https://doi.org/10.31891/2307-5732-2023-317-1-224-229
УДК 004.41:004.9

КОЗУБ Юрій
Луганський національний університет імені Тараса Шевченка
https://orcid.org/0000-0002-3366-6031
e-mail: kosub.yg@gmail.com
КОЗУБ Галина
Луганський національний університет імені Тараса Шевченка
https://orcid.org/0000-0001-5387-050X
e-mail: kozubg@luguniv.edu.ua

ОСОБЛИВОСТІ РОЗРОБКИ МУЛЬТИПЛАТФОРМНИХ ЗАСТОСУНКІВ НА KOTLIN

Проведено дослідження методології розробки мультиплатформних застосунків на мові програмування Kotlin. Представлено методику розробки мультиплатформного застосунку для операційних систем Windows, Android, macOS та Linux, що дозволяє створювати музичні мелодії у новому форматі. Досліджено принципи декларативного програмування та фреймворки для мультиплатформної розробки. Для програмної реалізації окремих нативних додатків, обрано нативні рішення. Такому рішенню сприяє використовування фреймворків Kotlin Multiplatform та Compose Multiplatform. Kotlin Multiplatform дозволяє створювати універсальний код логіки мультиплатформного застосунку, у поєднанні з UI фреймворком Compose Multiplatform. Використання такого поєднання дає можливість написання єдиного коду логіки та інтерфейсу застосунку для декількох платформ одночасно, що допомагає економити час і уникати значної кількості помилок. Описано архітектурний патерн MVI, який найкраще підходить під декларативний стиль мультиплатформного фреймворку Compose Multiplatform. Розглянуто шаблон проєктування Dependency Injection, а також інші засоби проєктування мультиплатформних застосунків, таких як: бібліотека Kotlin Coroutines для підтримки асинхронності і паралельних обчислень у Kotlin, система збірки проєктів Gradle Kotlin DSL, бібліотека Decompose та фреймворк MVIKotlin. Запропоновано методику розробки архітектури компонентів Android-додатку. Модульна структура архітектури проєкту поділена на дві частини: на common модуль, який містить у собі основну логіку додатку, а також платформні реалізації компонентів, та платформні модулі, що виконують ініціалізацію та запуск застосунку на певній платформі. Для налагодження доступу до репозиторіїв з компонентів бізнес-логіки застосунку, використовується впровадження залежностей. Логіка впровадження залежностей описується у файлах-модулях Koin. Запропоновано методику, що узагальнює методологію розробки мультиплатформних застосунків на мові програмування Kotlin для розробки інтерфейсів користувача під декілька платформ.

Ключові слова: Kotlin, Gradle, Common, Jetpack Compose, Dependency Injection, Kotlin Multiplatform, Compose Multiplatform, Koin.

KOZUB Yurii, KOZUB Halyna
Luhansk Taras Shevchenko National University

FEATURES OF MULTIPLATFORM APPLICATION DEVELOPMENT ON KOTLIN

A study of the methodology of developing multi-platform applications using the Kotlin programming language was conducted. The method of developing a multi-platform application for Windows, Android, macOS and Linux operating systems is presented, which allows you to create musical melodies in a new format. The principles of declarative programming and frameworks for multi-platform development are studied. For the software implementation of individual native applications, native solutions have been chosen. This solution is facilitated by the use of Kotlin Multiplatform and Compose Multiplatform frameworks. Kotlin Multiplatform allows you to create a universal logic code of a multiplatform application, in combination with the Compose Multiplatform UI framework. Using such a combination makes it possible to write a single logic code and application interface for several platforms at the same time, which helps to save time and avoid a significant number of errors. The MVI architectural pattern is described, which best fits the declarative style of the Compose Multiplatform framework. The Dependency Injection design pattern is considered, as well as other tools for designing multiplatform applications, such as: the Kotlin Coroutines library to support asynchrony and parallel computing in Kotlin, the Gradle Kotlin DSL project assembly system, the Decompose library, and the MVIKotlin framework. A methodology for developing the architecture of Android application components is proposed. The modular structure of the project architecture is divided into two parts: the common module, which contains the main logic of the application, as well as platform implementations of components, and platform modules that perform initialization and launch of the application on a certain platform. Dependency implementation is used to establish access to repositories from application business logic components. The logic of implementing dependencies is described in Koin module files. A methodology is proposed that summarizes the methodology of developing multi-platform applications in the Kotlin programming language for developing user interfaces for several platforms.

