НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
__________ Сергій СТІРЕНКО
(підпис)
“__” _________2023 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Бібліотека для побудови інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android

Виконав : студент __4__ курсу, групи ІП-93
(шифр групи)

_______________Стронов Іван Сергійович_____________________________
(прізвище, ім’я, по батькові) (підпис)

Керівник _________асистент, Пономаренко А. М._____________________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) доцент, к.т.н. Волокита А. М. __________
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ________________доцент, к.т.н Коган А. В. ____________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент ___________
(підпис)

Київ – 2023 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
__________ Сергій СТІРЕНКО
(підпис)
“__” __________2023 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Стронова Івана Сергійовича

1. Тема проєкту *Бібліотека для побудови інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android*
керівник проєкту Пономаренко А. М., асистент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 31 травня 2023 року №2101-с

2. Термін здачі студентом закінченого проєкту *13 червня 2023 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Реактивне програмування в контексті Android-розробки.

Розділ 2. Огляд та порівняння технологій для розробки бібліотеки.

Розділ 3. Деталі розробки бібліотеки.

5. Перелік графічного матеріалу (з точним позначенням обов’язкових креслень) схема розширень бібліотеки (функціональна схема), алгоритм бібліотеки (принципова схема), структурна схема бібліотеки.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання «30» серпня 2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*20.02.2023-25.02.2023*	
2.	*Вивчення та аналіз завдання*	*26.02.2023-16.04.2023*	
3.	*Розробка архітектури та загальної структури системи*	*17.04.2023-23.04.2023*	
4.	*Розробка структур окремих підсистем*	*24.04.2023-30.04.2023*	
5.	*Програмна реалізація системи*	*01.05.2023-07.05.2023*	
6.	*Оформлення пояснювальної записки*	*08.05.2023-20.05.2023*	
7.	*Захист програмного продукту*	*21.05.2023*	
8.	*Передзахист*	*07.06.2023*	
9.	*Захист*	*20.06.2023*	

Студент-дипломник ______________________ Іван СТРОНОВ
(підпис)

Керівник проєкту ______________________ Артем ПОНОМАРЕНКО
(підпис)

АНОТАЦІЯ

В бакалаврському дипломному проєкті розглянуто історію розвитку та основні принципи реактивного програмування та реактивних потоків даних як його частину. Досліджено методи використання парадигми реактивного програмування в контексті платформи Android. Проаналізовано різні бібліотеки реактивного програмування, які використовуються у Android-розробці. На основі аналізу обрано бібліотеку та розроблено систему для побудови інтерфейсу користувача з її допомогою.

Система (бібліотека) дозволяє інтегрувати реактивні потоки даних у рівень інтефейсу користувача додатку на платформі Android. Бібліотека створена на мові програмування Kotlin у середовищі Android Studio.

ABSTRACT

The bachelor diploma project considered the history of development and the basic principles of reactive programming and reactive data flows as part of it. Methods of using the reactive programming paradigm in the context of the Android platform have been studied. Various reactive programming libraries used in Android development were analyzed. Based on the analysis, a library was selected and a system for building a user interface was developed.

The system allows user to integrate reactive data flows into the user interface level of the application on the Android platform. The library was written in the Kotlin programming language in the Android Studio environment.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпляра	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Бібліотека для побудови інтерфейсу	3		
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Бібліотека для побудови інтерфейсу	70		
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Бібліотека для побудови інтерфейсу	1		
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Схема розширень бібліотеки			
			(функціональна схема)			
	А4	***ІАЛЦ.4672008.005 Д2***	Бібліотека для побудови інтерфейсу	6		
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Алгоритм бібліотеки			
			(принципова схема)			
	А4	***ІАЛЦ.4672008.006 Д3***	Бібліотека для побудови інтерфейсу			
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Структурна схема бібліотеки			
	А4	***ІАЛЦ.4672008.007 Д4***	Бібліотека для побудови інтерфейсу			
			користувача із застосуванням			
			реактивних потоків даних			
			на платформі Android			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	***ІАЛЦ.467200.001 ОА***		
Розроб		Стронов І. С			*Бібліотека для побудови інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android*	Літ.	Аркуш	Аркушів
Перев		Пономаренко А.М.					1	1
					Опис альбому	***КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93***		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Бібліотека для побудови інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android*»

Київ – 2023

ЗМІСТ

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 2

2. ПРИЧИНИ ДЛЯ РОЗРОБКИ 2

3. ЦІЛЬ ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 2

4. ДЖЕРЕЛА РОЗРОБКИ 2

5. ТЕХНІЧНІ ВИМОГИ 3

5.1 Вимоги до розробленого продукту 3

5.2 Вимоги до програмного забезпечення 3

5.3 Вимоги до апаратної частини 3

6. ЕТАПИ РОЗРОБКИ 3

					ІАЛЦ.467200.002 ТЗ			
		№ докум.	Підпис	Дата				
Розробив		Стронов І. С.			Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android	Літ.	Аркуш	Аркушів
Перевірив		Пономаренко А. М.					1	3
						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93		
Н. Контр.		Волокита А. М.						
Затвердив					Технічне завдання			

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку бібліотеки для побудови інтерфесу користувача із використанням реактивних потоків даних на платформі Android, а також підтримку та вдосконалення цієї бібліотеки.

Область застосування: альтернатива бібліотекам для проектування інтерфейсу користувача, які використовуються для розробки застосунків на платформі Android.

2. ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврського проєкту по освітньо-професійної програми “Інженерія програмного забезпечення комп’ютерних систем” спеціальності 121 “Комп’ютерна інженерія”, затверджене кафедрою Обчислювальної техніки Національного технічного Університету України “Київський Політехнічний інститут імені Ігоря Сікорського”.

3. ЦІЛЬ ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою даної роботи є розробка бібліотеки для побудови інтерфейсу користувача із використанням реактивних потоків даних, яка дозволить розробникам ефективніше створювати додатки на платформі Android.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література по реактивному програмуванню, офіційні документації бібліотек, публікації в Інтернеті з даних питань.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до продукту, що розробляється

- Простота у використанні з мінімальною кількістю залежностей, які має використовувати кінцевий користувач
- Покриття максимальної кількості елементів інтерфейсу користувача розширеннями для використання бібліотеки.
- Використання реактивних потоків даних для заміни методів зворотного виклику у всіх можливих випадках.
- Вичерпна документація до функцій, які запроваджує бібліотека

5.2. Вимоги до програмного забезпечення

- Операційна система Windows, Linux або MacOS
- Android Studio версії 3.6 або вище або Android плагін для Intellij IDEA версії 3.6 або вище

5.3. Вимоги до апаратної частини

- Центральний процесор не гірше Intel® Core (TM) i5-2100T.
- Вільний простір жорсткого диску не менше 32 ГБ
- Оперативної пам'яті не менше 4 ГБ.

6. ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	20.02.2023-25.02.2023
Вивчення та аналіз завдання	26.02.2023-16.04.2023
Розробка архітектури та загальної структури системи	17.04.2023-23.04.2023
Розробка структур окремих частин системи	24.04.2023-30.04.2023
Програмна реалізація системи	01.05.2023-07.05.2023
Виправлення помилок	07.05.2023-19.05.2023
Оформлення пояснювальної записки	20.05.2023

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Бібліотека для побудови інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android*»

Київ – 2023

ЗМІСТ

ВСТУП 4

РОЗДІЛ 1. РЕАКТИВНЕ ПРОГРАМУВАННЯ В КОНТЕКСТІ ANDROID РОЗРОБКИ 5

1.1 Передумови виникнення парадигми реактивного програмування 5

1.2 Передумови приходу методів реактивного програмування до Android-розробки 7

1.3 Впровадження реактивних підходів в Android розробці 10

1.4 Новітня реалізація реактивних шаблонів проектування в рамках мови програмування Kotlin 14

ВИСНОВОК ДО РОЗДІЛУ 1 20

РОЗДІЛ 2. ОГЛЯД ТА ПОРІВНЯННЯ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ БІБЛІОТЕКИ 22

2.1 Вибір мови програмування 22

2.1.1 Мова програмування Java 22

2.1.2 Мова програмування Kotlin 24

2.1.3 Переваги Java над Kotlin 25

2.1.4 Переваги Kotlin над Java 26

2.1.5 Результати порівняння мов програмування 29

2.2 Вибір реактивної бібліотеки 29

2.2.1 Реактивна бібліотека RXJava (RXKotlin) 29

2.2.2 Реактивні компоненти модуля kotlinx.coroutines 31

2.2.3 Порвіняння бібліотеки RXKotlin з реактивними компонентами Kotlin Coroutines + Flow 32

2.2.4 Результати порівняння та вибір бібліотеки 37

ВИСНОВОК ДО РОЗДІЛУ 2 38

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ БІБЛІОТЕКИ 40
3.1 Аналіз поставлених вимог 40
3.2 Загальна структура бібліотеки 43
3.3 Огляд розширюваних компонентів 46
3.4 Розробка компонентів бібліотеки 48
3.4.1 Структура розширень 48
3.4.2 Розширення класу View 50
3.4.3 Розширення класу TextView 54
3.4.4 Розширення класів CheckBox та SwitchCompat 55
3.4.5 Розширення класу RadioGroup 56
3.4.6 Розширення класу Spinner 57
3.4.7 Розширення класу SeekBar 58
3.4.8 Розширення класу ViewPager2 59
3.4.9 Розширення класу TabLayout 60
3.4.10 Розширення класу RecyclerView 61
3.5 Сценарій використання бібліотеки 62
ВИСНОВОК ДО РОЗДІЛУ 3 64
ВИСНОВКИ 65
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 67

ВСТУП

Під час вирішення великої кількості задач в області інженерії програмного забезпечення розробники звертаються до певних парадигм програмування, які можуть надати вказівки з приводу того, які техніки застосовувати та якими принципами керуватися в процесі розробки. Однією з таких парадигм є парадигма реактивного програмування.

В контексті розробки програмного забезпечення на платформі Android парадигмою реактивного програмування почали керуватися відносно нещодавно. Реактивне програмування прийшло в Android розробку в 2013 році з заснуванням проекту ReactiveX для мови програмування Java. До цього в розробці під дану платформу домінували імперативна та об'єктно-орієнтовна парадигми, які властиві попереднім програмам, написаним на мові Java.

З 2017 року при розробці під Android розробники почали також використовувати нову мову програмування Kotlin. Ця мова привнесла до процесу розробки й інші парадигми, такі як функціональне програмування, а також і власну реалізацію парадигми реактивного програмування, незалежну від ReactiveX.

Оскільки парадигма реактивного програмування є відносно новою в контексті розробки програмного забезпечення під платформу Android, розробники досі створюють нові підходи для вирішення проблем, специфічних для процесу розробки саме під цю платформу. Досі є велика кількість проблем, які потребують рішення із застосуванням принципів та імплементацією підходів реактивного програмування на платформі Android, а велика кількість підходів та відсутність єдиної системи рекомендацій ускладнює розуміння процесу для нових розробників. Однією з проблем, які необхідно вирішити для запровадження реактивної парадигми програмування в нові проекти є Memory leaking з втратою об’єкта контексту.

Ця проблема притаманна для і старих Android проєктів, написаних до приходу реактивної парадигми програмування в процес розробки, але в

контексті реактивного програмування частина логіки та принципи, що відповідають за керування та контроль об'єкта контексту, майже не змінилися. Однак нові реактивні підходи потребують нових рішень, які здатні покращити якість написаного коду та полегшити процес розробки для сучасних розробників.

Оскільки проблема із втратою контексту найчастіше виникає у випадках роботи над створенням інтерфейсу користувача у додатках на платформі Android, дана робота ставить перед собою мету запровадження зручного та безпечного інтерфейсу для розробників, використовуючи який вони зможуть уникнути даної проблеми в цілому.

РОЗДІЛ 1. РЕАКТИВНЕ ПРОГРАМУВАННЯ В КОНТЕКСТІ ANDROID РОЗРОБКИ

1.1 Передумови виникнення парадигми реактивного програмування

Реактивне програмування — це парадигма програмування, яка зосереджена на обробці та реагуванні на зміни в даних і подіях реактивним способом. Вона характеризується використанням асинхронних потоків даних, методів функціонального програмування та декларативного стилем написання коду.

Історію реактивного програмування можна простежити до 1980-х років, коли були розроблені функціональні мови програмування, такі як Lisp і Scheme. Ці мови представили концепцію функцій вищого порядку, тобто функцій, які можуть приймати функції в якості аргументів і повертати функції як результати. Ця концепція заклала основу для парадигми реактивного програмування, оскільки вона дозволила полегшити обробку подій і даних та зробити цей процес більш функціональним.

Першою реалізацією реактивного програмування була парадигма функціонального реактивного програмування (FRP), розроблена Коналом Еллиоттом і Полом Худаком наприкінці 1990-х років. Вона дозволила розробникам описувати та проектувати поведінку систем у вигляді моделі безперервних потоків даних [1]. Цей підхід особливо корисний для додатків, які мають користувацький інтерфейс, оскільки FRP дозволяє розробникам створювати програми, які реагують на дії користувача програми у реальному часі.

У той же час в середі розробників з’являється концепція реактивної системи. Реактивна система – це система, яка здатна в реальному часі реагувати на події та зміни у внутрішньому або зовнішньому середовищі

виконання програми. Реактивний маніфест, опублікований у 2014 році, визначив основні характеристики реактивних систем [2]:

- Чуйність: система має своєчасно реагувати на події, що надсилаються користувачем
- Стійкість: система не має втрачати можливість відповідати навіть в разі помилки
- Гнучкість: система не має втрачати можливість відповідати в разі перенавантаження
- Керованість повідомленнями: система має використовувати асинхронну передачу повідомлень для взаємодії з користувачем

У 1990-х роках також був розроблений шаблон проектування Спостерігач (Observer), який реалізовував сповіщення про зміни внутрішнього стану об’єктів до інших об’єктів. Цей шаблон широко використовувався в програмуванні графічного інтерфейсу користувача (GUI), оскільки він дозволяв розробникам створювати адаптивні та інтерактивні інтерфейси користувача.

На початку 2000-х років корпорація Майкрософт представила концепцію реактивних розширень (RX, Reactive Extensions), яка систематизувала стандарти та структуру для складання асинхронних програм і програм заснованих на подіях із використанням реактивних потоків даних [3]. Rx базувався на принципах функціонального програмування і дозволяв розробникам писати реактивний код у стислий і декларативний спосіб.

Популярність реактивного програмування зросла наприкінці 2000-х і на початку 2010-х років, коли розробка стала більш складною та інтерактивною. Було розроблено такі фреймворки, як AngularJS і React, які використовували концепції реактивного програмування, щоб допомогти розробникам створювати адаптивні та динамічні веб-додатки.

1.2 Передумови приходу методів реактивного програмування до Android-розробки

Однією з проблем, з якою часто стикаються розробники на платформі Android є асинхронна обробка даних. У випадках, коли треба завантажити деякі дані з мережі Інтернет або з внутрішньої бази даних користувацького додатку, ця робота має виконуватися асинхронно для того, щоб не блокувати інтерфейс користувача. Оскільки мобільний додаток має реагувати на дії користувача постійно, ці задачі виконуються у задньому фоні, поки користувач бачить інтерфейс із повідомленням про завантаження. Користувач повинен мати можливість перервати завантаження даних у будь-який момент, тому програма має слідкувати та виконувати операції з менеджменту задачі, що виконується так контролювати потік її виконання.

До приходу реактивної парадигми програмування у Android розробку для цих задач використовувалися стурктури даних та шаблони проектування, створені із розрахунком на об'єктно-орієнтовну парадигму та імперативний стиль написання коду. До таких структур можна віднести AsyncTask, CompletableFuture та інші [4].

Подібний підхід, як і більшість підходів у асинхронному програмуванні заснований на ідеї методів зворотнього виклику. Метод зворотнього виклику – це функція, не викликається одразу, а передається як аргумент у іншу функцію для виклику по завершенні деякої операції чи після виконання деякої роботи. У мові програмування Java функції не є повноцінними об’єктами, тому передача їх як аргументів не є можливим. Для обходу цього обмеження була створена концепція функціональних інтерфейсів – інтерфейсів з єдиним або декількома методами, об’єкти яких можна створити на місці і передати у функцію в якості аргумента [5]. Хоча строго кажучи такий об’єкт не можна називати функцією зворотного виклику, функціонально він виконує ту ж роль, тож у контексті асинхронних операцій можна вважати їх еквівалентними.

