Міністерство освіти та науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«РОЗРОБКА КРОСПЛАТФОРМЕНОГО ФРЕЙМВОРКУ МОВОЮ KOTLIN»**

Виконав: студент 4-го року навчання,

Спеціальності

121 Інженерія програмного забезпечення

Козир Владислав Ігорович

Керівник _Радзієвська О.В_____

асистент

Рецензент _____

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК_____

«____» _____ 20____р.

Київ – 2024

ЗМІСТ

АНОТАЦІЯ

Робота присвячена дослідженню можливостей кросплатформеного програмування для ефективної розробки мобільних застосунків. Було досліджено архітектурні шаблони для мобільних застосунків. Було досліджено особливості побудови Kotlin Multiplatform бібліотек. Було проведено адаптацію фреймворку TCA для розробки Android та iOS додатків та приведено приклад інтеграції в застосунок.

Ключові слова: KOTLIN MULTIPLATFORM, KOTLIN NATIVE, АРХІТЕКТУРНІ ШАБЛОНИ, ІНТЕГРАЦІЯ KOTLIN MULTIPLATFORM БІБЛІОТЕКИ, МОВА ПРОГРАМУВАННЯ KOTLIN, МОВА ПРОГРАМУВАННЯ SWIFT, ФРЕЙМВОРК TCA.

ВСТУП

Кожен бізнес прагне зменшення вартості та термінів розробки, щоб досягти максимально низького показнику Time To Market. Це зумовлено стрімкими змінами та бажанням бізнесу адаптуватися до них. За допомогою використання кросплатформенних рішень бізнеси мають змогу проводити цикл розробки, щонайбільше в 2 рази швидше, оскільки ці технології потребують написання коду тільки один раз. Також це дозволяє зменшувати розмір команди, щоб оптимізувати операційні витрати на ведення бізнесу, якщо існує така необхідність.

Кросплатформені рішення активно розвиваються та є актуальним інструментарієм для розробки мобільних застосунків. Певна кількість великих компаній зацікавлена в особистій розробці подібних рішень, що дозволяє очікувати підтримку розвитку. Google, Facebook, Microsoft, JetBrains - кожна з компаній вкладає ресурси у підтримку та адаптацію кросплатформенних рішень. Також наявність відкритого вихідного коду дозволяє покращувати ці технології й іншим спеціалістам галузі. Це є дуже важливою складовою, оскільки більшість рішень потребують сторонніх бібліотек або додаткових модулів для вирішення стандартних задач: доступу до API платформи, стандартизація підходів до розробки або рішення для побудови архітектури.

У даній роботі розглядаються різноманітні популярні кросплатформенні рішення та обгрунтування чому Kotlin Multiplatform є досить перспективним напрямком до розвитку подібних рішень. Оскільки, дана технологія тільки отримала статус Stable, у зв’язку з цим виникає потреба в зручних інструментах та фреймворках для стандартизації процесу розробки.

У першому розділі будуть розглянуті теоретичні особливості кросплатформенних рішень. Для цього будуть досліджені дані технології за списком критеріїв. Також буде досліджено технологію Kotlin Multiplatform, а саме проблематику розробки бібліотек для даного фреймворку та обґрунтування потреби в інструментах для стандартизації розробки.

У другому розділі будуть досліджено вимоги для архітектурних шаблонів в контексті використання для Kotlin Multiplatform. Для цього будуть досліджені основні архітектурні шаблони для розробки мобільних застосунків.

У третьому розділі буде описано розробку адаптації Swift фреймворку для даної технології. Для цього будуть досліджені особливості інтероперабельності Swift та Kotlin та виокремлено правила до використання публічного API Kotlin в iOS додатку. Будуть розглянуті функціональні вимоги та бібліотеки необхідні для розробки фреймворку. Результатом даного розділу є розробка даного рішення та приклад інтеграції в Android та iOS застосунки.

Актуальність дослідження полягає в тому, що безпосередня швидкість розробки залежить від наявного інструментарію та бібліотек. Відповідно стандартизація архітектурних підходів до розробки кросплатформенних рішень є важливою задачею для швидкого циклу розробки мобільних застосунків.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд ключових понять

Багатоплатформна мобільна розробка - це підхід, який дозволяє створити єдиний мобільний додаток, що безперебійно працює на декількох операційних системах. У крос-платформних додатках можна спільно використовувати частину або навіть весь вихідний код. Це означає, що розробники можуть створювати і розгортати мобільні ресурси, які працюють як на Android, так і на iOS, без необхідності перекодовувати їх для кожної окремої платформи.

Загалом, з точки зору користувача, різниці між кросплатформенністю та мультиплатформенністю немає. Однак розробники обирають кросплатформну розробку мобільних додатків, якщо вони використовують інструменти, які дозволяють писати код для багатьох платформ. З іншого боку, багатоплатформність означає, що код працює на багатьох платформах.

1.2 Дослідження мобільних кросплатформенних технологій

Розробники мобільних застосунків використовують кросплатформені рішення, щоб створювати додатки для різних платформ (Android, iOS) з спільною кодовою базою. Можливість розповсюджувати код між платформ є вагомою перевагою понад нативною розробкою. Наявність спільної кодової бази дозволяє економити час, оскільки з таким підходом написання коду та бізнес логіки потрібно виконувати тільки раз, а не для кожної з платформ окремо.

Задля оптимізації процесу мобільної розробки на ринку існує доволі велика кількість рішень, котрі надають інструментарій для написання

мультиплатформенных застосунків. В розрізі цього дослідження можемо розглянути кілька найпопулярніших рішень:

- Flutter
- React Native
- Kotlin Multiplatform
- .NET MAUI

Для аналізу та дослідження цих рішень визначимо критерії оцінки, щоб обгрунтувати .

- Інструментарій написання UI
- Швидкодія
- Інструментарій для налагодження застосунку
- Вплив на розмір додатку
- Можливості інтеграції нативних бібліотек

1.2.1 Flutter

Flutter - це найпопулярніший фреймворк для розробки мобільних, веб та десктоп застосунків з спільною кодовою базою станом на 2022 рік за інформацією статистики Statista [2]. Він розробляється компанією Google та був випущений в 2017 році. Розробка на базі даного фреймворку потребує написання коду мовою Dart.

Інструментарій для написання UI базується на використанні абстрактної композиції Widgets [3]. Для підтримки стандартних візуальних стилів для Android та iOS використовуються пакети flatter/material та flatter/cupertino, що дозволяє створювати візуальні компоненти системи наближеними до нативних та загальноприйнятих. Особливістю інструментарію написання UI є можливість hot reload - перегляд змін

компонентів без необхідності повної компіляції коду, що прискорює написання UI.

Фреймворк Flutter складається з декількох архітектурних блоків [4], а саме:

- Embedder - це платформо залежна частина для впровадження API кожної з платформ;
- Engine (C/C++) - це рушій фреймворку котрий за допомогою інструментарію Embedder має змогу відображати UI;
- Framework (Dart) - це загальний інструментарій для роботи з Engine, що виконує потребу написання спільного UI;

Оскільки, фреймворк працює через певні рівні абстракції, то це впливає на швидкодію в порівнянні з нативними аналогами. Особливо, ці обмеження швидкодії впливають на роботу з відображенням анімацій та зображень.

Flutter має обширну підтримку інструментів для налагодження застосунків, а саме:

- Fluter та Dart плагін використовується для налагодження вихідних кодів в Android Studio або VSCode;
- Flutter Inspector - це програмне забезпечення для налагодження ієрархії візуальних компонентів та виявлення проблем пов’язаних з швидкодією;
- Для віддаленого налагодження Flutter Engine, що працює в процесі Android додатку, використовується Flutter Gdb;

Розмір Flutter застосунку для режиму debug є доволі вагомим, оскільки підключаються необхідні залежності для можливості налагодження та hot reload. Попри це release застосунок має більший розмір ніж нативний, через необхідність використання платформо-залежного рушія.

Інтеграція платформенного API відбуваються за допомогою інструментарію Platform Channel, але воно має свої обмеження у вигляді необхідності розробки плагіну для цієї бібліотеки або API платформи.

1.2.2 React Native

React Native - це другий за популярністю фреймворк для розробки мобільних застосунків з спільним кодом, але наразі має низхідний тренд популярності за аналітикою Statista [2]. Це рішення має відкритий вихідний код та було розроблено компанією Facebook в 2015 році. Написання коду для React Native відбувається за допомогою JavaScript або TypeScript.

Інструментарій для написання UI компонентів використовується такий самий як і для веб-фреймворку React, а саме за допомогою components з стандартної бібліотеки. Як і у випадку з Flutter є змога використовувати hot reload функціонал для швидкого перегляду змін в UI компонентах.

