DOI: https://doi.org/10.33216/2222-3428-2024-26-2
УДК 004.4'236

Щербаков Є.В., Щербакова М.Є.

ОСОБЛИВОСТІ РОЗРОБКИ ANDROID-ДОДАТКІВ НА БАЗІ JETPACK COMPOSE

Jetpack Compose — це абсолютно новий підхід до розробки додатків для всіх платформ операційних систем Google. Основні цілі UI-фреймворка Compose — зробити розробку додатків легшою, швидшою та менш схильною до типових помилок, які зазвичай з'являються під час розробки програмних проектів. Багато переваг Compose походять від того факту, що він одночасно декларативний і керований потоком даних. Декларативний синтаксис Compose забезпечує зовсім інший спосіб реалізації макетів і поведінки інтерфейсів користувача ніж було раніше. Замість того, щоб вручну розробляти складні деталі зовнішнього вигляду компонентів, які складають чергову сцену, Compose дозволяє описувати сцени за допомогою простого та інтуїтивно зрозумілого синтаксису. Іншими словами, Compose дозволяє створювати екрани, оголошуючи, як має виглядати інтерфейс користувача, не турбуючись про складність того, як будується кожен екран. Після написання цих оголошень усі заплутані та складні деталі позиціонування, обмежень, рендерингу та рекомпозиції екрана автоматично обробляються середовищем виконання Compose. З появою Jetpack Compose рекомендується розробляти сучасні Android-додатки тільки з однією активністю, де різні екрани для взаємодії з користувачем завантажуються як контент в рамках цієї активності. Окрім того, нова архітектура додатків рекомендує розділяти різні зони відповідальності в додатку на абсолютно окремі модулі (separation of concerns). Одним із ключів до такого підходу є компонент ViewModel, який реалізується як окремий клас і включає змінні стану, що містять дані моделі та функції, які можуть бути викликані для управління цими даними. Активність, яка містить інтерфейс користувача, спостерігає (observes) за значеннями стану і функціонує так, що будь-які зміни значень в моделі ініціюють рекомпозицію екрана.
Ключові слова: *інтерфейс користувача (UI), віджет, макет, декларативний синтаксис, потік даних, рендеринг, рекомпозиція.*

Вступ. Історично ієрархію представлень (views) на екрані Android-додатка можна було уявляти як дерево віджетів інтерфейсу користувача [1]. Оскільки стан додатка змінюється через такі речі, як взаємодія користувача, ієрархію віджетів потрібно оновлювати, щоб відображати поточні дані. Найпоширеніший спосіб оновлення інтерфейсу користувача — пройти по дереву за допомогою таких функцій, як findViewById(), і змінювати вузли, викликаючи такі методи, як button.setText(String), container.addChild(View) або img.setImageBitmap(Bitmap). Ці методи змінюють внутрішній стан віджета.

Маніпулювання представленнями при ручному кодуванні підвищує ймовірність помилок. Якщо частина даних відображається в кількох місцях, легко забути оновити одне з представлень, у якому вони відображаються. Також легко створити нелегальні стани, коли два оновлення конфліктують неочікуваним чином. Наприклад, оновлення може спробувати встановити значення вузла, який щойно було видалено з інтерфейсу користувача. Загалом, складність обслуговування програмного забезпечення зростає разом із кількістю представлень, які потребують оновлення.

За останні кілька років уся галузь IT почала переходити на декларативну модель інтерфейсу користувача, яка значно спрощує інженерію, пов'язану зі створенням і оновленням візуального інтерфейсу додатка для взаємодії з користувачем. Техніка працює шляхом концептуальної регенерації всього екрана з нуля, а потім застосування лише необхідних змін. Цей підхід дозволяє уникнути складності ручного оновлення ієрархії представлень при зміні стану. Compose — це і є декларативний UI-фреймворк для розробки додатків для ОС Android. Однією з проблем регенерації всього екрана є те, що це потенційно дорого з точки зору часу, обчислювальної потужності та використання батареї пристрою. Щоб зменшити ці витрати, Compose під час рекомпозиції розумно вибирає, які частини інтерфейсу користувача потрібно перерисовувати в будь-який момент часу. Це має певні наслідки для того, як проектуються та розробляються компоненти інтерфейсу користувача.

