Харківський національний університет радіоелектроніки
Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти - другий (магістерський)

Дослідження можливостей застосування односпрямованого підходу до проєктування мобільних архітектур на прикладі Redux

Виконав: студент 2 курсу, групи ІПЗм-20-3
Губар С.О.
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-наукової програми
(тип програми)

Інженерія програмного забезпечення
(повна назва освітньої програми)

Керівник доцент каф. ПІ, Каук В.І.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2022р.

Харківський національний університет радіоелектроніки

Факультет_____Комп'ютерних наук_____

Кафедра_____Програмної інженерії_____

Рівень вищої освіти - _____другий (магістерський)_____

Спеціальність _____121-Інженерія програмного забезпечення_____
(код і повна назва)

Тип програми _____освітньо-наукова програма_____

Освітня програма_Інженерія програмного забезпечення_____

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 20 ___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові _____Губар Сергію Олександровичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи _Дослідження можливостей застосування односпрямованого підходу до проєктування мобільних архітектур на прикладі Redux
затверджена наказом університету від "__" ___ р № _____
заповнюється вручну після отримання наказу

2. Термін подання студентом роботи до екзаменаційної комісії
15 травня 2022 р.

3. Вихідні дані до роботи *основні вимоги до створюваного модуля, загальні ресурси за обраною темою*_____

4. Перелік питань, що потрібно опрацювати в роботі *аналіз предметної області, постановка задачі, формування вимог до архітектурного модуля, формування вимог до програмної системи, UML-моделювання програмної системи, програмна реалізація системи, проектування експерименту, його проведення та оцінювання результатів*

5 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доцент каф. ПІ, Каук В.І.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	25.12.21 – 30.12.21	Виконано
2	Аналіз аналогів	05.01.22 – 15.01.22	Виконано
3	Постановка задачі	15.01.22 – 20.01.22	Виконано
4	Формування вимог до розроблюваного модуля	20.01.22 – 25.01.22	Виконано
5	Планування експерименту	25.01.22 – 29.01.22	Виконано
6	Проектування програмної системи	29.01.22 – 07.02.21	Виконано
7	UML моделювання	07.02.22 – 15.02.22	Виконано
8	Написання тез доповіді на міжнародну конференцію	15.02.22 – 19.02.22	Виконано
9	Програмна реалізація	19.02.22 – 09.03.22	Виконано
10	Тестування	09.03.22 – 18.03.22	Виконано
11	Отримання результатів експерименту	18.03.22 – 24.03.22	Виконано
12	Аналіз результатів експерименту	24.03.22 – 04.04.22	Виконано
13	Створення висновків та рекомендацій на основі результатів експерименту	04.04.22 – 07.04.22	Виконано
14	Написання пояснювальної записки	08.04.22 – 25.04.22	Виконано

15	Підготовка презентації	25.04.22 – 30.04.22	Виконано
16	Нормоконтроль	05.05.22 – 17.05.22	
17	Рецензування		
18	Занесення диплома в електронний архів		
19	Попередній захист		
20	Допуск до захисту роботи		

Дата видачі завдання __25_____січня_____ 2021р.

Студент _____ Губар С.О.
(підпис)

Керівник роботи _____ доц. Каук В.І.
(підпис)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 81 сторінку, 13 рисунків, 6 таблиць, 12 джерел.

МОБІЛЬНА АРХІТЕКТУРА, ПРОГРАМНА СИСТЕМА, KOTLIN, REDUX, MVP, MVVM.

Метою роботи є дослідження архітектурного модуля, який буде використовувати принципи односпрямованих підходів до проектування архітектури для вирішення недоліків існуючих рішень.

Методи розробки базуються на Kotlin, Android Framework з використанням Android Studio.

Результатом виконання роботи є спроектований архітектурний модуль а також його прототип, який можна використовувати у мобільних застосунках на базі ОС Android.

Explanatory note contains: 81 p., 13 fig., 6 tables, 12 sources.

MOBILE ACRHITECTURE, SOFTWARE SYSTEM, KOTLIN, REDUX, MVP, MVVM.

The purpose of this work is to design an architectural module, based on unidirectional architectural approaches to solve problems of existing approaches.

Development methods are based on Kotlin, Android Framework using Android Studio.

The result of this work is an architectural design of the module and prototype implementation which can be used in any Android application.

Я, Губар Сергій Олександрович, студент групи ІПЗм-20-3 здобувач вищої освіти на другому (магістерському) рівні кафедра програмної інженерії заявляю: моя кваліфікаційна робота на тему «Дослідження можливостей застосування односпрямованого підходу до проєктування мобільних архітектур на прикладі Redux» що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ПЗ – програмне забезпечення;

JWT – Json Web Token;

API – Application Programming Interface;

TTM – Time To Market;

ELK – Elasticsearch, Logstash and Kibana;

DSL – Domain Specific Language.

ВСТУП

У останні роки мобільні застосунки отримують все більшу популярність – багато популярних платформ є в першу чергу орієнтованими на мобільні пристрої, що робить розробку таких застосунків актуальною та необхідною задачею. Так як розробка застосунків для мобільних пристроїв є відносно новою галуззю, не існує єдиного підходу до проектування таких застосунків, що призводить до великої кількості поганих архітектурних рішень, які ускладнюють розробку та супровід програмного забезпечення.

Предметом дослідження є односпрямовані підходи до проектування програмного забезпечення.

Об’єктом дослідження є сучасні архітектурні підходи, які можуть бути використані для розробки мобільних застосунків.

Головною ідеєю роботи є аналіз існуючих рішень до проектування архітектур мобільних застосунків, виділення існуючих недоліків, та їх вирішення за допомогою проектування та реалізації архітектурного програмного модуля, який можна використовувати у будь-якому застосунку.

У роботі досліджується різниця двох основних підходів до проєктування архітектури – з односпрямованим потоком даних, та двоспрямованим. Наводяться приклади конкретних архітектур, які реалізують ці підходи – MVC (Model-View-Controller), MVP (Model-View-Presenter), MVVM (Model-View-ViewModel), MVI (Model-View-Intent), Redux. Кожна з цих реалізацій має як свої сильні, так і слабкі сторони у різних сценаріях використання – деякі архітектури краще підходять для розробки невеликих застосунків, де ключову роль грає швидкість та простота розробки, в той час як інші більше підходять для розробки великомасштабних проектів, над якими водночас працює велика кількість розробників, що робить їх підтримку та супровід набагато більш важливою та складною задачею.

Таким чином, метою роботи є виявлення недоліків існуючих підходів до проектування архітектури, аналіз можливих рішень цих недоліків, та реалізація

модуля, що дозволить розробляти мобільні застосунки стандартизовано, мінімізуючи витрати на супровід та розширення програмного коду.

В ході виконання роботи реалізовуються наступні програмні компоненти:

- архітектурний модуль Redux;
- клієнтський застосунок;
- серверний застосунок.

Для реалізації архітектурного модуля Redux було обрано мову програмування Kotlin [1] та залежності Android для інтеграції у клієнтські застосунки.

Клієнтський застосунок використовує ту ж мову програмування з використанням Android SDK. Слід відмітити, що цей застосунок реалізується двома різними способами – з використанням Redux [2], та з використанням MVVM – для подальшого порівняння.

Серверний застосунок побудовано з використанням Typescript та фреймворку Nest.js. Розгортання застосунку використовується за допомогою платформу AWS та сервісу Elastic Computing (EC2). В якості бази даних використовується NoSQL база даних MongoDB, яку розгорнуто у хмарному кластері MongoDB Atlas.

Розроблений програмний модуль порівнюється за визначеними метриками з підходом MVVM для визначення актуальності використання такого підходу у мобільний розробці. Також за результатами роботи було опубліковано тези доповіді у науково-технічній конференції «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління».

Актуальність роботи визначається популярністю мобільної розробки та відсутністю єдиного підходу до проектування архітектури таких застосунків.

1 АНАЛІТИЧНИЙ ОБЗОР

1.1 Аналіз предметної області

На сьогоднішній день, мобільна розробка є досить популярним напрямом, який сформувався нещодавно, і у якому немає єдиного підходу до розробку застосунків такого типу. Проблемою, яку вирішує дослідження, є недоліки конкретних існуючих архітектурних шаблонів (MVP, MVVM) а також зміна підходу до проектування архітектури - перехід до односпрямованого потоку даних у системі.

Існує декілька підходів до проектування архітектури, які є рекомендованими компанією Google – власником Android, одним з яких є підхід MVVM (Model-View-ViewModel). Однією з цілей дослідження є виявлення недоліків цього підходу, та спроба їх вирішити за допомогою використання іншої концепції роботи з даними та їх напрямом у системі. Також існує багато інших підходів – MVC, MVP, MVI, RIBs, VIPER, але жоден з них не є настільки популярним та поширеним як підхід MVVM.

Серед особливостей дослідження слід виділити непопулярність односпрямованих підходів саме для розробки застосунків під ОС Android, що робить реалізацію цього підходу досить складною, а результати дослідження складними для передбачення. Також слід відмітити, що Android SDK на даний момент не передбачує такого сценарію використання, що додає складності інтеграції з цією платформою.

Головне питання, на яке необхідно отримати відповідь у результаті дослідження та після проведення експерименту - “Чи доцільно використовувати односпрямований підхід у проектуванні архітектури мобільних застосунків?”, а також сформувати рекомендації щодо типів проектів, де використання такої архітектури має сенс.

1.2 Актуальність та мета дослідження

Метою роботи є дослідження односпрямованого підходу до проектування архітектури програмного забезпечення з ціллю покращення якості мобільних застосунків.

Актуальність дослідження зумовлено популярністю розробки мобільних застосунків під ОС Android, а також відсутністю чітко сформованого підходу до проектування архітектури таких застосунків.

Ще одним фактором, який обумовлює актуальність дослідження – поступовий перехід до декларативної парадигми проектування користувацького інтерфейсу, що відбувається у Android SDK останнім часом, який є можливим через наявність нового фреймворку – Jetpack Compose [3]. Саме декларативний стиль написання користувацького інтерфейсу дуже добре поєднується з односпрямованим підходом до проектування архітектури, що робить дослідження цього підходу корисним для покращення якості коду та можливості його розширення.

2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА АЛГОРИТМІВ

2.1 Опис існуючих підходів до проектування архітектур

Підходи до проектування архітектури мобільних застосунків за напрямом даних можна поділити на два типи – двоспрямовані (bidirectional) та односпрямовані (unidirectional [4]). Слід відзначити, що ці підходи є незалежними від конкретної платформи – це лише ідея, реалізація якої є базою деякої архітектури, як, наприклад, Redux є однією (але не єдиною) реалізацією односпрямованого підходу.

Прикладом двоспрямованої архітектури є рекомендована Google архітектура MVVM з використанням DataBinding. Суть двоспрямованого потоку даних полягає у наступному – існує два компоненти (View та ViewModel у MVVM), зміни в кожному з яких викликають зміни у іншому. Таким чином, зміна користувачем певного стану View викликає зміни у ViewModel, і навпаки (рис. 1).

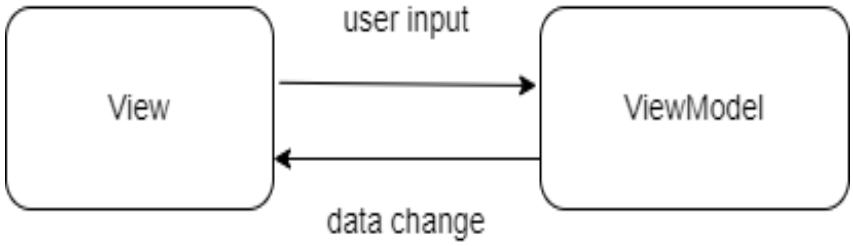

Рисунок 2.1 – Потік даних у двоспрямованій архітектури

Існує також односпрямований підхід до проектування, який відрізняється тим, що потік даних у системі іде у одному напрямку – два компоненти не можуть взаємодіяти один з одним напряму, як це трапляється, наприклад, у MVVM. Якщо одному компоненту потрібно взаємодіяти з іншим – він не робить це напряму, а викликає інший компонент, який за ланцюжком викликає інші компоненти, що призводить до повідомлення необхідного компоненту про зміну даних.

Прикладами односпрямованого підходу є архітектури Flux та Redux. Принцип роботи такого підходу зображено на рис. 2.2

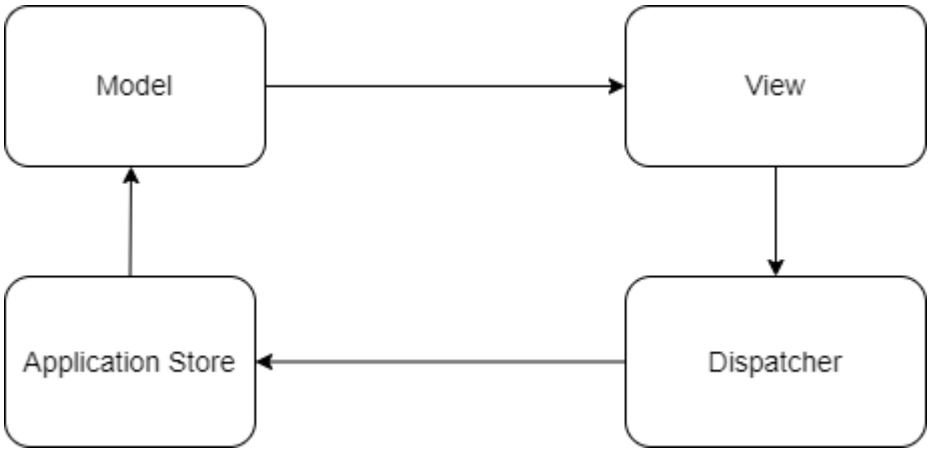

Рисунок 2.2 – Потік даних у односпрямованій архітектурі

Як можна бачити, компонент користувацького відображення даних (View) не може змінити напряму компонент з даними (Model). Для цього йому потрібно викликати певні методи компонента Dispatcher, який в свою чергу викликає зміну у Application Store, що призводить до зміни даних (Model).