Окрім функцій зворотного виклику структури подібні до AsyncTask, які використовувалися у мові Java підтримували також виконання операцій у

різних потоках. Базовий потік, на якому виконуються всі операції у додатках написаних на платформі Android – це потік Main (основний потік). Операції, які потребують доступу до файлової системи, а також бази даних чи доступ до даних через мережу Інтернет, зазвичай виконуються у потоці IO (Input/Output – Вводу/виводу) [6]. Це пов'язано з тим, що цей потік оптимізований для подібного роду операцій, оскільки він покладається на більшу степінь паралелізму, тобто розділяє загальний об'єм роботи на більшу кількість простих (атомарних) операцій, які можуть виконуватися одночасно на ядрі процесора мобільного пристрою.

Виконання різних типів задач у різних потоках, оптимізованих спеціально для цих конкретних задач, дозволяє покращити продуктивність та швидкість виконання цих задач, особливо на пристроях з обмеженою кількістю обчислювальних ресурсів. Мобільні пристрою відомі своєю нижчою продуктивністю у порівнянні зі стаціонарними девайсами. Ця проблема особливо гостро стояла на початку 2000-х, коли мобільні пристрої ще не були достатньо розповсюдженими та достатньо потужними. Більшість пристроїв на платформі Android мали не більше 2-х, максимум 4-х ядер. Через ці особливості платформи обмеження, які накладалися на розробників програмного забезпечення, які працювали на ній, змушували їх використовувати ресурси, які надає апаратне забезпечення пристрою через операційну систему, дуже відповідально та максимально ефективно. Проблема з розпаралелюванням асинхронних операцій на різні потоки стояла дуже гостро. І хоча на сьогоднішній день мобільні пристрої н платформі Android стали значно потужнішими, і ця проблема відійшла на задній план, вона досі є частково актуальною для додатків, які потребують великої кількості обчислювальних ресурсів.

Основним слабким місцем об'єктно-орієнтовних підходів, реалізованих у стандартних бібліотеках мови програмування Java, є відсутність гнучкого та зручного контролю за потоком даних, отриманих із локальної бази даних чи з мережі Інтернет. Отримані дані часто необхідно трансформувати там

змінювати відповідно до вимог розроблюваного застосунку, підвантажувати дані з інших джерел та збирати їх в єдину модель даних. Також на абстракціях типу AsyncTask у Java дуже складно [7], а в деяких випадках і взагалі неможливо, побудувати саме потік даних, який здатен розраховувати не лише єдиний результат виконання запиту до бази даних чи до мережі Інтернет, але й реагувати на зміни в цих даних, щоб оновлювати їх і після того, як був отриманий перший результат виконання через метод зворотнього виклику, переданий заздалегідь.

Ця проблема стає більш актуальною із врахуванням архітектури мобільного додатку на платформі Android. Хоча не існує єдиного стандарту архітектури додатків, і деякі з них можуть вирішувати дану проблему своїм дизайном, проте при проектуванні системи за стандартними рекомендаціями вона все ж грала значну роль у підвищенні складності розробки. Більшість моделей архітектури тим чи іншим чином включають у себе розділення програми на деяку кількість рівнів. Домінуюча архітектурна модель для розробки мобільних додатків під платформу Android у той час була модель MVC. Модель складалась із таких компонентів [8]:

- Модель: Включає в себе основну бізнес-логіку програми, яка не залежить від конкретної платформи реалізації системи.
- Представлення: Інтерфейс програми, який генерує події на основі взаємодії з користувачем.
- Controller: Зв’язує модель та представлення, відповідає за комунікацію між ними та потоком даних у застосунку.

Часто кожен із рівнів архітектури мав свою унікальну модель даних, яка використовувалася в середині цього рівня. В результаті при взаємодії між рівнями необхідно було перетворювати кожну модель даних на еквівалентну модель з відповідного рівня архітектури. Без зручних методів для перетворення моделей у асинхронних операціях як у єдиному потоці даних є доволі складною задачею, яка значно сповільнює процес розробки, перетягуючи на себе увагу розробників програмного забезпечення.

1.3 Впровадження реактивних підходів в Android розробці. Бібліотека RXJava

Для систематичного вирішення проблем написання процесу виконання асинхронних операцій Android-розробники почали використовувати бібліотеки, побудовані на основі реактивної парадигми програмування.

Парадигма реактивного програмування вперше з'явилася як інструмент для розробки додатків на платформі Android з появою бібліотеки RxJava. Ця бібліотека була вперше випущена у 2013 році як частина проекту Reactive Extensions.

Однією з головних особливостей бібліотеки була реалізація шаблонів спостерігач (Observer) та джерело (Observable), які забезпечували кращий контроль за потоком даних, ніж традиційні компоненти стандартної бібліотеки Java, які використовувалися для роботи з асинхронними операціями. Також бібліотека реалізувала механізми та інструменти для менеджменту протитиску (backpressure), характерних для будь-якої реалізації реактивних потоків даних за допомогою цих шаблонів проектування [9].

Для контролю та перетворення потоків даних у застосунках на платформі Android бібліотека впроваджувала реалізацію великої кількості операторів, які надавали розробникам трансформувати та комбінувати потоки даних між собою. Це вирішило значну кількість традиційних проблем та задач, які раніше стояли перед кожним Android-розробником. До одних із найчастіше використовуваних операторів відносяться:

- map – трансформує потік даних у інший потік даних, до кожного елемента якого застосовано переданий метод
- flatMap – трансформує потік даних, поєднуючи кожен його елемент із іншим потоком даних, який повертається з переданого метода
- groupBy – трансформує потік даних, згруповуючи елементи за певним предикатом, який повертається з переданого метода
- buffer – трансформує потік даних, поєднуючи елементи у групи з заданою кількістю в кожній

- filter – фільтрує потік даних за певною умовою, змушуючи його повертати лише елементи, які її задовольняють
- take – обмежує загальну кількість елементів, які може повертати потік даних заданою кількістю

Бібліотека значно покращила процес розробки для самих розробників програмного забезпечення під платформу Android, зробивши управління потоком даних більш зручним [9]. У перші роки з впровадження бібліотеки Android платформа в основному покладалася на традиційну архітектуру - MVC, яка характеризувалася центральним контролером, який керує зв’язком між представленням і моделлю. Однак у міру того, як мобільні пристрої стали потужнішими, розробники почали розуміти, що ця архітектура не підходить для створення швидко реагуючих і ефективних програм.

Це призвело до появи альтернативних архітектурних шаблонів, таких як Model-View-Presenter (MVP) і Model-View-ViewModel (MVVM), які забезпечили кращий розподіл проблем та контроль за потоком даних у мобільному додатку і покращили тестувальні можливості розроблюваних систем. Ці шаблони також полегшили впровадження парадигми реактивного програмування, яке набирало популярності в спільноті мобільних-розробників.

Із зростанням популярності реактивного програмування Google почав включати підтримку реактивного програмування в платформу Android та стандартну бібліотеку компонентів, яка використовується розробниками на платформі. У 2017 році Google випустив модуль Android Architecture Components [10], який включав в себе структури LiveData та ViewModel, два компоненти, розроблені для роботи з реактивним програмуванням. Ці компоненти дозволили розробникам створювати реактивні інтерфейси користувача, які автоматично оновлювалися у відповідь на зміни в даних, якими керується застосунок.

LiveData є по суті покращеною реалізацією шаблону Observable, до того реалізованого лише у бібліотеці RxJava для розробників на платформі Android. Деякі аспекти реалізації шаблону з цієї бібліотеки були налаштовані специфічно для робооти з Android застосуваннями, що зменшило загальну гнучкість та збільшило зв'язність компонента, але у той самий час полегшило його використання у рамках вирішення традиційних задач, які стояли перед розробниками.

Одним з недоліків застосування бібліотеки RxJava було те, що ця бібліотека оригінально не була створена для платформи Android. Через це її компоненти були повністю абстраговані від компонентів Android-застосунків і не мали уявлення про їх життєвий цикл та особливості взаємодії та передачі даних у відношенні до елементів інтерфейсу користувача.

Життєвий цикл більшості компонентів застосування у системі Android грає значну роль в роботі програми та її можливостями взаємодіяти з кінцевим користувачем. Для більшості компонентів він складається щонайменше з наступних методів зворотнього виклику, що викликаються системою при певних станах компонента [11]:

- onCreate – викликається після створення компонента в пам'яті, коли система готується до відображення створеного компонента на екрані мобільного пристрою
- onStart – викликається, коли елемент повністю ініціалізовано та він готовий бути відображеним на екрані пристрою
- onResume – викликається, коли компонента починає бути активним та користувач може взаємодіяти з ним
- onPause – викликається, коли компонент все ще видимий на екрані пристрою, але користувач не взаємодіє з ним активно
- onStop – викликається, коли компонент перестає бути видимим користувачеві на екрані мобільного пристрою

- onDestroy – викликається, коли компонент готовий бути знищеним системою для вивільнення пам’яті, коли користувач вже не має змоги взаємодіяти з ним

Стандартні реалізації реактивних компонентів з бібліотеки RxJava не мали змоги взаємодіяти з компонентами Android-застосунку із врахуванням їх життєвого циклу. Через це перед розробниками постійно стояла задача контроль та менеджменту за станом виконання асинхронних операцій через реактивні потоки даних. Тобто ‘підписування’ та ‘відписування’ від відповідні потоків даних розробники виконували в ручному режимі у методах зворотного виклику, які надає стандартна бібліотека Android.

Саме цю проблему вирішувала покращена реалізація шаблону Observable – LiveData. Цей компонент був прив’язаний до життєвого циклу компонентів застосунку на платформі Android. У його функціоналі було закладено механізм автоматичного припинення виконання асинхронних операцій у випадках, коли поточний стан життєвого циклу компонента не задовольняє вимог, вказаних розробником як необхідних для виконання цих операцій. Тобто у випадках, коли кінцевий користувач переходить з одного екрану додатку на інший, але не всі асинхронні операції на попередньому екрані ще завершені і деякі компоненти досі очікують на дані з реактивних потоків, реалізація LiveData здатна автоматично скасувати виконання операції [12] як для того, щоб не витрачати ресурси системи на виконання задач, які припинили бути актуальними для користувача, так і для того, щоб уникнути можливих витоків пам’яті через те, що компоненти досі очікують на результат виконання досі не завершеної операції.

ViewModel – ще один компонент, введений до середи Android-розробки в першу чергу для полегшення реалізації парадигми реактивного програмування на платформі. Цей компонент є одним з ключових у архітектурному шаблоні MVVM, який досі домінує і у сучасній Android-розробці.

Основна відмінність ViewModel від аналогічних до неї компонентів із альтернативних шаблонів проектування (Presenter та Controller) полягає в

тому, що цей компонент не зв’язаний із представленням для його модифікації при зміні внутрішнього стану чи даних, необхідних для відображення. Цей підхід зменшує залежність компонентів одне від одного в системі та допомагає краще розділити залежності для кожного рівня архітектури. Замість прямої модифікації стану представлення ViewModel реалізує більш реактивну взаємодію з ним, адже згідно з принципами реактивного програмування статичний стан має бути зведений до мінімуму. ViewModel виставляє для представлення лише реактивні потоки даних у вигляді компонентів LiveData, які отримуються з моделі та перетворюються у відповідні моделі представлення, зрозумілі для його методів відображення даних.

Компоненти представлення має змогу ‘підписатися’ на отримані потоки даних для виконання певних асинхронних операцій та отримання даних з моделі. А нативність компоненту LiveData забезпечує автоматичний контроль за процесом виконання, прив’язаний до життєвого циклу елементів інтерфейсу користувача.

1.4 Новітня реалізація реактивних шаблонів проектування у рамках мови програмування Kotlin

На бібліотеці RXJava та впровадженні компонентів, таких як LiveData та ViewModel, розвиток реактивної парадигми в Android-розробці не зупинився. У 2018 році Google випустив бібліотеку Android Jetpack, яка підтримувала програмування у реактивному стилі через корутини (співпрограми) мови програмування Kotlin [13]. У цей же час Google почав рекомендувати Kotlin як більш пріоритетну мову для розробниців на цій платформі.

Kotlin — це статично типізована кросплатформна мова програмування, розроблена JetBrains, тією ж компанією, яка розробляє популярне інтегроване середовище розробки IntelliJ IDEA (IDE) і також найбільш популярну на сьогоднішній день середу розробки під платформу Android – Android Studio. Його вперше було випущено в 2011 році, і він швидко завоював популярність серед розробників завдяки своїм функціональним можливостям, простоті

вивчення та використання для нових розробників та обширною стандартною бібліотекою.

Однією з головних переваг Kotlin над Java є його стислий синтаксис. Синтаксис Kotlin розроблений так, щоб бути більш інтуїтивно зрозумілим і виразним, що полегшує розробникам читання, написання та підтримку коду. Наприклад, система неявного визначення типів Kotlin дозволяє розробникам у багатьох випадках пропускати оголошення типів, зменшуючи необхідний обсяг шаблонного коду [14].

Kotlin також надає низку функцій, недоступних у Java, таких як функції розширення, нульовий захист і корутини (співпрограми). Функції розширення дозволяють розробникам додавати новий функціонал до існуючих класів, не змінюючи їхній вихідний код, тоді як безпека нульового вказівника допомагає запобігти виняткам нульового покажчика під час компіляції, які були одним з найбільш найрозповсюджених типів виключень у програмах, написаних за допомогою мови Java.

Окрім мовних функцій, Kotlin також забезпечує повну взаємозамінність коду з Java, що полегшує розробникам використання Kotlin разом із існуючими кодовими базами Java. Це дозволяє розробникам поступово переносити свій код на Kotlin, не переписуючи все з нуля. Саме ця особливість мови забезпечила Kotlin особливу увагу зі сторони спільноти розробників програмного забезпечення під платформу Android. Кодова база застосунків на цій платформі була значно більшою за подібні застосунки написані на мові Java для інших платформ, оскільки особливості операційної системи вимагали більшої кількості шаблонного коду для коректної та продуктивної взаємодії з компонентами апаратного забезпечення мобільного пристрою. Можливість інтегрувати код, написаний на мові Kotlin, напряму у кодову базу на мові Java дозволила розробникам, які підтримують більш старі додатки, плавно переписувати програму з менш ефективного об’єктно-орієнтовного та імперативного стилю на більш сучасний – реактивний.

Для виконання асинхронних операцій у той час Kotlin пропонував розробникам використовувати корутини. Корутини є потужною функцією паралелізму, яка дозволяє розробникам писати асинхронний і неблокуючий код більш природним і зрозумілим способом. Корутини можна використовувати для обробки тривалих завдань, таких як мережеві запити або операції введення/виведення файлів, не блокуючи основний потік.

На відміну від всіх попередніх реалізацій асинхронного програмування на платформі, таких як AsyncTask і більш сучасних модифікацій RxJava, корутини у мові програмування Kotlin побудовані не на концепції методів зворотнього виклику, а на концепції suspending функцій (функції, що здатні призупинятися).

Suspending функції в мові програмування Kotlin – це функція з особливим прапорцем, який індикує те, що функція може перервати своє виконання у певній точці з подальшою можливістю відновити і продовжити своє виконання без втрати стану. Переривання функції відбувається для того, щоб звільнити обмежені ресурси системи для більш пріоритетних операцій або для шнших suspending функцій. Грубо кажучи в таких випадках система виступає в ролі планувальника потоків, розподіляючи час, виділений для певних компонентів застосунку, на виконання заданих розробником асинхронних операцій між всіма компонентами, які його потребують. Самі корутини в мові програмування Kotlin позиціонуються розробниками як «легковісні потоки» [15]. Але не має великого сенсу напряму порівнювати реалізацію корутин у Kotlin з реалізацією потоків в Java, оскільки корутини також підтримують розширений функціонал за контролем потоку виконання асинхронних операцій і загалом були створені для інших цілей.