Фреймворк React Native складається з декількох частин[5]:

- React Native Android / iOS host platform - це платформа для змоги відслідковування подій системи та відмальовування UI представлення;
- Fabric (C++) - це система для рендерингу представлення, що використовує API host platform;
- React - це загальний інструментарій для розробки представлення, що базується на API Fabric;

Оскільки, даний фреймворк, як і Flutter, використовує певний рівень абстракції над нативними елементами системи, то швидкодія не може бути повністю наближен до платформеного інструментарію.

З версії 0.62 React Native використовує за замовчування забезпечення Flipper для налагодження застосунку. Ця розробка Facebook має змогу проводити налогодження Android, iOS та React Native застосунків.

Даний фреймворк дозволяє додавати нестандартні налаштування до системи збірки артефактів під кожну з платформ, але необхідність рушія та API платформи все рівно збільшує розмір додатку в порівнянні з нативними аналогами.

React Native в своєму інструментарії дає можливість розширення за допомогою платформних бібліотек. Для для цього використовується Fabric Native Components та Turbo Native Module [6], але дане розширення вимагає додаткове написання модуля для можливості інтеграції в JS коді.

1.2.3 .NET MAUI

.NET MAUI - це кросплатформний фреймворк з відкритим вихідним кодом розроблений компанією Microsoft. Він дозволяє розробляти мобільні та десктопні застосунки за допомогою мови програмування C# та XAML. Ця технологія надає доступ до API платформи, такі як: локація, камера і т.д.

Даний фреймворк дає можливість розробки користувацького інтерфейсу за допомогою різноманітних компонентів з дизайн систем та XAML. Особливістю даного рішення є можливість використання фреймворків з UI компонентами для кожної з платформ. На ринку існує декілька вендорів дизайн систем для MAUI, що полегшує розробку та стандартизацію UI компонентів. Також варто звернути увагу на залежність від підтримки UI фреймворків для кожної з платформ, оскільки оновлення призводять до адаптації платформеного API.

Фреймворк .NET MAUI складається з декількох шарів та частин системи [7]:

- Mono Runtime - імплементація оточення .NET для кожної з платформ;
- .NET BCL - це технологія, що дозволяє використовувати спільну бізнес логіку;
- .NET for Android/iOS - це фреймворк за допомогою котрого розробники мають змогу взаємодіяти з API платформи

Оскільки, цей фреймворк, як і попередні представники мають власне оточення та абстрактний інструментарій для роботи на кожні з платформ, то це накладає додаткові обмеження на швидкодію.

Компанія Microsoft пропонує вбудований інструментарій для налагодження застосунків за допомогою середовища Visual Studio. Він включає інструменти для оцінювання швидкодії UI та налагодження спільного коду для бізнес логіки.

Даний фреймворк використовує технологію Native AOT, що використовується для виконання коду без .NET Runtime. Це призводить до значного збільшення додатку.

.NET MAUI для інтеграції бібліотек використовує схожий підхід до інтеграції як і React Native, тому це накладає додаткові вимоги до інтеграції таких рішень.

1.2.4 Kotlin Multiplatform

Kotlin Multiplatform - це технологія розроблена компанією JetBrains, котра має відкритий вихідних код. Особливість цієї розробки в можливості написання спільного Kotlin коду з можливістю інтеграції в нативні, веб та десктоп застосунки. Наразі, Kotlin Multiplatform є не настільки популярною,

оскільки тільки нещодавно в перейшла в stable версію [2]. Перспективність та подальший розвиток даної технології підтверджується тим, що Google оголосила підтримку для Android [8] і починає впровадження для стандартних бібліотек з пакету Jetpack.

Даний фреймворк не має інструментарію для написання UI компонентів, тобто у розробників є змога реалізовувати UI шар додатку за допомогою стандартних бібліотек для Android та iOS. На базі Kotlin Multiplatform будується додатковий фреймворк для побудови спільного UI - Compose Multiplatform. Наразі, підтримка даної технології знаходиться в Alpha статусі для iOS платформи [9], але це є доволі перспективним напрямком розвитку для реалізації спільного UI коду. В даному випадку великою перевагою є можливість вибору інструментів для цієї задачі, оскільки це дозволяє, в залежності від бізнес задач, мати змогу зменшувати час на розробку або використовувати нативні компоненти.

Оскільки, Kotlin код в рамках цієї технології компілюється в нативні артефакти, то швидкодія системи є співрозмірною з стандартними Android та iOS застосунками.

Інструментарій Kotlin Multiplatform дозволяє проводити налагодження як і для Android застосунків так і для Kotlin Native коду. Також є можливість віддаленого налагодження за допомогою GDB[10]. Наразі ці підходи та підтримка компілятором для Kotlin активно розвиваються, тому вони мають обмежені можливості для налагодження програмного забезпечення.

Оскільки, Kotlin Multiplatform компілює спільний код в артефакти для кожної з платформ, то у випадку з Android - маємо співрозмірний вплив на розмір додатку, а для iOS - фреймворк може мати більший вплив на розмір через особливості інтероперабельності Kotlin до Swift та додаткову мета інформацію.

За допомогою механізмів expect / actual розробники мають змогу перевикористовувати існуючі платформні залежності або бібліотеки. Цей підхід забезпечує зручне використання вже існуючих рішень або адаптацію API платформи.

У висновку, проаналізувавши дані технології за вище згаданими критеріями оцінки, можемо виділити ряд таких переваг:

- KMM має гнучкість щодо можливостей реалізації UI, оскільки є змога реалізації за допомогою нативних інструментів або адаптація Compose Multiplatform для швидкої розробки;
- KMM має співрозмірну швидкодію відносно нативних рішень, в противагу іншим описаним рішенням;
- KMM дає здатність легкої інтеграції з існуючими бібліотеками та API для кожної з платформ;
- KMM має мінімальний вплив на розмір артефакту для Android застосунку, але iOS частина може потребувати додаткової оптимізації. У випадку з іншими рішеннями - це завжди призводить до збільшення розміру в порівнянні з нативними аналогами.

1.3 Дослідження Kotlin Native

Kotlin Native - це технологія для компіляції Kotlin коду в нативні бінарні файли [11]. Це дозволяє виконувати код без віртуальної машини, як наприклад Java Virtual Machine. Дана технологія представляє собою бекенд, котрий базується на LLVM [12] для стандартного компілятор мови Kotlin. Також Kotlin Native включає реалізацію всієї стандартної бібліотеки Kotlin.

Дана технологія підтримує такі платформи:

- Android NDK

- Windows (MinGW)
- Linux
- macOS
- iOS, tvOS, watchOS

Надалі в роботі ми будемо фокусуватися саме на мобільних платформах.

Дана технологія ідеально підходить для випадків, коли є необхідність розробки програмного забезпечення без певної віртуальної машини або середовища для виконання.

Kotlin Native компілятор має змогу створювати:

- Apple Framework для Swift, Objective C
- Статичну та динамічну бібліотеки для C/C++ проєктів
- Виконувані файли для великої кількості популярних платформ

Особливість Kotlin Native є можливість підтримки взаємодії з нативними мовами програмування й навпаки - взаємодії з Kotlin кодом з нативної частини програми. Щоб досягти такої поведінки Kotlin Native підтримує інтероперабельність для використання існуючих рішень:

- Apple Framework для Swift, Objective C
- Статичну та динамічну бібліотеки для проєктів на C

Наприклад, результат роботи компілятора Kotlin Native ми маємо змогу легко включати в проєкти, котрі написані мовами: C/C++, Objective-C, Swift. Відповідно і в обернену сторону ми маємо змогу використовувати нативний код, Apple фреймворки, статичні та динамічні бібліотеки мовою C.

Задля підтримки платформ в пакет Kotlin Native включено додаткові бібліотеки. Наприклад, для екосистеми Apple: POSIX, gzip, OpenGL, Metal, Foundation [12].

Kotlin Native є невід'ємною складовою Kotlin Multiplatform. За допомогою цієї технології KMM має змогу надавати платформне API для спільного Kotlin коду. Це дозволяє писати спільну бізнес логіку без привязки до певної платформи, а деталі кожної з них реалізовувати в окремих модулях (Рисунок 1.1).