На перший погляд, така складність може здатися непотрібною, але якщо застосунок має складну бізнес-логіку або велику кількість компонентів – такий підхід полегшує розширення системи, реалізацію складного функціоналу та додавання нових компонентів.

2.2 Опис існуючих архітектур

Найбільш популярними у розробці мобільних застосунків є наступні архітектурні підходи:

- MVC (Model – View – Controller);
- MVP (Model – View – Presenter);
- MVVM (Model – View – ViewModel);

– MVI (Model – View – Intent).

Спільною ідеєю усіх цих архітектурних підходів (окрім MVC) є одна особливість - виділення окремого презентаційного шару (presentation layer) та шару відображення (view layer), що робить модель незалежною від конкретного відображення. Таке розподілення відповідає принципу Single Responsibility, а також підвищує можливість тестування коду, а також зменшує кількість помилок, які можна допустити у великих проектах. Так у MVP за презентаційний шар відповідає сутність Presenter, а у MVVM – ViewModel.

Першою історично та найбільш поширеною у часи формування стандартів програмування застосунків під ОС Android була архітектура MVC, принцип роботи якої зображено на рис. 2.3.

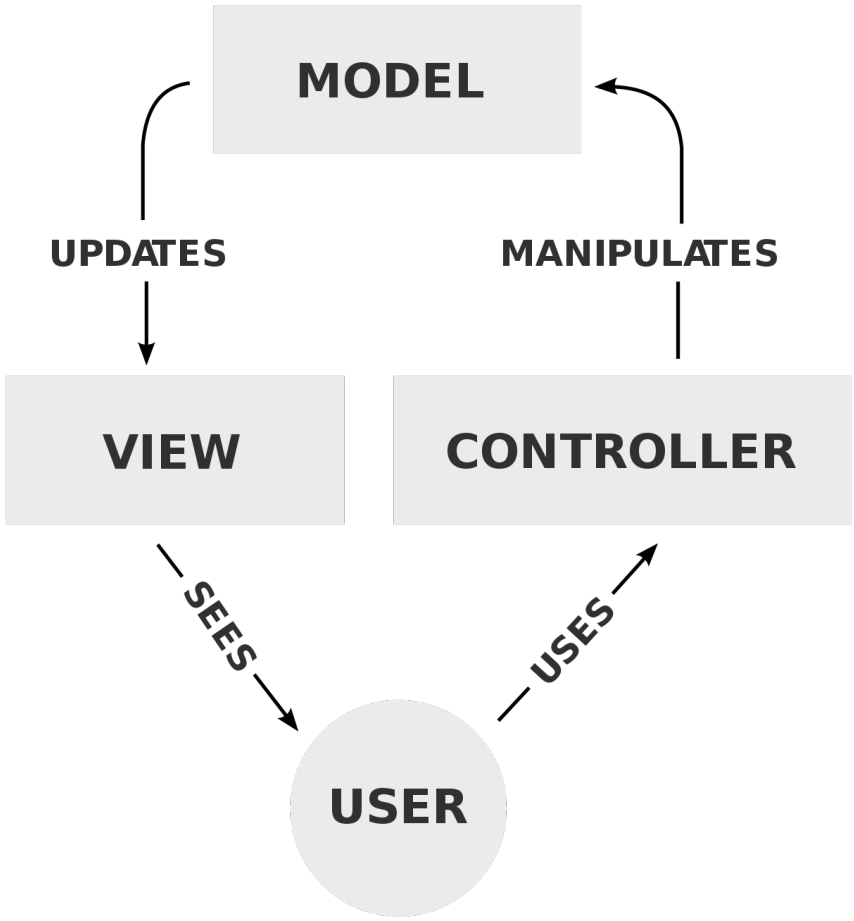

Рисунок 2.3 – Архітектура MVC [5]

Слід відмітити, що концептуально архітектура MVC не дуже сильно відрізняється від MVP, але у межах розробки мобільних застосунків склалися певні труднощі з використанням MVC. Суть цих труднощів складається у тому, що у Android SDK дуже складно виділити межі шарів View та Controller, через те, що фреймворк змішує декілька компонентів воєдино. Таким чином, найбільш простий Android застосунок складається з одного компоненту – Activity, який в свою чергу використовує певні елементи користувацького інтерфейсу за допомогою XML-розмітки. І якщо виділити шар Model у такому підході не складно – з шарами View та Controller виникають труднощі через те, що один клас (Activity) виконує обидві функції – відображає дані, та надає шар який контролює взаємодію с системою (Controller).

Для більш чіткого розподілу обов'язків між компонентами у Android застосунках було адаптовано інший архітектурний підхід – MVP (рис. 2.4).

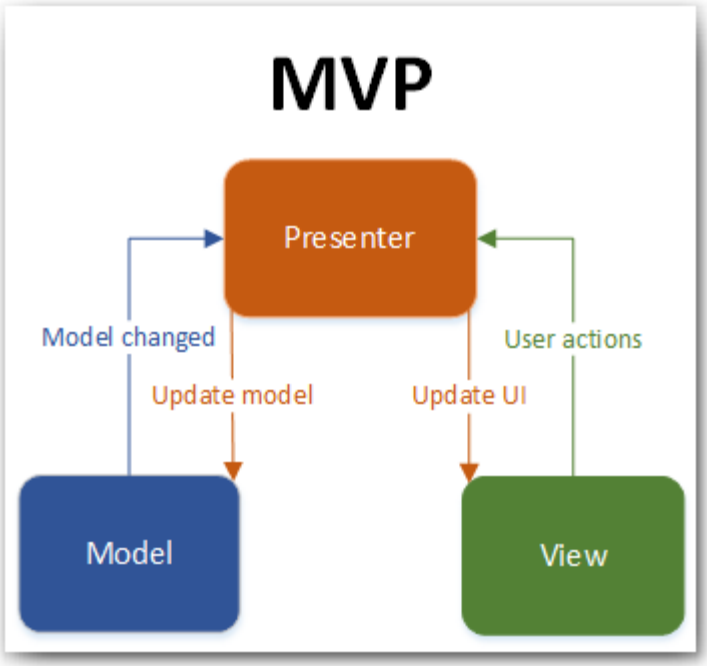

Рисунок 2.4 – Архітектура MVP [6]

Головна відмінність архітектури MVP від MVC полягає у тому, що межі презентаційного шару чітко виділені у вигляді окремої сутності Presenter, що

призводить до того, що компонент Activity стає шаром відображення, незважаючи на те, що в ньому є деякий код, який здебільшого викликає презентаційний шар.

Слід також відмітити, що у MVP підході шари Model та View ізольовані один від одного, на відміну від MVC, що призводить до значно більшої чіткості розподілу відповідальності між сутностями. Підхід MVP є популярним і на сьогоднішній день він має найбільш популярну реалізацію у вигляді бібліотеки Moxy [7].

Незважаючи на відносну простоту та зручність використання й розширення коду з архітектурою MVP, нещодавно з'явилася ще одна архітектура – MVVM, принцип роботи якої зображено на рис. 2.5. Слід відмітити, що MVVM у найбільш канонічному вигляді з'явився ще давно у платформах від Microsoft таких як Windows Presentation Foundation (WPF) та Silverlight. У екосистемі мобільних застосунків на базі ОС Android архітектурний підхід MVVM здобув дещо нестандартну реалізацію компанією Google у рамках бібліотек Android Architecture Components.

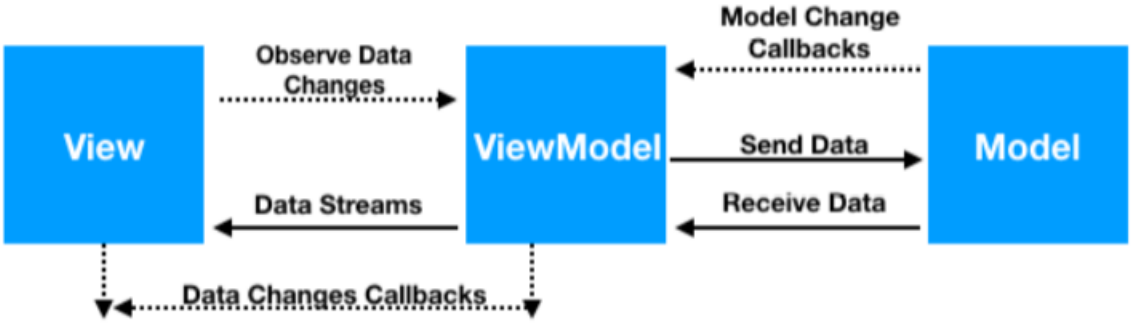

Рисунок 2.5 – Архітектура MVVM [8]

Як можна бачити з діаграми, принцип роботи MVVM дуже схожий на MVP, з єдиною різницею – компонент ViewModel не спілкується з View напряму, а замість цього використовується паттерн Observer, який надає можливість View спостерігати зміни ViewModel, та відповідно реагувати на них, змінюючи відображення.

У контексті Android застосунків використовується дещо специфічний механізм зв’язку ViewModel з View – використовуючи компоненти LiveData та DataBinding (рис. 2.6).

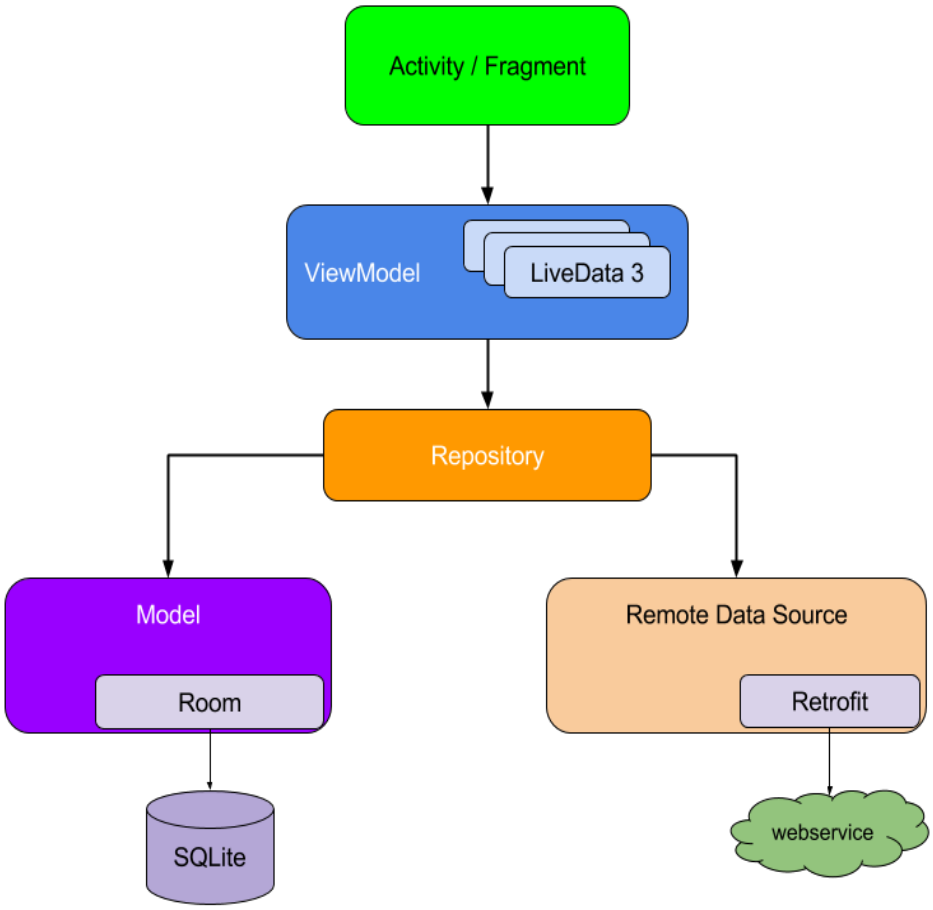

Рисунок 2.6 – Типова MVVM архітектура Android застосунку [9]

Компонент LiveData відповідає саме за той принцип Observer, та реалізує можливість View (в даному випадку Activity або Fragment) спостерігати зміни даних у ViewModel, та відображати їх. Потрібен цей компонент також для того, щоб

Компонент DataBinding використовується для автоматизації відображення даних за допомогою використання спеціального XML-синтаксису, який дозволяє вказати назви певних LiveData компонентів, які будуть автоматично відображені у елементах інтерфейсу. Слід відмітити, що аналогічний механізм не є специфікою

реалізації конкретного підходу у Android SDK, а присутній майже у будь-якій реалізації MVVM для зменшення кількості коду, необхідного для відображення даних.

Останньою з найбільш розповсюджених архітектур для розробки Android застосунків є MVI (Model – View – Intent). Цей підхід є найбільш новим, та відноситься до архітектур з односпрямованим потоком даних. Архітектурну діаграму MVI зображено на рис. 2.7.

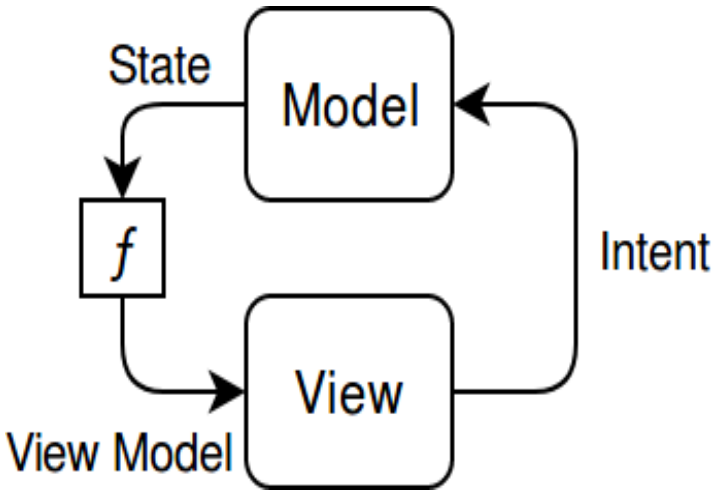

Рисунок 2.7 – Архітектура MVI

Як можна бачити, MVI загалом базується на тих же принципах, що й MVP та MVVM, але має дещо інший потік даних – View може спілкуватися з іншими компонентами лише генеруючи спеціальні об’єкти Intent (намір), які потім обробляються шаром Model. На етапі обробки намірів з’являється нове поняття – стан (state), яке відповідає за деякі дані, які необхідні системі. Цей стан застосунку змінюється відповідно до намірів, які надходять до системи, наприклад може змінитися флаг, який відображає процес завантаження даних, коли користувач викликає цей механізм. Після зміни стану, відбувається ще одна операція, яка є специфічною для MVI архітектури – трансформація даних (на рис. 2.7. зображена як функція *f*). Суть трансформації полягає у тому, щоб підготувати дані які зберігаються у стані для відображення у компонентах користувацького інтерфейсу

(View). Таким чином досягається ще більша (у порівняння з MVVM) ізоляція відповідальності компонентів та здатність до тестування.