Suspending функції побудовані за допомогою моделі кінцевого автомату. Компілятор автоматично генерує весь код, необхідний для роботи подібного роду функцій, тому розробники можуть не поглиблюватися в деталі реалізації, і писати код у більш декларативному стилі. На основі кожної suspending функції компілятор генерує кінцевий автомат, де кожна точка призупинки

(точки у послідовності операцій виконання функції, у якій вона здатна зберегти свій стан та передати контроль до іншої частини програми) відповідає одному із станів кінцевого автомата. По завершенні виконання частини заданих розробником асинхронних операції функція зберігає свій стан у об'єкт інтерфейсу Continuation (продовження). Цей об'єкт інкапсулює в собі поточний стан виконання у вигляді простого лічильника і дані, передані до функції або створені у самій функції у попередніх періодах виконання інших асинхронних операцій [16]. Інтерфейс Continuation містить метод resume, який викликається системою в момент, коли у неї з'явилися вільні ресурси для того, щоб можна було продовжити виконання асинхронної операції. В залежності від даних об'єкта Continuation згенерована функція відновлює свій стан на момент призупинки, перевіряє поточний стан виконання за допомогою лічильника і продовжує виконання наступної асинхронної операції чи повертає розраховане значення.

Корутини мови програмування Kotlin дозволяють розробникам писати блокуючий код, який складається з задач, які потрібно виконувати на задньому фоні від обробки взаємодії користувача з додатком, у стилі неблокуючого коду з чіткою послідовністю дій та отримання результату. Бібліотека Android Jetpack підштовхувала розробників до використання нової моделі корутин, яку пропонувала мова програмування Kotlin, включивши їх підтримку в тому числі в до стандартних компонентів, які були введені Google для підтримки реактивної парадигми програмування серед спільноти Android розробників.

Так за допомогою функцій-розширень Kotlin були створені розширення таких класів як ViewModel, LifecycleOwner та інших, які впроваджували стандартний інтерфейс для взаємодії з корутинами та запуску асинхронних операцій із методів, знайомих звичайному розробнику під мобільну платформу Android.

Незважаючи на порівняно більшу зручність та функціональність корутин мови програмування Kotlin в порівнянні як з традиційними методами використання та виконання асинхронних операцій, побудованих на методах

зворотнього виклику, так і більш сучасних підходах, основаних на реактивній парадигмі програмування, які пропонують реактивні бібліотеки, такі як RXJava, деяка частина розробників все ж сумнівалося, що нові підходи мови Kotlin зможуть повністю замінити функціонал реактивних бібліотек. Частиною проблеми вважалося те, що корутини все ще не підтримували саме реалізацію реактивних потоків даних імплементацією стандартного шаблону реактивного програмування Observable. Через це деякий час в середі Android розробки досі домінували реактивні підходи бібліотеки RXJava, яку також потрували до мови програмування Kotlin як частину реалізації проекту ReactiveX, або комбіновані підходи, які поєднували реактивні потоки RXJava із корутинами з Kotlin.

Однак ця ситуація змінилася з впровадженням реалізаціїї реактивних потоків даних як частину стандартної бібліотеки Kotlin через реактивну модель Flow (потік).

Kotlin Flow — це реалізація архітектурної моделі реактивного потоку даних, яка дозволяє розробникам писати асинхронний неблокуючий код у більш природний та інтуїтивно зрозумілий спосіб. Вона створена на основі корутин Kotlin, які забезпечують потужну та легку модель паралелізму для обробки асинхронних операцій.

На високому рівні Kotlin Flow надає набір операторів і функцій для створення та керування потоками даних [17]. Як і інші реалізації реактивного потоку, такі як RxJava, Kotlin Flow використовує шаблон «Observable/Observer» для керування потоком даних між різними компонента розроблюваного застосування.

Після введення Kotlin Flow як одного з альтернативних варіантів реалізації реактивних потоків даних при розробці на платформі Android розробники отримали ще один інструмент для обробки та менеджменту виконання асинхронних подій. Більш того, цей інструмент впроваджував деякі переваги над стандартними компонентами, реалізованими у бібліотеці RXJava. Також Kotlin Flow є нативним інструментом для мови Kotlin з

обширної бібліотекою підтримуваного функціоналу, що ще більше прискорило перехід розробників з мови програмування Java до її більш сучасного аналогу.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було розглянуто історію розвитку парадигми реактивного програмування в контексті розробки програмного забезпечення під платформу Android. Першими варіантами реалізації шаблонів проектування цієї парадигми почали використовуватися розробниками у контексті бібліотеки RXJava, оригінально створеної не специфічно для платформи Android, а для більш узагальнених систем, написаних на мові програмування Java.

Реактивні підходи застосовувалися у програмуванні під дану платформу в першу чергу для вирішення проблеми виконання асинхронних задач на задньому плані від взаємодії користувача з інтерфейсом програми. Шаблони реактивного програмування допомогли розробникам створювати більш стійкі та зручні для використання системи з написанням меншої кількості кодової бази.

Проте перші реактивні бібліотеки на платформі Android мали значний недолік, оскільки їх не було конфігуровано для зв’язування з життєвим циклом компонентів застосування, що вимушувало розробників писати додатковий шаблонний код для деталізованим контролем за процесом виконання асинхронних операцій.

Для вирішення цього роду проблем Google випустив нову біблітеку, яка включала в себе такі компоненти, як LiveData та ViewModel. Ці компоненти підтримували шаблони та принципи реактивног програмування, дозволяючи розробникам писати код в реактивному стилі. Також ці компоненти були прив'язані до платформи Android, тому вони значно покращили контроль роробників над виконанням асинхронного коду, зв’язуючись із традиційними компонентами платформи Android.

Після приходу мови програмування Kotlin до спільноти Android-розробників з'явилась необхідність в розробці нових бібліотек для підтримки

парадигми реактивного програмування в рамках цієї мови. Бібліотека Android Jetpack від Google створила рекомендовану на поточний час реалізацію стандартних шаблонів реактивного програмування із застосуванням новітніх функціональних можливостей мови програмування Kotlin. До бібліотеки входила реалізація корутин для роботи з асинхронними операціями, а згодом до бібліотеки було додано покращену реалізацію шаблону Observer через модель Flow.

РОЗДІЛ 2. ОГЛЯД ТА ПОРІВНЯННЯ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ БІБЛІОТЕКИ

2.1 Вибір мови програмування

2.1.1 Мова програмування Java

Java — це мова програмування загального призначення, яка була вперше випущена компанією Sun Microsystems у 1995 році. Вона була розроблена для написання коду, який є простим, безпечним, портативним і незалежним від платформи, а також зосередженим на принципах об’єктно-орієнтованого програмування та синтаксисі, який легко читати та писати.

Історію Java можна віднести до початку 1990-х років, коли команда інженерів Sun Microsystems під керівництвом Джеймса Гослінга вирішила створити мову, яку можна було б використовувати для написання програмного забезпечення для різних пристроїв та платформ, включаючи вбудовані системи, персональні комп’ютери, мобільні пристрої та сервери. Команда почала з розробки мови під назвою Oak, яка використовувалася для створення програмного забезпечення для проекту приставки. Під час роботи над цією мовою програмування, команда розробників зрозуміла, що цю мову можна використовувати для значно ширшого спектру програм і платформ, і почала додавати до неї нові функції та можливості [19].

У 1995 році Sun Microsystems випустила першу загальнодоступну версію Java, яка включала віртуальну машину (JVM), яка могла виконувати байт-код Java на будь-якій платформі, роблячи його максимально незалежним від платформи виконання. Java швидко набула популярності серед розробників, особливо завдяки своїй кросс-платформенності та легкості, з якою її можна було використовувати для створення графічних інтерфейсів користувача. У 1996 році компанія Sun Microsystems випустила Java Development Kit (JDK),

який включав набір інструментів і бібліотек для розробки програм на мові Java.

Сьогодні Java є однією з найпоширеніших мов програмування у світі, на якій написано безліч різноманітних додатків від корпоративного програмного забезпечення до мобільних програм і відеоігор [19]. Її популярність значною мірою пояснюється простотою використання, гнучкістю та великою екосистемою інструментів, бібліотек і фреймворків, які були побудовані навколо неї.

В контексті розробки програмного забезпечення під платформу Android Java займала місце найбільш широко використовуваної мови протягом багатьох років і була основною мовою для розробки Android-застосунків з моменту появи платформи. Java стала основною мовою для розробки під Android через її простоту використання, стабільність і популярність серед розробників.

Java має кілька ключових характеристик, які зробили її привабливим варіантом для розробки на платформі Android. По-перше, це об’єктно-орієнтована мова, яка дозволяє розробникам легко створювати складні програми. Підтримка об’єктно-орієнтовної парадигми програмування робить цю мову придатною для створення великих масштабованих програм, які можна легко підтримувати та розширювати з часом. Окрім об’єктно-орієнтованих функцій, Java має низку бібліотек і фреймворків, спеціально розроблених для розробки застосунків під платформу Android. До них відноситься Android SDK, який надає набір інструментів і API для створення програм на цій платформі, а також бібліотеки написані сторонніми розробниками, такі як Retrofit, OkHttp і Gson, які можна використовувати для спрощення розробки типових задач, таких як виконання операцій чи отримку даних за допомогою мережі Інтернет, доступ до локальної бази даних на мобільному пристрої і серіалізація даних у формати Json або XML – найбільш популярні формати на сьогоднішній день.

Спільнота Android-розробників за роки існування Java на платформі створила величезну кількість ресурсів та документації, яку нові розробники можуть використовувати для вирішення стандартних задач в процесі розробки під дану платформу. Java існує вже понад 25 років і має величезну екосистему інструментів, бібліотек і фреймворків, які можна використовувати для розробки під Android.

Хоча на сьогоднішній день мова Java починає поступатися місцем більш сучасним аналогам, вона все ще залишається однією з найпопулярніших мов для розробки під Android платформу, оскільки вона пропонує стабільну, надійну та добре налагоджену систему для створення програм, а також має велику спільноту розробників і ресурси, які спрощують початок роботи та створення високоякісних застосунків.

2.1.2 Мова програмування Kotlin

Kotlin — мова програмування загального призначення, яка була вперше представлена в 2011 році компанією з розробки програмного забезпечення JetBrains. Під час розробки мови програмісти ставили перед собою задачу створити мову таку, щоб вона була лаконічною, безпечною і сумісною з Java, що в результаті зробило її привабливою альтернативою Java для багатьох розробників [21].

Однією з ключових особливостей мови Kotlin є її лаконічність. Код, написаний на Kotlin, часто коротший і більш виразний, ніж еквівалентний код, написаний на мові Java, завдяки таким можливостям, як неявне визначення типів, функції розширення та перевантаження операторів. Ці особливості можуть полегшити та пришвидшити написання та підтримку коду, особливо для складних проектів.

Ще однією важливою особливістю Kotlin є його безпека. Kotlin містить ряд інструментів, які допомагають запобігти поширеним помилкам програмування, таким як винятки нульового покажчика (NullPointerException), які є характерними для більшості програм, написаних

на Java. Ці інструменти включають в себе ненульові типи, які гарантують, що змінні не можуть бути нульовими, якщо вони явно не оголошені як нульові, і безпечні виклики, які дозволяють розробникам безпечно отримувати доступ до методів і властивостей об'єктів, які можуть бути нульовими.

Kotlin також є повністю сумісним з Java, тобто розробники можуть використовувати Kotlin разом із існуючим кодом Java в одному проекті. Це полегшує поступове впровадження Kotlin в проектах, без необхідності переписувати існуючий код або вивчати абсолютно нову мову.

Kotlin швидко набув популярності серед розробників після свого випуску, і багато розробників та організацій прийняли його як свою основну мову для нових проектів. Його використовували для створення широкого діапазону додатків, від мобільних додатків і веб-сервісів до мобільних додатків і серверних систем.

Окрім сумісності з Java, Kotlin містить низку функцій, спеціально розроблених для розробки Android. Серед них – першокласна підтримка розробки Android у формі розширень Android, які дозволяють розробникам легко отримувати доступ до представлень і ресурсів у своїх макетах, а також бібліотека Anko, яка надає набір інструментів і утиліт для створення програм Android.

У 2017 році Google оголосив про офіційну підтримку Kotlin на платформі Android, зробивши його офіційно підтримуваною мовою для розробки Android поряд з Java [22]. Цей крок допоміг ще більше підвищити популярність Kotlin серед розробників Android і призвів до зростання спільноти розробників і ресурсів, пов'язаних із цією мовою.

2.1.3 Переваги мови програмування Java над мовою Kotlin

Порівнюючи між собою мови програмування Java та Kotlin, можна виділити наступні переваги мови Java:

1. Більш широка спільнота розробників

Оскільки мова програмування Java існує значно більше часу за мову програмування Kotlin, а також використовується в більшій кількості сфер, спільнота розробників навколо мови Java є значно більшою ніж спільнота навколо мови Kotlin. Це означає, що у разі необхідності допомоги легше буде знайти підтримку, програмуючи на мові Java.

2. Більша кількість бібліотек та фреймворків

   Kotlin – відносно нова мова програмування, і тому з часу її створення було реалізовано не так багато бібліотек та фреймворків, які можуть допомогти розробникам в процесі програмування застосувань під платформу Android. Однак цей недолік частково нівелюється тим, що Kotlin є повністю сумісною з Java. Завдяки цій характеристиці значна частина бібліотек та фреймворків скоріше за все буде працювати коректно як з кодовою базою на Java, так і на Kotlin

3. Більша надійність у рамках платформи Android

   Мобільна платформа Android створювалася із задумкою про використання саме мови програмування Java для розробки застосувань на ній. У той час мова Kotlin ще не існувала. Таким чином Java є більш сумісною та заточеною під платформу Android, аніж мова програмування Kotlin

4. Менший час компіляції та збірки застосувань

   Оскільки мова Java використовує меншу кількість абстракцій, ніж Kotlin, вона компілюється швидше, що дозволяє розробникам скоріше перевіряти результат своєї роботи. Однак різниця в часі є доволі малою, тому не можна вважати цю перевагу значною

2.1.4 Переваги мови програмування Kotlin над Java

Мова програмування Kotlin також має низку значних переваг у порівнянні з мовою Java:

1. Простіший синтаксис.

В порівнянні з мовою програмування Java мова Kotlin має значно більш простий синтаксис. Це забезпечується тим, що в мові Kotlin прибрана значна кількість шаблонного коду, який необхідно було б писати при програмування на Java

2. Нульовий захист

Однією з найбільш розповсюджених помилок, з якими стикаються Java розробники є NullPointerException. Ця помилка виникає у разі спроби звернення до властивості об'єкта за нульовим показником. Kotlin вирішеє цю проблему двома способами. По-перше впровадженням ненульових типів, які не можуть мати нульове значення, якщо розробник не вказав це інакше. По-друге, безпечні виклики для нульових типів, які автоматично перевіряють чи є вказівник на об'єкт нульовим перед спробою звернутися до його полів.

3. Функції-розширення

Мова програмування Kotlin підтримує функції-розширення. Це функції, які здатні розширювати базовий функціонал класів без їх наслідування для написання коду у більш лаконічному стилі. Однак ці функції мають певні обмеження. Так, вони не можуть визначати нові поля, які не були визначені в розширюваному класі. Це зумовлено необхідністю дотримуватися принципів статичної типізації.

4. Data-класи

На відміну від Java, де розробникам необхідно писати шаблонний код для визначення get та set методів у кожному класі, ціль якого виступати моделлю представлення даних, у мові Kotlin можна автоматично згенерувати ці методи під час компіляції, позначивши необхідний клас ключовим словом data.

5. Перевантаження операторів

Мова Kotlin підтримує перевантаження операторів, тобто створення функцій, назвою яких фактично є один або декілька спеціальних символів та які можна застосовувати у вигляді предикатів.

6. Розумне приведення типів

   Після перевірки змінної деякого типу на те, чи є вона конкретною реалізацією якогось типу Kotlin буде вважати, що ця змінна має даний більш специфічний тип у обраному контексті. Через це при подальшому використання розробники можуть явно не приводити змінну до цього типу, як це необхідно робити в Java

7. Базові та іменовані значення параметрів

   В Java в разі виклику метода необхідно передавати усі параметри в порядку, заданому у сигнатурі визначення цього методу. Єдиним способом кастомізації порядку чи списку параметрів є перевантаження методів, що може призвести до створення дуже великої кількості методів з однаковою назвою, але різною сигнатурою. Kotlin вирішує цю проблему, дозволяючи надавати базові значення параметрам (в такому випадку вони не є обов’язковими для передачі в момент виклику метода), і також передавати параметри, іменуючи значення (в такому випадку можна передавати іменовані параметри до методу у довільному порядку)

8. Функції як повноцінні компоненти мови. Функції вищого порядку

   Функціональна парадигма програмування реалізована у мові програмування Kotlin у значно більшому обсязі, ніж у мові програмування Java. Таким чином Kotlin дозволяє писати більш ефективний код із застосуванням різних парадигм для конкретних випадків. До функціональних можливостей мови відносяться функції вищого порядку, які здатні приймати інші функції в якості аргументів чи повертати функції як свій результат. Хоча реалізувати аналогічний функціонал можна і у рамках мови програмування Java за допомогою функціональних інтерфейсів і лямбда-виразів, такі реалізації все ще є значно обмеженими в порівнянні з реалізаціями у мові Kotlin. Наприклад, Java не підтримує реалізацію повноцінного замикання для лямбда-виразів.

