com.vkr.bluetoothchat.ui.theme.BluetoothChatTheme
import com.vkr.bluetoothchat.ui.theme.OldRose
import com.vkr.bluetoothchat.ui.theme.Vanilla

@Composable
fun ChatMessage(
    message: BluetoothMessage,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .clip(
                RoundedCornerShape(
                    topStart = if (message.isFromLocalUser) 15.dp else 0.dp,
                    topEnd = 15.dp,
                    bottomStart = 15.dp,
                    bottomEnd = if (message.isFromLocalUser) 0.dp else 15.dp
                )
            )
            .background(
                if (message.isFromLocalUser) OldRose else Vanilla
            )
            .padding(16.dp)
    ) {
        Text(
            text = message.senderName,
            fontSize = 10.sp,
            color = Color.Black
        )
        Text(
            text = message.message,
            color = Color.Black,
            modifier = Modifier.widthIn(max = 250.dp)
        )
    }
}

@Preview
@Composable
fun ChatMessagePreview() {
    BluetoothChatTheme {
        ChatMessage(
            message = BluetoothMessage(
                message = "Hello World!",
                senderName = "Pixel 6",
                isFromLocalUser = false
            )
        )
    }
}
```

ChatScreen.kt

```
@file:OptIn(ExperimentalComposeUiApi::class)

package com.vkr.bluetoothchat.presentation.components

import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.Icon
import androidx.compose.material.IconButton
import androidx.compose.material.Text
import androidx.compose.material.TextField
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Close
import androidx.compose.material.icons.filled.Send
import androidx.compose.runtime.Composable
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.saveable.rememberSaveable
import androidx.compose.ui.Alignment
import androidx.compose.ui.ExperimentalComposeUiApi
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalSoftwareKeyboardController
import androidx.compose.ui.unit.dp
import com.vkr.bluetoothchat.presentation.BluetoothUiState

@Composable
fun ChatScreen(
    state: BluetoothUiState,
    onDisconnect: () -> Unit,
    onSendMessage: (String) -> Unit
) {
    val message = rememberSaveable {
        mutableStateOf("")
    }
    val keyboardController = LocalSoftwareKeyboardController.current
    Column(
        modifier = Modifier
            .fillMaxSize()
    ) {
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            Text(
                text = "Messages",
                modifier = Modifier.weight(1f)
            )
            IconButton(onClick = onDisconnect) {
                Icon(
                    imageVector = Icons.Default.Close,
                    contentDescription = "Disconnect"
                )
            }
        }
        LazyColumn(
```

```
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f),
            contentPadding = PaddingValues(16.dp),
            verticalArrangement = Arrangement.spacedBy(16.dp)
        ) {
            items(state.messages) { message ->
                Column(
                    modifier = Modifier.fillMaxWidth()
                ) {
                    ChatMessage(
                        message = message,
                        modifier = Modifier
                         .align(
                           if(message.isFromLocalUser) Alignment.End else
Alignment.Start
                            )
                    )
                }
            }
        }
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(16.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            TextField(
                value = message.value,
                onValueChange = { message.value = it },
                modifier = Modifier.weight(1f),
                placeholder = {
                    Text(text = "Message")
                }
            )
            IconButton(onClick = {
                onSendMessage(message.value)
                message.value = ""
                keyboardController?.hide()
            }) {
                Icon(
                    imageVector = Icons.Default.Send,
                    contentDescription = "Send message"
                )
            }
        }
    }
}
```

DeviceScreen.kt

```
package com.vkr.bluetoothchat.presentation.components

import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.Button
```

```
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import com.vkr.bluetoothchat.domain.chat.BluetoothDevice
import com.vkr.bluetoothchat.presentation.BluetoothUiState

@Composable
fun DeviceScreen(
    state: BluetoothUiState,
    onStartScan: () -> Unit,
    onStopScan: () -> Unit,
    onStartServer: () -> Unit,
    onDeviceClick: (BluetoothDevice) -> Unit
) {
    Column(
        modifier = Modifier
            .fillMaxSize()
    ) {
        BluetoothDeviceList(
            pairedDevices = state.pairedDevices,
            scannedDevices = state.scannedDevices,
            onClick = onDeviceClick,
            modifier = Modifier
                .fillMaxWidth()
                .weight(1f)
        )
        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceAround
        ) {
            Button(onClick = onStartScan) {
                Text(text = "Start scan")
            }
            Button(onClick = onStopScan) {
                Text(text = "Stop scan")
            }
            Button(onClick = onStartServer) {
                Text(text = "Start server")
            }
        }
    }
}

@Composable
fun BluetoothDeviceList(
    pairedDevices: List<BluetoothDevice>,
    scannedDevices: List<BluetoothDevice>,
    onClick: (BluetoothDevice) -> Unit,
    modifier: Modifier = Modifier
) {
    LazyColumn(
        modifier = modifier
    ) {
        item {
            Text(
```