Останній момент, який є не дуже логічним у архітектурі MVI – використання компоненту ViewModel, який використовується у MVVM. Незважаючи на однакову назву, цей компонент виконує кардинально різні функції – у MVVM він містить майже усю логіку презентаційного шару, в той час як в MVI цей клас є необхідним лише для зв'язку стану застосунку та моделі з користувацьким відображенням.

Слід відмітити, що самостійна реалізація підходу MVI є складною задачею (на відміну від, наприклад, MVP), що призводить до використання сторонніх рішень. Найбільш популярною реалізацією MVI на сьогоднішній день є бібліотека MVICore.

Окрім описаних вище архітектур також у деяких випадках використовуються більш специфічні підходи – такі як VIPER (View – Interactor – Presenter – Entity – Router), RIBs (Router – Interactor – Builder), та інші - але вони не розглядаються через дуже низьку популярність та вузьку направленість – кожна з цих архітектур використовується лише у одному випадку, або ж не використовується зовсім у сучасній Android розробці.

2.3 Переваги та недоліки існуючих підходів

Як вже було відзначено, існує два основних типи проектування архітектур для застосунків с користувацьким інтерфейсом – архітектури з односпрямованим потоком даних (unidirectional) та з двостороннім (bidirectional).

Кожен з цих підходів має як свої переваги, так і недоліки. Серед переваг односпрямованого потоку даних можна відмітити наступні:

- масштабованість;
- простота модифікації;

- можливість інтеграції будь-якої бізнес-логіки;
- можливість відстежувати усі зміни у системі;
- відсутність необхідності у спеціальній інфраструктурі (наприклад, DataBinding).

Серед недоліків архітектурних підходів з односпрямованим потоком даних можна виділити наступні:

- велика кількість коду для реалізації базової архітектури, необхідність у створенні бібліотек;
- велика (в порівнянні з двоспрямованим підходом) кількість коду для реалізації простого відображення;
- когнітивне навантаження – необхідно завжди пам’ятати, що шар відображення треба оновити після зміни даних.

Що стосується двоспрямованого потоку даних, можна виділити наступні переваги:

- простота використання;
- автоматичні оновлення відображення після зміни даних – і навпаки, завжди актуальний стан даних;
- невелика (порівняно з односпрямованим підходом) кількість коду для відображення даних;
- наявність готової інфраструктури – DataBinding та LiveData.

Але незважаючи на велику кількість переваг, двоспрямований підхід має також серйозні недоліки:

- залежність від інфраструктури – DataBinding працює з використанням кодогенерації, що зменшує швидкість збірки проекту та дуже часто призводить до незрозумілих помилок;
- необхідність у вивченні спеціального синтаксису у XML;
- неможливість реалізації складної бізнес логіки – увесь процес обмежений спеціальним синтаксисом у XML, якому бракує дуже великої кількості функціоналу;

- нескінченні цикли оновлення – трапляються ситуації, у яких оновлення відображення приводить до оновлення ViewModel, яке циклічно оновлює відображення;
- проблеми з швидкодією – автоматичне оновлення одразу обох компонентів може викликати велику кількість методів та перемальовувань користувацького інтерфейсу;
- інтеграційні проблеми – бібліотека DataBinding має помилки, що дуже часто складно відслідкувати;
- складність роботи з сторонніми компонентами – за наявності такого (наприклад, модальне вікно) необхідно також реалізувати певний функціонал, який буде адаптувати його інтерфейс до використання DataBinding.

Підбиваючи підсумки слід відмітити, що як односпрямований так і двоспрямований підхід мають свої сценарії використання. Двоспрямований підхід є більш доцільним для використання у невеликих застосунках, де критичною рисою є швидкість розробки, в той час як односпрямований підхід більш доцільно використувати у великих застосунках, насичених бізнес-логікою.

2.4 Порівняння конкретних архітектурних реалізацій

Кожна з архітектурних реалізацій (MVP, MVVM, MVC та інші) є похідною від певного підходу (наприклад, односпрямованого потоку даних) та наслідує усі переваги та недоліки від нього. Але також кожна архітектура має власні риси, які слід відзначити.

Найбільш застарілим підходом вважається MVC, який не має чіткого розподілу на зони відповідальності, а також найменш відповідає принципам чистої архітектури (clean architecture), які були сформовані Робертом Мартіном. Через те,

що недоліки цього підходу вже давно вивчені, а інші архітектури їх вже позбулися - немає сенсу надалі розглядати цей підхід.

Спільною ідеєю усіх цих архітектурних підходів (окрім MVC) є одна особливість - виділення окремого презентаційного шару (presentation layer) та шару відображення (view layer), що робить модель незалежною від конкретного відображення. Незважаючи на це, недоліки у цих підходів дуже різні, тому виділимо їх у кожній архітектурі, починаючи з MVP:

- складність проектування для забезпечення завжди актуального стану застосунку - необхідність використовувати парадигму реактивного програмування;
- великий розмір та складність підтримки об’єкту Presenter;
- велика кількість інтерфейсів та абстракцій (навіть у простих сценаріях), що робить код складним до сприйняття.

Наступним (історично) після MVP є підхід MVVM, який намагається вирішити його проблеми, але має свої недоліки:

- велика зв’язність (coupling) коду та майже повне ігнорування принципів чистої архітектури для забезпечення простоти використання;
- необхідність у додаткових інструментах для найбільш ефективного використання (наприклад, можливість задавати зв’язок з ViewModel у xml файлах, які описують рівень відображення);
- складність тестування.

Останнім та найбільш актуальним на сьогоднішній день підходом є MVI, який має багато спільного з архітектурами з односпрямованим потоком даних, але також має наступні відмінності та недоліки:

- найбільша складність у реалізації та підтримці;
- відсутність глобального стану застосунку - кожен екран має свій стан, що робить складним реалізацію спільної логіки та робить неможливим використання спільних даних у багатьох місцях;

Таким чином, для того, щоб реалізувати архітектуру, якою буде зручно користуватися, та яка буде надавати можливість розширювати код з найменшою кількістю помилок, необхідно реалізувати наступні вимоги:

- простота використання – важливі сутності є чітко виділеними та задокументованими, відповідальності декількох різних сутностей не змішуються між собою;
- незалежність від фреймворку – архітектура повинна постачатися окремим модулем, який має найменшу можливу кількість залежностей (в ідеальному випадку – жодної залежності);
- можливість використання глобального стану застосунку – дані не повинні бути прив’язані до екранів або інших компонентів;
- незалежність від сторонніх інструментів (наприклад, Databinding).

3 ВИБІР ТА ПОСТАНОВКА ЗАДАЧІ

3.1 Постановка задачі

У якості завдання було обрано дослідження можливості застосування односпрямованих архітектур для проектування мобільних застосунків.

Головне питання, на яке необхідно знайти відповідь - «Чи доцільно використовувати Redux для розробки мобільних застосунків?». Для того, щоб визначити це, необхідно виділити певні метрики, реалізувати однаковий за функціоналом застосунок з використанням декількох різних архітектурних підходів, та провести порівняння.

3.2 Аналіз аналогічних досліджень

Тема односпрямованого архітектурного підходу до проектування програмного забезпечення є досить розповсюдженою, тому існує декілька популярних варіантів реалізації цього підходу:

- The Elm Architecture;
- Redux;
- Flux.

Кожен з цих підходів використовується у спеціальному середовищі - наприклад, The Elm Architecture найбільш розповсюджений у програмах написаних з використанням мови програмування Elm. Redux (як архітектура похідна з Flux) є бібліотекою, та широко використовується у проектуванні сучасних веб застосунків з використанням мови програмування JavaScript.

Найбільш популярним аналогічним дослідження на сьогодні є дослідження архітектури Elm [10], яке описує реалізацію концепту односпрямованого потоку даних у архітектурі Model-View-Update, яку також називають The Elm Architecture. У даному дослідженні розглядається побудова архітектурної моделі, та доказ її

валідності. Також досліджується побудування графічного інтерфейсу, та його зв’язок з механізмом unidirectional data flow для реалізації управління станом застосунку.

Flux (та похідний від нього Redux) на даний момент не мають формальних досліджень через свій відносно невеликий вік (Flux було вперше реалізовано лише у 2014 р., а похідний від нього Redux - ще пізніше).

Слід також відзначити, що визначною рисою дослідження від інших є орієнтованість на розробку саме мобільних застосунків.

3.3 Обґрунтування методів дослідження

Порівняння архітектур – це в першу чергу емпіричне дослідження. Для того, щоб порівнювати декілька архітектур, потрібно проводити експериментальні дослідження. Експериментальне дослідження проводиться у вигляді набору тестів. Для цього створюється стабільне середовище, розробляються тести. Мета експерименту - довести наукову теорію з визначеними залежними і незалежними змінними. В даному випадку проводяться тести швидкодії та тести на здатність коду для розширення і модифікації.

3.4 Послідовність етапів проведення наукового дослідження

Головним завданням дослідження є порівняння односпрямованого архітектурного підходу з вже існуючими рішеннями на основі реалізації модулю Redux, та побудови мобільного застосунку на базі ОС Android з його використанням.

В результаті виконання дослідження потрібно притримуватися наступного плану:

- дослідити найбільш популярні архітектурні підходи у сфері проектування мобільних застосунків;
- визначити переваги та недоліки кожного підходу;
- спроектувати модуль з реалізацією односпрямованого підходу на прикладі архітектури Redux;
- реалізувати архітектурний модуль;
- використати модуль у реалізації мобільного застосунку;
- виділити найбільш важливу та складну функціональність та метрики і порівняти аналогічну реалізацію з використанням усіх архітектур;
- сформувати порівняння підходів за метриками, та визначити, чи доцільно застосовувати односпрямований підхід до проектування у сфері мобільних застосунків.

3.5 Формування вимог до програмних компонентів

Першим програмним компонентом є архітектурний модуль Redux, який реалізує односпрямований підхід потоку даних. Для нього визначено наступні вимоги:

- надає механізм збереження стану застосунку;
- надає механізм зміни стану застосунку;
- автоматично повідомляє користувацький інтерфейс про зміни в стані застосунку;
- надає механізм контролю сторонніх ефектів (side effects) у асинхронному режимі;
- надає методи для розширення функціоналу модуля за допомогою сутності Middleware.

Для проведення експерименту та отримання результатів реалізується також клієнтський застосунок, який використовує реалізований модуль Redux. В даному випадку, для порівняння існують дві версій клієнтського застосунку – одна з використанням Redux, та інша з використанням MVVM. Обидві версії мають ідентичний функціонал, та надають користувачеві можливість робити замовлення медіа-контенту у професійних музикантів, та виконання цих замовлень (зі сторони музиканта).

Разом з реалізацією серверної частини, ця програмна система має відповідати наступним функціональним вимогам:

- можливість зареєструватися / авторизуватися як музикант або звичайний користувач;
- можливість переглянути список виконавців, які можуть виконати замовлення у ролі користувача;
- перегляд рейтингу виконавців;
- створення замовлення у ролі користувача з наявністю зручних налаштувань, які дозволяють описати необхідне аудіо;
- отримання замовлень у ролі виконавця, можливість відмовитися від замовлення;
- підтвердження результату - завантаження отриманого аудіо зі сторони виконавця, та підтвердження результату зі сторони користувача з послідуючим наданням рейтингу виконавцю.

4 РЕАЛІЗАЦІЯ

4.1 Опис програмних компонентів

В результаті роботи реалізовано три програмних компоненти – архітектурний модуль Redux, клієнтський застосунок на базі ОС Android, та серверний застосунок. Слід відмітити, що клієнтський застосунок містить два ідентичних набори функціоналу для експерименту та демонстраційних цілей – один з використанням MVVM архітектури, а інший – з використанням розробленого модуля Redux.

Клієнтський застосунок та архітектурний модуль використовують мову програмування Kotlin та Android SDK. Також використовуються бібліотеки Timber – для логування, Kotlin Coroutines – для побудування асинхронних структур, та Retrofit – для HTTP запитів. Уся комунікація між клієнтом та сервером здійснюється за допомогою REST API та формату JSON. Клієнтська серіалізація / десеріалізація виконується з використанням бібліотеки Moshi.

Тестування клієнтського застосунку виконано за допомогою JUnit4, бібліотек mockk для створення об’єктів заглушок та strikt для перевірки тверджень (assertions).

Серверну частину реалізовано з використанням мови програмування Typescript та фреймворку Nest.js. Дані користувачів зберігаються у хмарному сховищі MongoDB Atlas, яке уявляє з себе NoSQL базу даних, яку можна легко масштабувати на будь яку кількість користувачів за допомогою використання вбудованих сервісів балансування трафіку та горизонтального масштабування у хмарі.

Для розгортання програмної системи обрано також хмарний провайдер AWS та сервіс Elastic Computing. Для того, щоб серверну частину можна було легко перенести у будь-яку іншу інфраструктуру було використано Docker контейнеризацію.

В цілому, архітектура програмної системи уявляє з себе класичний зразок N-tier [11]. Діаграму розгортання розробленої програмної системи зображено на рисунку 4.1.