Композиційні функції. Композиційні функції (також відомі як composables або композиційні елементи) — це спеціальні функції Kotlin, які використовуються для створення інтерфейсів користувача для роботи з Compose [2, 3]. Композиційна функція відрізняється від звичайних функцій Kotlin у коді за допомогою анотації @Composable.

Коли викликається композиційна функція, їй зазвичай передаються деякі дані та набір властивостей, які визначають, як відповідна секція інтерфейсу користувача має поводитися та відображатися під час рендерингу користувачеві у запущеному на виконання додатку. По суті, композиційні функції трансформують дані в елементи інтерфейсу користувача (рис. 1).

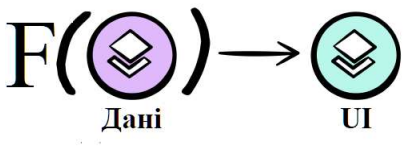

Рисунок 1 - Інтерфейс користувача в Compose є функцією даних (стану)

Композиційні функції не повертають значення в традиційному розумінні функції Kotlin, а натомість надають елементи інтерфейсу користувача системі виконання Compose для рендерингу. Композиційні функції можуть викликати інші композиційні функції для створення ієрархії компонентів. Хоча композиційна функція може викликати також стандартні функції Kotlin, стандартні функції не можуть викликати композиційні функції. Типовий інтерфейс користувача на основі Compose зазвичай складається з комбінації вбудованих в Compose і спеціально створених композиційних функцій.

Композиційні функції класифікуються як такі, що мають стан або не мають стану [4]. Стан, у контексті Compose, визначається як будь-яке значення, що може змінюватися під час виконання додатку. Наприклад, значення положення повзунка, рядок, введений у текстове поле, або поточна установка прапорця — усі вони є формами стану. Збереження стану досягається за допомогою ключового слова *remember* і функції mutableStateOf().

Композиційні функції оголошуються за допомогою анотації @Composable і пишуться майже так само, як і стандартні функції Kotlin. Можна, наприклад, оголосити композиційну функцію, яка нічого не робить, наступним чином:

```
@Composable
fun MyFunction() {
}
```

Композиційна функція може викликати інші композиційні функції та приймати аргументи:

```
@Composable
fun CustomText(text: String, fontWeight: FontWeight, color: Color) {
    Text(text = text, fontWeight = fontWeight, color = color)
}
```

При розробці додатків за допомогою Compose використовується суміш користувацьких композиційних функцій в поєднанні з набором готових до використання компонентів, наданих комплектом розробки Compose (наприклад, композиційні функції Text, Button, Column і Slider).

Композиційні функції, що постачаються в комплекті з Compose, поділяються на три категорії: макетні (*Layout*) компоненти, базові (*Foundation*) компоненти та компоненти матеріального дизайну (Material Design). Макетні компоненти дають змогу визначати розташування компонентів на екрані та поведінку цих компонентів відносно один одного. Макетні композиційні функції включають: Box, BoxWithConstraints, Column, ConstraintLayout і Row.

Базові композиційні функції - це набір мінімальних компонентів, які забезпечують базову функціональність інтерфейсу користувача. Хоча ці компоненти за умовчанням не нав'язують певний стиль або тему, їх можна налаштувати так, щоб забезпечити будь-який вигляд і поведінку, необхідні для додатка. До них належать: BaseTextField, Canvas, Image, LazyColumn, LazyRow, Shape та Text.

З іншого боку, композиційні функції матеріального дизайну розроблено таким чином, щоб вони відповідали правилам теми Google’s Material. Вони включають, наприклад, композиційні функції AlertDialog, Button, Card, CircularProgressIndicator, DropdownMenu, Checkbox, FloatingActionButton, LinearProgressIndicator, ModalDrawer, RadioButton, Scaffold, Slider, Snackbar, Switch, TextField, TopAppBar і BottomNavigation.