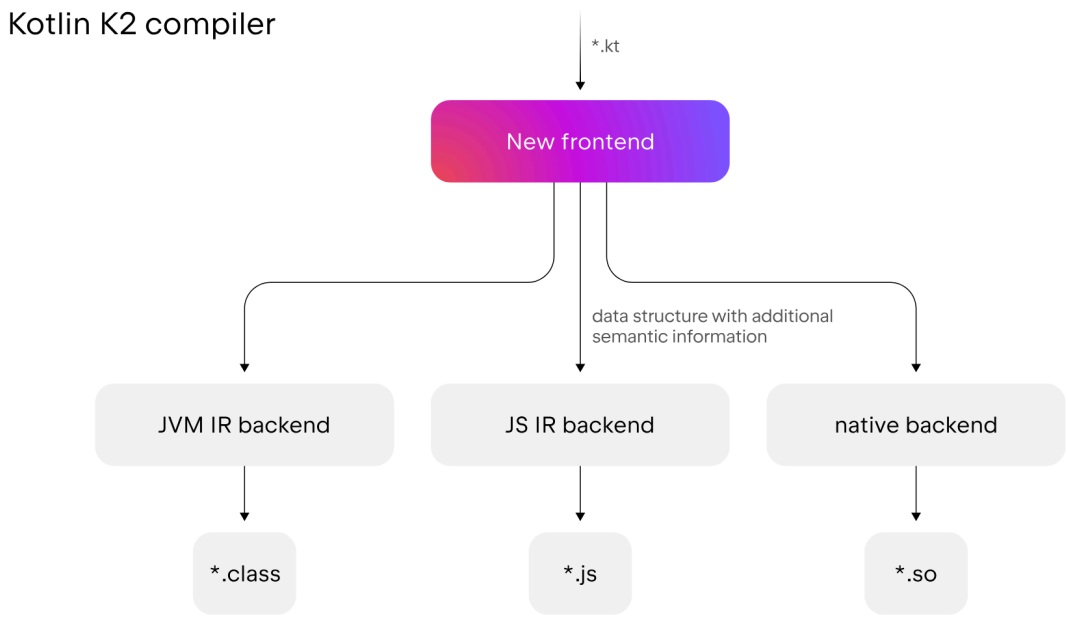

Рисунок 1.1 - Структура Kotlin K2 компілятора

При розробці за допомогою Kotlin Native та Kotlin Multiplatform можна виділити пару значних проблем:

- Кількість кросплатформенних бібліотек доволі обмежена і не всі з них нормально підтримують обидві платформи. Здебільшого дані рішення орієнтуються початково тільки для Android застосунків і реалізація та

адаптація iOS частини залишається незавершеною. Обмеження кількості сторонніх бібліотек є вагомою проблемою для певного типу задач при розробці подібних кросплатформенних рішень

- Попередня проблема безпосередня пов’язана з особливостями інтероперабельності Kotlin та Swift в Kotlin Native. Оскільки, мови доволі сильно відрізняються, то існує проблематика впровадження зручного та звичного API для iOS розробників
- Також варто відзначити певну проблему та нестабільність роботи середовища розробки Android Studio, а саме підтримку синтаксису для подібних проєктів. Це є вагомим джерелом проблем для авто доповнення імпортування існуючих бібліотек та роботи з переходження до вихідної реалізації певних об'єктів

Оскільки, Kotlin Multiplatform активно розвивається та має Stable версію, то дані проблеми вирішуються з ростом популярності даної технології. Також, як згадано було раніше, Google долучається до активної підтримки, що додасть певної популярності до технології. З ростом активною аудиторії розробників, котрі все більше будуть створювати бібліотек з вихідним кодом, Kotlin Multiplatform буде все більше привабливою для використання, навіть нових проєктів.

1.4 Постановка задачі

Проведений огляд даної предметної області показав, що мобільні інженери використовують кросплатформні фреймворки для мобільної розробки, щоб створювати нативні додатки для різних платформ, таких як Android та iOS, використовуючи єдину кодову базу. Спільний доступ до коду -

одна з ключових переваг цього підходу над нативною розробкою додатків. Наявність єдиної кодової бази означає, що мобільні інженери можуть заощадити час, уникаючи необхідності писати код для кожної операційної системи, що прискорює процес розробки.

В Kotlin Multiplatform для спільної кодової бази та бізнес логіки використовують shared модуль, а модуль iOS та Android для коду специфічного для кожної з платформ.

Вагому частину часу розробки займає саме бізнес логіка, тому правильним рішенням є стандартизація підходів до роботи з компонентами системи. Для вирішення цієї проблеми вирішенням є застосування архітектурних шаблонів, що дозволить писати новий функціонал за певними правилами. Даний підхід використовуються в усіх великих проєктах. Багато архітектурних шаблонів вимагають певного базового функціоналу для повсяк місного перевикористання. За допомогою реалізації або розширення компонентів шаблону ми маємо змогу ефективно впроваджувати новий функціонал для написання нової бізнес логіки. Варто відмітити, що не існує ідеальних рішень шаблонів, котрими можна вирішити бізнес потреби для кожного з бізнесів і систем. У зв'язку з цим в мобільній розробці використовується велика варіація цих шаблонів.

Kotlin Multiplatform стрімко розвивається і дедалі частіше з’являються фреймворки та бібліотеки, котрі реалізують уніфікований досвід роботи з мультиплатформеними застосунками. Тобто розширення реалізацій архітектурних шаблонів є актуальною задачею.

Для вирішення даної проблеми сформульовано наступні задачі:

- Дослідити архітектурні шаблони для мобільної розробки;
- Дослідити особливості взаємодії нативного коду з Kotlin Multiplatform бібліотеками;

- Розробити фреймворк з реалізацією архітектурного шаблону;
- Провести інтеграцію в Android та iOS застосунки;

Висновки до розділу 1

В даному розділі було проведено огляд предметної області з розробки кросплатформенних рішень, що дозволив виділити ряд переваг Kotlin Multiplatform перед подібними рішеннями такі як: Flutter, React Native, .NET MAUI. Також було проведено аналіз ефективності розробки за допомогою Kotlin Native, що окреслило проблематику складності розробки зручного API для Swift коду.

Оскільки, Kotlin Multiplatform є доволі новою технологією, що швидко розвивається, то стандартизація процесу розробки є актуальною проблемою для пришвидшення даного процесу.

2 ДОСЛІДЖЕННЯ ТА АНАЛІЗ АРХІТЕКТУРНИХ ШАБЛОНІВ В МОБІЛЬНІЙ РОЗРОБЦІ

2.1 Дослідження архітектурних шаблонів

MVC (Model - View - Controller) - це шаблон проєктування, котрий повсюдно використовувався в розробці програмного забезпечення протягом великого терміну часу. Перші згадування та реалізації можна відслідкувати ще у мові програмування Smalltalk [13] у 1970-х роках. За цей час даний шаблон став дуже популярним та розповсюдженим, як і веб розробці, так і в мобільній.

Даний патерн поділяє логіку програмного забезпечення на такі три компоненти (Рис. 2.1):

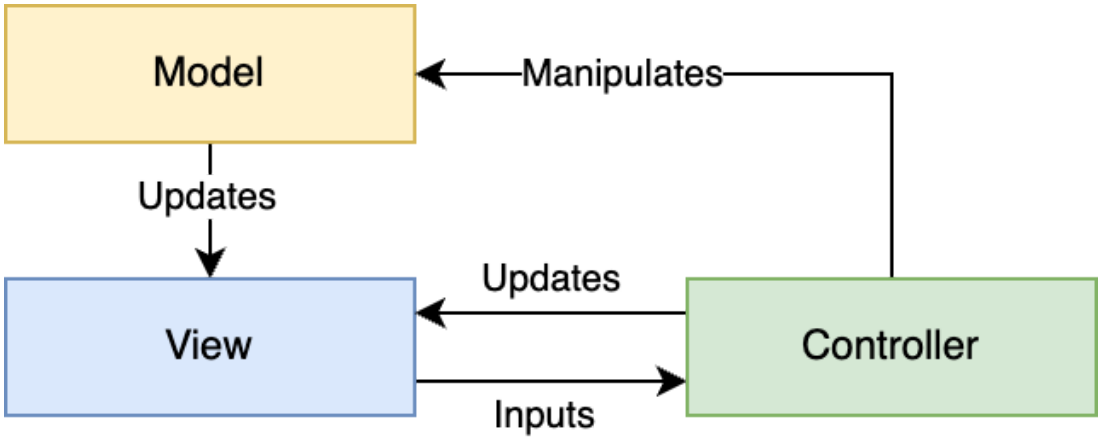

Рисунок 2.1 Діаграма шаблону MVC

Model - це компонент котрий відповідає за бізнес логіку домена та подальшу обробку наявних даних. Ця сутність являється центральним компонентом, котрий має відповідальність оновлення та зберігання даних. Також варто зазначити, що моделі забовязанні проводити валідацію вхідних даних.

View - це компонент, котрий відповідає за обробку подій користувача та візуальне представлення даних системи. На прикладі Android застосунку, це всі елементи користувацького інтерфейсу: поля для вводу, зображення і т.д. У випадку оновлення стану або даних системи має відображати оновлений варіант.

Controller - це компонент, котрий слугує спільною точкою поєднання View та Model. Зазвичай він в залежності від вхідних даних користувача оновлює модель та переносить відповідні зміни для оновлення на користувацький інтерфейс. Тобто він має гарантувати, що при оновленні даних, представлення буде йому відповідати.

Цей архітектурний шаблон забезпечує створення модульності системи, що безпосередньо впливає на складність підтримки системи. Також варто згадати, що шаблон дозволяє поліпшити процес тестування, бо можна відокремити компоненти системи.MVC доволі часто використовується для систем, котрі мають велику кількість бізнес логіки.