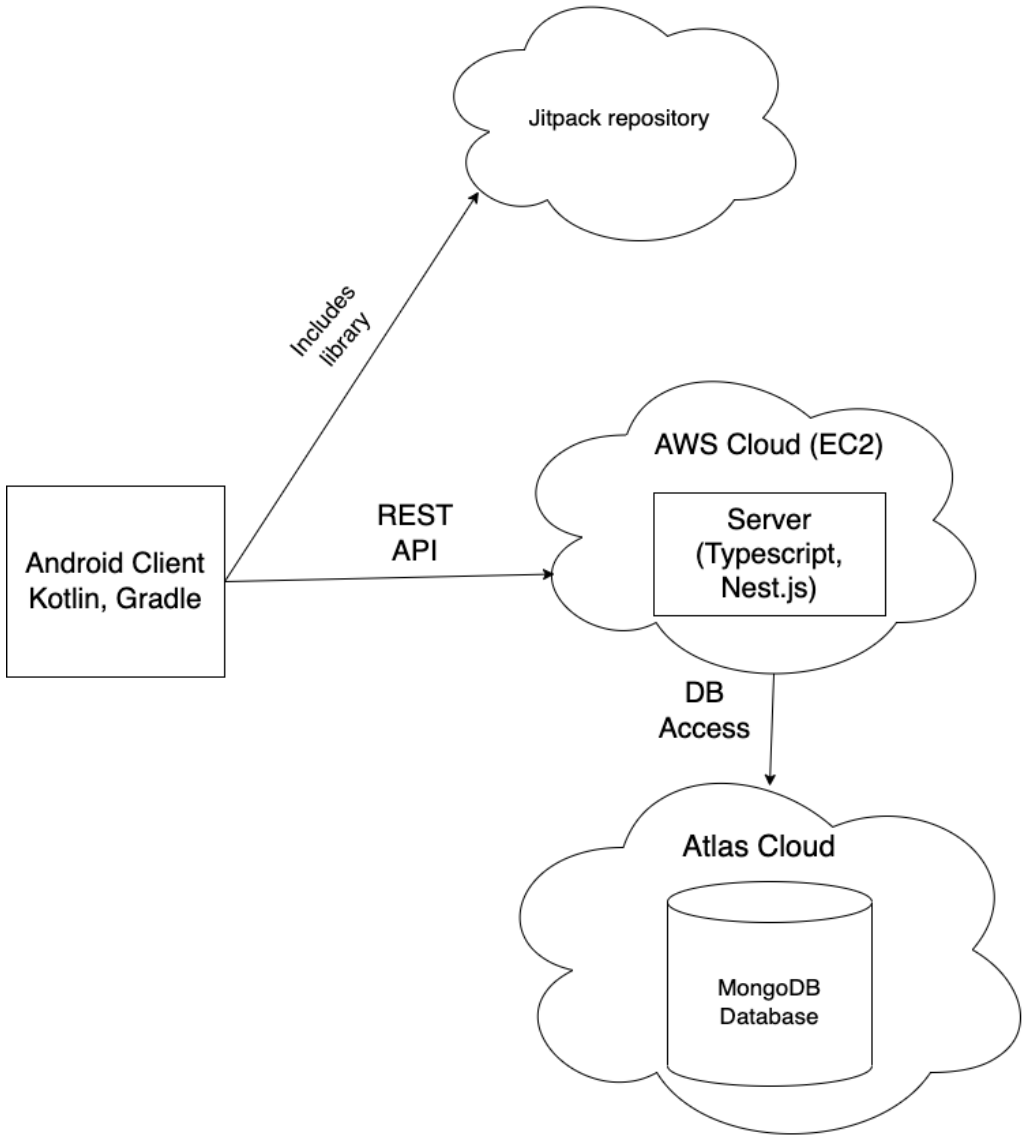

Рисунок 4.1 – Діаграма розгортання системи

Як можна бачити, більшу частину системи (архітектурний модуль, серверний застосунок та база даних) розгорнуто у хмарних сервісах – репозиторію Jitpack, хмарному середовищі AWS, та MongoDB Atlas. Зокрема слід відмітити механізм постачання архітектурного модуля – завдяки репозиторію Jitpack він є доступним для використання майже у будь якому Android застосунку за допомогою додання відповідної залежності у build.gradle.

4.2 Механізм використання архітектури

Для того, щоб визначити спосіб, яким буде використовуватися реалізована архітектура – слід визначити, які механізми підключення коду можуть бути використані.

На сьогоднішній день найбільш популярними системами збірки нативних застосунків під ОС Android є Bazel та Gradle. Кожна з цих систем збірок дозволяє підключати окремі програмні модулі – локально, чи з віддаленого репозиторію. Таким чином, найбільш комфортним для кінцевого користувача буде публікація архітектурного модуля у віддалений репозиторій Maven, залежності з якого можна буде використовувати у будь якому проекті, який використовує типові механізми збірки проекту. Приклад схеми залежностей модулів типового проекту наведено на рис. 4.1.

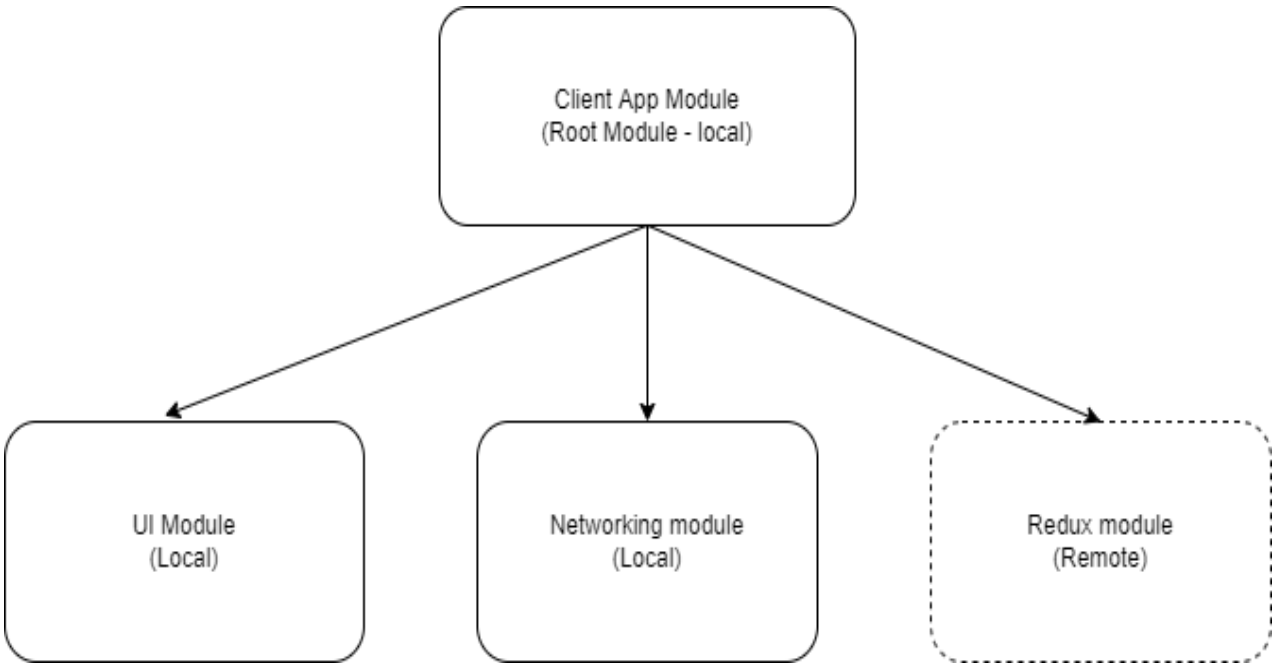

Рисунок 4.1 – Приклад системи модулів проекту

Як можна бачити, типовий сценарій використання бібліотеки клієнтом включає у себе підключення модулю з репозиторію Maven до кореневого модулю

застосунку (App). Після цього необхідно реалізувати певні інтерфейси та виконати процедуру налаштування модулю, і далі вже можна використовувати усі сутності та механізми, які надає архітектурний модуль.

4.3 Специфікація програмного модуля

В результаті дослідження потрібно розробити архітектурний модуль, який реалізує концепцію односпрямованого потоку даних на прикладі архітектури Redux.

Опис функціонування модуля можна виконати за допомогою сигнатур головних функцій, які наведено у додатку А.

Цілком роботу та зв’язок архітектури з компонентами мобільного застосунку можно описати наступним чином:

- система має первинний стан, який відображається на екрані за допомогою об’єкту Props;
- користувач виконує певні дії, що призводить до виклику функції dispatch з певними параметрами Action;
- кожна дія користувача визиває функцію reduce, яка є чистой функцією - приймаючи дію та попередній стан, формується новий стан застосунку;
- новий стан додатку використовується для формування нового об’єкту відображення даних - Props;
- після зміни стану, функція applyMiddleware отримує виконану дію користувача, та передає її кожному зареєстрованому обробнику дій, що може виконати певні ефекти (наприклад, виконати запит до зовнішнього сервера);
- об’єкти Middleware асинхронно виконують необхідні дії та виконують dispatch об’єктів Action з результатами виконання.

Спроектовану структуру програмного модуля, що реалізовує архітектуру Redux, наведено на рис. 4.2.

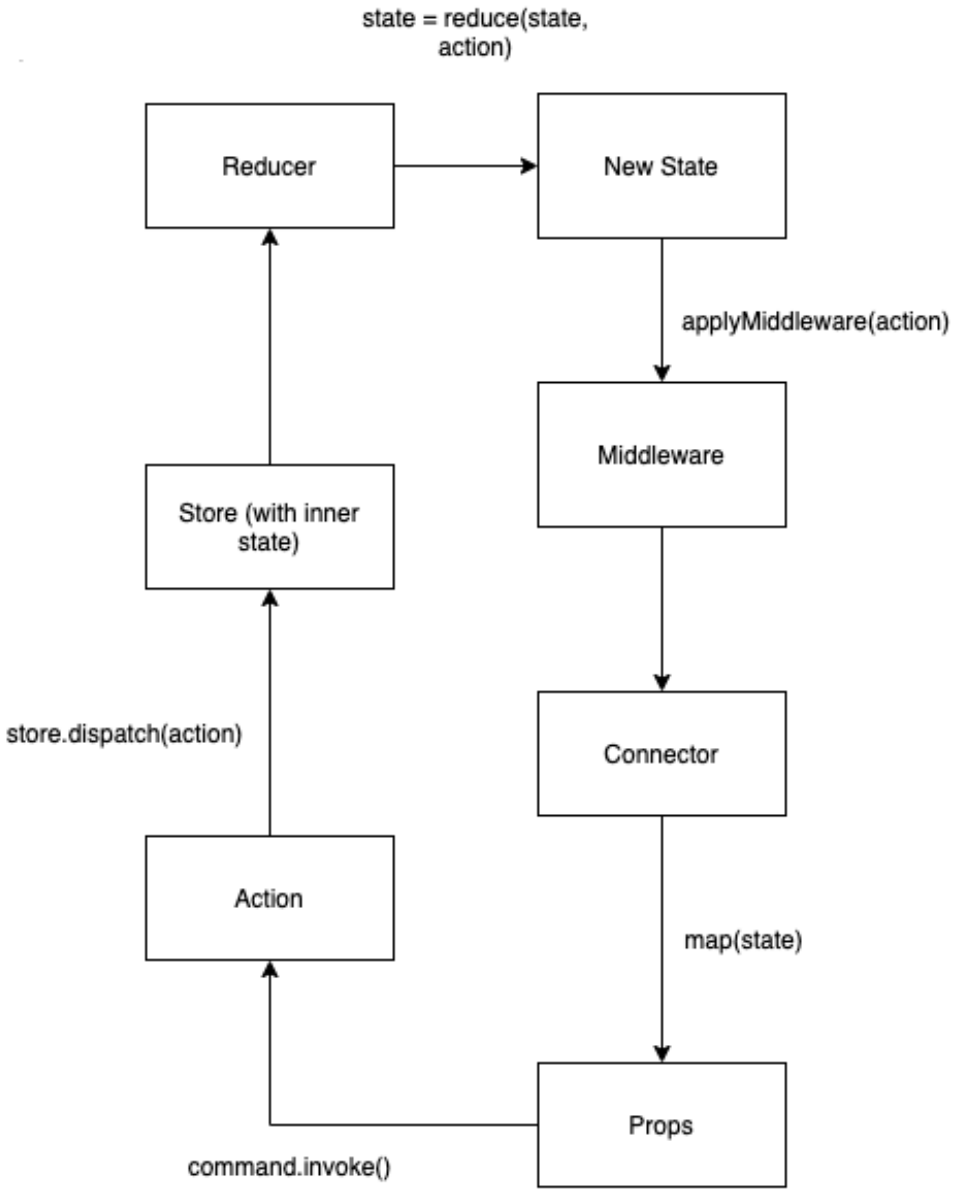

Рисунок 4.2 – Принцип роботи архітектури Redux

Таким чином, за допомогою реалізації наведеного принципу роботи Redux можна досягнути односпрямованого потоку даних – коли два компоненти не можуть звертатися друг до друга, а замість цього викликають інші компоненти, які формують коло викликів.

5 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ

5.1 Методологія проведення експерименту

Основна ціль експерименту - порівняння розробленого односпрямованого підходу з найбільш розповсюдженими на даний момент аналогами. Для того, щоб провести це порівняння, визначимо набор головних метрик, які ми можемо порівнювати:

- швидкодія застосунку;
- здатність до тестування;
- здатність до розширення (додавання нового функціоналу);
- здатність до підтримки (простота внесення змін у вже існуючий код);
- кількість допущених помилок під час розробки функціоналу.
- час на первинну інтеграцію архітектури у проект;
- час, необхідний на додавання нового функціоналу.