При виборі композиційних функцій важливо відзначити, що базові композиційні функції та композиційні функції матеріального дизайну не є взаємовиключними. При розробці неминуче будуть використовуватися композиційні функції з обох категорій, оскільки категорія матеріального дизайну має композиційні функції, для яких немає еквівалентів в наборі базових композиційних функцій, і навпаки.

На додаток до трьох згаданих вище категорій Compose забезпечує використання в композиційних функціях API слотів (Slot API). Реалізуючи API слотів, контент композиційної функції може бути динамічно заданий в точці, в якій вона викликається. Це контрастує зі статичним контентом типової композиційної функції, де контент визначається в точці оголошення функції і не може бути згодом змінений. Композиційна функція з використанням API слотів по суті є шаблоном інтерфейсу користувача, що містить один або кілька слотів, в які інші композиційні функції можуть бути вставлені під час виконання додатка.

Стан та рекомпозиція в Compose. До появи Compose додаток для Android містив специфічний доволі складний код, відповідальний за контроль за поточними значеннями даних в додатку [5]. Compose вирішує цю складність, надаючи систему підтримки, яка базується на стані. Дані, які зберігаються як стан, гарантують, що будь-які зміни цих даних автоматично будуть відображатися в інтерфейсі користувача без необхідності писати додатковий код для виявлення змін. Будь-який компонент інтерфейсу користувача, який отримує доступ до стану, по суті, робить підписку на цей стан. Коли стан змінюється в будь-якій частині коду додатка, всі елементи підписки на ці дані знищуються та створюються заново, щоб відобразити зміни, що відбулися. Це

гарантує, що коли будь-який стан, від якого залежать інтерфейси користувача, змінюється, усі компоненти, які покладаються на ці дані, автоматично оновлюються.

У декларативних системах, таких як Compose, станом додатка зазвичай називають групу значень, які можуть змінюватися з часом. На перший погляд, це схоже на будь-які інші дані в додатку. Стандартна змінна Kotlin, наприклад, за визначенням призначена для зберігання значення, яке може змінюватися в будь-який момент під час виконання. Стан, однак, відрізняється від множини стандартних змінних двома суттєвими ознаками.

По-перше, потрібно запам'ятовувати (remember) значення, присвоєне змінній стану у композиційній функції. Іншими словами, кожного разу, коли викликається композиційна функція, що містить стан (функція зі станом), вона повинна запам'ятовувати будь-які значення стану з моменту останнього виклику. Це відрізняється від стандартної локальної змінної, яка повторно ініціалізується кожного разу, коли виконується виклик функції, в якій вона оголошена.

Друга ключова відмінність полягає в тому, що модифікація значення будь-якої змінної стану має далекосяжні наслідки для всього дерева ієрархії композиційних функцій, які складають інтерфейс користувача в Compose. Як було описано раніше, кожну композиційну функцію в дереві можна розглядати як приймаючу дані і використовуючу ці дані для створення секцій макету інтерфейсу користувача. Потім ці секції рендеряться на екрані системою виконання Compose. У більшості випадків дані, що передаються від однієї композиційної функції до іншої, оголошуються як змінні стану в батьківській функції. Це означає, що будь-яка зміна значення стану в батьківській композиційній функції повинна бути відображена в будь-якій дочірній композиційній функції, якій було передано стан. Compose вирішує це, виконуючи операцію, яка називається рекомпозицією.

Рекомпозиція відбувається щоразу, коли змінюється значення стану в ієрархії композиційних функцій. Щойно Compose виявляє зміну стану, він обробляє всі композиційні функції в активності та виконує рекомпозицію всіх функцій, на які впливає зміна значення стану. Рекомпозиція просто означає, що функція викликається знову з передачею їй нового значення стану.

Першим кроком у оголошенні значення стану є обгортання його в об'єкт MutableState, наприклад:

```
var myState = remember { mutableStateOf("Hello!") }
```

MutableState — це клас Compose, який називають спостережуваним типом (observable type). Кажуть, що будь-яка функція, яка читає значення стану, підписана на цей спостережуваний стан. Як наслідок, будь-які зміни значення стану ініціюють рекомпозицію всіх підписаних функцій.