Розглянувши компоненти шаблону можемо сформувати переваги та недоліки. Переваги використання MVC:

- Оскільки ми маємо змогу розділяти систему на окремі компоненти, це покращує здатність до тестування;
- Також можливість розділення системи на компоненти дозволяє простіше модифікувати певні частини без вагомого впливу на інші;
- Розділення на логічні компоненти спрощує підтримку системи;
- Шаблон надає можливість перевикористання компонентів в інших системах або різних сценаріях використання;

Недоліки використання MVC:

- При не вірній реалізації даного шаблону може призвести до

- У випадку, коли Controller та View сильно пов'язані, то це призводить до ускладнення змін в представленні, котрі не впливають на Controller, та на тестування компонентів
- При збільшенні контролеру є ймовірність ускладнення підтримки даної частини системи

MVP (Model - View - Presenter) - це дуже розповсюджений архітектурний шаблон, що був дуже популярним рішенням для Android застосунків. Він був розроблений як наступний крок розвитку патерну MVC і представлений наприкінці 1990-х років [14]. Він допомагає вирішувати певні недоліки використання MVC, котрі було приведено раніше.

Цей шаблон можемо розкласти на відповідні компоненти та зобразити принцип взаємодії (Рисунок 2.2).

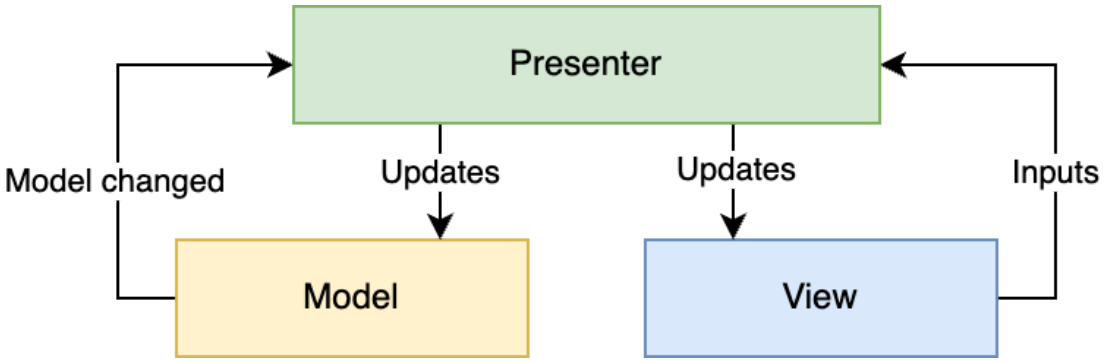

Рисунок 2.2 Діаграма шаблону MVP

Model - це компонент, котрий відповідає маніпулювання даними та має надавати інтерфейс взаємодії для інших компонентів системи. Тобто можна зробити висновок, що даний компонент працює схожим чином як і для MVC.

View - це компонент представлення, аналогічно до шаблону MVC.

Presenter - це композитний компонент, котрий дозволяє об'єднати попередні Model та View. Він допомагає керувати взаємодією між моделлю

та користувацьким представленням. Даний компонент в залежності від вхідних даних користувачів дозволяє оновлювати модель та її відповідне представлення. Така логіка дозволяє чітко розділювати View та Model, що покращує складність супроводу системи.

MVP був дуже популярним рішенням в світі Android розробки, оскільки він дозволяє покращувати здатність до тестування модульних компонентів системи. Ці переваги найбільше кристалізуються для додатків котрі мають вагому кількість логіки представлення.

Розглянувши компоненти шаблону можемо сформувати переваги та недоліки. Переваги використання MVP:

- Компонент дозволяє поліпшити модуляризацію та можливості до керування ними;
- Оскільки, компонент є більш незалежними, то це покращує можливості до тестування;
- Розділення компонентів поліпшує перевикористання існуючого коду;
- Компонент View має змогу ліпше обробляти велику кількість логіки представлення;

Недоліки використання MVP:

- Оскільки відбувається розділення компонентів це може призводити до додаткової складності в їх синхронізації
- Через наявність Presenter ми маємо додатковий рівень абстракції, котрий додає складнощів в розумінні потоку данних
- Для використання MVP доволі часто потрібна певна кількість шаблонного коду, що додає складнощів для розуміння кодовії бази
- Через озвучені раніше пункти, MVP вимагає більшої кількості часу для розробки

MVVM (Model-View-ViewModel) - це дуже популярний архітектурний шаблон для Android розробки. Він допомагає зручно організовувати код і надає можливість простого обслуговування. Вперше даний шаблон було представлено в 2005 році компанією Microsoft для WPF [15], як варіацію Presentation Model шаблону від Мартіна Фаулера [16].

Даний патерн поділяється на такі три компоненти (Рис. 2.3):

Рисунок 2.3 Діаграма шаблону MVVM

Model - це компонент, котрий відповідає за дані та бізнес логіку, але на відміну від MVC, модель в MVVM немає відповідальності стосовно обробки логіки та оновлення користувацького представлення.

View - це компонент представелнння користувацького інтерфейсі. В порівнянні з MVC даний компонент має єдину відповідальність по обробці користувацьких подій та не займається обробкою бізнес логіки.

ViewModel - це є композитним компонентом для View та Model. Він відповідає за зміну представлення при оновленні Model та всю логіку обробки користувацького інтерфейсу: валідація даних та їх обробку. Якщо порівнювати даний підхід з MVC, то MVVM має зобов'язання обробки логіки представлення та його оновлення без необхідності роботи з бізнес логікою.

З прикладного досвіду можна зробити висновок, що MVVM добре підходить для великих застосунків, що оперують значними обсягами даних та бізнес логіки. Другим сценарієм влучного використання можна назвати необхідність обробки складної логіки користувацького представлення, а саме:

валідація вхідних даних, їх обробка і т.д. На противагу таким складним системам, при розробці простих рішень MVP може закривати більшість необхідних задач.

Розглянувши компоненти шаблону можемо сформувати переваги та недоліки. Переваги використання MVVM:

- Можливість простого тестування окремих компонентів;
- Чітке розподілення відповідальності компонентів;
- Поліпшення масштабованості для великих проєктів;

Недоліки використання MVVM:

- Для простих застосунків це може додавати складності для ознайомлення
- Дани архітектурний шаблон потребує додатковий шаблонний код, що може призводити до податкової складності

MVI (Model View Intent) - це доволі популярний в світі Android архітектурний шаблон, що був розроблений для функціонального фреймворку мовою JavaScript [17]. Даний шаблон отримав свою популярність, як доволі зручна альтернатива MVP та MVC. Це зумовлено тим, що він базується на UDF архітектурі та використовує функціонале реактивне програмування.

Даний архітектурний шаблон складається з (Рисунок 2.4):

- Model - це компонент, що відповідає за стан додатку. Реалізація Model базується на використанні незмінних структур даних, що в свою чергу забезпечує передбачуваність та консистентність зміни стану. Це є відмінністю у порівнянні з MVC та MVP, в котрих робота з моделлю даних може бути розпорошена між кількома компонентами;

- View - це компонент представлення. Він реалізований як реактивний компонент, що дозволяє оновлювати представлення відповідно до змін стану застосунку;
- Intent - це компонент намірів користувача. Він являється представленням намірів користувача. Інформація про Intent передається в Model, котрі використовуються для зміни стану або запуску побічних ефектів;

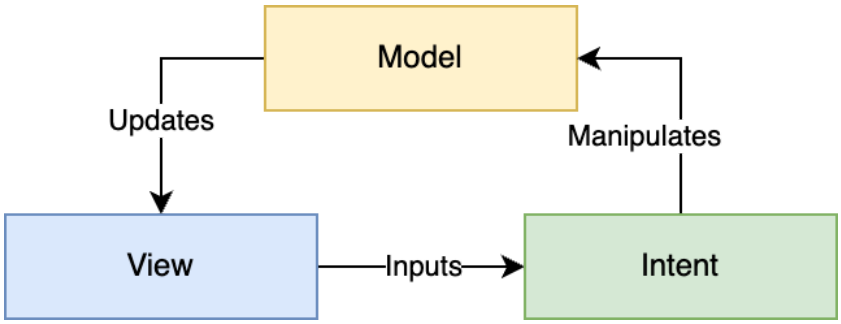

Рисунок 2.3 Діаграма шаблону MVI

Розглянувши компоненти шаблону можемо сформувати переваги та недоліки. Переваги використання MVI:

- Даний підхід дозволяє чітко розмежовувати взаємодію користувацького представлення та іншої бізнес логіки застосунку
- MVI дозволяє спростити взаємодію з UI, оскільки даний компонент є реактивним
- Вагомою перевагою відокремлення відповідальності компонентів є покращення процесу тестування
- Він забезпечує консистентність данних, оскільки наслідує принципи UDF, котрі будуть розглянути надалі

Недоліки використання MVI:

- Потребує використання шаблонного коду;
- Цей підхід додає нову абстракція для використання;
- Через вище перелiковані пункти це ускладнює розуміння підходу;

2.2 UDF шаблон

UDF (unidirectional data flow) - це шаблон проєктування, котрий має низхідний потік даних для представлення та висхідний для користувацьких подій. Якщо дотримуватися даного підходу, то це дозволяє виокремлювати компоненти, котрі відповідають за представлення даних та котрі займаються модифікацією та збереженням цих даних.