Враховуючи усі наведені метрики, придатність до використання певної архітектури (*Q)* можна визначити як значення функції:

$$Q = f(h_r, h_c, t, e, m, t_i, t_s)$$

де h_r - використання оперативної пам’яті, h_c - використання CPU, t - показник здатності до тестування, e - показник розширюваності, m - здатність до подальшої підтримки, t_i - час первинної інтеграції, t_s - витрати часу на додавання подальшого функціоналу.

Найбільш чіткими та легко вимірюваними є показники швидкодії (завантаженість процесору, зайнятий обсяг оперативної пам’яті, час, необхідний на обробку операції, тощо), які можна вимірювати за допомогою спеціального програмного забезпечення, - можливість отримання помилок та похибка є мінімальною.

Більш розмитими, але в той же час не менш важливими під час вибору архітектури є показники здатності до тестування, розширення, та підтримки. Здатність до розширення коду можна виміряти як час або кількість рядків коду /

сутностей, необхідних для реалізації нового функціоналу у системі. Здатність до підтримки аналогічно вимірюється як кількість рядків коду та витрати часу на внесення змін у вже реалізований функціонал.

Здатність до тестування архітектури можна визначити через два головних параметри - кількість рядків коду, які необхідні для тестування конкретного функціоналу, а також крихкість (fragility) коду. Крихкість коду є показником, який відображає яка кількість написаних тестів для певного функціоналу зламається в результаті внесення змін у логіку. Також для вимірювання здатності коді до тестування можна застосувати параметр покриття коду тестами (test coverage), але він може не відображати реальну здатність коду до тестування, тому у даному випадку слід очікувати помилки.

Кількість допущених під час розробки помилок є також важливою метрикою, але в той же час, це найбільш хаотичний показник з усіх перерахованих. Досягти адекватних для порівняння показників цієї метрики майже неможливо через те, що у випадку реалізації функціоналу одним розробником, кількість помилок залежить від порядку проведення експерименту - перша архітектура буде майже завжди мати більше помилок, ніж остання (через те, що розробник отримує досвід та не допускає помилок у подальші рази). У випадку реалізації функціоналу декількома розробниками, показник також не є чітким через досить очевидну причину - вплив людського фактору.

Час, який потрібен на первинну інтеграцію архітектури у проект, та час, необхідний для додавання нового функціоналу, також є дуже важливими метриками, які слід вимірювати використовуючи не значення у годинах, а перш за все відношення показника однієї архітектури до показника аналога, з яким потрібно провести порівняння. Також зазначимо, що не можна розглядати значення цих двох параметрів як одне - витрати часу - через те, що час первинної інтеграції може дуже сильно змінити результати експерименту - деякі архітектури мають дуже великий показник, який компенсується потім на великому проміжку часу за рахунок зменшення часу додавання нового функціоналу. Враховуючи, що експеримент буде проводитися на досить обмеженому проміжку часу (в порівнянні з реалізацією

великого проекту) для адекватного порівняння слід вимірювати саме два показника.

Також введемо поняття величини *C* для відносного порівняння двох архітектур.

$C_{AB} = (q_{ab} / q_m)$, де q_{ab} - кількість метрик архітектури А, які є кращими за відповідні метрики В, а q_m - загальна кількість метрик для порівняння.

Маючи показники кожної архітектури, відповідь на питання “чи є доцільним використання архітектури А замість В” можна розглядати як нечіткий параметр з наступними можливими інтервалами значень:

- $C_{AB} >= 0.75$ - Архітектура А є рекомендованою замість В майже в усіх випадках;
- $0.5 <= C_{AB} < 0.75$ - А є рекомендованою замість В у певних випадках;
- $C_{AB} < 0.5$ - А є рекомендованою замість В у виключних випадках.

Порівнявши певну архітектуру з усіма аналогами, отримати відповідь на більш обширне питання “чи є використання архітектури А доцільним” можна аналогічним чином, просумувавши усі показники $C_{A(i)}$ та поділивши на кількість аналогів.

Таким чином, ітог експерименту – зібрані метрики відносно кожної з порівнюваних архітектур, які надають можливість зробити висновки відносно того, чи можна використовувати певну архітектуру. Слід також відмітити, що підбір метрик було виконано таким чином, щоб порівнювати найбільш універсальні показники, що забезпечує актуальність дослідження для більшості випадків.

Слід відмітити також, що для розробки специфічних застосунків (наприклад, застосунки які дуже сильно залежать від можливостей телефона – таких як аналіз звуку, даних Bluetooth Low Energy, датчиків гіроскопу, акселерометра та інших) потрібно проводити окремий експеримент з визначенням специфічних метрик – це можуть бути показники швидкодії, можливості інтеграції з системними інтерфейсами (API) та інші. В даному експерименті такі сценарії використання не розглядаються.

Також слід відмітити, що аналіз результатів експерименту може різнитися від конкретного випадку – наприклад, у командах з великою кількістю розробників може вважатися більш важливими показники масштабованості та розширюваності коду, в той час як в невеликих командах – швидкість написання коду та простота.

5.2 План експерименту. Опис процесу тестування

Для того, щоб визначитися числові дані повинні бути зібрані для всіх метрик кожної архітектури, тому використовується контрольований метод експерименту. Для проведення експерименту вибрано стабільне і постійне середовище виконання - фізичний пристрій Android, технічні характеристики якого наведено у таблиці 1.

Таблиця 5.1 - Характеристики тестового пристрою

Device Name	Google Pixel 4
Processor	Qualcomm Snapdragon 855, Pixel Neural Core
RAM	6GB
Screen Resolution	2280 × 1080
OS	Android 10.0
GPU	Adreno 640

Незалежною змінною в експерименті є конкретні випадки використання, які будуть проводитися для заміру показників швидкодії застосунку. Під випадком

використання мається на увазі проведення певної кількості дій користувача, яка є типовою для реалізованого застосунку (наприклад, відкрити екран, ввести дані, натиснути кнопку, перейти на наступний екран). Слід зазначити, що для того, щоб виключити помилки, які можуть бути допущені під час тестування людиною, потрібно автоматизувати цей процес за допомогою програмних інструментів автоматизованого тестування (Espresso, UI Automator). У результатах тестування швидкодії застосунку (навіть за умови використання автоматизації) слід очікувати похибки та помилки через те, що є залежність від взаємодії між клієнтом і сервером (час якої не є константним) а також через певну невизначеність швидкодії системних компонентів. Для компенсації впливу цих факторів на результати слід збільшити кількість повторень експерименту, що не є проблемою за умови наявності ресурсів для автоматизованого тестування.

Інші показники якості архітектури, а саме - здатність до тестування, розширювання та підтримки - вимірювати потрібно після реалізації необхідного функціоналу та написання тестів. Незалежною змінною в даному випадку є певні функціональні вимоги, які реалізуються за допомогою різних архітектурних підходів - таким чином, логіка застосунку залишається незмінною, а на реалізацію впливають лише особливості структурної організації коду (архітектури).

Останніми показниками, які необхідно отримати в результаті експерименту, - час на початкову інтеграцію та підтримку певної архітектури. Вимірюється цей час у відносній формі – наприклад, час реалізації певного функціоналу, використовуючи одну архітектуру приймається за N. Тоді час реалізації цього ж функціоналу за допомогою іншої архітектури можна умовно представити як більший (1.25N) або менший (наприклад, 0.75N).

Вимірюється цей час реалізуючи один і той же функціонал декількома способами, але похибка в даному вимірі може бути дуже великою за рахунок двох речей:

- людський фактор;
- обмеженість у часі - не має можливості повторити експеримент багато разів, щоб зменшити похибки.

Враховуючи складність об’єктивної оцінки цього параметру, у результатах експерименту його треба розглядати з найменшим пріоритетом.

5.3 Результати експерименту

Як було відзначено раніше, експеримент проводився за допомогою порівняння ідентичного реалізованого у двух випадках – з використанням архітектури MVVM, та з використанням розробленого архітектурного модуля Redux. У результаті було виміряно показники для кожного етапу розробки проекту, що надає можливість порівняти ці дві архітектури.

Окремо відзначимо, що функціонал застосунку було обрано з урахуванням типового застосунку під ОС Android, який має шаблонний функціонал – наприклад, авторизацію, обробку помилок, відображення списків. Таке рішення було прийнято з оглядом на те, що архітектура планує використовуватися саме для типових застосунків, а не вузькоспеціалізованих клієнтських застосунків – у такому випадку архітектуру повинно вибирати за потребою конкретних функціональних та нефункціональних вимог.

5.3.1 Первинна інтеграція у застосунок

Під час порівнянь архітектури існує певний час та обсяг коду, який необхідно написати для того, щоб обрана архітектура почала працювати. Цей час та обсяг існує ще до того, як починається реалізація функціоналу. Порівняємо ці показники для двох обраних архітектур.

У випадку розробленого модуля Redux ситуація є дуже простою – підключення бібліотеки виконується за допомогою додавання одного рядка у build.gradle файл:

```
dependencies {
      implementation “io.github.gubarsergey.redux:1.0.0”
 }
```

Таким чином, залежність додається до проекту, і завантажується з репозиторію Jitpack. Якщо Jitpack у проекті не використовується, його необхідно додати також до build.gradle файлу відповідно офіційній документації.

Після завантаження залежності, необхідно виконати первинні налаштування та надати модулю усі необхідні для функціонування залежності. Для роботи модулю необхідно надати наступні залежності:

- початковий стан застосунку;
- чисту функцію, яка дозволить змінювати стан застосунку (та кожного з вкладених станів);
- функцію виконання дії на головному потоці (необхідну для внутрішньої роботи бібліотеки);
- список сутностей Middleware, які перехоплюють дії у системі та виконують відповідні реакції (side effects);
- список сутностей Configurator, який відповідає за зв’язок користувацького інтерфейсу з глобальним станом.

Надання усіх цих залежностей виконується у манері схожій на DSL (Domain Specific Language), тільки суттєво необхідні залежності надаються за допомогою конструктора. Наведемо приклад конфігурації модуля для двох частин функціоналу – авторизація користувача, та перегляд списку усіх замовлень (в ролі користувача):

```
core = setupRedux(
       defaultState = ReduxAppState(
           auth = AuthState.default,
           myOrders = OrdersState.default,
       ),
       applyReducers = { state, action ->
           ReduxAppState(
               auth = Reduce.authState(state.auth, action),
```

```
                myOrders = Reduce.orders(state.myOrders, action),
            )
        },
        runOnUiThread = { action ->
            handler.post(action)
        }
    ) {
        withMiddlewares(
            listOf(
                LoggingMiddleware,
                AnalyticsMiddleware,
                AuthMiddleware(this),
                OrdersMiddleware(this),
            )
        )


        withConfigurators(
            listOf(
                CounterConfigurator(this),
                AuthConfigurator(this),
                OrdersConfigurator(this),
            )
        )
    }
```

Повну версію коду для конфігурації модуля наведено у додатку В. Як можна бачити, первинне додавання модуля та його конфігурація майже не витрачають часу і мають малий обсяг коду (більша частина коду, наведеного вище, відноситься до прикладу додавання нового функціоналу, та наведена для того, щоб зрозуміти як конфігурація буде виглядати під час реалізації подальшого функціоналу).

У випадку MVVM залежність також одна – бібліотека Model-View-ViewModel, яка додається наступним чином у build.gradle:

```
    dependencies {
            implementation "androidx.lifecycle:lifecycle-viewmodel-
ktx:$lifecycle_version"
        }
```

Змінна lifecycle_version в даному випадку – це версія бібліотеки, яку необхідно знайти у репозиторію MavenCentral.

Також слід відмітити, що окрім прямої залежності на бібліотеку ViewModel, існують також додаткові залежності, які надають певні інтеграції з іншими компонентами – наприклад, залежність для використування корутін (coroutines) у ViewModel, або ж для збереження даних між змінами конфігурацій. Також існують залежності для тестування компоненту ViewModel. Кожна з наведених

залежностей не є обов'язковою, та використовуються в залежності від потреб кінцевого користувача.

Після додавання відповідної залежності, використовувати компонент ViewModel можна наступним чином:

```
class MyViewModel: ViewModel() {
}
```

На перший погляд, здається що конфігурація MVVM архітектури є значно простішою, ніж конфігурація Redux – але це не зовсім вірно. Наприклад, для реалізації такого базового функціоналу як забезпечення параметрів у конструкторі, необхідно використовувати шаблон фабрики, який не є інтуїтивним, та реалізується наступним чином:

```
class SomeViewModelFactory(private val someString: String):
ViewModelProvider.NewInstanceFactory() {
    override fun <T : ViewModel?> create(modelClass: Class<T>): T =
SomeViewModel(someString) as T
}
```

Ще одним прикладом неочевидності, яка стосується базового функціоналу є наявність компонентів ViewModel, які відносяться до спільної області видимості. За замовчуванням, кожен компонент користувацького інтерфейсу (Fragment) має відповідний компонент ViewModel – але у випадку, коли необхідно використовувати одні й ті ж дані у декількох компонентах користувацького інтерфейсу – цей підхід не дає такої можливості, тому потрібно реалізовувати окрему глобальну область видимості, у якій буде існувати компонент ViewModel, який буде доступним для декількох компонентів Fragment. Такою областю у більшості випадків є Activity, що робить використання архітектури більш складним та збільшує складність системи.

Таким чином, час та витрачені зусилля на реалізацію початкового функціоналу у випадку MVVM суттєво збільшуються, та використання такої архітектури вже не є настільки простим, як здається. Більшість з наведених проблем можна уникнути, використовуючи спеціальні бібліотеки – наприклад, проблему надання залежностей вирішують бібліотеки ін'єкції залежностей (Dependency Injection), прикладом якої може бути бібліотека Koin, яка має

інтеграцію з ViewModel, але у даному порівнянні розглядаються тільки базові версії бібліотек, без використання сторонніх залежностей.

У результаті порівняння часу на первинну інтеграцію та обсягу коду можна вважати, що жодна з архітектур не має суттєвих переваг над іншою. У випадку Redux необхідно виконати складні первинні налаштування, але у випадку MVVM у більшості випадків необхідно використовувати сторонні компоненти, та вирішувати базові проблеми архітектури, що робить приблизно однаковим для обох цей показник у порівнянні.