Key words: Kotlin, Gradle, Common, Jetpack Compose, Dependency Injection, Kotlin Multiplatform, Compose Multiplatform, Koin.

Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Поширення смартфонів призвело до розробки великої кількості програмних застосунків, що робить їх розробку важливою сферою. Оскільки один і той самий сервіс у формі застосунку потрібно розробляти на кількох платформах, почали з'являтися різні технології нативного рішення – спочатку кросплатформні, а тепер мультиплатформенні. Ці технології були розроблені головним чином для скорочення витрат і підвищення ефективності процесу розробки програм. Плюсами кросплатформової розробки є висока швидкість та низька вартість розробки. До недоліків відносяться: складна підтримка низкорівневих

платформних функцій; відносно низька продуктивність, у порівнянні з нативними застосунками; забирає більш місця у пам'яті; нові платформні можливості з'являються пізніше ніж у нативних застосунках.

Аналіз досліджень та публікацій

Серед розробників мобільних застосунків, спостерігається тенденція вміння швидко вносити зміни у застосунок на потребу ринку. Внаслідок чого виникає необхідність визначення ефективних підходів до розробки конкурентних мобільних мультиплатформних застосунків.

Виробник кожної платформи чи операційної системи (ОС), надає набір для розробки програмного забезпечення (SDK), який містить усе необхідне для розробки програм на певній платформі чи ОС. Кросплатформні фреймворки мають окремий SDK, який зазвичай є шаром поверх рідного SDK. При програмній реалізації окремих мобільних застосунків під декілька платформ, у розробників, часто виникають проблеми з нативною розробкою, кросплатформенність не може вирішити деякі з цих проблем за допомогою компромісів, зростають витрати на синхронізацію і мультиплатформенність може бути кращим рішенням [1, 2].

Програмна реалізація окремих мобільних застосунків під декілька платформ сприяє збільшенню витрат на їх підтримку. На швидкість розробки, впровадження нових функціональностей, якість та зручності використання програмного застосунку впливає застосування кросплатформових інструментів. Для прискорення процесу написання коду, краще використовувати Xamarin.Forms, в якому майже всі елементи повністю сумісні з будь-якими платформами. Xamarin – платформа від Microsoft для створення застосунків під Android, iOS, Windows, Linux, macOS, watchOS та tvOS. Xamarin включає єдину загальну кодову базу C# і надає можливість тестувати застосунки на декількох пристроях з використанням Xamarin Cloud. Xamarin має два основних інструменти: Xamarin.Android, Xamarin.iOS і Xamarin.Forms. По частині кросплатформної розробки Xamarin пропонує використовувати єдиний API Xamarin.Essentials. Xamarin.Android і Xamarin.iOS наділяють застосунок тими ж можливостями і інтерфейсом, які є у нативних рішень [3].

React Native – це платформа з відкритим кодом для розробки мобільних додатків, дозволяє писати застосунки для IOS, Android, Windows, Web, Windows Phone, VR, Android TV, macOS, tvOS на мові JavaScript. Середовище поставляється з великим набором готових компонентів, однак вони не завжди адаптуються під різні платформи, що вимагає додаткових коригувань в коді. В гонитві за продуктивністю розробники віддають перевагу саме цьому фреймворку. Native дозволяє використовувати кастомні модулі на мовах для нативної розробки, але їх необхідно писати окремо для кожної платформи [4].

Безкоштовний кросплатформовий фреймворк Flutter від Google з відкритим вихідним кодом для швидкої розробки застосунків під Android, iOS, Windows, Linux, macOS, Web та Google Fuchsia, використовує об'єктно-орієнтовану мову програмування Dart використовує один і той же код для всіх платформ, перевершує конкурентів і демонструє найвищу продуктивність завдяки власному движку рендерингу і сучасній мові Dart, яка була розроблена Google. Flutter включає сторонні SDK, API для 2D, анімації, власні віджети Material Design і надає можливість повторно використовувати існуючий код Java, Swift та Objective-C [5].

Незважаючи на значну кількість досліджень в області кросплатформного програмування проблема використання декларативного підходу до створення мультиплатформних застосунків потребує подальших досліджень.

Формулювання цілей статті

Метою роботи є дослідження особливостей технологій Kotlin при створенні мультиплатформних застосунків.