Рекомпозиція всього композиційного дерева інтерфейсу користувача кожного разу, коли змінюється значення стану, було б дуже неефективним підходом до рендерингу та оновлення інтерфейсу користувача з точки зору накладних витрат. Compose уникає цих накладних витрат, використовуючи техніку, що називається інтелектуальною рекомпозицією, яка включає лише рекомпозицію тих функцій, на які безпосередньо впливає зміна стану. Іншими словами, рекомпозиція буде виконуватись лише для тих композиційних функцій, які спостерігають і читають значення стану, коли значення цього стану змінюється.

Односпрямований потік даних. Односпрямований потік даних - це підхід до розробки додатків, згідно з яким стан, що зберігається в композиційній функції, не повинен безпосередньо змінюватися в жодній з дочірніх композиційних функцій [3, 6]. Розглянемо, наприклад, композиційну функцію з іменем FunctionA, що містить значення стану у вигляді булевого значення switchState. Ця композиційна функція викликає іншу композиційну функцію з ім'ям FunctionB, яка містить компонент Switch (перемикач). Мета полягає в тому, щоб перемикач оновлював значення стану кожного разу, коли користувач змінює положення перемикача. У цій ситуації, при дотриманні односпрямованого потоку даних, забороняється FunctionB безпосередньо змінювати значення стану. Замість цього, FunctionA оголошує обробник події (зазвичай у формі лямбда-функції) і передає його як аргумент дочірній композиційній функції разом зі значенням стану. Switch всередині FunctionB повинен тоді бути налаштований на виклик обробника події кожного разу, коли змінюється положення перемикача, передаючи йому поточне встановлене значення. Обробник події у FunctionA оновить стан новим значенням.

Як правило, дані передаються вниз через дерево ієрархії композиційних функцій, тоді як події розповсюджуються вгору зворотніми викликами обробників у батьківських компонентах, як показано на рис. 2.

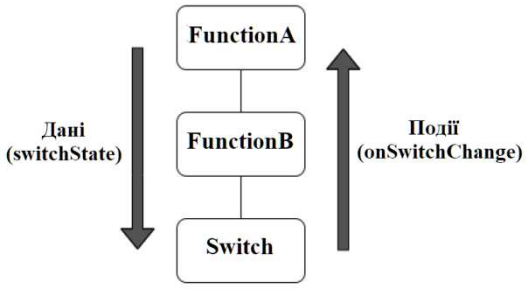

Рисунок 2 - Розповсюдження даних і подій в ієрархії композиційних функцій

Ключовим моментом, на який слід звернути увагу щодо цього процесу, є те, що значення, присвоєне switchState, змінюється лише всередині функції FunctionA і ніколи не оновлюється безпосередньо функцією

FunctionB. Установка Switch не перекидається з положення «on» у положення «off» безпосередньо за допомогою функції FunctionB(). Замість цього стан змінюється шляхом зворотного виклику обробника подій, розташованого у FunctionA, і дозволу рекомпозиції для регенерації Switch з новою установкою положення.

Архітектура Android-додатків в Compose. Ще кілька років тому Google не рекомендувала конкретного підходу до створення додатків для Android, окрім надання інструментів і наборів для розробки, дозволяючи розробникам вирішувати, що найкраще підходить для конкретного проекту або індивідуального стилю програмування. Це змінилося в 2017 році з появою компонентів архітектури Android, які стали частиною Android Jetpack, коли він був випущений у 2018 році [4]. Звичайно, з тих пір Jetpack був розширений, в основному за рахунок додавання UI-фреймворка Compose.

До появи Jetpack найпоширенішою була архітектура, при якій додатки складалися з декількох видів активностей (по одній на кожен екран у додатку), причому кожен клас активності певною мірою змішував інтерфейс користувача та код серверної обробки. Такий підхід призвів до низки проблем, пов'язаних із життєвим циклом додатка (наприклад, активність знищується та створюється заново щоразу, коли користувач обертає пристрій, що призводить до втрати будь-яких даних додатка, які не були збережені в деякому різновиді пам’яті постійного зберігання), а також до таких проблем, як неефективна навігація, пов'язана із запуском нової активності для кожного екрана додатка, до якого користувач має доступ.