```
                text = "Paired Devices",
                fontWeight = FontWeight.Bold,
                fontSize = 24.sp,
                modifier = Modifier.padding(16.dp)
            )
        }
        items(pairedDevices) { device ->
            Text(
                text = device.name ?: "(No name)",
                modifier = Modifier
                    .fillMaxWidth()
                    .clickable { onClick(device) }
                    .padding(16.dp)
            )
        }
        item {
            Text(
                text = "Scanned Devices",
                fontWeight = FontWeight.Bold,
                fontSize = 24.sp,
                modifier = Modifier.padding(16.dp)
            )
        }
        items(scannedDevices) { device ->
            Text(
                text = device.name ?: "(No name)",
                modifier = Modifier
                    .fillMaxWidth()
                    .clickable { onClick(device) }
                    .padding(16.dp)
            )
        }
    }
}
```

BluetoothUiState.kt

```
package com.vkr.bluetoothchat.presentation

import com.vkr.bluetoothchat.domain.chat.BluetoothDevice
import com.vkr.bluetoothchat.domain.chat.BluetoothMessage

data class BluetoothUiState(
    val scannedDevices: List<BluetoothDevice> = emptyList(),
    val pairedDevices: List<BluetoothDevice> = emptyList(),
    val isConnected: Boolean = false,
    val isConnecting: Boolean = false,
    val errorMessage: String? = null,
    val messages: List<BluetoothMessage> = emptyList()
)
```

BluetoothViewModel.kt

```
package com.vkr.bluetoothchat.presentation

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.vkr.bluetoothchat.domain.chat.BluetoothController
```

```
import com.vkr.bluetoothchat.domain.chat.BluetoothDeviceDomain
import com.vkr.bluetoothchat.domain.chat.ConnectionResult
import dagger.hilt.android.lifecycle.HiltViewModel
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import javax.inject.Inject

@HiltViewModel
class BluetoothViewModel @Inject constructor(
    private val bluetoothController: BluetoothController
): ViewModel() {

    private val _state = MutableStateFlow(BluetoothUiState())
    val state = combine(
        bluetoothController.scannedDevices,
        bluetoothController.pairedDevices,
        _state
    ) { scannedDevices, pairedDevices, state ->
        state.copy(
            scannedDevices = scannedDevices,
            pairedDevices = pairedDevices,
            messages = if(state.isConnected) state.messages else emptyList()
        )
    }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), _state.value)
    private var deviceConnectionJob: Job? = null
    init {
        bluetoothController.isConnected.onEach { isConnected ->
            _state.update { it.copy(isConnected = isConnected) }
        }.launchIn(viewModelScope)
        bluetoothController.errors.onEach { error ->
            _state.update { it.copy(
                errorMessage = error
            ) }
        }.launchIn(viewModelScope)
    }
    fun connectToDevice(device: BluetoothDeviceDomain) {
        _state.update { it.copy(isConnecting = true) }
        deviceConnectionJob = bluetoothController
            .connectToDevice(device)
            .listen()
    }
    fun disconnectFromDevice() {
        deviceConnectionJob?.cancel()
        bluetoothController.closeConnection()
        _state.update { it.copy(
            isConnecting = false,
            isConnected = false
        ) }
    }
    fun waitForIncomingConnections() {
        _state.update { it.copy(isConnecting = true) }
        deviceConnectionJob = bluetoothController
            .startBluetoothServer()
            .listen()
    }
    fun sendMessage(message: String) {
        viewModelScope.launch {
```

```
            val bluetoothMessage = bluetoothController.trySendMessage(message)
            if(bluetoothMessage != null) {
                _state.update { it.copy(
                    messages = it.messages + bluetoothMessage
                ) }
            }
        }
    }
    fun startScan() {
        bluetoothController.startDiscovery()
    }
    fun stopScan() {
        bluetoothController.stopDiscovery()
    }
    private fun Flow<ConnectionResult>.listen(): Job {
        return onEach { result ->
            when(result) {
                ConnectionResult.ConnectionEstablished -> {
                    _state.update { it.copy(
                        isConnected = true,
                        isConnecting = false,
                        errorMessage = null
                    ) }
                }
                is ConnectionResult.TransferSucceeded -> {
                    _state.update { it.copy(
                        messages = it.messages + result.message
                    ) }
                }
                is ConnectionResult.Error -> {
                    _state.update { it.copy(
                        isConnected = false,
                        isConnecting = false,
                        errorMessage = result.message
                    ) }
                }
            }
        }
            .catch { throwable ->
                bluetoothController.closeConnection()
                _state.update { it.copy(
                    isConnected = false,
                    isConnecting = false,
                ) }
            }
            .launchIn(viewModelScope)
    }
    override fun onCleared() {
        super.onCleared()
        bluetoothController.release()
    }
}
```

MainActivity.kt