Виклад основного матеріалу

Для розробки мультиплатформних застосунків використовується мова програмування Kotlin інструменти та бібліотеки з підтримкою Kotlin Multiplatform, такі як: Kotlin Coroutines, UI-фреймворк Jetpack Compose, патерн MVI, бібліотеки SQLDelight та Koin.

Kotlin – кросплатформова, статично типізована мова програмування загального призначення з підтримкою виводу типів. Kotlin розроблений для повної взаємодії з Java, де версія стандартної бібліотеки Kotlin для JVM залежить від бібліотеки класів Java, але тип взаємодії дозволяє зробити його синтаксис більш коротким і лаконічним. Kotlin в основному орієнтований на JVM, але також компілюється на JavaScript (наприклад, для інтерфейсних Web-додатків з використанням React) або власного коду (через LLVM), наприклад, для власних додатків iOS, що використовують бізнес-логіку спільно з додатками Android. [2, 6].

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

У Kotlin підтримка асинхронності і паралельних обчислень втілена у вигляді корутин (coroutine). Корутина являє собою блок коду, який може виконуватися паралельно з іншим кодом. Базова функціональність, пов'язана з корутинами, зосереджена в бібліотеці kotlin.coroutines [7, 8]. Переваги

використання корутин полягає: у швидкому опрацюванні та витрачанні малої частини ресурсів процесора, у порівнянні зі звичайними потоками; займає менше місця у коді.

Для розробки багатомовного програмного забезпечення використовуємо інструмент автоматизації збірки проєктів Gradle. Він контролює процес розробки в завданнях компіляції та упаковки проєктів для їх тестування, розгортання та публікації. Підтримувані мови включають Kotlin (а також Java, Groovy, Scala), C/C++ та JavaScript. Іншою, якщо не основною функцією Gradle, є збір статистичних даних про використання бібліотек програмного забезпечення по всьому світу. Gradle ґрунтується на концепціях Apache Ant і Apache Maven, та представляє доменну мову на основі Groovy і Kotlin, на відміну від конфігурації проєкту на основі XML, використовуваної Maven. Gradle використовує орієнтований ациклічний граф для визначення порядку, в якому можуть виконуватися завдання, за допомогою забезпечення управління залежностями. Gradle працює на JVM.

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

DSL Kotlin від Gradle надає альтернативний синтаксис традиційному DSL Groovy з поліпшеними можливостями редагування в підтримуваних IDE, з чудовою підтримкою контенту, рефакторингом, документацією і багатьом іншим [9].

При створенні швидких і реактивних інтерфейсів для Android, Desktop та Web-додатків на Kotlin використовується фреймворк Compose Multiplatform, який складається з трьох частин: Jetpack Compose; Compose for Desktop; Composer for Web. За допомогою механізмів, наданих Kotlin Multiplatform, частини фреймворку стали об'єднані під одним плагіном Gradle і групою артефактів. Це спростило і прискорило розробку інтерфейсів додатків завдяки використанню єдиного коду під декілька платформ одночасно [10, 11]. В цільових платформах (targets) у Kotlin Multiplatform, на рис. 1 показано, як налаштовуються в нативний код необхідні нам операційні системи, де і компілюється код на Kotlin [6].