На самому базовому рівні Google тепер виступає за додатки з однією активністю, де різні екрани завантажуються як контент в рамках цієї активності. Сучасні методичні вказівки з архітектури також рекомендують розділяти різні зони відповідальності в додатку на абсолютно окремі модулі (концепція, яка називається «separation of concerns (поділ відповідальності)»). Одним із ключів до такого підходу є компонент ViewModel.

Компонент ViewModel. Мета ViewModel полягає в тому, щоб відокремити модель даних і логіку додатка, пов'язану з інтерфейсом користувача, від коду, відповідального за відображення та управління інтерфейсом користувача та взаємодію з операційною системою [7]. Якщо додаток спроектований таким чином, він складатиметься з одного або кількох *UI*-контролерів, таких як активність, а також екземплярів ViewModel, які відповідають за обробку даних, потрібних цим контролерам.

Компонент ViewModel реалізується як окремий клас і включає змінні стану, що містять дані моделі та функції, які можуть бути викликані для управління цими даними. Активність, яка містить інтерфейс користувача, спостерігає (*observes*) за значеннями стану моделі так, що будь-які зміни значень ініціюють рекомпозицію. Події інтерфейсу користувача, пов'язані з даними моделі, такі як натискання кнопки, налаштовуються на виклик відповідної функції в ViewModel. Це, по суті, пряма реалізація концепції односпрямованого потоку даних. Діаграма на рис. 3 ілюструє цю концепцію, і як вона пов'язує активність та ViewModel додатка.

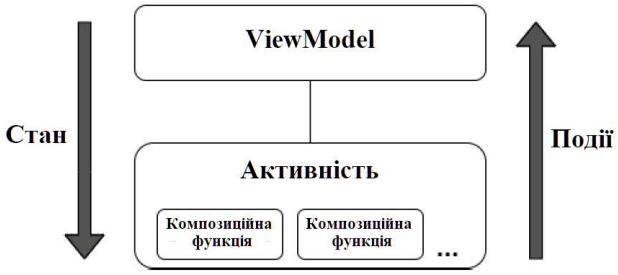

Рисунок 3 - Взаємодія ViewModel, активності та композиційних функцій в сучасному Android-додатку

Такий поділ відповідальності вирішує питання, пов'язані з життєвим циклом активностей. Незалежно від того, скільки разів протягом життєвого циклу додатка відтворюється активність, екземпляри ViewModel залишаються в пам'яті, зберігаючи таким чином узгодженість даних. Наприклад, екземпляр ViewModel, який використовується активністю, залишатиметься в пам'яті, доки активність не фінішує, а в додатках з однією активністю — доти, доки додаток не завершить роботу.

Основним призначенням ViewModel є зберігання даних, які можуть спостерігатися (observed) інтерфейсом користувача активності. Це дозволяє інтерфейсу користувача реагувати на зміни в даних ViewModel. Існує два способи оголошення даних у ViewModel, щоб їх можна було спостерігати. Одним з варіантів є використання механізму стану Compose. Альтернативним підходом є використання Jetpack-компонента LiveData.

Подібно до стану, оголошеного в композиційних елементах, стан ViewModel оголошується за допомогою групи функцій mutableStateOf(). Наступне оголошення ViewModel, наприклад, оголошує стан, що містить цілочисельне значення count з початковим значенням 0:

```
class MyViewModel : ViewModel() {
    var customerCount by mutableStateOf(0)
}
```

Після того, як якісь дані інкапсульовані в моделі, наступним кроком є додавання функції, яку можна викликати з інтерфейсу користувача для зміни значення лічильника:

```
class MyViewModel : ViewModel() {
    var customerCount by mutableStateOf(0)
    fun increaseCount() {
        customerCount++
    }
}
```

Навіть комплексні моделі є нічим іншим, як продовженням цих двох основних будівельних блоків стану і функції.