Розглянемо діаграму оновлення UI додатку, котрий використовує UDF шаблон проєктування (Рисунок 2.5).

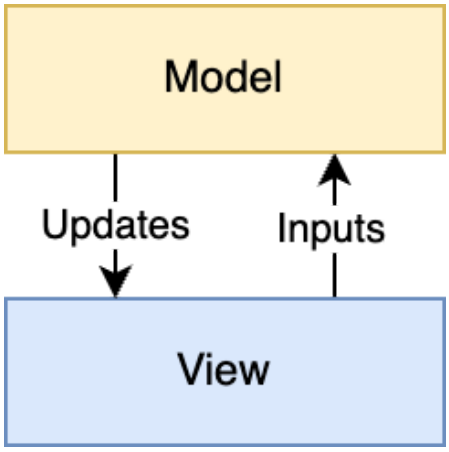

Рисунок 2.6 Діаграма UDF

Даний цикл оновлення користувацького представлення відбувається таким чином:

- Інтерфейс користувача генерує певні події при взаємодії з ним. Вони передаються до Model або компоненту котрий відповідальний за основну бізнес логіку;
- Після отримання події Model може опціонально оновити стан або викликати побічні ефекти;
- Після зміни стану Model передає його до користувацького представлення, котрий оновлює вигляд відповідно до змін;

Також варто відмітити, що даний підхід є рекомендований Google для використання з декларативним представленням Jetpack Compose [18]. При дотриманні UDF з даним представлення ми маємо ряд переваг:

- Оскільки ми відокремлюємо стан додатку від представлення, котре його відображає, то це спрощує процес тестування для ізольованих компонентів;
- Оскільки ми маємо єдине джерело стану, бо оновлення відбуваються тільки в одному місці - Model, то це зменшує кількість помилкових ситуацій через неузгодженість актуального стану застосунку;
- Оскільки відображення користувацького представлення відбуваються за допомогою публічного тримачу стану, котрий реактивно оновлюється при зміні, то це зменшує неузгодженість UI;

2.3 Огляд фреймворку TCA

Оглянувши вищезгадані шаблони проєктування ми можемо виділити спільні риси: простота в реалізації, впровадженні та розробці ізольованих функціональностей бізнес-логіки. Але варто помітити, що при збільшенні кодової бази можуть виникати такі недоліки:

- Поєднання компонентів та дублікати;
- Складність інтеграційного тестування;
- Складність передачі даних з однієї частини системи в іншу, що може призводить до зв'язності між ними;
- З попередніх пунктів витікає складність розділення системи на модулі;

Всі ці проблеми можна нівелювати строгими правилами до розробки, але нові розробники або не дуже досвідчення мають змогу пропустити щось та отримати не очікувану поведінку в застосунку. Фреймворк TCA покликаний, щоб вирішити ці проблеми за допомогою строгої типізації можливих дій станів.

TCA (The Composable architecture) - це Swift фреймворк, котрий містить інструментарій для розробки комплексних додатків. Дана бібліотека дозволяє вирішувати повсякденні проблеми при створенні додатків:

- Керування станом додатку та можливості до розповсюдження його між декількома екранами. Наприклад, при зміні даних на одному екрану ми маємо мати змогу спостерігати зміни на іншому
- Декомпозиція компонентів застосунку на декілька модулів, щоб мати змогу в батьківському модулі використовувати їх композицію
- Організація взаємодії певних частин програми з зовнішнім оточенням
- Організація тестування не тільки окремих компонентів системи, але й змога зручного інтеграційного тестування. Також вплив зовнішніх ефектів на здатність проведення тестування певних частин

Composable Architecture - це адаптація фреймворку Redux, яка чудово поєднується зі Swift UI, UIKit, Combine і Modern Concurrency (async/await) [19]. Також варто відмітити, що як і Redux та MVI вони наслідують

архітектурний шаблон UDF, що дозволяє використовувати всі переваги декларативного UI.

Дане рішення базується на декількох концептах:

- ізоляція
- композиція
- чисті функції
- керовані побічні ефекти

Основними складовими TCA є: Store, Reducer, State, Effect, Action, Env. Основними рушіями дій та змін в системі є Store та Reducer, а інші компоненти слугують певним контрактом взаємодії. Розглянемо більш детально зміст даних частин фреймворку та їх взаємодії (Рисунок 2.6).

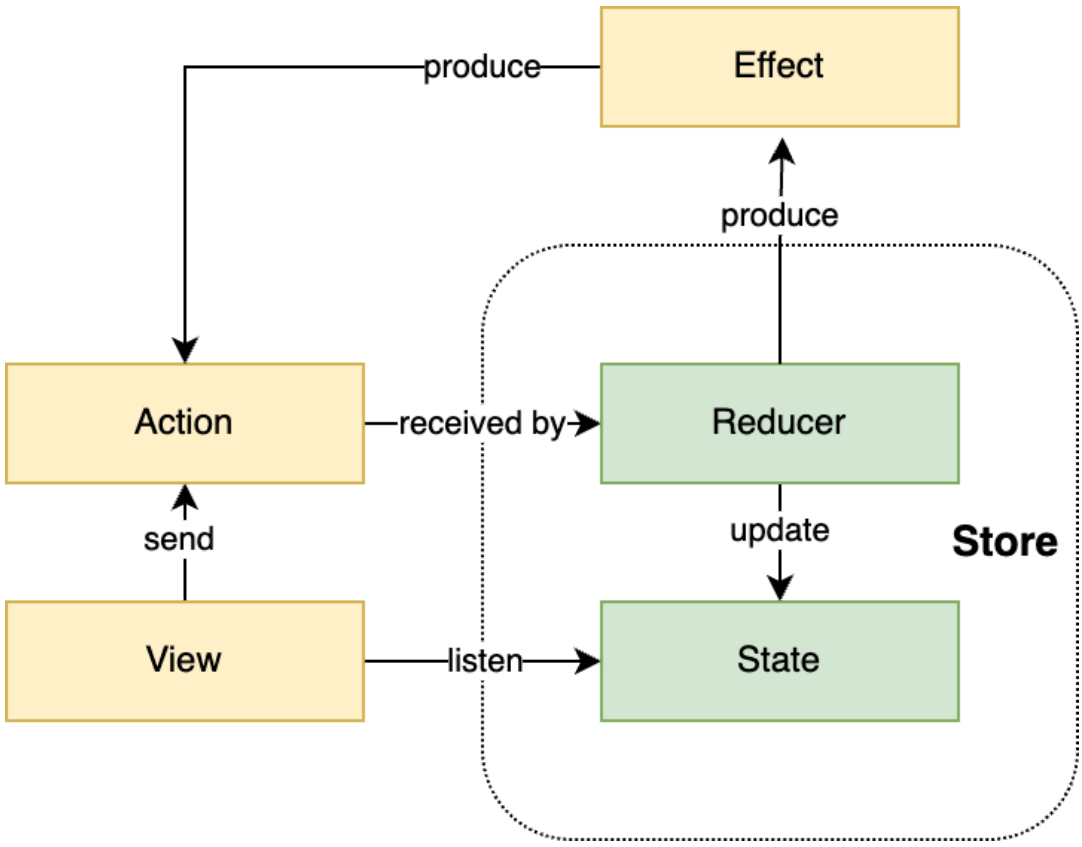

Рисунок 2.6 Діаграма фреймворку TCA

- State - це єдине джерело стану системи;
- Action - це подія, які можуть змінювати стан або виконувати побічні ефекти, такі як виклики API;
- Reducer - це чиста функція, яка описує, як перетворити поточний стан у наступний за допомогою певного Action і які побічні ефекти потрібно виконати;
- Store - сховище керує усім вищезгаданим. Він отримує дію і запускає reducer для зміни стану та опрацювання побічних ефектів;
- Effect - побічний ефект, котрий продукує події за межами локального контексту;

Найбільшою цінністю даного рішення є можливість контролювання дій та станів в застосунку. Використовуючи комбінації редукторів, є змога реагувати на будь-які зміни в системі: введення текстової інформації, отримання координат GPS, отримання результатів запиту з бекенд частини системи і т.д. Також великою перевагою є можливість розрізнення редукторів та винесення їх в окремі модулі системи. Гнучкий API та передвизначенні розширення для компонентів фреймворку дозволяють зручно моделювати стан застосунку.