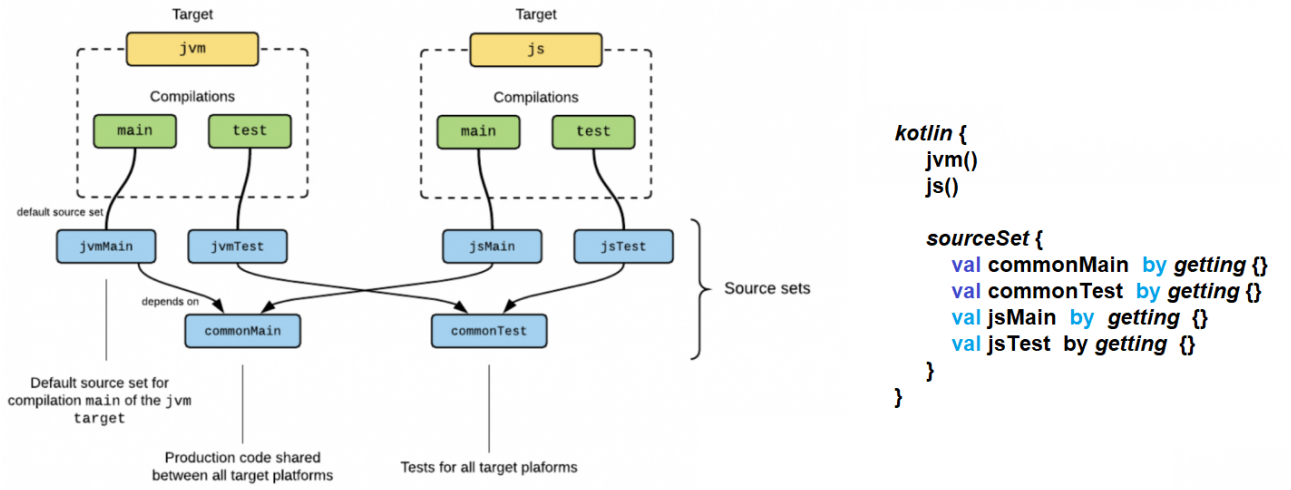

Рис. 1. Налаштування jvm та js targets [6]

Існує загальний набір source Sets вихідних кодів і платформних (їх стільки, скільки в проєкті targets, за цим стежить IDE). Для реалізації такої особливості використовується механізм Expect-actual. Expect-actual дозволяє із загального модуля звертатися до платформозалежного коду. Можна оголосити Expect декларацію в Common модулі і реалізувати її в платформних модулях [6].

Підтримка інших платформ реалізується аналогічно. За допомогою Kotlin Multiplatform витрачається менше часу на написання та підтримку одного й того ж коду для різних платформ і використовується механізм спільного коду. Фреймворк знаходиться на стадії розробки і, ще не вистачає готових рішень, але він, вже, має свої суттєві переваги: єдиний код, який можна в будь-який час дописувати і змінювати; поділ бізнес-логіки між платформами зі збереженням власного коду для кожного клієнтського інтерфейсу; будь-які зміни та коригування відбуваються одночасно на всіх платформах. Розглянемо приклад використання механізму Kotlin Multiplatform для отримання дати на пристроях:

```
Common:
internal expect val timestamp: Long

Android/JVM:
Internal actual val timestamp: Long
```

```
get() = java.lang.Ayatem.currentTimeMikkis

ios:
internal actual val timestamp: Long
get() = platform.Fundation.NSDate().timAnternalSince1970.toLong
```

Для програмного забезпечення (ПЗ), якому потрібна постійна підтримка, при написанні коду необхідно мінімізувати зв'язаність і максимізувати його цілісність. Щоб згрупувати якомога більше зв'язаного коду разом, підтримувати та масштабувати його, коли застосунок зростає, користуємося добре відомим принципом розробки ПЗ - Separation of concerns. Для опису інтерфейсу користувача використовуються Composable функції, що заснованні на функціональному підході до програмування у Jetpack Compose:

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

У цьому прикладі Composable функція отримує дані як параметри з класу appData та перетворює їх у елемент інтерфейсу користувача. Отже, при використанні будь-якого коду на Kotlin, обрані дані застосовують їх для опису ієрархії (наприклад: виклик Composable-функцій Header() і Body()). Це означає, що при виклику інших Composable-функцій, вони відображають структуру UI, при використанні надаються всі примітиви Kotlin, включаючи оператори if і цикли for для управління структурою UI, щоб вирішувати більш складну логіку інтерфейсу користувача.

Переваги Jetpack Compose полягають у його незалежності від конкретних версій цільової платформи; вся робота з UI виконується за допомогою мови програмування Kotlin; використання композиції замість спадкування. UI-компонент описується у вигляді функції з анотацією Composable, яка відповідає тільки за обмежений функціонал, тобто без зайвої логіки; однонаправленість потоку даних; зменшення кількості коду для UI-логіки. Таке прагматичне рішення, що спрощує інкапсуляцію стану в Composable функціях при зберіганні між викликами, його сигнатуру що дозволяє виконувати деякі оптимізації, особливо зручно використовувати для анімацій та інших речей, які не потрібно змінювати або спостерігати зовні [12].

Статично типізовану мову програмування Kotlin використано для розробки мультиплатформних застосунків. Вона працює поверх JVM та компілюється в JavaScript [6]. При використанні Jetpack Compose обрано патерн MVI, який найкраще підходить для опису логіки інтерфейсу користувача [13]. Взаємодія з інтерфейсом користувача у MVI обробляється бізнес-логікою, це вносить зміни в стан, що впливає на відображення інтерфейсу користувача та призводить до односпрямованого і циклічного потоку даних. За допомогою засобів Kotlin (Flow, StateFlow та Channel) реалізовано реактивне отримання оновлених даних [14].