Від ViewModel мало користі, якщо цей модуль не можна використовувати в композиційних елементах, з яких складається інтерфейс користувача додатка. Все, що для цього потрібно, це передати екземпляр ViewModel як аргумент до композиційного елемента, з якого можна отримати доступ до значень стану та функцій. Угода по програмуванню рекомендує, щоб ці кроки виконувалися в композиційних елементах, призначених виключно для цієї задачі і розташованих на вершині ієрархії композиційних елементів екрана. Потім стан моделі та функції обробників подій можуть бути передані дочірнім композиційним елементам, якщо це необхідно.

Реалізація ViewModel із використанням LiveData. Jetpack-компонент LiveData з'явився ще до появи Compose і може використовуватися як обгортка навколо значень даних у моделі представлення. Після того, як ці дані розміщені в екземплярі LiveData в якості змінних, вони стають спостережуваними (observable) для композиційних елементів в межах активності. Екземпляри LiveData можуть бути оголошені змінюваними (mutable), використовуючи клас MutableLiveData, що дозволяє функціям ViewModel вносити зміни до основних значень даних. Приклад моделі представлення, призначеної для зберігання імені клієнта, може, наприклад, бути реалізований таким чином, використовуючи MutableLiveData замість state:

```
class MyViewModel : ViewModel() {
    var customerName: MutableLiveData<String> = MutableLiveData("")
    fun setName(name: String) {
        customerName.value = name
    }
}
```

Треба звернути увагу, що нові значення повинні бути присвоєні змінній живих даних (live data) за допомогою властивості value.

Як і у випадку зі станом, першим кроком при роботі з LiveData є отримання екземпляра моделі представлення при ініціалізації композиційного елемента. Після того, як отримано доступ до екземпляра моделі представлення, наступним кроком буде зробити живі дані доступними для спостереження. Це досягається шляхом виклику методу observeAsState() на об'єкті живих даних.

Висновки. Jetpack Compose реалізує інший підхід до розробки додатків, ніж той, який пропонує редактор екранних макетів Android Studio. Замість того, щоб напряму реалізовувати методи, якими інтерфейс користувача повинен рендеритися, Compose дозволяє оголошувати візуальний інтерфейс додатка в описових термінах, а потім вирішує, як найкраще виконати рендеринг під час роботи додатка. Compose також є керованим даними, оскільки зміни даних управляють поведінкою та зовнішнім виглядом додатка. Це досягається за допомогою змінних стану і рекомпозиції.

Рекомпозиція — одна з найважливіших концепцій Compose. Це механізм, який використовує композиційні елементи для оновлення інтерфейсу користувача на основі змін стану додатка. Рекомпозиція дає можливість повторно викликати будь-яку композиційну функцію в будь-який час для повторного рендерингу компонента на основі нових даних. Щоразу, коли стан змінюється, Compose повторно викликає всі композиційні елементи, які залежать від цього стану, і оновлює інтерфейс користувача. Треба мати на увазі, що ця концепція означає, що не потрібно вручну програмувати оновлення інтерфейсу користувача, коли змінюється стан додатка Android.

При розробці додатків за допомогою Compose важливо мати чітке розуміння того, як стан додатка і рекомпозиція працюють разом, щоб гарантувати, що інтерфейс користувача завжди є актуальним. Цьому сприяють концепція односпрямованого потоку даних, які течуть через композиційну ієрархію зверху вниз, тоді як зміни даних здійснюються шляхом виклику вгору обробників подій, оголошених у функціях-предках зі станом. Важливою метою при написанні композиційних функцій є максимальне їх повторне використання. Цього можна досягти, зокрема, шляхом підняття стану з поточної композиційної функції до викликаючої батьківської або вищої функції в композиційній ієрархії.