Оскільки, дане рішення є доволі популярним в світі iOS розробки, має вище перелiковані переваги та реалізує UDF підхід, то це є добрим кандидатом для адаптації в Kotlin Multiplatform фреймворк. Для розробників iOS логіка роботи з даним рішення буде доволі знайомою, що спростить досвід написання кросплатформенних рішень.

Висновки до розділу 2

В даному розділі було проведено дослідження найвідоміших архітектурних шаблонів для мобільних застосунків. Виокремлення переваг та недоліків дозволило обрати оптимальний шаблон для подальшої розробки - UDF. Для подальшої адаптації було обрано Swift фреймворк TCA, що дозволяє спростити процес ознайомлення з основними принципами роботи та його інструментарієм.

3 РОЗРОБКА ФРЕЙМВОРКУ

3.1 Дослідження особливостей інтероперабельності Kotlin & Swift

В першому розділі при огляді технології Kotlin Native було вказано, що дана технологія має можливість підтримки взаємодії з нативними мовами програмування й навпаки - взаємодії з Kotlin кодом з нативної частини програми. В розрізі мобільної розробки цікавить саме сумісність з мовою Swift, оскільки при побудові API фреймворку варто враховувати зручність та ефективність використання спільного коду.

Наразі Kotlin Native не має прямої підтримки Swift. Сумісність та взаємодія між Kotlin та Swift відбувається за допомогою взаємодії з Objective-C. З Swift коду ми маємо можливість викликати спільний код для платформа за допомогою bridge заголовків в Objective-C[20]. Така інтероперабельність пов'язана з необхідністю підтримки доволі старих проєктів котрі написані мовою Objective-C. При такому підході є змога викликати спільний код на Kotlin, як і з Swift так і з Objective-C. В планах у команди, котра займається розробкою Kotlin Native в дорожні карті окреслені плани про додавання взаємодії напряму з Swift для значної оптимізації інтероперабельності, використовуючи повні можливості даної мови програмування.

Основні структурні одиниці мови програмування Kotlin ми можемо використовувати без проблем, а саме: класи, властивості та функції. З більшістю інших мовних засобів потрібно шукати обхідні шляхи або використовувати бібліотеки, котрі дозволяють нівелювати дані проблеми.

Щоб забезпечити найліпший користувацький досвід для iOS розробників, вартує використовувати обмежений мовний функціонал Kotlin.

Враховуючи, що Swift розробники можуть погано розбиратися в особливостях мови, то потрібна увага до даних особливостей або інтеграція з сторонніми бібліотеками, щоб більш вірно інтерпретувати Kotlin в Swift. Оскільки, це є вагомим рушієм до вигляду спільного API, то варто розглянути, що з мовного функціоналу допустиме до використання, як публічний API для iOS додатку. В наступному переліку будуть продемонстровані тільки особливості інтероперабельності.

Щоб отримати доступ до функції або властивості верхнього рівня з Kotlin, потрібно викликати клас обгортку з назвою файлу + Kt:

```
MainKt.main()
```

Якщо Kotlin код може викликати виключну ситуацію, то потрібно декларувати його з анотацією @Throws, бо в іншому випадку це призведе до фатальної помилки зі сторони Swift:

```
@Throws(Exception::class)
fun throwSomeException(){
    throw Exception()
}
```

Якщо викликається suspend функція, то без анотації буде повертатися тільки помилка про відміну виконання - CancellationException.

Для того, щоб переназвати для Swift певну властивість або функцію, то потрібно додати анотацію @ObjCName:

```
   @ObjCName("swiftName")
   fun name(@ObjCName("for") name: String): String = name
```

Для того, щоб прибрати певну властивість або функцію для Swift/Objective-C потрібно додати анотацію @HiddenFromObjC:

```
@HiddenFromObjC
fun main(){
    println("For kotlin")
}
```

Якщо перевантажити функцію з однаковою назвою параметра, то отримаємо додаткові підкреслення для назви цих параметрів. Щоб уникнути подібної поведінки, варто використовувати різні назви для параметрів:

```
fun returnValue(param: Int) = param

fun returnValue(param: Long) = param
```

В результаті в Swift:

```
MainKt.returnValue(param: 0)

MainKt.returnValue(_param: 0)
```

Для використання стандартних значень для функції можна використовувати анотацію з бібліотеки SKIE:

```
@DefaultArgumentInterop.Enabled
fun returnDefault(param: Int = 0) = param
```

При використані функції, що використовують лямбду з приймачем, в Swift потрібно використовувати функцію з аргументом приймача:

```
fun callBlock(block: Int.() -> Unit) = 0.block()
```

```
MainKt.funcWithExtension(block: { number in

})
```

При використанні Kotlin колекцій з примітивними типами (Int, Boolean і т.д.) в Swift отримаємо колекцію з Kotlin обгорткою, а не з примітивом Swift:

```
List<Int> -> [KotlinInt]
```

Оскільки, Swift та Objective-C не мають підтримки абстрактних класів, то при наслідуванні код компілюється без необхідності імплементації методів. У випадку коли буде викликаний метод даного абстрактного класу, то це спричиняє фатальну помилку NSGenericException. Також в Swift та Objective-C немає підтримки класів анотацій, дата класів, вбудованих класів та анонімних класів. Тому потрібно уникати використання даного функціоналу для API.

Також існує доволі вагома проблема при взаємодії з загальними типами. Взаємодія з ними можлива тільки для класів і тільки якщо ці типи не є примітивами. В більшості випадків при роботі з загальними типами потрібно зводити його до бажаного, що призводить до потенційних багів. Інтерфейси взагалі не підтримують загальні типи, що є проблемою при використанні пакету coroutines для Kotlin. Оскільки, один з базових елементів є Flow<T>, то це накладає обмеження та складнощі використання. Для того, щоб мати змогу працювати з даною бібліотекою, потрібно використовувати бібліотеки KMP-NativeCoroutines або вищезгадану SKIE. Вони за допомогою анотації генерують властивість котра повертає обгортку для Flow та ховають анотацією @HiddenFromObjC властивість для Swift коду.

Також варто звернути увагу на відсутність механізму для відміни suspend функцій та Flow. Дані проблеми допомагає виправити вищезгадані бібліотеки KMP-NativeCoroutines або SKIE. З Swift 5.5 з'явилась підтримка механізму async/await для suspend функцій, але поки в експериментальному статусі [21].

Виходячи з цих особливостей інтероперабельності Swift та Kotlin можна скласти певний перелік правил для написання ефективного та зручного API для Swift коду:

- Для функцій додавати через анотацію можливі помилки, щоб не отримати фатальну помилку;
- Не використовувати абстрактні класи та надавати перевагу інтерфейсам, якщо це є можливим;
- При необхідності використання загальних типів потрібно надавати перевагу класам;
- При використання бібліотеки coroutines додатково вартує інтегрувати одну з бібліотек KMP-NativeCoroutines або SKIE, що покращить досвід розробників;

Попри вищезгадані особливості та проблеми інтероперабельності Kotlin та Swift, можна звернути увагу на те що розробники додають свої особисті рішення даних проблем. Також команда Kotlin Native працює над покращення досвіду розробників та планує додавання підтримки взаємодії на пряму з Swift.

3.2 Архітектура фреймворку

Враховуючи вище описані рекомендації до використання мовного функціоналу Kotlin для покращення інтероперабельності до Objective-C та Swift, розглянемо як саме варто проводити адаптацію фреймворку TCA для Kotlin Multiplatform.

Для початку потрібно розділити API фреймворку на три складові: внутрішній інструментарій для реалізації рушія, API самої бібліотеки для інтеграції в shared модулі застосунку та API для взаємодії з бібліотекої зі

сторони Swift. Щоб досягнути подібного розділення можемо скористатися особливістю інтероперабельності модифікаторів доступу Kotlin.