9. Обширна стандартна бібліотека операторів:

   Kotlin має значну кількість операторів для роботи, наприклад, зі списками прямо «з коробки», у той час як в мові Java розробники мають підключати сторонні бібліотеки для їх підтримки. Це прискладнює процес розробки для нових програмістів, які лише вивчають мову та додають більшу кількість шаблонного коду до програм, написаних на мові Java

10. Підтримка корутин (співпрограм)

    В модулі kotlinx.coroutines стандартної бібліотеки Kotlin реалізовано корутини, як більш ефективний інструмент написання неблокуючого коду для виконання асинхронних операції. В той же час Java не має підтримку співпрограм навіть при застосування сторонніх бібліотек.

2.1.5 Результати порівняння мов програмування

Порівнявши функціональні можливості обох найбільш перспективних мов програмування, які застосовуються в процесі Android-розробки, обираємо мову програмування Kotlin. Ця мова має значно більше переваг над мовою Java і є рекомендованою мовою для розробки під платформу Android в цілому. Також значно ширші функціональні можливості цієї мови забезпечують кращу інтеграції шаблонів проектування парадигми реактивного програмування, з допомогою яких буде реалізовано бібліотеку.

2.2 Вибір реактивної бібліотеки

2.2.1 Реактивна бібліотека RXJava

RxJava — це популярна бібліотека з відкритим вихідним кодом для реактивного програмування на мові Java, яка яка була популярною у перші роки впровадження реактивної парадигми програмування у середі Android-розробки. RxJava дозволяє розробникам писати асинхронний код, керований подіями, у більш стислий і виразний спосіб ніж попередні методи роботи з асинхронними задачами, використовуючи такі концепції реактивного

програмування, як джерела (Observables), спостерігачі (Observers) та оператори.

В основі RxJava лежать компоненти Observable та Observer [23]. Observable — це потік даних, який надсилає значення з часом, а Observer — це об’єкт, який слухає Observable і реагує на отримані значення. Observable-об’єкти можуть видавати будь-яку кількість значень, включаючи відсутність значення взагалі, одне або декілька значень. Вони також можуть видавати помилку або сигнал завершення, які є перериваючими сигналами. В таких випадках Observable-об’єкт зазвичай припиняє надсилати подальші значення.

RxJava містить широкий спектр операторів, які можна використовувати для перетворення, фільтрації та комбінування потоків даних з Observable-об’єктів різними способами. Ці оператори можна об’єднувати разом, щоб виконувати більш складні операції над реактивними потоками даних і значної мірою трансформувати отримані значення.

Додатково до реалізацій шаблонів Observable і Observer, RxJava також включає в себе планувальники (Schedulers). Планувальники використовуються для вказівки потоку, в якому слід спостерігати зміни у об’єкті Observable. Використання цих компонентів полегшує обробку асинхронних операцій, дозволяючи виконувати певні типи операцій на специфічних планувальниках, оптимізованих для виконання подібного роду задач.

RxJava також добре взаємодіє з іншими бібліотеками, написаними для мови програмування Java, і бібліотека має велику та активну спільноту розробників, які роблять внесок у її постійний розвиток і підтримку. RxJava активно використовується в розробці на мові Java під платформу Android. Її можна використовувати як для обробки асинхронних операцій, таких як мережеві виклики та запити до бази даних, так і для обробки введених користувачем даних, анімацій та інших операцій, керованих подіями, створення реактивних компонентів інтерфейсу користувача, які динамічно реагують на зміни стану моделі даних у застосунку.

Оскільки було обрано мову програмування Kotlin для розробки бібліотеки, то при порівнянні буде розглядатися портована версія RXJava у мову програмування Kotlin – RXKotlin. Вона не має значних функціональних відмінностей від RXJava, але дозволяє використовувати інструменти мови Kotlin для зменшення об'єму шаблонного коду, написаного при розробці мобільного застосунку.

2.2.2 Реактивні компоненти модуля kotlinx.coroutines. Kotlin Flow

Kotlin Flow — це реалізація реактивних потоків даних специфікації ReactiveX, представлена у версії мови програмування 1.3. Вона забезпечує простий і ефективний спосіб обробки асинхронних операцій і потоків даних у більш реактивний і функціональний спосіб.

Реалізація реактивних потоків даних існувала і у більш ранніх версіях мови програмування. Зокрема компонент Channel був типічною реалізацією шаблону Observable у контексті Kotlin. Однак Channel вважався частиною більш низкорівневого стилю написання коду з ручним контролем за процесом виконання асинхронних операцій і не підходив для розв'язання більшості традиційних задач, які виникають в процесі розробки [24]. В решті решт реалізація Kotlin Flow була побудована саме на Channel, розширивши його функціонал, додавши велику кількість операторів та підтримку корутин як вже існуючого інструмента для роботи з асинхронністю.

Kotlin Flow забезпечує елегантний спосіб обробки потоків даних, таких як мережеві запити, запити до бази даних або події інтерфейсу користувача відповідно до технік та методів реактивної на функціональної парадигм програмування. Kotlin Flow схожий на RxJava та інші реалізації реактивних потоків, але розроблений таким, щоб бути легшим і простішим у використанні.

Реалізація Kotlin Flow підтримує зворотний тиск [25], що означає, що компонент може обробляти потоки даних, які надходять швидше, ніж вони можуть споживатися. Це допомагає запобігти витокам пам'яті та іншим проблемам, які можуть виникнути під час роботи з великими об'ємами даних,

які часто змінюються. Також Kotlin Flow підтримує скасування, що означає, що обробку даних, які надсилає реактивний потік можна скасувати у будь-який момент його виконання. Це допомагає зменшити використання ресурсів і підвищити загальну продуктивність системи.

2.2.3 Порівняння бібліотеки RXKotlin з компонентами Kotlin Coroutines + Kotlin Flow

Порівняємо базовий функціонал бібліотек, який може використовуватися в традиційних задачах Android розробки

1. Моделі об'єктів

   У бібліотеці RXKotlin маємо наступні об'єкти:

   - *fun observeEventsA(): Observable<Data>* - 0-n подій без контролю зворотного тиску
   - *fun observeEventsB(): Flowable<Data>* - 0-n подій з контролем зворотного тиску
   - *fun observeEvent(): Single<Data>* - рівно 1 подія
   - *fun observeEvent(): Maybe<Data>* - 0-1 подій
   - *fun observeNone(): Completable* – просто виконання задачі без специфічного результату

   При використанні Kotlin Flow + Coroutines не має необхідності у такій великій кількості методів, адже можна комбінувати підходи Flow з асинхронним виконання через корутини:

   - *fun observeEvents(): Flow<Data>* - 0-n подій з автоматичним контролем зворотного тиску
   - *suspend fun observeEvent(): Data* – рівно 1 подія
   - *suspend fun observeEvent(): Data?* – 0-1 подій
   - *suspend fun observeNone()* – просто виконання задачі без специфічного результату

2. Використання операторів

Бібліотека RXKotlin впроваджує дуже велику кількість різноманітних операторів для трансформації реактивних потоків даних [26]. І більшість з цих

операторів також мають реалізації для кожної з об'єктних моделей. У порівнянні Kotlin Flow + Coroutines не має необхідності підтримувати таку велику кількість операторів.

Наприклад у випадках, коли при написанні коду через RXKotlin необхідно виконати операцію над єдиною подією: *Single.map { }*, Kotlin Flow + Coroutines здатна оброблювати дані напряму, використовуючи функціонал корутин та suspending функцій [27]. Таким чином ця бібліотека позбавляється від додаткового рівня вкладеності методів зворотнього виклику, роблячи структуру коду більш лаконічною та зрозумілою.

Також частину операторів, реалізованих у RXKotlin специфікація Kotlin Coroutines + Flow здатна замінити єдиним оператором. Наприклад комбіновані функціональні можливості операторів *delay(), delaySubscription(), startWith(value), startWith(observable), concatWith(observable)* з бібліотеки RXKotlin можна замінити єдиним оператором *onStart { ... }*.

Таким же чином замість ускладненої обробки помилок за допомогою операторів *onErrorResumeWith(), onErrorReturn(), onErrorComplete()* бібліотеки RXKotlin можна всі необхідні дії виконати в рамках єдиного оператора *catch { ... }*.

3. Розширення бібліотеки власними операторами

У бібліотеці RXKotlin всі оператори є частиною визначення основних компонентів бібліотеки з дуже складними можливостями розширення та додавання власних операторів. Для цього застосовуються інші оператори, такі як *Observable.compose()* та *Observable.lift()*.

В реалізації Kotlin Flow + Coroutines всі оператори визначені не як частини компонентів бібліотеки, а як методи-розширення до цих компонентів. Цей підхід дозволив залишити інтерфейси базових інструментів для створення реактивних потоків даних відносно малими та не загромаджувати їх деталями реалізації кожного оператора. Завдяки цьому створювати нові оператори, розширюючи базовий функціонал є простою задачею, яка не ставить перед

розробниками необхідність поринати у деталі імплементації моделей компонентів бібліотеки.

4. Протікання контексту з вхідного потоку даних до вихідного

В коді, побудованому на методах зворотнього виклику часто виникає ситуація, коли методи, передані для виклику у сусідніх операторах викликаються у різних потоках системи. Це пов'язано з тим, що передаючи метод зворотного виклику розробник не має контролю над тим, коли і де цей метод буде викликано [28]. Якщо реалізація системи викличе цей метод у певному потоці, то весь код, прописаний у цьому методі, буде виконуватися у контексті даного потоку.

В рамках розробки застосунків під мобільну платформу Android ця проблема є дуже критичною при роботі з користувацьким інтерфейсом. Справа в тому, що всі операції у рамках взаємодії з користувачем та стандартними компонентами представлення системи Android мають виконуватися виключно у основному (Main) потоці. У випадку порушення цього обмеження процес виконання застосунку одразу припиниться з виключення IllegalStateException. Особливо часто така проблема виникає при роботі з реактивними потоками даних для виконання асинхронних операцій типу звернення до бази даних чи до мережі Інтернет, які зазвичай виконуються у потоці IO. При поверненні контролю для роботи з представленням у операторах реактивного потоку даних розробник має проконтролювати те, що контекст виконання перейшов назад до потоку Main. Ця проблема називається протіканням контексту з вхідного потоку даних до вихідного.

Для вирішення даної проблеми у бібліотеці RXKotlin існують 2 оператори: *observeOn()* та *subscribeOn()*, які контролюють контексти відповідно вхідного та вихідного потоку даних з оператора. Для уникнення проблеми з протіканням контексту розробник, який використовує дану бібліотеку, має розставляти подібні оператори у кожному місці, де я неоднозначність можливого контексту виконання метода зворотного виклику, зазначеного в операторі.

У реалізації Kotlin Flow + Coroutines ця проблема не виникає взагалі через те, яким способом відбувається підписка на реактивний потік даних. Ця бібліотека має лише один оператор *flowOn()*, який визначає контекст, у якому буде виконуватися оператори вхідного потоку даних. Виконання ж вихідного потоку даних залежить лише від того, у якому контексті відбулася підписка на цей потік. Оскільки у даній імплементації реактивної парадигми неможливо підписатися і запустити виконання операторів потоку даних без контексту і необхідний контекст можна отримати лише зі стандартних компонентів представлення Android-застосунку, це гарантує, що контекст виконання операторів завжди в кінцевому потоці буде загорнутий у Main контекст для безпечної взаємодії з користувацьким інтерфейсом.

5. Структурована конкуренція

У RxKotlin при підписці на реактивні потоки даних метод підписки повертає об'єкт типу *Disposable*. Цей об'єкт є свого роду втіленням контракту між розроблюваною системою та потоком даних про виконання останнім деякого набору асинхронних операції, заданих в операторах. Він використовується для менеджменту та контролю за цим потоком виконання та надсиланню певних сигналів для спілкування з ним [29]. Розробник власноруч має подбати про утилізацію цього об'єкта, тобто скасування підписка, коли він перестає бути потрібним. Стандартною практикою вважається збереження посилання на цей об'єкт, або розміщення його в *CompositeDisposable*, щоб пізніше викликати для нього *dispose()*, коли він більше не потрібен. Якщо розробник цього не зробить, це створить витік пам'яті у програмі.

І хоча така модель поведінки є значно кращою за стандартні засоби мови програмування Java, такі як Thread, над яким розробник взагалі не має реального контролю після моменту запуску (*Thread.stop()* є застарілим, шкідливим, а остання реалізація фактично нічого не робить, а *Thread.interrupt()* змушує потік завершитися з виключенням), такий спосіб реалізації шаблону проектування все ще залишає величезне поле для ризику,

яке розробникам доводиться контролювати власноруч, пишучи додатковий код.

Kotlin Flow + Coroutines ж інверсує таку модель, відходячи від ідеї «запустити і забути» про асинхронну операцію. В рамках інструментів цієї бібліотеки неможливо запустити асинхронну операцію через виклик корутини без *CoroutineContext* [30]. Цей контекст визначає область виконання корутини, обмежуючи її доступ до обмежених ресурсів системи. Кожна дочірня корутина, створена всередині початкової, матиме однаковий контекст, таким чином надаючи можливість скасувати операцію та всі дочірні до неї операції одним викликом методу скасування на об'єкті контексту. Альтернативно можна і окремо зберігати посилання на кожну запущену корутину та скасовувати їх окремо.

6. Підтримка гарячих та холодних потоків даних

Реактивні потоки даних поділяють на 2 типи: гарячі та холодні в залежності від того, як вони контролюють процес виконання заданих у операторах задач в залежності від поточної кількості підписників на відповідний потік даних [31].

Холодні потоки даних схожі на звичайні функції в коді: вони існують як визначення і нічого не роблять, доки розробник їх не викличе (тобто доки не підпишеться на вихідний потік). Як і з функцією, яку викликано двічі і більше разів, при підписці на холодний потік декілька разів кожного разу буде створюватися новий об'єкт зі своїм незалежним процесом виконання асинхронних операцій, заданих у операторах. (Наприклад, холодний потік для виконання http-запиту виконає запит двічі, якщо на нього підписатися двічі).

Гарячі потоки працюють іншим чином. Декілька підписників на єдиний гарячий потік мають спільний процес виконання операцій. Тобто гарячий потік запускається на виконання один раз і може мати велику кількість підписників одночасно.

В бібліотеці RXKotlin гарячий потік даних можна створити з холодного потоку даних за допомогою оператора *publish()*. Після чого всі підписники на

цей потік будуть отримувати одні й ті самі дані з вихідного потоку з моменту створення підписки.

Для контролю за виконанням асинхронних операцій в залежності від кількості підписників використовуються наступні оператори:

- *refCount()* – гарячий потік починає виконання з моменту появи першого підписника і не зупиняться
- *connect()* – гарячий потік починає виконання з моменту створення і не зупиняється
- *autoConnect()* – гарячий потік починає виконання з моменту появи першого підписника і зупиняється при відсутності підписників

Бібліотека Kotlin Flow + Coroutines також реалізує функціонал гарячих та холодних реактивних потоків за допомогою операторів *shareIn()* та *stateIn()*. Для контролю за асинхронними операціями використовується параметр цих операторів *started*, який має наступні можливі значення:

- *SharingStarted.Lazily* – гарячий потік починає виконання з моменту появи першого підписника і не зупиняться
- *SharingStarted.Eagerly* – гарячий потік починає виконання з моменту створення і не зупиняється
- *SharingStarted.WhileSubscribed()* – гарячий потік починає виконання з моменту появи першого підписника і зупиняється при відсутності підписників

2.2.4 Результати порівняння та вибір бібліотеки

Проаналізувавши переваги та недоліки RXKotlin та Kotlin Flow + Coroutines, обираємо останню для реалізації розроблюваної бібліотеки. Основні причини вибору на таку користь – краща інтегрованість з мовою програмування Kotlin та зі стандартними компонентами системи Android. Це забезпечить простіший і більш лаконічний стиль написання коду для бібліотеки.