До недавнього часу Google, як правило, не рекомендував будь-який конкретний підхід до структурування додатка для Android. Все змінилося з появою фреймворка Android Jetpack, який складається з набору інструментів, компонентів, бібліотек і методичних вказівок щодо архітектури. Ці архітектурні вказівки рекомендують розділяти проект додатку на окремі модулі, кожен з яких відповідає за певну область функціональності (separation of concerns). Зокрема, рекомендується відокремлювати модель даних представлення додатка від коду, який відповідає за обробку інтерфейсу користувача. Це може бути досягнуто за допомогою компонента ViewModel, який може спостерігати за змінами та отримувати доступ до даних моделі представлення з активності, використовуючи як стан, так і модуль живих даних LiveData.

Література

1. Thinking in Compose [Електронний ресурс] / Android developers. - Режим доступу: https://developer.android.com/jetpack/compose/mental-model
2. Jetpack Compose Tutorial [Електронний ресурс] / Android developers. - Режим доступу: https://developer.android.com/jetpack/compose/tutorial
3. Smyth N. Jetpack Compose 1.4 Essentials / N. Smyth. - Cary, NC: Payload Media Inc., 2023. – 606 p.
4. Getting started with Android Jetpack [Електронний ресурс] / Android developers. - Режим доступу: https://developer.android.com/jetpack/getting-started
5. Künneth T. Android UI Development with Jetpack Compose. Second Edition / T. Künneth. - Birmingham: Packt Publishing Ltd., 2023. - 278 p.
6. Buketa D. Jetpack Compose by Tutorials. Second Edition / D. Buketa, P. Prasad. – McGaheysville, VA: Kodeco Inc., 2023. – 441 p.
7. Wambua M. S. Modern Android 13 Development Cookbook / M. S. Wambua. - Birmingham: Packt Publishing Ltd., 2023. - 322 p.

References

1. Thinking in Compose [Electronic resource] / Android developers. - Access mode: https://developer.android.com/jetpack/compose/mental-model
2. Jetpack Compose Tutorial [Electronic resource] / Android developers. - Access mode: https://developer.android.com/jetpack/compose/tutorial
3. Smyth N. Jetpack Compose 1.4 Essentials. / N. Smyth. - Cary, NC: Payload Media Inc., 2023. – 606 p.
4. Getting started with Android Jetpack [Electronic resource] / Android developers. - Access mode: https://developer.android.com/jetpack/getting-started
5. Künneth T. Android UI Development with Jetpack Compose. Second Edition / T. Künneth. – Birmingham: Packt Publishing Ltd., 2023. - 278 p.
6. Buketa D. Jetpack Compose by Tutorials. Second Edition / D. Buketa, P. Prasad. – McGaheysville, VA: Kodeco Inc., - 2023. – 441 p.
7. Wambua M. S. Modern Android 13 Development Cookbook / M. S. Wambua. – Birmingham: Packt Publishing Ltd., 2023. - 322 p.

Jetpack Compose is a completely new approach to app development for all Google operating system platforms. The main goals of the Compose UI framework are to make app development easier, faster, and less prone to common mistakes that commonly occur when developing software projects. Many of Compose's advantages come from the fact that it is both declarative and data-flow driven. Compose's declarative syntax provides a completely different way to implement the layouts and behavior of user interfaces than before. Instead of manually designing the intricate details of the components appearance that make up the next scene, Compose lets you describe scenes using a simple and intuitive syntax. In other words, Compose allows you to create screens by declaring what the user interface should look like without worrying about the complexity of how each screen is built. Once these declarations are written, all the intricate and complex details of positioning, constraints, rendering, and screen recomposition are automatically handled by the Compose runtime. With the introduction of Jetpack Compose, it is recommended to develop modern Android applications with only one activity, where different user interaction screens are loaded as content within that activity. In addition, the new application architecture recommends separating different areas of responsibility in the application into completely separate modules (separation of concerns). One key to this approach is the ViewModel component, which is implemented as a separate class and includes state variables containing model data and functions that can be called to manage that data. The activity that contains the user interface observes the state values and functions so that any changes in the values in the model trigger a screen recomposition.
***Keywords**: user interface (UI), widget, layout, declarative syntax, data flow, rendering, recomposition.*

Щербаков Є.В. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: gkvarc@gmail.com

Щербакова М.Є. – к.т.н., доцент, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету ім. В. Даля, e-mail: m.shcherbakova432@gmail.com