З модифікатором public все доволі просто, оскільки дані частини коду будуть доступні і для Swift. Оскільки, protected не має підтримки з сторони мов Objective-C та Swift, то воно рівноцінне до Public і не є бажаним до використання. У Kotlin є змога позначатити класи, функції та властивості ключовим словом internal. Це обмежує області видимості даних компонент на рівні блоку компіляції. У даному випадку це дозволяє організувати приховування внутрішніх реалізацій в середині бібліотеки не передаючи API, а ні в shared модуль, до котрого підключена бібліотека, а ні до Swift коду. Також варто окреслити, що ми маємо змогу ховати від компіляції до Objective-C певні властивості, функції та класи за допомогою анотації @HiddenFromObjC. Тобто можемо окреслити правила використання модифікаторів та приховування API для Swift:

- Якщо потрібно приховати внутрішню реалізацію в середині блоку компіляції, то варто використовувати internal;
- Якщо API потрібно приховати для Swift коду, то використовуємо @HiddenFromObjC;
- В інших випадках нам достатньо public;

Далі розглянемо приклади адаптації основних компонентів даного фреймворку TCA. Для адаптації редуктора (Рисунок 2.7) варто додати відсутність можливості мутації стану, а просто повертати його. Оскільки, ще нам потрібно повертати побічні ефекти, то ми можемо зробити композитний тип StateWithEffect<State, Action>. Оскільки, редуктор це приклад чистої функції, то ми можемо окреслити його функціональни протоколом, щоб мати

змогу викликати безпосередній об'єкт реалізації. Це нам дозволяє отримати такий інтерфейс взаємодії з редуктором:

```
@HiddenFromObjC
fun interface Reducer<State, Action> {

   fun reduce(state: State, action: Action): StateWithEffect<State, Action>
}
```

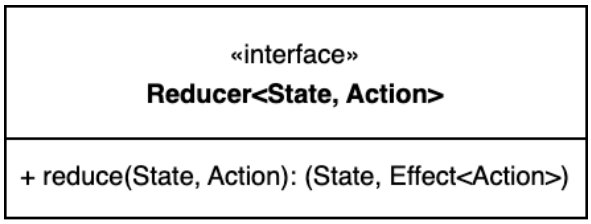

Рисунок 2.7 Редуктор

В фреймворку TCA використовується інструментарій мови Swift - KeyPath. Він допомагає лаконічно описувати отримання з root типу іншого результуючого. Для мови Kotlin не має зручного відповідника, тому це змушує використовувати лямбди з параметром для схожого функціоналу.

Відповідно до специфікації TCA розроблено такі реалізації редукторів:

- ForEachReducer - це редуктор, котрий дозволяє комбінувати з редуктором для списку вкладених станів;
- CombineReducer - це редуктор, котрий дозволяє акумулятивно об’єднувати декілька з них;
- ScopeReducer - це редуктор, котрий дозволяє комбінувати з редуктором для вкладеного стану;
- OptionalReducer - це редуктор, котрий дозволяє комбінувати з редуктором для nullable вкладеного стану;

- EmptyReducer - це редуктор, котрий повертає оригінальний стан без додаткових ефектів;

Для оптимізації роботи з побічними ефектами ми можемо узагальнити підхід до асинхронного використання та розглядати їх як не окремий набір операцій (без подій, асинхронний результат та потік), а як Flow очікуваних подій. За допомогою функціонального інтерфейсу та додаткових розширень ми можемо отримати підтримку подібних операцій.

```
fun interface Effect<out Action> {

   @NativeCoroutines
   fun run(): Flow<Action>

   companion object {
       fun none(): Effect<Nothing> = NoEffect
   }
}
```

Для розширення підтримуваних операцій достатньо перевизначити потік даних, наприклад:

```
fun <Action> Effect.Companion.create(
   block: suspend () -> Action
): Effect<Action> = Effect { block.asFlow() }
```

Аналогічно ми можемо додавати реалізацію схожих методів.

Найбільшою частиною фреймворку є його рушій, а саме компонент Store. Саме з ним найбільше взаємодіють реалізації користувацького представлення для надсилання новий подій та відслідковування стану застосунку. Оскільки, Swift код буде взаємодіяти з певними реалізаціями, то ми уникаємо проблем з інтероперабельність загальних типів для інтерфейсів.

Загальний протокол взаємодії виглядає наступним чином:

```
interface Store<State, Action> {

   @NativeCoroutines
   val state: Flow<State>

   val currentState: State

   fun send(actions: List<Action>)

   fun send(action: Action) = send(listOf(action))

   fun <O> withState(block: (State) -> O): O = block(currentState)

   fun dispose()
}
```

3.3 Інструментарій фреймворку

Для підтримки різноманітних сценаріїв використання варто розширювати дане рішення, щоб мати змогу адаптуватися до них. Однією з найпопулярніших проблем в мобільній розробці є управління та відміна запущених задач. Це безпосередньо відноситься до запущених побічних ефектів. Потрібно додати підтримку декількох режимів виконання відкладених задач: fire and forget, перемикання виконання задачі та можливість їх відміни. Для вирішення даної проблеми можна використати обгортку над стандартним ефектом за допомогою функціоналу делегування в Kotlin (Рисунок 2.8):

```
data class CancellableEffect<Action>(
   val id: String,
   val effect: Effect<Action>,
   val cancelInFlight: Boolean = false
) : Effect<Action> by effect
```

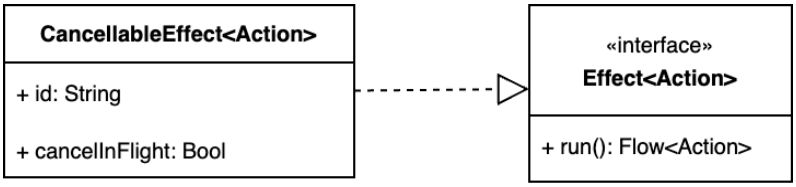

Рисунок 2.8 Побічний ефект з відміною

Це є зручним базисом для роботи з ефектами, але потребує перевизначення розширення для стандартного ефекту, щоб не втрачати мета інформацію про ефект при приведенні типів:

```
inline fun <IAction, OAction> CancellableEffect<IAction>.map(
   crossinline transform: (IAction) -> OAction
): CancellableEffect<OAction> = CancellableEffect(
   id = id,
   cancelInFlight = cancelInFlight,
   effect = effect.map(transform)
)
```

Хоча й інтерфейс побічного ефекту дає змогу до реалізації різноманітної поведінки, але також варто окреслити й попередньо реалізовані розширення.

Наприклад, для оптимізації пошукових запитів, щоб не навантажувати бекенд частину постійними запитами, доволі часто використовують debounce для потоку вхідних данних. Він працює таким чином, що після введення нової літери в пошуку, відраховується певна затримка, якщо за цей час було додано нове значення, то відлік починається знову (Рисунок 2.9).

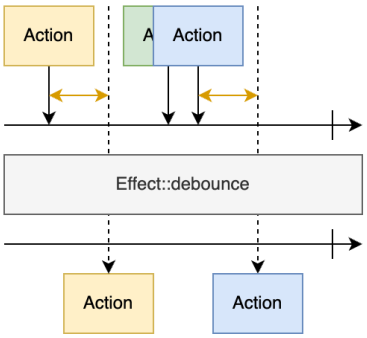

Рисунок 2.9 Побічний ефект з debounce

Також наприклад, для фільтрації даних з GPS можна використовувати throttle для вхідних данних з системи. Даний ефект працює таким чином, що дозволяє фільтрувати значення в рамках певного періоду (Рисунок 2.10).

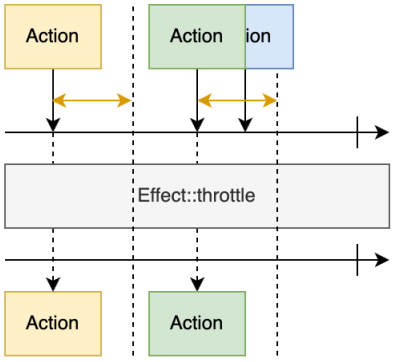

Рисунок 2.10 Побічний ефект з throttle

В рамках інструментарію фреймворку варто розглянути і здатність до тестування. Дана необхідність забезпечується за допомогою впровадження CoroutineScope до Store, що дозволяє зручно тестувати рушій за допомогою пакету kotlinx-coroutines-test. Також це дозволяє зручно налаштовувати обробник виключних ситуацій та диспетчер для середовища виконання.

3.4 Інтеграція фреймворку в Android та iOS застосунки

Для інтеграції даного фреймворку в Kotlin Multiplatform достатньо підключити залежність до build.gradle файлу shared модулю і задекларувати цю залежність в каталогу версій:

```
kaleidoscope-core = { module = "com.ukma.kaleidoscope:kaleidoscope-core",
version.ref = "kaleidoscope-core" }
```

```
kotlin {
   sourceSets {
       commonMain.dependencies {
           api(libs.kaleidoscope.core)
       }
```

```
    }
}
```

В даному випадку ми використовуємо область видимості api, щоб в нас був доступ до API бібліотеки з застосунків.

Після реалізації основної бізнес логіки за допомогою інструментарію, котрий надає фреймворк, ми можемо передавати об’єкти рушія до платформенних частин. Це можна робити, як і через Dependency Injection за допомогою бібліотеки Koin, так і мануально.