Якщо розглядати час на реалізацію, його також можна вважати рівним, так як жодна з архітектур не змушує витрачати велику кількість часу на її розгортання та конфігурацію.

5.3.2 Функціонал авторизації користувача

Першим функціоналом для порівняння виступає механізм авторизації користувача. Для реалізації було обрано найбільш типовий вид – авторизація за допомогою JWT (JSON Web Token), який отримується після надання логіна та пароля користувача.

Дослідження цього та подальшого функціоналу на предмет кількості строк коду/необхідних класів та інших сутностей проводилося за допомогою порівняння git diff.

Також слід відмітити, що деякі компоненти використовувалися одні й ті ж в обох досліджуваних архітектурах для того, щоб максимально ізолювати вплив сторонніх компонентів да дослідити виключно архітектуру. До таких компонентів входять:

- ресурси користувацького інтерфейсу (layout);
- сервісні класи (наприклад, доступ до локальних даних диску - SharedPreferences);

- класи, які відповідають за ізольовану комунікацію з сервером (створення та відправка HTTP запитів);
- класи, які відповідають за реалізацію ін’єкції залежностей;
- класи ресурсів застосунку.

Отримані результати порівняння наведено у таблиці 5.1.

Таблиця 5.2 – Результати порівняння функціоналу авторизації

Показник / Архітектура	MVVM	Redux
Кількість доданих рядків коду	172	234
Кількість створених сутностей (класів)	2 (ViewModel та Fragment)	6 (Fragment, Props, Connector, Configurator, State, Middleware)
Найбільша кількість відповідальностей у класі	4	1
Кількість об’єктів, для тестування яких необхідні окремі залежності	2	0
Відносний час, витрачений на реалізацію	1x	1.5x
Кількість рядків, необхідних для тестування	54	62

Як можна бачити, обидві архітектури мають свої переваги та недоліки. Розберемо кожний з показників окремо.

Кількість доданих рядків коду – показник розширюваності коду (менше – краще). Слід відмітити, що при вимірі цього показника виключався автоматично згенерований код – наприклад, методи equals та hashcode, які необхідні для відображення списків з Redux та DiffUtil механізмом. В даному випадку, MVVM має перевагу – кількість рядків коду, які необхідно додати, суттєво менша за Redux.

Кількість створених класів – також показник розширюваності. У випадку MVVM потрібно створити лише два класи – ViewModel та Fragment, в той час як Redux потребує шість класів – Fragment, Props, Connector, Configurator, State, Action.

Кількість об’єктів, для тестування яких необхідні окремі залежності – показник придатності коду до тестування. В даному випадку, Redux має суттєву перевагу – для тестування коду з використанням цієї архітектури не потрібно жодних бібліотек, або сторонніх залежностей. У випадку MVVM ми не можемо створити ані компонент ViewModel, ані LiveData, що робить тестування написаного коду більш складною задачею.

Найбільша кількість відповідальностей у класі – показник підтримки коду та розширюваності. Позначає те, наскільки створені сутності ізольовані та чи перетинаються їх відповідальності одна з одною (менше – краще). В даному випадку маємо явну перевагу Redux – кожна з сутностей відповідає тільки за одну річ, в той час як в MVVM клас ViewModel відповідає за завантаження даних, збереження, бізнес логіку та відображення.

Відносний час, витрачений на реалізацію – показник розширюваності коду. Порівняння є відносним, за базову величину береться та архітектура, час реалізації якої зайняв менше часу. Як вже було відзначено, цей показник не є надійним через те, що його дуже складно виміряти, та його не слід вважати основним, але в даному випадку перевага MVVM є досить значною – реалізація аналогічного функціоналу з використанням Redux зайняла майже в півтора рази більше часу. Така різниця виникає в більшості через те, що на екрані авторизації присутні компоненти, взаємодія з якими є складною у Redux – текстові поля вводу та відображення помилок. Це компоненти Android SDK мають свій власний стан, за рахунок чого

взаємодія з ними ускладнюється у архітектурі з централізованим механізмом керування станом застосунку. Більш детально цю проблему та інші описано у пункті 5.6.

Кількість рядків, необхідних для тестування – показник придатності коду до тестування (менше – краще). В даному випадку маємо незначну перевагу MVVM через те, що у Redux необхідно створити більшу кількість сутностей та надати більше залежностей.

Окремо слід відмітити, що під час реалізації авторизації виникла перша проблема архітектури MVVM – відсутність глобального стану. Як вже було відзначено, авторизація здійснюється за допомогою допомогою запиту, який повертає Json Web Token, який є необхідним для усіх подальших запитів – тому до нього необхідно мати доступ майже у всіх компонентах, але у випадку MVVM глобального стану не існує – кожна ViewModel не може обмінюватися даними з іншими, що робить збереження токену та доступ до нього нетривіальною задачею. В даному випадку було обрано механізм запису даних до диску SharedPreferences та створено окремий клас, якій постачається у вигляді залежності у всі компоненти ViewModel.

5.3.3 Функціонал відображення списку замовлень

Відображення списку даних є, мабуть, найбільш типовою задачею у Android застосунках. У дослідженні було реалізовано наступний функціонал – виконання запиту на сервер, трансформація даних, відображення списку у компоненті користувацького інтерфейсу. Також було реалізовано обробку помилок мережі, та випадок, коли замовлень немає. Результати порівняння двох архітектур для такого функціоналу наведено у таблиці 5.3.

Таблиця 5.3 – Порівняння функціоналу списку замовлень

Показник / Архітектура	MVVM	Redux
Кількість доданих рядків коду	215	251
Кількість створених сутностей (класів)	4	8
Найбільша кількість відповідальностей у класі	4	1
Кількість об’єктів, для тестування яких необхідні окремі залежності	2	0
Відносний час, витрачений на реалізацію	1x	1.2x

Як можна бачити з порівняльної таблиці, результати є схожими з функціоналом авторизації, та вже починає бути видно закономірність зв’язку двох архітектур.

Кількість доданих рядків коду – майже ідентична з авторизацією ситуація, реалізація Redux перевищує MVVM за обсягом на ~30%.

Найбільша кількість відповідальностей у класі – ідентичний показник з функціоналом авторизації. У порівнянні подальшого функціоналу він наводитися не буде через те, що висновок можна зробити вже на основі двох частин функціоналу – у інших місцях кардинальних змін не очікується через те, що

розподіл відповідальностей у певній архітектурі майже не змінюється в залежності від функціоналу.

Кількість об’єктів, для тестування яких необхідні окремі залежності – також ідентичний з авторизацією показник. Усі основні сутності та сценарії використання архітектури відносно цієї метрики вже розглянуто, тому далі вона також наводитися окремо не буде.

Відносний час, витрачений на реалізацію – в даному випадку, тенденція порівняння зберігається (Redux забирає більше часу, ніж MVVM), але різниця вже не є настільки суттєвою (~20% можна вважати на грані похибки, через те, що ця метрика не є чітко вимірюваною).

Як можна бачити, з двох реалізованих частин функціоналу вже видно певну тенденцію порівняння двох архітектур, але для того, щоб зробити висновки порівняємо ще декілька частин застосунку, включаючи більш складні екрани та функціонал.

5.3.4 Функціонал відображення музикантів

Відображення списку музикантів є прикладом одного з найбільш типових екранів мобільних застосунків. Основна задача екрану – відобразити колекцію даних, надати можливість їх сортувати, фільтрувати, та взаємодіяти з ними.

Цей екран є більш складним для реалізації (відносно розглянутих раніше частин функціоналу), тому він буде розглядатися у дві стадії – реалізація та підтримка.

У стадію реалізації входить базовий функціонал – завантаження доступних музикантів, у яких можна зробити замовлення, відображення цієї інформації на екрані, та обробка можливих помилок (немає інтернет з’єднання, невалідний ключ авторизації, тощо).

У стадію підтримки входить додавання функціоналу – сортування, фільтрації, та навігація до перегляду детальної інформації про музиканта. Таким чином, порівняння є дуже схожим на типовий проект, який має певний життєвий цикл – додавання нового функціоналу, модифікація існуючого функціоналу, видалення коду, рефакторінг та інші етапи. Замір показників у два етапи продемонструє, чи відрізняються результати при реалізації та супроводі функціоналу.

Для надання контексту, цей екран зображено на рис. 5.1.

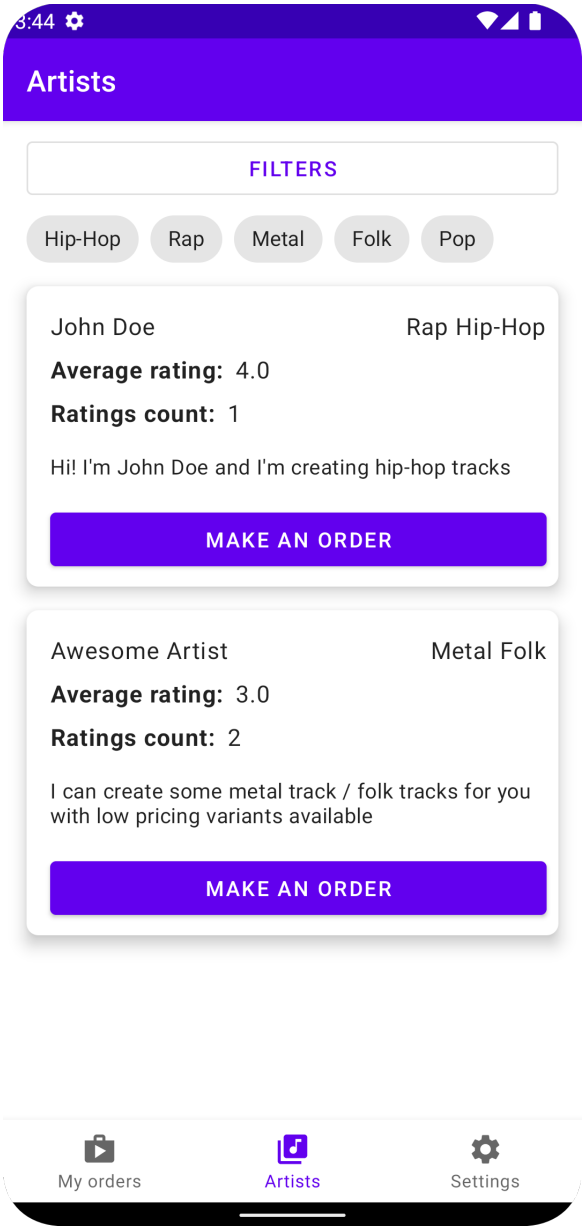

Рисунок 5.1 – Екран відображення музикантів

Результати порівняння для початкової реалізації функціоналу наведено у таблиці 5.4. Показники, результати порівняння яких є ідентичними до попередніх порівнянь у таблиці не наводяться.

Таблиця 5.4 – Порівняння початкової реалізації

Показник / Архітектура	MVVM	Redux
Кількість доданих рядків коду	204	243
Кількість створених сутностей (класів)	3	7
Відносний час, витрачений на реалізацію	1x	1.2x

Як можна бачити, результати початкової реалізації функціоналу є схожими з попередніми вимірами – Redux займає більше рядків коду та сутностей, і більше на у середньому 20% часу.

Проведемо порівняння показників розширення коду – додамо функціонал сортування, фільтрації та навігації до детального відображення. Результати порівняння наведено у таблиці 5.5.

Таблиця 5.5 – Порівняння показників розширення

Показник / Архітектура	MVVM	Redux
Кількість доданих рядків коду	84	89
Кількість створених сутностей (класів)	1	1

Кінець таблиці 5.5

Показник / Архітектура	MVVM	Redux
Відносний час, витрачений на реалізацію	1.2x	1x

Як можна бачити з результатів порівняння, показники при додаванні функціоналу та розширенні значно відрізняються. При розширенні функціоналу різниця у кількості створених сутностей та рядків коду у Redux та MVVM майже відсутня – усі необхідні сутності уже створено та пов’язано між собою, залишається лише модифікувати їх для реалізації нових вимог.

Час на реалізацію також змінився – модифікація коду з використанням архітектури Redux є більш простою, ніж реалізація функціоналу через те, що усі необхідні класи уже додано, кожен з яких має чітко визначену відповідальність. Таким чином, якщо необхідно модифікувати функціонал – розробник чітко знає, у які сутності йому необхідно внести зміни, що дуже сильно спрощує роботу. В даному випадку ці ознаки й вплинули на те, що час модифікації Redux є меншим за модифікацію коду з використанням MVVM архітектури.

Окремо слід відмітити, що під час модифікації коду на MVVM виникли труднощі, які пов’язані з відсутністю механізму збереження даних у ViewModel. Ідіоматична реалізація ViewModel полягає у тому, що цей компонент взаємодіє з певними джерелами даних (локальна БД, запити до сервера), і віддає результати взаємодії у компоненти LiveData, які потім отримують компоненти користувацького інтерфейсу за допомогою використання патерну Observer. Таким чином, ViewModel не має проміжного стану, який зберігається у оперативній пам’яті окрім значень LiveData, яких не завжди достатньо для реалізації необхідного функціоналу. Простий приклад такого випадку – додавання будь-якої фільтрації, яка з колекції розміром у N елементів робить колекцію розміром K елементів, K <= N. Якщо застосувати цю фільтрацію на значені LiveData і отримати нове значення, повернутися до попереднього значення буде неможливо –

інформація про елементи, які були до використання фільтру, вже втрачена. Для вирішення цієї проблеми створюють або ще один компонент LiveData, або зберігають дані у класі ViewModel. Створення нового об’єкту LiveData привносить ще більше проблем – тепер необхідно слідкувати одночасно за двома станами, а не за одним. Таке рішення потенціально при додаванні деякої кількості функціоналу майже лишає код можливостей розширення через те, що у об’єкті ViewModel накопичується велика кількість LiveData (одна на кожну частину функціоналу), що робить розширення коду без помилок майже неможливим вже при наявності 5-10 таких компонентів, в той час як на складному екрані застосунку великого масштабу може бути у кілька разів більше компонентів LiveData.