ВИСНОВОК ДО РОЗДІЛУ 2

У даному розділі було розглянуто інструменти, які будуть використовуватися для розробки бібліотеки для побудови інтерфейсу користувача з застосування реактивних потоків даних на платформі Android.

Було порівняно дві основні мови програмування, які використовуються для розробки застосунків на цій платформі: Java та Kotlin. Провівши порівняльний аналіз було визначено, що мова програмування Kotlin краще підходить до технічних вимог, які необхідно реалізувати у рамках розробки бібліотеки. Ключовими перевагами цієї мови над альтернативою в контексті даного проекту визначено:

- Пріоритетність мови програмування Kotlin у розробці під платформу Android, визначену спільнотою розробників та власниками платформи
- Більш обширний інструментарій функціональних можливостей та шаблонів реактивного програмування, реалізованих за допомогою них
- Лаконічніший синтаксис із значно меншою кількістю шаблонного коду, необхідного для написання в процесі вирішення поставленої задачі

Окрім мови програмування було обрано бібліотеку з реалізацією основних компонентів та принципів реактивної парадигми програмування, яка буде використовуватися при розробці бібліотеки. Було розглянуто 2 варіанти реалізації: RXJava (та порт цієї бібліотеки до мови програмування Kotlin – RXKotlin) та Kotlin Flow + Coroutines. Після порівняння бібліотек було обрано реалізацію Kotlin Flow + Coroutines за наступними причинами:

- Краща інтегрованість компонентів бібліотеки із обраною мовою програмування Kotlin, що дозволить писати код в більш лаконічному та стандартизованому до мови стилі
- Підтримка роботи з корутинами як дуже важливим компонентом роботи з асинхронними операціями при програмування на мові Kotlin
- Реалізація структурованої конкуренція на рівні компонентів бібліотеки – підходу, що дозволяє уникнути проблеми протікання контексту з вхідного потоку даних до вихідного
- Підтримка сучасних підходів, рекомендованих для розробки застосувань під платформу Android та інтеграція з компонентами бібліотеки Android Jetpack, такими як LiveData та ViewModel.

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ БІБЛІОТЕКИ

3.1 Аналіз поставлених вимог

У попередніх розділах було описано історію розвитку та основні проблеми використання реактивних потоків даних розробниками на мобільній платформі Android. Для вирішення цих проблем було створено певні рекомендації розробникам щодо загальної архітектури мобільних додатків та систем, що розроблюються під цю платформу.

У стандартних архітектурних задачах рекомендується використовувати UDF – Unidirectional Data Flow (односторонній потік даних). Згідно з цим підходом стан системи передається «вниз», тобто до рівня інтерфейсу користувача, а події, викликані взаємодією користувача з мобільним додатком – «вгору», тобто до рівня стану та бізнес-логіки системи. Цей підхід дозволяє представити весь спектр операції, які може виконати застосування, у вигляді простого потоку перетворень та дій, а односторонність потоку даних забезпечує захист від непередбачуваних факторів, які можуть вплинути на процес його обробки.

Стан у стандартній модель архітектури мобільного додатку представляється у вигляді класів даних та репозиторіїв, які з ними взаємодіють. На цьому рівні рекомендується використовувати Kotlin Flow для передачі та отримання даних з інших рівнів. Зазвичай робота репозиторіїв – забезпечити отримання та збереження даних з певних «кінцевих точок» (ендпоінтів), з якими взаємодіє програмний додаток. Для мобільного застосування на платформі Android це зазвичай внутрішня база даних та/або зовнішній API. Розглянемо ці випадки по черзі.

У випадку використання у застосунку бази даних, варіанти для розробників є дещо обмеженими. Платформа не підтримує нативну імплементації більшості сучасних СУБД, тому найчастіше використовується база даних на основі SQLite у вигляді одного файлу. Для роботи з базою даних

розробникам рекомендується використовувати бібліотеку Room, яка впроваджує моделі та інтерфейси для простого створення моделей даних та взаємодії з ними. Room – доволі стара бібліотека, написана ще на мові Java, але більш нові версії бібліотеки включають у себе підтримку як Kotlin Coroutines для одноразового отримання та роботи з даними, так і Kotlin Flow для постійного спостереження за змінами у даних. Саме останній випадок більше цікавить нас у плані проектування бібліотеки. Якщо застосування має підтримку бази даних на основі Room/Kotlin Flow, така база даних часто стає реалізацією шаблону SSoT (Single Source of Truth) у архітектурі системи, адже підтримка реактивних потоків даних дозволяє спостерігати за поточним станом навіть якщо додаток не має доступу до мережі Інтернет.

У випадку використання у застосунку серверних викликів розробники мають більший спектр бібліотек для вибору. Рекомендований варіант – бібліотека Retrofit, доволі стара, як і Room, написана ще на Java. Альтернативний до неї варіант – бібліотека Ktor, написана виключно для мови програмування Kotlin. Обидві бібліотеки наразі підтримують Kotlin Coroutines для запитів до серверу. Випадки використання реактивних потоків даних при комунікації клієнт-сервер є дещо обмеженими, адже зачасту сервер не може підтримувати броадкастинг даних для всієї маси потенційних користувачів, які можуть підключитися для взаємодії з ним. Проте цей функціонал все ж можна реалізувати на основі Веб-сокетів зі сторони серверу. В такому разі при відкритті з’єднання клієнт може активно прослуховувати події та дані з сервера, не роблячи при цьому активних запитів зі своєї сторони. Для використання веб-сокетів обидві бібліотеки також підтримують Kotlin Flow.

Отож у обох типічних випадках використання ендпоінтів в мобільному застосунку на платформі Android вже створені компоненти, побудовані на реактивних потоках даних (зокрема Kotlin Flow), і вони активно застосовуються сучасними розробниками. При реалізації рекомендованої моделі архітектури з використанням UDF репозиторії мають залежності від таких ендпоінтів, які надають інтерфейс для спілкування у вигляді suspend-

функцій та Flow для отримання даних. Отримуючи необхідні дані з декількох джерел репозиторій оброблює їх для кращого представлення у рамках вимог мобільного застосунку та виставляє з рівня стану архітектури інтерфейс у вигляді результуючого реактивних потік Flow з даними, готовими для відображення для користувача чи подальшої обробки наступним рівнем.

Наступний рекомендований рівень архітектури для стандартного мобільного додатку на платформі Android – рівень ViewModel. Цей компонент був описаний детальніше у попередніх розділах роботи. Стандартна бібліотека Andoid SDK реалізує компонент ViewModel, що дозволяє розробникам легко використовувати його в своїх застосунках. Також стандартна бібліотека Android реалізує ряд корисних розширень, які дозволяють інтегрувати цей компонент з іншими компонентами програми.

ViewModel має залежність від репозиторіїв рівня стану архітектури програми. Її робота – отримати актуальний потік даних та перетворити його у моделі даних, які здатен розуміти та відображати рівень представлення для користувача. Рекомендований спосіб досягти цієї мети – використання компоненту LiveData, або альтернативно знову ж таки Kotlin Flow для виставлення моделей до рівня представлення. При використанні Kotlin існує також певний набір розширень для компонента ViewModel спеціально створений для роботи з Kotlin Coroutines та Kotlin Flow. Одним з таких розширень є viewModelScope, який прив’язаний до життєвого циклу ViewModel та буде скасованим при знищенні компонента. За допомогою цього розширення можна зручніше контролювати отриманий з репозиторіїв потік даних, запускаючи окремі корутини для асинхронної обробки.

Останній, найнижчий рекомендований рівень архітектури – рівень представлення. Сюди входять стандартні компоненти системи Android для побудови користувацького інтерфейсу, такі як Activity, Fragment, а в їх реалізації – компоненти ієрархії View. Цей рівень – кінцева точка потоку даних, тому на цьому рівні відбувається підписка. Activity чи Fragment посилається на ViewModel та отримує потік даних, необхідних для

відображення користувачеві. Для підписки на Flow використовуються методи *collect, collectLatest* або *onEach/launchIn*. При цьому для вирішення проблеми, пов'язаної з життєвими циклами компонентів цього рівня архітектури також застосовуються методи *repeatOnLifecycle* та *flowWithLifecycle*. Для підписки на реактивні потоки даних, реалізовані за допомогою Kotlin Flow стандартна бібліотека Android SDK також надає розширення *lifecycleScope* для компнентів Activity та Fragment цього рівня. Для використання реактивниз потоків даних на більш низьких рівнях представлення View, необхідне більш ручне управління контекстами та скоупами корутин та Flow.

Однак саме тут з'являється проблема, адже цей рівень представлення не має сам по собі впроваджено ї реалізації підтримки реактивних потоків даних для обробки подій від користувача. Вся ця логіка поведінки застосування все ще будується на старих компонентах системи Android, написаних на мові Java, і тому використовує лише методи зворотного виклику для обробки цих подів. Методи зворотного виклику дуже погано взаємодіють з реактивними потоками даних, і тому поєднати їх в єдину структуру, як того вимагає модель архітектури UDF, практично неможливо.

Вирішення саме цієї проблеми взаємодії реактивних потоків даних з рівня ViewModel з методами зворотного виклику рівня представлення є першочерговою метою розробки цієї бібліотеки.

3.2 Загальна структура бібліотеки

Оскільки розроблювана система є бібліотекою, в основі структури проекту буде лежати стандартний бібліотечний модуль для Android-застосунків. Для побудови бібліотеки буде застосовуватися система збірки додатків Gradle. Ця система генерує файл під назвою build.gradle у кореневій директорії проекту, у якому вказуються залежності та різні конфігурації створюваного компонента.

Виставляємо значення параметрів compileSdk та targetSdk на 33, осклькі це є останньої версією операційної системи Android із підтримуваним SDK.

Значення параметра minSdk виставляємо як 21, адже це найбільш рання версія Android, з якою стабільно працюють корутини та реактивні потоки даних мови Kotlin. В опціях компіляції виставляємо значення параметрів sourceCompatibility і targetCompatibility як Java версії 1.8. Це наймолодша версія мови програмування Java, яка стабільно почала використовуватися для Android-розробки, і код, скомпілюваний з такими параметрами сумісності, все ще залишиться сумісним із кодом на мові Java цієї версії.

Серед залежностей бібліотеки додаємо наступні компоненти:

1) *'androidx.core:core-ktx:1.10.1'* – Базова підтримка програмування під платформу Android на мові Kotlin. Вона додає ряд компонентів, адаптерів та розширень, які дозволяють використовувати Kotlin при роботі з компонентами стандартної бібліотеки Android SDK, написаній на мові Java.
2) *'androidx.appcompat:appcompat:1.6.1'* – Розширення стандартної бібліотеки Android SDK, що включає в себе певні компоненти та міграції, які забезпечуються сумісність системи з більш старими версіями операційної системи Android.
3) *'com.google.android.material:material:1.9.0'* – Бібліотеки від Google, яка впроваджує ряд компонентів для побудови інтерфейсу користувача, які дуже часто використовуються розробниками на цій платформі. Частина з них буде використовуватися нарівні зі стандартними компонентами з Android SDK для розширення розроблюваною бібліотекою.

По завершенню написання файлу build.gradle, створюється базова структура проекту. Всі файли розроблюваної бібліотеки лежать у папці src на верхньому рівні модуля. Також на верхньому рівні модуля лежить папка libs. Вона використовується для додавання бібліотек у файйловому вигляді. Але оскільки ми використовуємо систему збірки проекту Gradle, то можемо повністю видалити цю папку. В цій папці знаходяться 3 підпапки:

- Test для юніт-тестів
- AndroidTest - для інструментальних тестів
- Main – для файлів програми

В папці main знаходиться файл маніфесту бібліотеки. Оскільки бібліотека не буде визначати жодного із головних акторів операційної системи Android (Activity, Service, BroadcastReceiver, ContentProvider), у файлі маніфесту можна не визначати жодного компоненту, тобто функціонально залишити його порожнім. Але він все ж таки має існувати, бо без нього система Android не зможе правильно ідентифікувати його залежності та вид діяльності.

Окрім цього на тому ж рівні знаходиться папка res для ресурсів бібліотечного модуля. Але оскільки розроблювана бібліотека не має власного інтерфейсу чи його компонентів, ця папка може бути видалена повністю з модуля.

В папці Java знаходяться кодові файли бібліотеки. Вони лежать у певній підструктурі пакетів та підпакетів мови програмування Java. Зазвичай ця структура складається з трьох слів:

- Домен ресурсу бібліотеки
- Ідентифікатор автора бібліотеки
- Назва бібліотеки

Однак насправді цю структуру пакетів можна іменувати як завгодно, адже система збірки Gradle не матиме ніяких обмень при однозначному розпізнаванні цієї бібліотеки. Для розроблюваної бібліотеки структура пакетів виглядає як *com.stronov.kavex*. Kavex – назва бібліотеки. Вона походить від скорочення абревіатури Kotlin Android View EX-tensions.

Найнижчий пакет цієї структури розглядається розробниками як кореневий файл кодової бази модуля, тобто це означає, що код буде знаходитися на цьому рівні, або рівняє ще нижче у випадках подальшого створення підпакетів у проекті.

Оскільки описана бібліотека буде писатися на мові програмування Kotlin, було вирішено застосовувати функціонал цієї мови для мінімізації

шаблонного коду, який довелося би писати у випадку програмування мовою Java. Одним з шаблонів мови Kotlin, який буде застосовуватися для програмування системи є функції-розширення.

Функції-розришення – це функції, які дозволяють розробникам розширювати функціональні можливості певного класу без застосування наслідування. Такий підхід більше відповідає функціональній парадигмі програмування, і, як наслідок, реактивній парадигмі програмування також. Розширюватися будуть компонентя для побудови інтерфейсу користувача, які надаються базовою бібліотекою Android SDK, а також компонентами з Google Material. Для створення реактивних потоків даних на основі взаємодії цих компонентів з користувачем мобільного застосування, базові класи цих компонентів будуть розширені методами для побудови реактивних потоків даних.

Таким чином вся бібліотека буде представляти із себе набір функцій-розришень, згрупованих у файлах відповідно до базових компонентів, які вони будуть розширювати. Іменування файлів буде відповідати назві базового класу розширюваного компоненту з суфіксом Ext (від англ. Extension – розширення). Наприклад, файл для функцій-розширень, які додають функціонал реактивних потоків даних для компонента TextView, буде називатися TextViewExt.kt. Аналогічним чином створюватимуться і файли для розширення інших компонентів.

3.3 Огляд розширюваних компонентів

Для доповнення елементи побудови інтерфейсу користувача функціоналом для взаємодії з реактивними потоками даних будуть розширюватися наступні компоненти:

1) *android.view.View* – Цей компонент є базовим компонентом усієї системи побудови інтерфейсу користувача, яку надає для розробників Android SDK. Від цього компонента наслідуються усі інші

компоненти, які мають можливість відображатися на екрані мобільного пристрою.

2) *android.widget.TextView* – Цей компонент представляє з себе базове текстове поле. В цьому полі може відображатися текст, написаний розробником або користувачем застосунку. Також від цього компонента наслідуються майже усі компоненти у інтерфейсі користувача, які використовуються для роботи з текстом
3) *android.widget.CheckBox* – Цей компонент представляє з себе квадрат, у якому може з'являтися галочка, коли користувач натискає на нього. Він має 2 стани, які змінюються між собою через взіємодію з користувачем.
4) *androidx.appcompat.widget.SwitchCompat* – Компонент перемикач. Аналогічно до CheckBox має 2 стани: ввімкнено та вимкнено. Перемикається коли користувач натискає на компонент.
5) *android.widget.RadioGroup* – Абстракція для групи кнопок, серед якої одночасно ввімкненою може бути лише одна. Саму базову кнопку цієї групи RadioButton розширювати не має потреби, адже майже весь стан цього компонента зберігається саме у RadioGroup.
6) *android.widget.Spinner* – Цей компонент представляє з себе випадаюче меню. При натисканні на базове вікно перед користувачем з'являється список опцій, серед яких можна обрати лише одну. Обрана опція потім відображається на згорнутомі вікні.
7) *Android.widget.SeekBar* – Лінія завантаження. Може як відображати прогрес якоїсь операції, яку виконує застосунок, так і взаємодіяти з користувачем для більш тонкого налаштування деякого параметра у застосуванні.
8) *androidx.viewpager2.widget.ViewPager2* – Компонент для відображення декількох сторінок контенту на одному екрані з можливістю переключення між ними. Розширюється саме друга версія цього компоненту, адже оригінальна має велику кількість проблем і

вважається застарілою, тому майже не використовується в сучасній розробці під платформу Android.