Для налагодження написання спільного коду, обрано потужні інструменти, такі як логування та “подорож у часі”, що надає фреймворк MVIKotlin при використанні шаблону MVI для Kotlin Multiplatform. MVIKotlin не застосовує жодної конкретної архітектури. Його відповідальність описується тим, що надає одне джерело істини для стану, забезпечує абстракцію для UI з ефективними оновленнями та забезпечує прив’язку до життєвого циклу між входами та виходами.

Для опису контракту бізнес-логіки компонентів інтерфейсу мультиплатформного застосування, обрано модель Store [13]. Наступний код демонструє реалізацію контракту бізнес-логіки компонентів у інтерфейсах Store, що виконується у класах MusicStore з розробленого ПЗ “Composer” [15]:

```
interface MusicStore : Store<MusicStore.Intent, MusicStore.State, Nothing> {
  sealed class Intent {
    data class TabChanged(val newTab: Tab) : Intent()
    object SearchClick : Intent()
    data class TypeOfSortChanged(val typeOfSort: TypeOfSort) : Intent()
    data class SearchFilterChanged(val searchFilter: String) : Intent()
    object SearchCancelClick : Intent()
    data class SelectedItemsChanged(val selectedItems: Int) : Intent()
    object RemoveSelection : Intent()
  }
  data class State(
    val selectedTab: Tab = Tab.AllMusic,
    val currentTypeOfSort: TypeOfSort = TypeOfSort.Ascending,
    val screenState: MusicScreenState = MusicScreenState.Default
  )
```

```
  enum class Tab {
    AllMusic, AllPlaylists, AllSamples
  }
  sealed class MusicScreenState {
    object Default : MusicScreenState()
    data class Search(val searchFilter: String = "") : MusicScreenState()
    data class ItemsSelection(val selectedItems: Int = 1) : MusicScreenState()
  }
}
```

Налаштування спільного модуля додатку виконано у файлі build.gradle.kts модуля common. Налаштування Gradle android додатку виконані у файлі build.gradle.kts модуля android. Налаштування Gradle desktop додатку виконані у файлі build.gradle.kts модуля desktop.

Архітектура проєкту поділена на дві частини:

1. Common модуль містить у собі основну логіку застосунку, а також платформні реалізації компонентів;

2. Платформні модулі виконують ініціалізацію головного компоненту застосунку зі спільного модуля, вмикають DI та запускають застосунок.

Впровадження залежностей застосунку здійснюється за допомогою шаблону проєктування Dependency Injection, щоб отримати поділ завдань на створення та використання об'єктів та підвищити читабельність і повторне використання коду. Це шаблон проєктування, при якому об'єкт отримує інші об'єкти, від яких він залежить. Як правило, приймаючий об'єкт називається клієнтом, а переданий об'єкт називається сервісом. Код, який передає сервіс клієнту, називається інжектором. Замість того, щоб клієнт вказував, який сервіс він буде використовувати, інжектор повідомляє клієнту, який сервіс використовувати. Впровадження відноситься до передачі залежності (сервісу) клієнту, який її використовує. Сервіс стає частиною стану клієнта. Передача сервісу клієнту, замість дозволу клієнту створювати або знаходити сервіс, є основною вимогою шаблону.

Для налагодження доступу до репозиторіїв з компонентів бізнес-логіки ПЗ, використовується впровадження залежностей. Логіка впровадження залежностей описується у файлах-модулях Koin. DatabaseModule описує створення об’єкту бази даних для кожної платформи окремо. PreferencesModule створює об’єкт Settings, який використовується для роботи з платформними сховищами налаштувань додатку:

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

RepositoryModule реалізує створення об’єктів репозиторіїв:

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

SoundMashineModule створює об’єкт утиліти для відтворення звуків:

- Контракт у common модулі:

```
expect val soundMachineModule: Module
```

- Реалізація для android:

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

- Реалізація для desktop:

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

Всі модулі впровадження залежностей ініціюються у функції initKoin:

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

Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі

В роботі розглянуто принципи декларативного підходу для розробки інтерфейсів користувача під декілька платформ. Описано відмінності для різних платформ для системи налаштування Gradle, застосування Dependency Injection та логіку впровадження залежностей у файлах-модулях Koin в застосуванні. Проведено аналіз сучасних інструментів та методик розробки мультиплатформних застосунків: Compose Multiplatform, Jetpack Compose, MVIKotlin, Dependency Injection. За запропонованою методикою, що узагальнює методологію розробки мультиплатформних застосунків на мові програмування

Kotlin, з використанням декларативного стилю мультиплатформного фреймворку Compose Multiplatform створено проєкт структури застосунку.