Під час модифікації функціоналу було використано як додавання нових LiveData, так і збереження стану у класі ViewModel. Розглянемо на прикладі, які проблеми це спричинило. Компоненти LiveData для реалізації цього екрану мають наступний вигляд:

```
 private val _artists =
MutableLiveData<List<AvailableArtistsProps.ArtistProps>>()
    val artists: LiveData<List<AvailableArtistsProps.ArtistProps>> =
_artists

    private val _errors = MutableLiveData<APIError>()
    val errors: LiveData<APIError> = _errors

    private val _sorting = MutableLiveData<Sorting>()
    val sorting: LiveData<Sorting> = _sorting

    private val _chipsSelected: MutableLiveData<MutableList<String>> =
MutableLiveData()
    val chipsSelected: LiveData<MutableList<String>> = _chipsSelected
```

Таким чином, ми маємо чотири компоненти – відображення списку музикантів, помилок взаємодії з сервером, сортування (окремою кнопкою) та фільтрації (з використанням компоненту chips). Слід відмітити, що тільки один з цих компонентів є ізольованим та незалежним від інших – помилки взаємодії з сервером, усі інші компоненти пов’язані між собою. Компонент відображення музикантів є тісно зв’язаним з компонентом сортування і фільтрації, що робить їх ізоляцію один від одного майже незалежним. Таким чином виникає проблема необхідності модифікації декількох об’єктів стану за раз. Наведемо приклад – при

застосуванні певного фільтру користувачем нам необхідно змінити стан _sorting для того, щоб кнопка сортування відобразила обраний тип сортування. Але в той же час, зміна лише одного компоненту _sorting не є достатньою – список музикантів ніяк не зміниться, що призведе до некоректної поведінки. Таким чином, розробнику необхідно не забути про усі компоненти LiveData, які необхідно змінити, що дуже сильно ускладнює модифікацію функціоналу. Як подальший приклад можна навести фільтрацію результатів – якщо виникає необхідність модифікувати і стан фільтрів в залежності від сортування (наприклад, заборонити певні фільтрації відсортованих даних) код може виглядати наступним чином:

```
fun selectBestRatingSort() {
        _sorting.value = Sorting.BEST_RATING
        _artists.value    =    _artists.value?.sortedByDescending    {
it.averageRating }
         _chipsSelected.value  =  _chipsSelected.removeFirst  {  it  is
RestrictedFilter }
    }
```

Як можна бачити з прикладу коду, одна взаємодія користувача (вибір методу сортування) викликає зміни трьох компонентів - _sorting, _artists, _chipsSelected, що робить модифікацію такого коду дуже складною та збільшує кількість помилок. При виникненні будь-якої події у системі (чи то взаємодія користувача, чи внутрішня взаємодія компонентів) необхідно кожний раз пам’ятати про консистентність усіх даних – якщо не змінити якийсь один з них, чи змінити неправильно – виникне функціональна помилка.

У реаліях сучасних систем, де певні екрані можуть мати 50 компонентів LiveData та більше такий підхід робить розширення та модифікацію коду без внесення помилок майже неможливою. Архітектура Redux дуже добре вирішує саме цю проблему за допомогою використання об’єкту Props. Props – це єдиний компонент, який необхідно сформувати для відображення користувацького інтерфейсу. В даному випадку об’єкт Props виглядає наступним чином:

```
data class AvailableArtistsProps(
    val artists: List<ArtistProps>,
    val viewLoaded: Command,
    val selectNoneFilter: Command,
    val selectBestRatingFilter: Command,
    val selectMostOrdersFilter: Command,
    val filter: Filter,
    val chips: Map<String, ChipInfo>
```

```
    ) {

        data class ChipInfo(
            val click: Command,
            val isSelected: Boolean,
        )
```

Як можна бачити, один обєкт описує весь стан користувацького інтерфейсу – список користувачів, стан фільтрації та сортування. Будь яка зміна у системі має створити новий об’єкт Props, що дуже сильно спрощує модифікацію – у такому випадку забути про зміну певного компоненту є майже неможливою через те, що створення об’єкту вже описує усі залежності між компонентами користувацького інтерфейсу:

```
        override fun map(appState: ReduxAppState): AvailableArtistsProps {
            return AvailableArtistsProps(
                artists = core.state.availableArtists.byId.map { (id,
stateArtist) ->
                    AvailableArtistsProps.ArtistProps(
                        id = id,
                        fullName = stateArtist.fullName,
                        profileDescription =
stateArtist.profileDescription,
                        email = stateArtist.email,
                        genres = stateArtist.genres,
                        averageRating =
stateArtist.ratingInfo.averageRating,
                        ratingCount =
stateArtist.ratingInfo.numberOfRatings,
                        makeAnOrder = Command.nop(),
                    )
                }.appliedFilters(appState),
                viewLoaded = core.bind(LoadAvailableArtists),
                selectNoneFilter =
core.bind(AvailableArtistsFilterReset),
                selectBestRatingFilter =
core.bind(AvailableArtistsBestRatingSelected),
                selectMostOrdersFilter =
core.bind(AvailableArtistsMostOrdersSelected),
                filter = when (appState.availableArtists.appliedFilter) {
                    AvailableArtistsState.ArtistFilter.NONE        ->
AvailableArtistsProps.Filter.NONE
                    AvailableArtistsState.ArtistFilter.BEST_RATING ->
AvailableArtistsProps.Filter.BEST_RATING
                    AvailableArtistsState.ArtistFilter.MOST_ORDERS ->
AvailableArtistsProps.Filter.MOST_ORDERS
                },
                chips =
appState.availableArtists.genresSelection.mapValues { (genre, isChecked) ->

AvailableArtistsProps.ChipInfo(core.bind(AvailableArtistsGenreSelection(gen
re)), isChecked)
                }
            )
```

```
        }
```

Як можна бачити з наведеного коду, увесь зв'язок між сутностями виражається у створенні об’єкту – колекція завжди містить відфільтровані та відсортовані елементи, а компонент сортування завжди містить актуальний стан для відображення відповідної кнопки. Таким чином, кількість окремих місць які необхідно модифікувати у MVVM для розширення функціоналу становить від 1 до N, де N – кількість компонентів LiveData, в той час як в Redux для зміни користувацького інтерфейсу завжди необхідно модифікувати лише одне місце таким чином, щоб при будь-якому стані системи завжди формувався коректний об’єкт Props.

Також слід більш детально описати проблему відсутності механізму збереження стану, яку було згадано раніше. Під час реалізації функціоналу фільтрації було знайдено помилку, яка виникла при обиранні жанру фільтрації – якщо обрати жанр, та потім відмінити фільтрацію, дані відображалися все одно відфільтровані. Наведемо приблизний код фільтрації (без деталей реалізації зі сторони комунікації Fragment-ViewModel):

```
    fun selectChip(chip: String) {
            // Calculate new filters
            _chipsSelected.value = newChips
            _artists.value       =       _artists.value?.filter       {
it.genres.containsAll(chips) }
        }
```

Як можна бачити, фільтрація результатів використовує попереднє значення _artists для формування нового значення. Таким чином, якщо відфільтрувати деяку кількість результатів – більше їх ніколи у списку не буде через те, що значення відфільтрованого списку заміняє повний список. Вирішити цю проблему можна додаванням проміжного стану у об’єкт ViewModel, таким чином список музикантів буде виглядати наступним чином:

```
    private var artistsData: AvailableArtistsResponseDto? = null

    private val _artists =
MutableLiveData<List<AvailableArtistsProps.ArtistProps>>()
    val artists: LiveData<List<AvailableArtistsProps.ArtistProps>> =
_artists
```

Як можна бачити, тепер дані про музикантів зберігаються одразу в двох місцях – компоненті LiveData та у artistsData. Такий підхід приводить до великої

кількості помилок, яких дуже складно уникнути. Серед найбільш розповсюджених проблем можна виділити наступні:

- незрозуміло, яке поле зберігає останні дані про музикантів - _artists чи artistsData;
- незрозуміло, чи необхідно оновлювати обидва місця, чи достатньо лише одного.

Ці дві проблеми відносяться до єдиного класу – можлива неконсистентність одних й тих же даних в двох різних місцях. Redux такої проблеми не має ніколи через те, що у цій архітектурі реалізовано підхід єдиного джерела правди (single source of truth), що означає що в будь який момент часу дані у одному місці системи є завжди актуальними, і тільки їм можна довіряти. В даному випадку єдине джерело правди – об’єкт State, з якого вже формуються Props для усіх екранів. Виглядає він наступним чином:

```
data class AvailableArtistsState(
    val byId: MutableMap<String, Artist>,
    val appliedFilter: ArtistFilter,
    val genresSelection: MutableMap<String, Boolean>,
) {
data class Artist(
        val id: String,
        val fullName: String,
        val email: String,
        val ratingInfo: RatingInfo,
        val profileDescription: String,
        val genres: List<String>
    ) {
        data class RatingInfo(
            val averageRating: Double,
            val numberOfRatings: Int,
        )
```

Як можна бачити, об’єкт State зберігає інформацію про музикантів у хеш таблиці за ключом id. Кожен раз, коли екрану необхідно отримати якусь інформацію, відповідний об’єкт Connector проводить необхідні перетворення, не змінюючи State – лише створюючи нові Props. Таким чином, стан застосунку ніяк не змінюється від деталей відображення, він проектується універсальним чином для того, щоб з будь-якого компоненту можна було отримати доступ до найбільш актуальних даних.

За допомогою такого механізму ніколи не має проблеми неактуальних даних, що вирішує багато проблем одразу – наприклад, якщо один екран редагує певну сутність, а інший її відображає – у MVVM виникає проблема синхронизації, необхідно певним чином передавати результат редагування для того, щоб оновити дані на попередньому екрані. У Redux такої проблеми бути не може – кожен екран отримує доступ до одного джерела даних, таким чином усі компоненти завжди мають актуальний стан застосунку.

5.4 Порівняння швидкодії

Швидкодія двох архітектур було оцінено двомам різними способами – шляхом профайлінгу застосунку використовуючи вбудовані в Android Studio інструменти та заміром часу виконання юніт тестів типічних сценаріїв використання застосунку.

Під час профайлінгу виконувався певний сценарій використання застосунку, який включає в себе наступні дії: авторизуватися, відкрити список замовлень, відкрити список музикантів, виконати фільтрацію даних, зробити замовлення та вийти з застосунку. Слід відмітити, що сценарії виконувалися в однаковому середовищі та без запущених інших застосунків (для більш точних результатів). Результати порівняння наведено у таблиці 5.6.

Таблиця 5.6 – Результати профайлінгу

Показник / Архітектура	Середнє навантаження CPU (тільки застосунок)	Середній показник використаної RAM
Redux	4.8%	154.0 mb
MVVM	5%	152.4 mb

Як можна бачити з таблиці, виміряні показники відрізняються в рамках похибки. Слід відмітити, що такий метод заміру не є найбільш точним, але він

підходить для того, щоб зрозуміти загальну тенденцію – якщо в одному випадку показники відрізняються дуже сильно, то можна проводити дослідження методами з більш точними результатами. В випадку порівняння MVVM та Redux показники майже ідентичні, що робить використання більш точних методів неактуальним.

На рисунку 5.2 графік навантаження CPU під час використання застосунком архітектури MVVM – спочатку йде навантаження майже у 50% під час запуску, далі навантаження становить максимум 20% під час навігації між екранами.

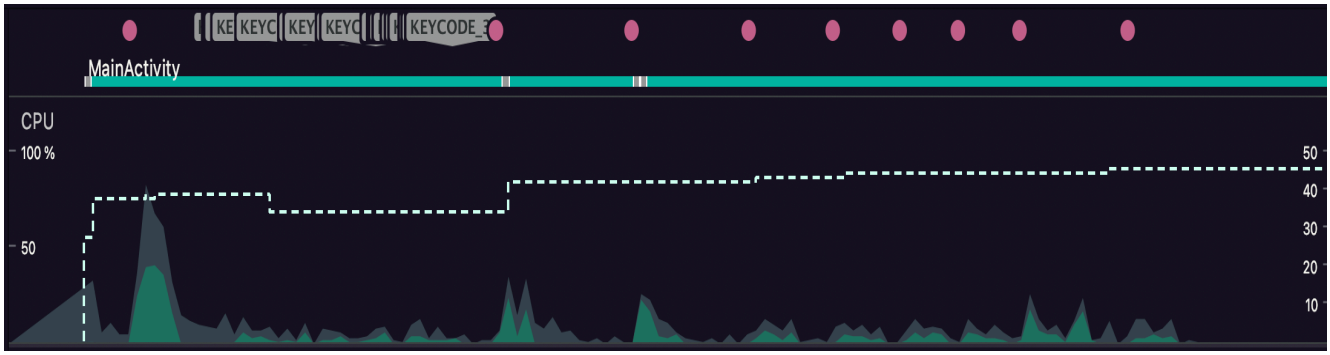

Рисунок 5.2 – Завантаження CPU (MVVM архітектура)

Заміри для архітектури Redux зображено на рисунку 5.3. Результати майже не відрізняються від MVVM

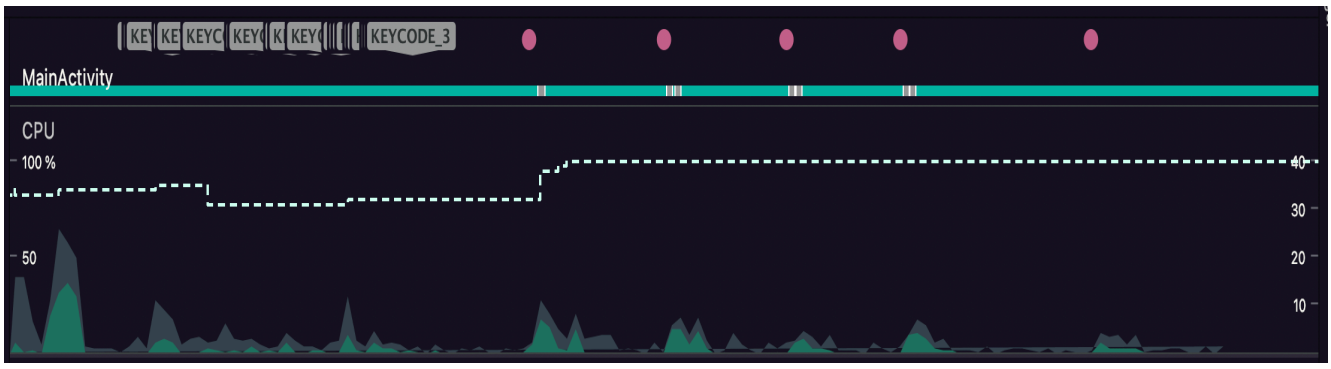

Рисунок 5.3 – Завантаження CPU (Redux архітектура)

Також швидкодія архітектур оцінювалася методом написання юніт тестів, які описують основні сценарії використання – наприклад, введення користувачем своєї електронної адреси та паролю та авторизація у системі.