9) *com.google.material.tabs.TabLayout* – Цей компонент представляє з себе меню з декількома вкладками, між якими користувач може перемикатися. Цей компонент часто використовується в парі з *ViewPager2*

10) *androidx.recyclerview.widget.RecyclerView* – Базовий компонент для створення списків з довільним наповненням. Цей компонент здатен перевикористовувати контейнери для дочірніх елементів списку в цілях покращення продуктивності використання оперативної пам’яті мобільним пристроєм.

Ці компоненти були обрані, оскільки вони є найчастішими компонентами, які використовуються розробниками при створенні мобільних додатків на платформі Android. Не всі компоненти були представлені у цьому списку, адже для деяких з них не має сенсу у створенні реактивних потоків даних на основі подій, які вони можуть надсилати при взаємодії з користувачем, а для інших компонентів цей іункціонал буде покрито завдяки поліформізму функцій-розширень. Наприклад, функції-розришення компонента View можуть використовуватися для будь-яких компонентів для побудови інтерфейсу користувача, адже вони всі наслідуються від View.

3.4 Розробка компонентів бібліотеки

3.4.1. Структура розширень

В рамках реалізації та розробки даної бібліотеки основною задачею для вирішення поставлених цілей буде перетворення інтерфейсу компонентів для побудови інтерфейсу користувача з методів зворотного виклику на реактивні потоки даних Kotlin Flow. Це потрібно для того, щоб забезпечити зручний спосіб взаємодії компонентів рівня представлення мобільного застосування з компонентами рівня ViewModel, побудованими за допомогою реактивних потоків даних.

Для перетворення інтерфейсу методів зворотного виклику буде застосовуватися функція *callbackFlow*, яку надає стандартна бібліотека мови програмування Kotlin. Ця функція працює таким чином, що усі події, які генеруються функцією зворотного виклику, переданою до неї, будуть надсилатися у реактивний потік, який і буде повернений з *callbackFlow*.

Надсилати події можна двома способами: *trySend* та *trySendBlocking*. Різниця між ними в тому, що *trySendBlocking* – блокує поточний потік при спробі надіслати подію у реактивний потік. У випадку коли він заблокований програма буде очікувати, поки він звільниться для надсилання наступної події. *trySend* з іншого боку є неблокуючим методом. Він представляє із себе suspend функцію, яка призупинить виконання допоки даний реактивний потік не буде розблоковано для подальшого надсилання подій. Через цю особливість в рамках даної бібліотеки буде складно його використовувати, оскільки для виклику suspend функції необхідний контекст корутини, в якому відбуватиметься очікування. Цей контекст можна було б створити на місці за допомогою стандартних конструкторів корутин *CoroutineScope()*, однак такий спосіб реалізації створить лише більше проблем, оскільки він прибере всяку можливість розробникам контролювати стан виконання цього потоку даних зовні без посилання на цей контекст. Таким чином в рамках розробки даної бібліотеки зручніше буде використовувати метод *trySendBlocking*.

Для того, щоб реактивний потік даних не закінчив надсилати події одразу, в його тілі потрібно викликати певну suspend функцію, щоб вона призупинила знищення потоку до моменту, поки від нього не відпишеться розробник у своєму коді. На щастя, така функція вже реалізована в рамках стандартної бібліотеки Kotlin Coroutines. Ця функція – *awaitClose()*. Вона очікує, поки реактивний потік даних буде знищено, і тоді завершує своє виконання, щоб не витрачати більше зайвих ресурсів системи мобільного пристрою. До цієї функції можна передати додатковий метод зворотного виклику, який буде викликано перед закриттям реактивного потоку даних. Цей функціонал необхідно буде використовувати у випадках, коли треба звільнити певні

ресурси, створені у тілі потоку. В нашому випадку він використовуватиметься для прибирання методів зворотного виклику, які будуть надані компонентам для побудови інтерфейсу користувача для надсилання події у реактивний потік даних.

Всі асинхронні роботи на архітектурному рівні представлення мобільного застосування на платформі Android мають виконуватися у потоці Main. Для цього базовий потік для створення контекстів корутин у компонентах цього рівня через розширення, надані бібліотекою Android SDK, такі як *lifecycleScope* та *viewModelScope*, є якраз потік Main. У разі запуску обробки подій реактивного потоку даних, створеного за допомогою цієї бібліотеки, через ці контексти, жодних проблем та неочікуваної поведінки в процесі їх виконання не буде. Однак неуважний розробник все ж може допустити помилку та змінити контекст виконання цього потоку даних. На щастя, специфікація Kotlin Flow надає рішення цієї проблеми у вигляді розширення *flowOn* для реактивних потоків даних. При використанні цього методу на потоці, він змінює контекст операцій, виконаних у ланцюгу до виклику *flowOn* на контекст, вказаний у параметрах до цього методу. Таким чином в разі використання оператора *flowOn* з параметром *Dispatchers.Main* бібліотека зможе гарантувати, що навіть у випадку, коли розробник зробив помилку у визначенні контексту, де мають оброблюватися події, надіслані цим реактивним потоком, бібліотека працюватиме коректно та без виключень через використання не-Main потоку для роботи з компонентами інтерфейсу користувача.

3.4.2 Розширення класу View

Як було зазначено раніше, клас View є базовим класом для усіх інших компонентів інтерфейсу користувача, наданих бібліотекою Android SDK. Через це усі розширення, реалізовані для цього класу в рамках даної бібліотеки можуть бути перевикористані для усіх інших компонентів без необхідності писати їх знову для кожного підкласу. В якості подій, на яких будуть

побудовані реактивні потоки даних для цього класу будуть виступати найбільш розповсюджені способи взаємодії користувача з цими елементами. До таких подій відносяться:

1) Кліки. Будь-який компонент, що наслідується від класу View може бути клікнутий користувачем на мобільному пристрої. Для обробки такої взаємодії клас надає метод setOnClickListener. Цей метод може бути перетворений у реактивний потік даних наступною реалізацією:

```
val View.clicks get() = callbackFlow {
    setOnClickListener {  it: View!
        trySendBlocking(Unit)
    }
    awaitClose {
        setOnClickListener(null)
    }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.1 – Реалізація розширення *clicks* для обробки кліків

Оскільки події кліків не мають певної характеристики чи величини, яка могла б оброблятися особливим чином, у створений реактивний потік даних надсилається *Unit* – об'єкт, який не має функціонального значення та представляє з себе мінімальну одиницю інформації. У блоці *awaitClose()* викликіється *setOnClickListener* з параметром null для того, щоб компонент перестав намагатися надіслати нові події у вже закритий потік.

2) Натискання. Окрім кліків, більш складні способи взаємодії з компонентами можуть бути оброблені за допомогою натискань через метод зворотного виклику *setOnTouchListener*. Переданий метод повертає об’єкт *MotionEvent*, який інкапсулює в собі спосіб взаємодії користувача з даним компонентом візуального інтерфейсу програми. В рамках даної взаємодії нас цікавлять події DOWN та UP.

```kotlin
val View.touches get() = callbackFlow {  this: ProducerScope<Boolean>
    setOnTouchListener { _, event ->
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                trySendBlocking( element: true)
                true  ^setOnTouchListener
            }
            MotionEvent.ACTION_UP -> {
                trySendBlocking( element: false)
                performClick()
                true  ^setOnTouchListener
            }
            else -> false  ^setOnTouchListener
        }
    }
    awaitClose { setOnTouchListener(null) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.2 – Реалізація розширення *touches* для обробки натискань

Реалізований реактивний потік даних надсилатиме істину (true), коли користувач покладе палець на екран мобільного пристрою, та хибу (false), коли користувач зніме палець з екрану. Сам використаний метод має повертати true у випадках, коли подія була оброблена даним методом зворотного виклику а інакше – false. Тому ми реагуємо лише на дві задані події, а у всіх інших випадках передаємо подію на обробку іншим компонентам вище за ієрархією.

3) Фокус. Кожен компонент інтерфейсу користувача може знаходитися у фокусі чи поза ним. Зазвичай компонент отримує фокус після того, як користувач висловив бажання з ним взаємодіяти. Розробник може слідкувати за станом фокусу елемента, щоб розуміти, коли він знаходиться на першому плані. Для обробки цих подій існує поле *onFocusChangeListener*.

```kotlin
val View.focus get() = callbackFlow { this: ProducerScope<Boolean>
    onFocusChangeListener = View.OnFocusChangeListener { _, hasFocus ->
        trySendBlocking(hasFocus)
    }
    awaitClose { onFocusChangeListener = null }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.3 – Реалізація розширення *focus* для спостереженням за станом фокусу

Даний потік надсилатиме розробнику істину, коли компонент отримує фокус і хибу, коли компонент його втрачає. Перший параметр позначено символом «_», оскільки він не використовується в даній реалізації.

4) Видимість. Реактивний потік створено для того, щоб спостерігати за станом видимості необхідного компонента. Хоча цей стан може змінюватися лише програмно через виконуваний код, це розширення все ще можна розглядати як утиліту для спостереження за цією властивістю. Для спостереження за цим станом використовується поле *tag* класу View, а також метод зворотного виклику *ViewTreeObserver.OnGlobalLayoutListener*, який викликається при будь-якій зміні стану зображуваних на екрані мобільного пристрою елементів для повторного їх відображення.

```kotlin
val View.currentVisibility get() = callbackFlow { this: ProducerScope<Boolean>
    val listener = ViewTreeObserver.OnGlobalLayoutListener {
        if (tag as? Int != visibility) {
            tag = visibility
            trySendBlocking(isVisible)
        }
    }
    viewTreeObserver.addOnGlobalLayoutListener(listener)
    awaitClose { viewTreeObserver.removeOnGlobalLayoutListener(listener) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.4 – Реалізація розширення *currentVisibility* для спостереженням за видимістю компонента

5) Пересування. У випадках коли користувач виконує рух типу свайпу на компоненті інтерфейсу, цю подію можна обробити перевизначивши метод зворотного виклику *setOnDragListener*.

```
val View.drags get() = callbackFlow {  this: ProducerScope<Pair<Float, Float>>
    setOnDragListener { _, event ->
        trySendBlocking( element: event.x to event.y)
        true  ^setOnDragListener
    }
    awaitClose { setOnDragListener(null) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.5 – Реалізація розширення *drags* для обробки пересувань

Метод зворотного виклику повертає 2 параметра: посилання на об'єкт View, який нас не цікавить, а також об'єкт *DragEvent*. У цій події розробників зазвичай цікавить зміщення для визначення напрямку руху, у якому користувач пересуває компонент. Через це у реактивний потік ми надсилаємо координати x та y цієї події у вигляді пари елементів.

3.4.3 Розширення класу TextView

Робота розробників з цим компонентом, як правило, розгортається навколо використання тексту. Стандартні методи отримання тексту від користувача, реалізовані такими компонентами, як EditText, основані також на взаємодії з цим компонентом. Сам клас TextView є базовим класом для реалізації всіх типів такого роду взаємодій та маніпуляцій як з вхідним, так і з вихідним текстом.

Через це для цього компонента буде реалізовано єдине розширення для використання реактивного потоку – *currentText*. Цей потік буде реагувати на зміни тексту як зі сторони користувача, так і зі сторони мобільного застосування, надсилаючи поточний текст у вихідний потік.

Робота цього потоку буде реалізована за допомогою реалізації інтерфейсу методу зворотного виклику *TextWatcher*. Цей інтерфейс впроваджує 3 методи для реалізації: *beforeTextChanged*, *onTextChanged* та *afterTextChanged*. Перші

два методи не мають великого значення в рамках цієї реалізації, оскільки зазвичай розробників цікавить кінцевий результат введеного користувачем тексту, або кінцевий результат змін до внутрішнього стану цього компонента, зроблених програмним шляхом. Останній метод повертає об'єкт *Editable*, який можна перетворити на строку, викликавши метод *toString()*.

```kotlin
val TextView.currentText get() = callbackFlow {  this: ProducerScope<String>
    val listener = object : TextWatcher {
        override fun beforeTextChanged(text: CharSequence?, start: Int, num: Int, after: Int) = Unit

        override fun onTextChanged(text: CharSequence?, start: Int, before: Int, count: Int) = Unit

        override fun afterTextChanged(text: Editable?) {
            trySendBlocking(text.toString())
        }
    }
    addTextChangedListener(listener)
    awaitClose { removeTextChangedListener(listener) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.6 – Реалізація розширення *currentText* для спостереження за поточним встановленим текстом

3.4.4 Розширення класів CheckBox та SwitchCompat

Ці компоненти використовуються в мобільному застосунку для обробки взаємодії з користувачем, яка може мати лише два варіанти кінцевого значення. Відповідно ці компоненти мають лише два варіанти внутрішнього стану, між якими вони можуть перемикатися.

Обидва компоненти наслідуються від єдиного класу CompoundButton, і тому реалізації розширень реактивних потоків даних для них буде доволі схожою, оскільки вона посилається на одні й ті ж самі програмні інтерфейси. Це зумовлено тим, що як правило в мобільному застосуванні мони виконують одну й ту ж саму функцію, і відрізняються між собою лише дизайном компонента.

Для спостереження за станом цих компонентів використовується метод зворотного виклику *setOnCheckedChangeListener*. Другий параметр цього метода повертає одне з двох значень – істину чи хибу, яка і буде надсилатися до реактивного потоку даних.

```
val CheckBox.checked get() = callbackFlow {  this: ProducerScope<Boolean>
    setOnCheckedChangeListener { _, checked ->
        trySendBlocking(checked)
    }
    awaitClose { setOnCheckedChangeListener(null) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.7 – Реалізація розширення *checked* для спостереження за внутрішнім станом компонента

Розширення для класу SwitchCompat реалізовано аналогічним чином.

3.4.5 Розширення класу RadioGroup

Компонент RadioGroup використовується розробниками, коли користувачеві мобільного додатку треба надати можливість вибору декількох варіантів. В такому випадку створюється єдина група RadioGroup, всередині якої визначаються декілька опцій RadioButton. Можливо було б робити розширення для спостереженням за станом конкретної обраної опції, однак значно більш корисним з точки зору розробника буде спостерігати з всією групою, оскільки стан кожної конкретної опції часто залежить від стану інших опцій.

Визначити цю поведінку можна за допомогою методу зворотного виклику *setOnCheckedChangeListener*. Цей метод повертає в якості результату ідентифікатор обраної опції кожен раз, коли користувач мобільного додатку змінює свій вибір. За цим ідентифікатором потім можна отримати посилання на конкретний об’єкт RadioButton, який відповідає обраній опції. Оскільки всі об’єкти типу RadioButton є дочірніми елементами до групи опції у ієрархії відображення, кожну з цих опцій можна знайти, скориставшись методом *findViewById.*

```kotlin
val RadioGroup.selectedItem get() = callbackFlow {  this: ProducerScope<Int>
    setOnCheckedChangeListener { _, id ->
        trySendBlocking(id)
    }
    awaitClose {
        setOnCheckedChangeListener(null)
    }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.8 – Реалізація розширення *selectedItem* для спостереження за поточною обраною опцією у групі

3.4.6 Розширення класу Spinner

Цей компонент має схожу функції з компонентом RadioGroup – вибір користувачем мобільного застосування однієї опції з набору варіантів. Однак внутрішня реалізації цього компоненту значно відрізняється від попереднього, тому буде використано інші інтерфейси для створення методів зворотного виклику, які будуть надсилати дані до реактивного потоку даних.

Оскільки клас Spinner є наслідником класу AdapterView, буде використовуватися інтерфейс *AdapterView.OnItemSelectedListener,* визначений в цьому класі. Цей інтерфейс визначає 2 методи: *onItemSelected* та *onNothingSelected*. На відміну від RadioGroup, цей компонент може не мати обраного елемента у початковому стані. Через це і необхідність визначити 2 методи.

Коли користувач обере необхідну опцію, у реактивний потік даних буде надіслано позицію цієї опції у загальному списку опцій. Такий підхід є значно більш зручним, аніж використовувати ідентифікатори для посилання на дочірні об’єкти опцій. Одна в той же час він може бути менш гнучким у деяких випадках.