```
package com.vkr.bluetoothchat.presentation
import android.Manifest
import android.bluetooth.BluetoothAdapter
```

```
import android.bluetooth.BluetoothManager
import android.content.Intent
import android.os.Build
import android.os.Bundle
import android.widget.Toast
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.result.contract.ActivityResultContracts
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.material.CircularProgressIndicator
import androidx.compose.material.MaterialTheme
import androidx.compose.material.Surface
import androidx.compose.material.Text
import androidx.compose.runtime.LaunchedEffect
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.hilt.navigation.compose.hiltViewModel
import com.vkr.bluetoothchat.presentation.components.ChatScreen
import com.vkr.bluetoothchat.presentation.components.DeviceScreen
import com.vkr.bluetoothchat.ui.theme.BluetoothChatTheme
import dagger.hilt.android.AndroidEntryPoint
@AndroidEntryPoint
class MainActivity : ComponentActivity() {
    private val bluetoothManager by lazy {
        applicationContext.getSystemService(BluetoothManager::class.java)
    }
    private val bluetoothAdapter by lazy {
        bluetoothManager?.adapter
    }
    private val isBluetoothEnabled: Boolean
        get() = bluetoothAdapter?.isEnabled == true
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val enableBluetoothLauncher = registerForActivityResult(
            ActivityResultContracts.StartActivityForResult()
        ) { /* Not needed */ }
        val permissionLauncher = registerForActivityResult(
            ActivityResultContracts.RequestMultiplePermissions()
        ) { perms ->
            val canEnableBluetooth = if(Build.VERSION.SDK_INT >=
Build.VERSION_CODES.S) {
                perms[Manifest.permission.BLUETOOTH_CONNECT] == true
            } else true
            if(canEnableBluetooth && !isBluetoothEnabled) {
                enableBluetoothLauncher.launch(
                    Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
                )
            }
        }
        if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.S) {
            permissionLauncher.launch(
                arrayOf(
                    Manifest.permission.BLUETOOTH_SCAN,
                    Manifest.permission.BLUETOOTH_CONNECT,
```

```
                )
            )
        }
        setContent {
            BluetoothChatTheme {
                val viewModel = hiltViewModel<BluetoothViewModel>()
                val state by viewModel.state.collectAsState()
                LaunchedEffect(key1 = state.errorMessage) {
                    state.errorMessage?.let { message ->
                        Toast.makeText(
                            applicationContext,
                            message,
                            Toast.LENGTH_LONG
                        ).show()
                    }
                }
                LaunchedEffect(key1 = state.isConnected) {
                    if(state.isConnected) {
                        Toast.makeText(
                            applicationContext,
                            "You're connected!",
                            Toast.LENGTH_LONG
                        ).show()
                    }
                }
                Surface(
                    color = MaterialTheme.colors.background
                ) {
                    when {
                        state.isConnecting -> {
                            Column(
                                modifier = Modifier.fillMaxSize(),
                                horizontalAlignment =
Alignment.CenterHorizontally,
                                verticalArrangement = Arrangement.Center
                            ) {
                                CircularProgressIndicator()
                                Text(text = "Connecting...")
                            }
                        }
                        state.isConnected -> {
                            ChatScreen(
                                state = state,
                                onDisconnect = viewModel::disconnectFromDevice,
                                onSendMessage = viewModel::sendMessage
                            )
                        }
                        else -> {
                            DeviceScreen(
                                state = state,
                                onStartScan = viewModel::startScan,
                                onStopScan = viewModel::stopScan,
                                onDeviceClick = viewModel::connectToDevice,
                                onStartServer =
viewModel::waitForIncomingConnections
                            )
                        }
                    }
```

```
                }
            }
        }
    }
}
```

BluetoothApp.kt

```
package com.vkr.bluetoothchat

import android.app.Application
import dagger.hilt.android.HiltAndroidApp
@HiltAndroidApp
class BluetoothApp: Application()
```

strings.xml

```
<resources>
    <string name="app_name">BluetoothChat</string>
</resources>
```

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    <uses-permission android:name="android.permission.BLUETOOTH"
android:maxSdkVersion="30" />
    <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
    <uses-permission android:name="android.permission.BLUETOOTH_SCAN"
        android:usesPermissionFlags="neverForLocation" />
    <uses-feature android:name="android.hardware.bluetooth" />
    <application
        android:name=".BluetoothApp"
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/Theme.BluetoothChat"
        tools:targetApi="31">
        <activity
            android:name=".presentation.MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.BluetoothChat">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```