Слід відмітити, що усі сторонні фактори – такі як запити до сервера в даному замірі було замінено синхроними операціями з фіксованим часом виконання, що виключає час на встановлення HTTP зв’язку та відправку запита та робить порівняння більш актуальним. Також було виключено час на початкове налаштування архітектури – вимірювалися лише показники для конкретного функціоналу. Для ще більш точних даних, заміри проводилися 1000 разів, та вибиралося середнє значення, що робить похибку менш значною.

У результаті замірів було отримано наступні показники – для MVVM час виконання сценарію авторизації становить 9 мс, а для Redux 10 мс. Таким чином, можна зробити висновок, що швидкодія двох архітектур майже не відрізняється між собою, та майже не має різниці для клієнтського застосунку – набагато більша частина часу витрачається на відмальовування користувацького інтерфейсу.

5.5 Аналіз результатів дослідження

Перед тим, як аналізувати результати дослідження та робити висновки, слід відмітити, що в даному випадку не може бути однозначного результату виду “Архітектура А є кращою за В” через те, що результати завжди залежать від того, у якому проекті ця архітектура буде використовуються. Деякі архітектури краще підходять для простих застосунків, деякі – для застосунків з великою кількістю модулей та розробників, а певні специфічні мобільні застосунки взагалі потребують архітектурних рішень, які вирішують конкретні проблеми, які виникають тільки у цих застосунках.

Під час проведення дослідження на предмет порівняння швидкодії двох архітектур можна зробити висновок, що практичної різниці у показниках швидкодії між Redux та MVVM немає. Такі результати є очікуваними через те, що архітектура в першу чергу впливає на показники розширюваності, здатності коду до підтримки та тестування, а також на час додавання та модифікації функціоналу.

Після проведення дослідження та порівняння двох архітектур на предмет показників розширюваності, здатності до підтримки та тестування можна зробити висновок, що архітектура Redux є рекомендованою до використання відносно MVVM у певних випадках. Як можна бачити з показників, які вимірювалися під час експерименту, реалізація Redux займає у середньому на 20 відсотків більше часу і рядків коду, але в той же час має набагато кращий показник розділення відповідальностей – кожна сутність має лише одну ціль та механізм використання, в той час як в MVVM об’єкт ViewModel відповідає майже за все. Також слід відмітити показники розширюваності коду – модифікація існуючого функціоналу з використанням Redux є дуже простою, в той час як аналогічні дії з використанням MVVM займають більше часу, та збільшують подальшу складність модифікації за рахунок додавання нових об’єктів LiveData та неочевидних зав’язків між ними. Таким чином, модифікація коду на Redux є більш швидкою та стійкою до випадкових помилок.

Підсумувавши отримані показники, отримаємо наступні переваги MVVM відносно Redux:

- швидкість початкової реалізації;
- менша кількість рядків коду;
- менша кількість створених класів / сутностей;
- більш інтуїтивна майже для усіх Android розробників.

В той час як Redux має такі переваги:

- розподіл відповідальностей – кожен клас / сутність відповідає за одну чітко поставлену задачу;
- більш проста модифікація коду – займає менше часу, потенціально викликає менше помилок;
- краща контрольованість – немає зав’язки на класи фреймворку для написання бізнес логіки;
- простота тестування – не має необхідності використовувати сторонні залежності.

Таким чином, можна зробити висновки, що проекти, у якіх рекомендовано Redux відносно MVVM – це середні та великі проекти (4+ розробники та активна розробка обсягом від півроку), у яких більшу роль грає простота розширення та підртимки проекту, ніж час на реалізацію прототипу. Такі проекти характеризуються великим обсягом часу підтримки та модифікації – ці сроки досягають декількох років, що робить показники розширюваності, підтримки та можливості тестування критичними.

Для невеликих проектів та стартапів навпаки – Redux не є рекомендованою архітектурою через більший час первинної реалізації та необхідність в навчанні розробників роботі з цією архітектурою. Такі проекти в більшості мають визначну метрику TTM (time to market), що робить критичним час на реалізацію, в той час як метрики розширюваності, можливості підтримки та тестування часто відходять на другий план.

5.6 Проблеми та рекомендації до використання

Під час реалізації типового застосунку з використанням Redux було сформовано декілька рекомендації щодо найбільш оптимального використання даної архітектури.

Однією з перших проблем, з якою можна зіткнутися – оновлення даних компонентів користувацького інтерфейсу, яке вводить застосунок у безкінечний цикл оновлення. Трапляється так тому, що Android SDK та його компоненти не пристосовані до односпрямованого підходу з єдиним джерелом правди (state) – кожен з компонентів має свій внутрішній стан (наприклад, компонент для введення тексту має внутрішній стан тексту), що викликає проблеми – в Redux об’єкт стану контролюється окремо. Таким чином, оновлення стану в Redux викликає оновлення внутрішнього стану компонента, яке в свою чергу знов оновлює стан Redux. Позбутися такої проблеми можна за допомогою написання функцій розширення

(extension function), які при оновленні компонента будуть робити перевірку на те, чи співпадає внутрішній стан компонента з станом Redux, і якщо так – не викликати оновлення, щоб позбутися введення у цикл.

Ще один сценарій використання, який потребує зміни типового підходу до розробки – робота зі списками. У застосунку з типовою архітектурою список оновлюється лише раз – коли завантажується з певного джерела даних, або ж кілька разів – наприклад, якщо користувач щось змінює у списку. У Redux ситуація відрізняється – кожна зміна стану застосунку викликає оновлення користувацького інтерфейсу, що в свою чергу викликає оновлення списку (навіть якщо змінився не список). Для вирішення цієї проблеми існує спеціальний клас ListAdapter та DiffCallback, які дозволяють проводити оновлення списків за допомогою розрахунку різниці списків з використанням алгоритму Майерса, що вирішує проблему постійного оновлення – якщо немає різниці між двома списками, то й оновлювати нічого не треба. Для роботи цього механізму у Redux необхідно знати про один неочевидний нюанс – здебільшого елементи списку представляються у вигляді класів з полями, для яких equals та hashCode генерується автоматично, але це може не завжди працювати, тому що у обєктах Props часто є об’єкти Command, які уявляють з себе лямбда вирази для взаємодії користувацького інтерфейсу з системою (наприклад, клік користувача на елемент списку). Якщо у об’єкті є такі поля Command – механізм автоматичного розрахунку різності двох об’єктів буде працювати некоректно (один лямбда вираз завжди не є рівним іншому через те, що це посилання, а не примітив), в зв’язку з цим необхідно реалізувати окремий клас, який буде порівнювати два об’єкти, незважаючи на усі об’єкти типу Command для коректної роботи алгоритму розрахунку різності.

Також однією з речей, про які треба пам’ятати при розробці на Redux – це те, що користувацький інтерфейс буде оновлюватися кожен раз, як оновлюється об’єкт стану застосунку, що може викликати проблем з швидкодією застосунку. Для того, щоб уникнути таких проблем, можна реалізувати механізм, який буде порівнювати, чи змінився певний екран, і не викликати створення нового об’єкту Props, якщо не потрібно. У реалізованій бібліотеці така оптимізація присутня

частково – якщо користувач не бачить екран, для нього не створюється новий об’єкт Props.

5.7 Перспективи та можливості для покращення

Як вже було відмічено раніше, реалізований модуль Redux є повністю функціональним і може бути використаний у будь-якому Android застосунку, але існує багато можливостей для подальшого розвитку та розширення цього модуля. Серед таких можливостей виділимо наступні:

- логування подій у сторонні сервіси та аналітику – наприклад, використовуючи технічний стек ELK;
- реалізація механізму Guards;
- інтеграція з Jetpack Compose;
- механізм Time Travel;
- реалізація DSL.

Розглянемо кожну можливість детальніше. Найбільш корисною та актуальною є інтеграція архітектури з сторонніми сервісами аналітики та логування – так, наприклад, за допомогою технологічного стеку ELK можна збирати усі події, які трапляються у системі, що надає можливість їх аналізувати та спрощує процес пошуку помилок у кінцевих користувачів.

Ще одним механізмом пошуку помилок є реалізація об’єктів спостерігачів (Guards), які отримують інформацію про усі зміни у системи і можуть аналізувати стан на предмет неконсистентності. Такі об’єкти можуть бути корисними та знаходити невідповідність у декількох станах – наприклад, якщо в нас є стан деталей певного користувача – там може міститися ідентифікатор користувача у системі. Об’єкт Guard може спостерігати за тим, щоб у кожен момент часу для обраного ідентифікатора була інформація у стані застосунку – таким чином, неможливо обрати ідентифікатор користувача, якого немає у системі, що виконує

роль аналогічну юніт тестам, але працює не окремо від застосунку, а у кожен момент часу, коли користувач використовує застосунок.

Інтеграція з Jetpack Compose є дуже важливою частиною для адаптації модуля під сучасні реалії розробки. На момент реалізації модуля, більшість Android застосунків все ще використовують механізми View та XML для реалізації користувацького інтерфейсу, але декларативний механізм Jetpack Compose набуває все більшої популярності, що робить інтеграцію однією з найбільш пріоритетних задач.

Механізм Time Travel – одна з можливостей, яка виникає через те, що усі події у системі централізовані. Така реалізація надає можливість зберігати усі події та стани застосунку за певний час, і в будь-який момент замінити стан застосунку іншим станом. Таким чином, можна реалізувати механізм інтерактивної відладки застосунку, що значно спрощує пошук помилок.

Останньою можливістю для покращення модуля можна відмітити оптимізацію та спрощення інтерфейсу взаємодії з Redux. Однім із способів це зробити є реалізація DSL для первинної конфігурації модуля.

ВИСНОВКИ

В ході виконання магістерської кваліфікаційної роботи було обрано та проаналізовано сучасні тенденції у проектуванні архітектур для мобільних застосунків. В результаті аналізу було знайдено проблемні місця існуючих рішень, та сформовано вимоги до архітектурного модуля, який може вирішити певні проблеми найбільш популярних на даний момент рішень.

В результаті проектування та розробки було створено програмний модуль, який реалізує архітектурний підхід з односпрямованим потоком даних та є аналогом популярного у Web програмуванні модуля Redux. Розроблений модуль було завантажено у відкритий доступ до репозиторію JitPack, що надає можливість використовувати його у будь-якому мобільному застосунку на базі ОС Android та системи збірки Gradle.

Для аналізу результатів проектування та розробки було обрано певні критерії та метрики, за якими проводилося порівняння розробленого модуля Redux з найбільш розповсюдженою та рекомендованою компанією Google архітектурою MVVM. У результаті порівняння було зроблено висновок про можливості використання Redux та рекомендації до того, як обрати архітектуру для різних видів проектів. За результатами аналізу Redux має сенс використовувати у проектах, в яких ключову роль грають показники розширюваності, підтримки та можливості тестування коду, а не швидкості початкової розробки. Результати роботи було опубліковано у науково-технічній конференції «Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління» [12].

Також було означено перспективи та можливості для подальшої розробки розробленого модуля – існує велика кількість можливих інтеграцій, покращень та змін у структурі, які можуть зробити програмний модуль ще більш підходящим для мобільної розробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1) Kotlin Documentation: веб-сайт. URL: https://kotlinlang.org/docs/home.html
2) Redux.js Architecture Documentation: веб-сайт. URL: https://redux.js.org/
3) Jetpack Compose (Declarative User Interface Toolkit): Documentation : веб-сайт. URL: https://developer.android.com/jetpack/compose
4) Unidirectional data flow (client side architectural pattern): веб-сайт. URL: https://en.wikipedia.org/wiki/Unidirectional_Data_Flow_(computer_science)
5) MVC (Framework) Architectural Pattern Introduction : веб-сайт. URL: https://tutorialspoint.com/mvc_framework/mvc_framework_introduction.htm
6) MVP Pattern: веб-сайт. URL: https://startandroid.ru/ru/blog/493-mvp-dlja-nachinajuschih-bez-bibliotek-i-interfejsov.html
7) Moxy Community (MVP library) Documentation: веб-сайт. URL: https://github.com/moxy-community/Moxy
8) MVVM Architectural Pattern (Android ViewModel Overview): веб-сайт. URL: https://developer.android.com/topic/libraries/architecture/viewmodel
9) Guide to app architecture (Android recommended): веб-сайт. URL: https://developer.android.com/jetpack/guide#recommended-app-arch
10) Model-View-Update-Communicate: Session Types meet the Elm : веб-сайт. URL: https://arxiv.org/abs/1910.11108
11) Investigation of Architecture and Technology Stack for e-Archive System // 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 2019, pp. 229-235, doi: 10.1109/PICST47496.2019.9061407.
12) Каук В.І., Губар С.О., Дослідження можливостей застосування односпрямованого підходу до проєктування мобільних архітектур на прикладі Redux // Сучасні напрями розвитку інформаційно-комунікаційних технологій та засобів управління. - Квітень 2022.