Якщо жоден елемент не обрано, у потік буде надсилатися значення -1, яке не відповідає жодній коректній позиції у списку доступних варіантів. Такий випадок розробник має розглядати як виключення та обробляти його особливим чином.

```kotlin
val Spinner.currentItem get() = callbackFlow { this: ProducerScope<Int>
    val listener = object : AdapterView.OnItemSelectedListener {
        override fun onItemSelected(p: AdapterView<*>?, v: View?, pos: Int, id: Long) {
            trySendBlocking(pos)
        }

        override fun onNothingSelected(p0: AdapterView<*>?) {
            trySendBlocking( element: -1)
        }
    }
    onItemSelectedListener = listener
    awaitClose { onItemSelectedListener = null }
}.distinctUntilChanged().flowOn(Dispatchers.Main)
```

Рисунок 3.9 – Реалізація розширення *currentItem* для спостереження за обраною опцією

3.4.7 Розширення класу SeekBar

Компонент Seekbar використовується для відображення прогресу деякої операції або для взаємодії з користувачем в цілях тонкого налаштування величини певного параметра у програмі. Саме прогрес є визначальною характеристикою цього компонента, тому його значення буде надсилатися у реактивний потік даних.

Проте перший випадок – відображення прогресу операції не буде реалізовано цим розширенням. Таке рішення було прийнято тому, що такого роду зміни обов’язково мають контролюватися з боку програми, тобто розробником застосунку. Однак спостереження за прогресом операції на основі елементів користувацького інтерфейсу є поганою практикою, яка може призвести до помилок у процесі виконання програми. Для спостереженням за цим прогресом розробнику слід використовувати інші засоби на інших архітектурних рівнях програми, наприклад на рівні ViewModel.

У випадку взаємодії з користувачем для обробки цих подій можна застосовувати інтерфейс *OnSeekBarChangeListener*. Цей інтерфейс визначає три методи для перевизначення, серед яких нас цікавить лише один – *onProgressChanged*. Цей метод зворотного виклику надсилатиме у реактивний

потік даних значення *progress* цього компонента, однак лише за умови, якщо значення параметра *fromUser* є істиною.

```
val SeekBar.currentProgress get() = callbackFlow {  this: ProducerScope<Int>
    val listener = object : SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(bar: SeekBar?, progress: Int, fromUser: Boolean) {
            if (fromUser) trySendBlocking(progress)
        }

        override fun onStartTrackingTouch(bar: SeekBar?) = Unit
        override fun onStopTrackingTouch(bar: SeekBar?) = Unit
    }
    setOnSeekBarChangeListener(listener)
    awaitClose { setOnSeekBarChangeListener(null) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.10 – Реалізація розширення *currentProgress* для спостереження за прогресом

3.4.8 Розширення класу ViewPager2

Компонент ViewPager2 використовується для відображення декількох сторінок з інтерфейсом користувачеві, між якими він може перемикатися. Для розробників корисно буде спостерігати за поточною сторінкою, яку бачить користувач. Це можна зробити за допомогою методу зворотного виклику *OnPageChangeCallback*. При зміні обраної сторінки у реактивний потік даних буде надсилатися позація поточної обраної користувачем сторінки у списку усіх сторінок.

Однак цей метод зворотного виклику може надсилати події навіть у випадку, коли користувач переобирає ту ж саму сторінку для відображення. Це може призвести до того, що одна й та ж сама логіка обробки цієї події буде виконуватися декілька разів. Для уникнення цієї проблеми слід відфільтрувати реактивний потік даних таким чином, щоб події надсилатися лише при зміні стану, а не кожного разу, коли метод зворотного виклику реагує на подію. Для цього компонент Kotlin Flow має розширення *distinctUntilChanged*, який реалізує саме таку поведінку. Цей оператор буде додано до результуючого потоку.

```
val ViewPager2.currentPage get() = callbackFlow { this: ProducerScope<Int>
    val callback = object : OnPageChangeCallback() {
        override fun onPageSelected(position: Int) {
            trySendBlocking(position)
        }
    }
    registerOnPageChangeCallback(callback)
    awaitClose { unregisterOnPageChangeCallback(callback) }
}.distinctUntilChanged().flowOn(Dispatchers.Main)
```

Рисунок 3.11 – Реалізація розширення *currentPage* для спостереження за поточною сторінкою інтерфейсу

3.4.9 Розширення класу TabLayout

Компонент TabLayout части використовується в парі з ViewPager2 для відображення поточної обраної користувачем сторінки інтерфейсу. Проте він також може використовуватися і окремо в цілях відображення будь-якого інтерфейсу за макетом вкладинок.

Для реалізації цього розширення було створено об’єкт, який імплементує інтерфейс *TabLayout.OnTabSelectedListener*. Цей інтерфейс визначає 3 методи для перевизначення. Метод *onTabSelected* використовується при виборі нової вкладки серед списку доступних. В цьому випадку у потік буде надсилатися позиція цієї вкладки у списку. Метод *onTabUnselected* викликається, коли користувач прибирає свій вибір зі списку доступних вкладок. Ця поведінка не обов’язково має бути специфікованою розробником, однак бібліотека покриватиме і такі можливі способи застосування. В цьому випадку до потоку буде надсилатися значення -1 як індекс, що не відповідає одній вкладці. Останній метод – *onTabReselected,* він викликається у випадку, коли користувач переобирає поточну вкладку додатковий раз. В такому випадку, як і в першому, у потік буде надсилатися індекс переобраної вкладки.

```kotlin
val TabLayout.selectedTab get() = callbackFlow { this: ProducerScope<Int>
    val listener = object : TabLayout.OnTabSelectedListener {
        override fun onTabSelected(tab: TabLayout.Tab?) {
            val position = tab?.position ?: -1
            trySendBlocking(position)
        }

        override fun onTabUnselected(tab: TabLayout.Tab?) {
            trySendBlocking( element: -1)
        }

        override fun onTabReselected(tab: TabLayout.Tab?) {
            val position = tab?.position ?: -1
            trySendBlocking(position)
        }
    }
    addOnTabSelectedListener(listener)
    awaitClose { removeOnTabSelectedListener(listener) }
}.distinctUntilChanged().flowOn(Dispatchers.Main)
```

Рисунок 3.12 – Реалізація розширення *selectedTab* для спостереження за поточною обраною вкладкою

3.4.10 Розширення класу RecyclerView

Цейс компонент є дуже універсальним інструментом для створення будь-яких списків. До нього можна написати велику кількість різноманітних розширень в залежності від способу використання, але ці розширення можуть обмежити універсальність компонента. Через це спектр розширень із застосуванням реактивних потоків даних є дещо обмеженим, тому кожне з них слід застосувати для спостереженням за величиною, яка є незмінною для будь-якої реалізації RecyclerView.

Однією з таких величин є прокрутка списку відносно початкової позиції. Ця величина впливає на роботу компонента, змушуючи його створювати та видаляти нові контейнери для елементів, які необхідно відобразити у списку. Прокрутка може бути горизонтальною та вертикальною – це залежить від способу реалізації та орієнтації компонента RecyclerView. Оскільки

компонент рекомендується використовувати для прокрутки одночасно лише в одному напрямку, можна створювати окремі потоки для кожного із видів прокручування та оброблювати їх окремо.

Для реалізації такої поведінки в рамках бібліотеки використовуватиметься метод зворотного виклику *RecyclerView.OnScrollListener*. Цей метод повертає переміщення прокрутки, створеної користувачем, у двох координатах – dx та dy. Оскільки одночасно нас цікавить лише одна величина, буде створено два розширення, кожне з яких відповідає за прокрутку по одній з двох осей координат.

```
val RecyclerView.verticalScroll get() = callbackFlow {  this: ProducerScope<Int>
    val listener = object : RecyclerView.OnScrollListener() {
        override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
            trySendBlocking(dy)
        }
    }
    addOnScrollListener(listener)
    awaitClose { removeOnScrollListener(listener) }
}.flowOn(Dispatchers.Main)
```

Рисунок 3.13 – Реалізація розширення *verticalScroll* для спостереження за вертикальною прокруткою списку

Розширення компонента з реактивним потоком для спостереження за горизонтальною прокруткою реалізовано аналогічним чином

3.5 Сценарій використання бібліотеки

Створені бібліотекою розширення до цих компонентів нададуть розробникам програмного забезпечення під платформу Android можливість інтегрувати реактивні потоки даних у архітектурний рівень представлення мобільного застосунку. Оскільки ці компоненти представляють з себе елементи інтерфейсу користувача, взаємодіяти з ними можна буде з компонентів Android додатку, які мають безпосередній доступ до контролю за створенням та менеджментом життєвого циклу цих компонентів, такими як Fragment або Activity.

Після створення View для цих компонентів, розробник має отримати посилання на візуальні компоненти, розширення до яких підтримуються даною бібліотекою. Це можна зробити або за допомогою методу *findViewById* або більш рекомендованим способом – застосовуючи *viewBinding*. Після отримання косилання на ці компоненти та імпортувавши розширення з бібліотеки, розробник зможе створювати ці реактивні потоки даних, просто звертаючись до визначених розширень.

Оброблювати ці потоки подій необхідно використовуючи елемент *lifecycleScope* або інші способи створення контексту корутин з прив'язкою до життєвого циклу компонента. Події можна оброблювати як окремо, змінюючи внутрішній стан представлення чи виконуючи дії для запуску операцій обробки введених користувачем інпутів, так і комбінуючи їх між собою, або з іншими реактивними потоками даних, наприклад з рівня *ViewModel*.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі цієї розробки було описано проблеми, з якими стикаються розробники програмного забезпечення на платформі Android та шляхи, якими ці проблеми можна адресувати та вирішити. На основі цього аналізу було розроблено бібліотеку, яка використовує інструменти мови програмування Kotlin, зокрема Kotlin Flow та функції-розширення, для вирішення поставлених задач та покращення процесу розробки для програмістів під Android.

Для створення розширень було взято групу найбільш розповсюджених та популярних компонентів для побудови інтерфейсу користувача. Враховуючи особливості застосування кожного з них було розроблено відповідні розширення, які впроваджують реактивні потоки даних для обробки взаємодії цих компонентів з користувачем мобільного додатку на платформі Android. Було описано сценарій використання бібліотеки у контексті архітектурного рівня представлення мобільного застосування з описом ситуацій, на які треба звернути увагу при її використанні.

ВИСНОВКИ

В процесі виконання дипломної роботи було створено бібліотеку для інтеграції функціональних та реактивних підходів в процес розробки користувацького інтерфейсу для мобільних застосувань на платформі Android. Бібліотека впроваджує для розробників спосіб підтримувати рекомендації програмної архітектури, засновані на парадигмах реактивного програмування, на всіх рівнях додатку, зокрема архітектурного рівня представлення та його взаємодії з рівнями вище.

В першому розділі цієї роботи було розглянуто історію парадигми реактивного програмування в цілому та її процес впровадження в спільноту розробників під мобільну платформу Android. Було проаналізовано передумови її появи та її переваги над тими підходами та шаблонами програмування, які застосовуватися для розробки під цю платформу раніше. Також було досліджено тенденції розвитку програмування на платформі та розвитку парадигм у її контексті.

В другому розділі цієї роботи було розглянуто інструменти, які використовуються розробниками для розробки застосувань під платформу Android в контексті створення бібліотеки. Було складено порівняльну характеристику мов програмування, які домінують на цій платформі та, враховуючи переваги та недоліки кожної з них, було обрано мову для розробки бібліотеки, яка найкращим чином відображатиме тенденції розвитку реактивної парадигми програмування та яка має найбільш широкий та глибокий вибір інструментів для реалізації цієї бібліотеки.

В третьому розділі цієї роботи було розглянуто деталі реалізації бібліотеки. Було представлено головні проблеми, з якими стикаються розробники мобільних застосувань під платформу Android, для вирішення яких в першу чергу і було описано технічне завдання для бібліотеки. Згідно з архітектурними принципами, які рекомендуються для розробників на цій

платформі було розроблено загальну архітектуру бібліотеки на основі інструментів мови програмування Kotlin – Flow та функцій-розширень. Серед компонентів для побудови користувацького інтерфейсу під Android було обрано найбільш актуальні в контексті сучасної розробки на платформі. Було проаналізовано особливості їх застосування та на їх основі було побудовано компоненті бібліотеки. Також був описаний сценарій використання бібліотеки з використанням стандартних компонентів у типічному Android-додатку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. An Overview of Functional Reactive Programming [Електронний ресурс] – Режим доступу до ресурсу: https://es.cs.rptu.de/publications/datarsg/Kock21.pdf
2. The Reactive Manifesto [Електронний ресурс] – Режим доступу до ресурсу: https://www.reactivemanifesto.org/
3. Introduction of ReactiveX [Електронний ресурс] – Режим доступу до ресурсу: https://reactivex.io/intro.html
4. Background processing in Android [Електронний ресурс] – Режим доступу до ресурсу: https://www.vogella.com/tutorials/AndroidBackgroundProcessing/article.html
5. Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft / Java 8 in action, 2014 – С. 56-76
6. Processes and threads overview [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/guide/components/processes-and-threads
7. Jeff Friesen / Learn Java for Android Development, 2nd Edition, 2012 – С. 525-530, 603-611
8. Design Patterns - MVC Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm
9. Tomasz Nurkiewicz, Ben Christensen / Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications, 2016 – С. 27-32, 61-69

10. Android Architecture Components [Електронний ресурс] –Режим доступу до ресурсу: https://www.kodeco.com/books/advanced-android-app-architecture/v1.0/chapters/4-android-architecture-components
11. Handling Lifecycles with Lifecycle-Aware Components [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/topic/libraries/architecture/lifecycle
12. Android LiveData vs RxJava [Електронний ресурс] – Режим доступу до ресурсу: https://www.digitalocean.com/community/tutorials/android-livedata
13. Android Jetpack Dev Resources [Електронний ресурс] – режим доступу до ресурсу: https://developer.android.com/jetpack
14. Dmitry Jemerov, Svetlana Isakova / Kotlin in Action First Edition – С. 30-33
15. Coroutines basics [Електронний ресурс] – Режим доступу до ресурсу: https://kotlinlang.org/docs/coroutines-basics.html
16. The suspend modifier — under the hood [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/androiddevelopers/the-suspend-modifier-under-the-hood-b7ce46af624f
17. Kotlin flows on Android [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/kotlin/flow
18. History and Retrospective - Java [Електронний ресурс] – Режим доступу до ресурсу: https://www.oreilly.com/library/view/java-the-legend/9781492048299/ch01.html
19. 11 Most In-Demand Programming Languages in 2023 [Електронний ресурс] – Режим доступу до ресурсу: https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/
20. What is Kotlin? The Java alternative explained [Електронний ресурс] – Режим доступу до ресурсу:

https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html

21. Aleksei Sedunov / Kotlin In-Depth [Vol-I]: A Comprehensive Guide to Modern Multi-Paradigm Language, 2020 – С. 26-33
22. Android's Kotlin-first approach [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/kotlin/first
23. Observable pattern in RXJava [Електронний ресурс] – Режим доступу до ресурсу: https://reactivex.io/documentation/observable.html
24. Difference Between Flows and Channels in Kotlin [Електронний ресурс] – Режим доступу до ресурсу: https://www.baeldung.com/kotlin/flows-vs-channels
25. From RxJava to Kotlin Flow: Backpressure [Електронний ресурс] – Режим доступу до ресурсу: https://proandroiddev.com/from-rxjava-to-kotlin-flow-backpressure-d1fb91e6dea8
26. Alphabetical List of Observable Operators [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/ReactiveX/RxJava/wiki/Alphabetical-List-of-Observable-Operators
27. Flow Operators in Kotlin [Електронний ресурс] – Режим доступу до ресурсу: https://androidgeek.co/how-to-use-flow-operators-in-kotlin-complete-guide-part-2-8c14adaa0349
28. Context Propagation with Project Reactor 3 [Електронний ресурс] – Режим доступу до ресурсу: https://spring.io/blog/2023/03/30/context-propagation-with-project-reactor-3-unified-bridging-between-reactive
29. When and How to Use RxJava Disposable [Електронний ресурс] – Режим доступу до ресурсу: https://cupsofcode.com/post/when_how_use_rxjava_disposable_serialdisposable_compositedisposable

30. Structured concurrency [Електронний ресурс] – Режим доступу до ресурсу: https://elizarov.medium.com/structured-concurrency-722d765aa952
31. Nickolay Tsvetinov / Learning Reactive Programming with Java 8, 2015 – С. 43-50

ДОДАТОК 1

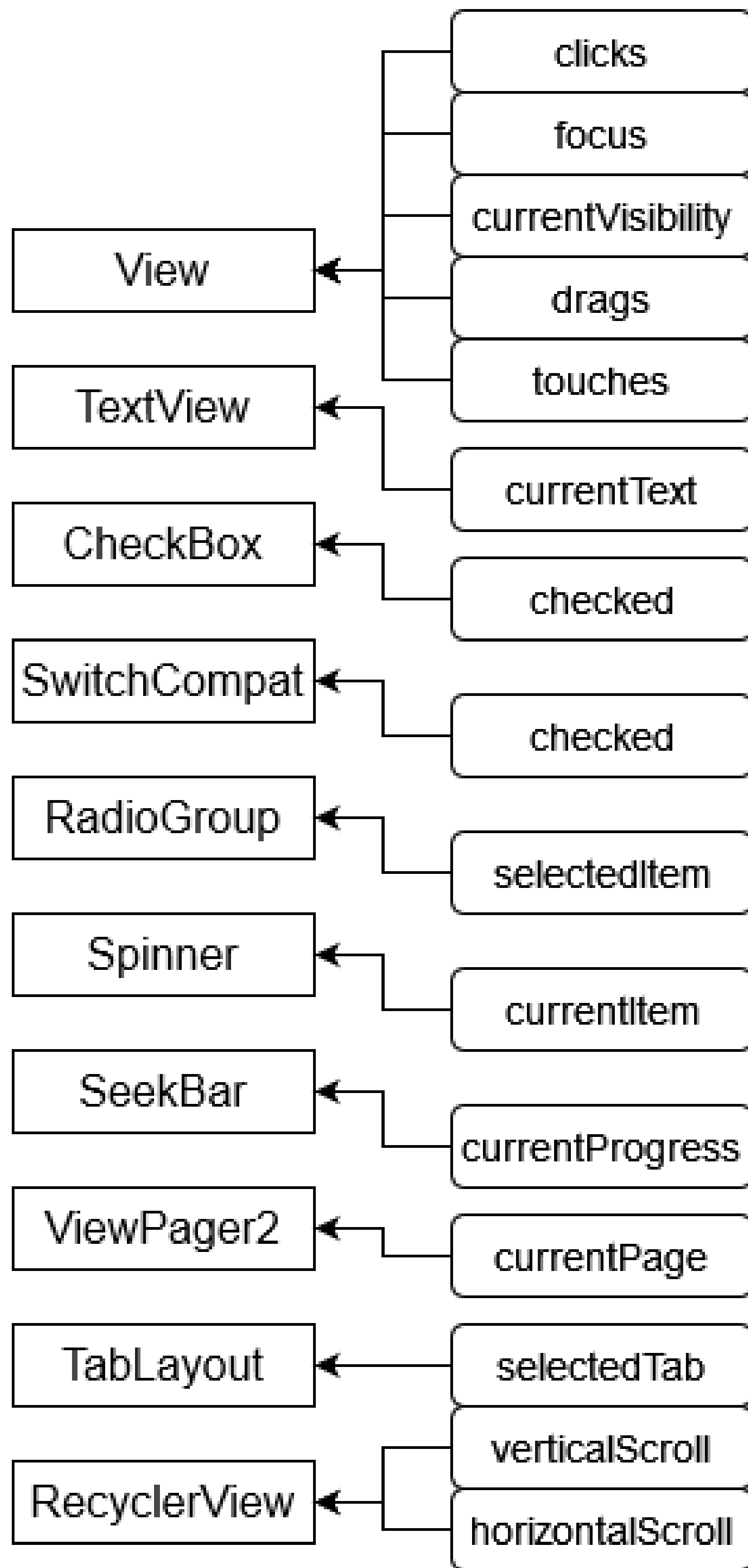
Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android

Схема розширень бібліотеки (функціональна схема)

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2023 р



					ІАЛЦ.467200.004 Д1			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Стронов І. С.			Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android Схема розширень бібліотеки (функціональна схема)	Літ.	Аркуш	Аркушів
Перевірив		Пономаренко А. М.					1	
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93		
Н. Контр.		Волокита А. М.						
Затвердив								

ДОДАТОК 2

Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android

Алгоритм бібліотеки (принципова схема)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2023 р

					ІАЛЦ.467200.005 Д2
Зм.	Арк.	№ докум.	Підпис	Дата	
Розробив		Стронов І. С.			**Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android** Алгоритм бібліотеки (принципова схема)
Перевірив		Пономаренко А. М.			Літ. / Аркуш: 1 / Аркушів
Реценз.					КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93
Н. Контр.		Волокита А. М.			
Затвердив					

ДОДАТОК 3

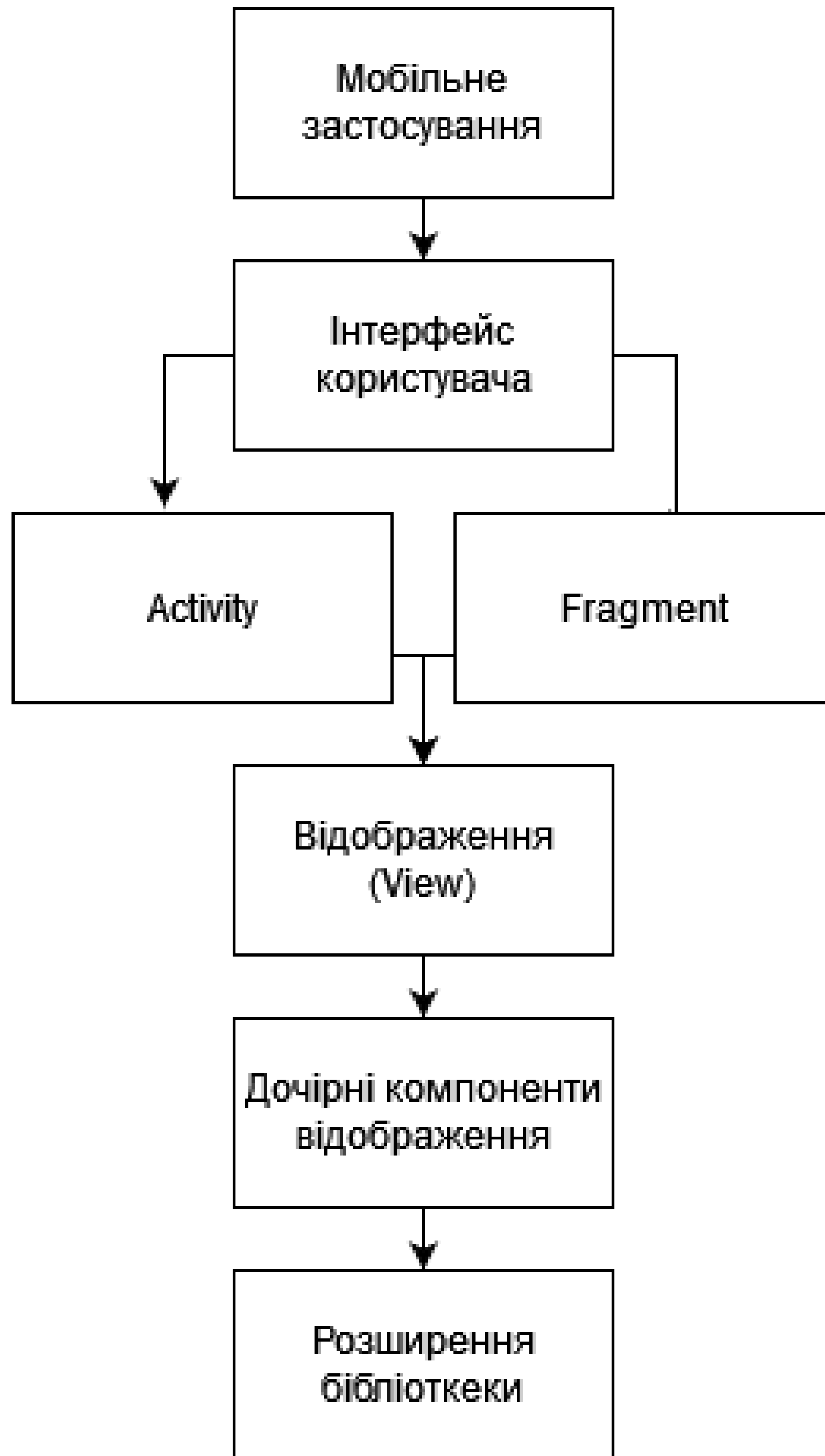
Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android

Структурна схема бібліотеки

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2023 р



Зм.	Арк.	№ докум.	Підпис	Дата	ІАЛЦ.467200.006 Д3		
Розробив		Стронов І. С.			Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android Структурна схема бібліотеки	Літ.	Аркуш	Аркушів
Перевірив		Пономаренко А. М.					1	
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-93		
Н. Контр.		Волокита А. М.						
Затвердив								

ДОДАТОК 4

Бібліотека для розробки інтерфейсу користувача із застосуванням реактивних потоків даних на платформі Android

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів

Київ 2023 р