Література

1. Nagy R. Simplifying Application Development with Kotlin Multiplatform Mobile. Packt, 2022. 184 p. https://www.packtpub.com/product/simplifying-application-development-with-kotlin-multiplatform-mobile/978180181 2580.
2. Soshin A. Kotlin Design Patterns and Best Practices - Second Edition. Packt. 2022. 356 p.
3. Xamarin documentation. URL: https://docs.microsoft.com/en-us/xamarin/ React Native. URL: https://reactnative.dev/
4. React Native. URL: https://reactnative.dev/
5. Flutter. URL: https://flutter.dev/
6. Kotlin Programming Language. URL: https://kotlinlang.org
7. Coroutines. URL: https://kotlinlang.org/docs/coroutines-overview.html
8. Корутины. URL: https://metanit.com/kotlin/tutorial/8.1.php
9. Gradle Kotlin DSL Primer. URL: https://docs.gradle.org/current/ userguide/kotlin_dsl.html
10. Compose Multiplatform. URL: https://www.jetbrains.com/ru-ru/lp/compose-mpp/
11. Build better apps faster with Jetpack Compose. URL: https://developer.android.com/jetpack/compose
12. Understanding Jetpack Compose–part 1 of 2. URL: https://medium.com/androiddevelopers/ understanding- jetpack-compose-part-1-of-2-ca316fe39050
13. MVIKotlin Overview. URL: https://arkivanov.github.io/MVIKotlin/
14. Козуб, Г., Козуб Ю., Могильний Г., Жуков А. Розробка мобільного Android-додатку з застосуванням принципів Clean Architecture. Вісник Східноукраїнського національного університету імені Володимира Даля, вип. 5 (269), Вересень 2021, С. 5–10.
15. Жуков А. В., Козуб Г.О. Застосування фреймворку Jetpack Compose у багатомодульному Android-додатку. Вітчизняна наука на зламі епох: проблеми та перспективи розвитку : зб. наук. праць. Переяслав, 2021. Вип. 67. С. 109–111.

References

1. Nagy R. Simplifying Application Development with Kotlin Multiplatform Mobile. Packt, 2022. 184 p. https://www.packtpub.com/product/simplifying-application-development-with-kotlin-multiplatform-mobile/978180181 2580.
2. Soshin A. Kotlin Design Patterns and Best Practices - Second Edition. Packt. 2022. 356 p.
3. Xamarin documentation. URL: https://docs.microsoft.com/en-us/xamarin/ React Native. URL: https://reactnative.dev/
4. React Native. URL: https://reactnative.dev/
5. Flutter. URL: https://flutter.dev/
6. Kotlin Programming Language. URL: https://kotlinlang.org
7. Coroutines. URL: https://kotlinlang.org/docs/coroutines-overview.html
8. Корутины. URL: https://metanit.com/kotlin/tutorial/8.1.php
9. Gradle Kotlin DSL Primer. URL: https://docs.gradle.org/current/ userguide/kotlin_dsl.html
10. Compose Multiplatform. URL: https://www.jetbrains.com/ru-ru/lp/compose-mpp/
11. Build better apps faster with Jetpack Compose. URL: https://developer.android.com/jetpack/compose
12. Understanding Jetpack Compose–part 1 of 2. URL: https://medium.com/androiddevelopers/ understanding- jetpack-compose-part-1-of-2-ca316fe39050
13. MVIKotlin Overview. URL: https://arkivanov.github.io/MVIKotlin/
14. Kozub, H., Kozub Yu., Mohylnyi H., Zhukov A. Rozrobka mobilnoho Android-dodatku z zastosuvanniam pryntsypiv Slean Architecture. Visnyk Skhidnoukrainskoho natsionalnoho universytetu imeni Volodymyra Dalia, vyp. 5 (269), Veresen 2021, S. 5–10.
15. Zhukov A. V., Kozub H.O. Zastosuvannia freimvorku Jetpack Compose u bahatomodulnomu Android-dodatku. Vitchyzniana nauka na zlami epokh: problemy ta perspektyvy rozvytku : zb. nauk. prats. Pereiaslav, 2021. Vyp. 67. S. 109–111.