```
package com.stronov.kavex

import android.widget.CheckBox
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val CheckBox.checked get() = callbackFlow {
    setOnCheckedChangeListener { _, checked ->
        trySendBlocking(checked)
    }
    awaitClose { setOnCheckedChangeListener(null) }
}.flowOn(Dispatchers.Main)


package com.stronov.kavex

import android.widget.RadioGroup
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val RadioGroup.selectedItem get() = callbackFlow {
    setOnCheckedChangeListener { _, id ->
        trySendBlocking(id)
    }
    awaitClose {
        setOnCheckedChangeListener(null)
    }
}.flowOn(Dispatchers.Main)


package com.stronov.kavex

import androidx.recyclerview.widget.RecyclerView
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val RecyclerView.verticalScroll get() = callbackFlow {
    val listener = object : RecyclerView.OnScrollListener() {
        override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
            trySendBlocking(dy)
        }
    }
    addOnScrollListener(listener)
    awaitClose { removeOnScrollListener(listener) }
}.flowOn(Dispatchers.Main)
```

```kotlin
val RecyclerView.horizontalScroll get() = callbackFlow {
    val listener = object : RecyclerView.OnScrollListener() {
        override fun onScrolled(recyclerView: RecyclerView, dx: Int, dy: Int) {
            trySendBlocking(dx)
        }
    }
    addOnScrollListener(listener)
    awaitClose { removeOnScrollListener(listener) }
}.flowOn(Dispatchers.Main)


package com.stronov.kavex

import android.widget.SeekBar
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val SeekBar.currentProgress get() = callbackFlow {
    val listener = object : SeekBar.OnSeekBarChangeListener {
        override fun onProgressChanged(bar: SeekBar?, progress: Int, fromUser:
Boolean) {
            if (fromUser) trySendBlocking(progress)
        }

        override fun onStartTrackingTouch(bar: SeekBar?) = Unit
        override fun onStopTrackingTouch(bar: SeekBar?) = Unit
    }
    setOnSeekBarChangeListener(listener)
    awaitClose { setOnSeekBarChangeListener(null) }
}.flowOn(Dispatchers.Main)
package com.stronov.kavex

import android.view.View
import android.widget.AdapterView
import android.widget.Spinner
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.distinctUntilChanged
import kotlinx.coroutines.flow.flowOn

val Spinner.currentItem get() = callbackFlow {
    val listener = object : AdapterView.OnItemSelectedListener {
        override fun onItemSelected(p: AdapterView<*>?, v: View?, pos: Int, id: Long)
{
            trySendBlocking(pos)
        }

        override fun onNothingSelected(p0: AdapterView<*>?) {
            trySendBlocking(-1)
        }
    }
    onItemSelectedListener = listener
    awaitClose { onItemSelectedListener = null }
}.distinctUntilChanged().flowOn(Dispatchers.Main)
```

```
package com.stronov.kavex

import androidx.appcompat.widget.SwitchCompat
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val SwitchCompat.checked get() = callbackFlow {
    setOnCheckedChangeListener { _, checked -> trySendBlocking(checked) }
    awaitClose { setOnCheckedChangeListener(null) }
}.flowOn(Dispatchers.Main)
package com.stronov.kavex

import com.google.android.material.tabs.TabLayout
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.distinctUntilChanged
import kotlinx.coroutines.flow.flowOn

val TabLayout.selectedTab get() = callbackFlow {
    val listener = object : TabLayout.OnTabSelectedListener {
        override fun onTabSelected(tab: TabLayout.Tab?) {
            val position = tab?.position ?: -1
            trySendBlocking(position)
        }

        override fun onTabUnselected(tab: TabLayout.Tab?) {
            trySendBlocking(-1)
        }

        override fun onTabReselected(tab: TabLayout.Tab?) {
            val position = tab?.position ?: -1
            trySendBlocking(position)
        }
    }
    addOnTabSelectedListener(listener)
    awaitClose { removeOnTabSelectedListener(listener) }
}.distinctUntilChanged().flowOn(Dispatchers.Main)


package com.stronov.kavex

import android.text.Editable
import android.text.TextWatcher
import android.widget.TextView
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val TextView.currentText get() = callbackFlow {
    val listener = object : TextWatcher {
        override fun beforeTextChanged(text: CharSequence?, start: Int, num: Int,
after: Int) = Unit
```

```
        override fun onTextChanged(text: CharSequence?, start: Int, before: Int,
count: Int) = Unit

        override fun afterTextChanged(text: Editable?) {
            trySendBlocking(text.toString())
        }
    }
    addTextChangedListener(listener)
    awaitClose { removeTextChangedListener(listener) }
}.flowOn(Dispatchers.Main)

package com.stronov.kavex

import android.view.MotionEvent
import android.view.View
import android.view.ViewTreeObserver
import androidx.core.view.isVisible
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.flowOn

val View.clicks get() = callbackFlow {
    setOnClickListener {
        trySendBlocking(Unit)
    }
    awaitClose {
        setOnClickListener(null)
    }
}.flowOn(Dispatchers.Main)

val View.touches get() = callbackFlow {
    setOnTouchListener { _, event ->
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                trySendBlocking(true)
                true
            }
            MotionEvent.ACTION_UP -> {
                trySendBlocking(false)
                performClick()
                true
            }
            else -> false
        }
    }
    awaitClose { setOnTouchListener(null) }
}.flowOn(Dispatchers.Main)

val View.focus get() = callbackFlow {
    onFocusChangeListener = View.OnFocusChangeListener { _, hasFocus ->
        trySendBlocking(hasFocus)
    }
    awaitClose { onFocusChangeListener = null }
}.flowOn(Dispatchers.Main)

val View.drags get() = callbackFlow {
    setOnDragListener { _, event ->
        trySendBlocking(event.x to event.y)
```

```
true
    }
    awaitClose { setOnDragListener(null) }
}.flowOn(Dispatchers.Main)

val View.currentVisibility get() = callbackFlow {
    val listener = ViewTreeObserver.OnGlobalLayoutListener {
        if (tag as? Int != visibility) {
            tag = visibility
            trySendBlocking(isVisible)
        }
    }
    viewTreeObserver.addOnGlobalLayoutListener(listener)
    awaitClose { viewTreeObserver.removeOnGlobalLayoutListener(listener) }
}.flowOn(Dispatchers.Main)

package com.stronov.kavex

import androidx.viewpager2.widget.ViewPager2
import androidx.viewpager2.widget.ViewPager2.OnPageChangeCallback
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.channels.trySendBlocking
import kotlinx.coroutines.flow.callbackFlow
import kotlinx.coroutines.flow.distinctUntilChanged
import kotlinx.coroutines.flow.flowOn

val ViewPager2.currentPage get() = callbackFlow {
    val callback = object : OnPageChangeCallback() {
        override fun onPageSelected(position: Int) {
            trySendBlocking(position)
        }
    }
    registerOnPageChangeCallback(callback)
    awaitClose { unregisterOnPageChangeCallback(callback) }
}.distinctUntilChanged().flowOn(Dispatchers.Main)

plugins {
    id 'com.android.library'
    id 'org.jetbrains.kotlin.android'
}

android {
    namespace 'com.stronov.kavex'
    compileSdk 33

    defaultConfig {
        minSdk 21
        targetSdk 33

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        consumerProguardFiles "consumer-rules.pro"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
        }
```

```
}
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
    }
}

dependencies {

    implementation 'androidx.core:core-ktx:1.10.1'
    implementation 'androidx.appcompat:appcompat:1.6.1'
    implementation 'com.google.android.material:material:1.9.0'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.5'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
}
```