На прикладі роботи простого TODO застосунку розглянемо принцип реалізації бізнес логіки. Складемо ряд вимог до функціоналу:

- Можливість створення задач, зміна статусу виконання та оновлення заголовку задачі;
- Можливість фільтрувати виконані, активні задачі;
- Можливість очищувати виконані задачі;
- Сортування списку задач з затримкою після їх зміни;

Для керування виключними ситуаціями в рушії ми можемо створювати бажаний CoroutineScope, котрий виконується на головному потоці та має обробник виключень.

```
val appStore = RootStore(
    initialState = TodosFeature.State(),
    reducer = appReducer,
    coroutineScope = CoroutineScope(
        SupervisorJob()
                + Dispatchers.Main
                + CoroutineExceptionHandler { context, throwable -> }
    ),
    exceptionHandler = ExceptionHandler {}
)
```

Розглянемо функціонал запуску побічних ефектів при зміні задачі. Оскільки, нам потрібно сортувати цей список з затримкою, то можемо використати CancellableEffect з відміною попередньої задачі, щоб UI не оновлювався декілька разів. Далі наведено реалізацію логіки в середині редуктора для цього стану.

```
state.copy(isLoading = true)
   .withSuspendEffect(id = "sort", cancelInFlight = true) {
       delay(1000L)
       Action.SortCompletedTodos
   }
```

Всі інші події працюють відповідної логіки з використанням оновлення стану без додаткових побічних ефектів.

Для того, щоб проінтегрувати дану бізнес логіку в Android застосунок, потрібно додати залежність на shared модуль в buid.gradle:

```
implementation(projects.shared)
```

Для відслідковування стану рушія ми можемо скористатися розширенням Flow collectAsState для Compose. Воно забезпечує рекомпозицію нашого представлення.

```
val appState by appStore.state.collectAsState(TodosFeature.State())
```

Для надсилання подій з користувацького представлення достатньо мати об’єкт рушія та передати перед визначену подію в метод send.

```
todoStore.send(TodoFeature.Action.Complete(isCompleted)))
```

Як результат роботи маємо застосунок, котрий відповідає вищезгаданим критеріям роботи (Рисунок 2.11).

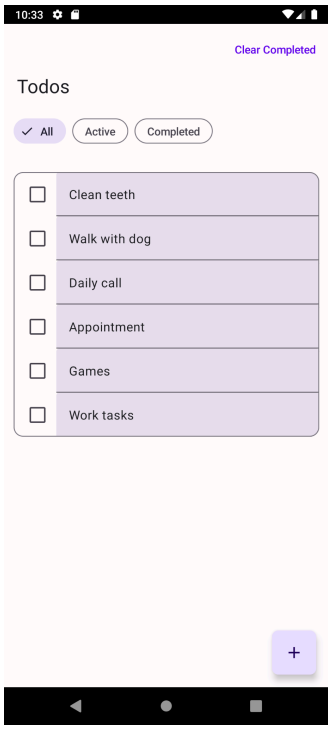

Рисунок 2.11 Реалізація TODO застосунку на Android

Для інтеграції в iOS застосунок можемо скористатися реалізацією ObservableObject. Використовуючи розширення бібліотеки KMP-NativeCoroutines - createPublisher ми межемо оновити значення стану за допомогою Combine:

```
class ObservableAppStore: ObservableObject {
    @Published public var state: TodosFeature.State = TodosFeature.State(
        todos: [],
        filter: TodosFeature.Filter.all,
        isLoading: false
    )

    public let store: Kaleidoscope_coreRootStore<TodosFeature.State, TodosFeature.Action>
    init(store: Kaleidoscope_coreRootStore<TodosFeature.State, TodosFeature.Action>) {
        self.store = store

        createPublisher(for: store.state)
            .receive(on: DispatchQueue.main)
            .sink { [weak self] state in
                self?.state = state
            }
    }
```

```
    deinit {
        self.store.dispose()
    }
}
```

Надалі даний об’єкт використовується в Swift UI представленні:

```
struct AppView: View {
    @ObservedObject private var store: ObservableAppStore

    public init() {
        self.store = ObservableAppStore(store: AppKt.appStore)
    }
```

Висновки до розділу 3

В рамках даного розділу було проведено дослідження інтероперабельності Swift & Kotlin, було розроблено правила для використання мовних функціональностей Kotlin для публічного API, було проведено адаптацію фреймворку TCA, розроблено зручний API для Swift та проведено інтеграцію в TODO застосунок.

Архітектура фреймворка були ретельно продумана для зручного використання переваг архітектурного шаблону, на котрому базується TCA. Дане рішення дозволяє зручно розробляти застосунки та проводити тестування спільної бізнес логіки.

ВИСНОВКИ

В рамках роботи було проведено дослідження інструментарію розробки кросплатформенних рішень. Для цього було проведено огляд предметної області з розробки кросплатформенних рішень, що дозволив виділити ряд переваг Kotlin Multiplatform перед подібними рішеннями такі як: Flutter, React Native, .NET MAUI. Також було проведено аналіз ефективності розробки за допомогою Kotlin Native, що висвітлює проблематику складності розробки зручного API для Swift коду. Було проаналізовано необхідність стандартизації процесу розробки для її пришвидшення.

Було проаналізовано найвідоміші архітектурні шаблонів для мобільних застосунків. Шаблони котрі було розглянуто: MVC, MVP, MVVM та MVI. Було проведено порівняльний аналіз даних шаблонів, що дозволило обрати оптимальний шаблон для подальшої розробки - UDF. Для подальшої адаптації було обрано Swift фреймворк TCA на базі UDF, що дозволяє спростити процес ознайомлення з основними принципами роботи та його інструментарієм для iOS розробників.

Також було проведено дослідження інтероперабельності Swift & Kotlin, було розроблено правила для використання мовних функціональностей Kotlin для публічного API, було проведено адаптацію фреймворку TCA, розроблено зручний API для Swift та проведено інтеграцію в TODO застосунок.

Архітектура фреймворка були ретельно продумана для зручного використання переваг архітектурного шаблону, на котрому базується TCA. Дане рішення дозволяє зручно розробляти застосунки та проводити тестування спільної бізнес логіки.

Результатом дослідження є зручний та оптимальний фреймворк для Kotlin Multiplatform, що допомагає стандартизувати підхід до розробки бізнес

логіки. У подальшому можна розширити функціонал фреймворку за допомогою наявного базису для певних прикладних задач в розробці мобільних застосунків: підтримка зручної навігації, додавання кодової генерації для зменшення шаблонного коду та додатковий інструментарій для інтеграційного тестування. Дане дослідження показало, що використання кросплатформенних рішень дозволяє пришвидшувати цикл розробки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sedunov, Aleksei. Kotlin In-Depth: A Guide to a Multipurpose Programming Language for Server-Side, Front-End, Android, and Multiplatform Mobile. BPB Publications (2022) pp. 131-182
2. Cross-Platform Mobile Frameworks Used by Global Developers 2022 [Електронний ресурс] // Statista. – 2022. — Режим доступу до ресурсу: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/
3. Building User Interfaces with Flutter [Електронний ресурс] // Google. — Режим доступу до ресурсу: https://docs.flutter.dev/ui
4. Flutter Architectural Overview [Електронний ресурс] // Google. — Режим доступу до ресурсу: https://docs.flutter.dev/resources/architectural-overview
5. Architecture Overview React Native [Електронний ресурс] // Facebook. — Режим доступу до ресурсу: https://reactnative.dev/architecture/overview
6. Native Modules Intro React Native [Електронний ресурс] // Facebook. — Режим доступу до ресурсу: https://reactnative.dev/docs/next/native-modules-intro
7. What Is .NET MAUI? [Електронний ресурс] // Microsoft. — Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui
8. Android Support for Kotlin Multiplatform to Share Business Logic Across Mobile, Web, Server, and Desktop Platforms [Електронний ресурс] // Google. — Режим доступу до ресурсу: https://android-developers.googleblog.com/2024/05/android-support-for-kotl

in-multiplatform-to-share-business-logic-across-mobile-web-server-desktop.html

9. Compose Multiplatform UI Framework [Електронний ресурс] // Jetbrains. — Режим доступу до ресурсу: https://www.jetbrains.com/lp/compose-multiplatform/
10. Debugging Kotlin/Native [Електронний ресурс] // Jetbrains. — Режим доступу до ресурсу: https://kotlinlang.org/docs/native-debugging.html
11. Kotlin Native [Електронний ресурс] // Jetbrains. — Режим доступу до ресурсу: https://kotlinlang.org/docs/native-overview.html
12. The LLVM Compiler Infrastructure Project [Електронний ресурс] // The LLVM Foundation. — Режим доступу до ресурсу: https://llvm.org/
13. Krasner, Glenn E., and Stephen T. Pope. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming, vol. 1, no. 3, Aug. 1988, pp. 26–49.
14. Archiveddocs. The Model-View-Presenter (MVP) Pattern [Електронний ресурс] // Microsoft. — Режим доступу до ресурсу: https://learn.microsoft.com/en-us/previous-versions/msp-n-p/ff649571(v=pandp.10)
15. Introduction to Model/View/ViewModel Pattern for Building WPF Apps [Електронний ресурс] // Microsoft. — Режим доступу до ресурсу: https://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps
16. Presentation Model [Електронний ресурс] // Martin Fowler. — Режим доступу до ресурсу: https://martinfowler.com/eaaDev/PresentationModel.html
17. Cycle.Js - Model-View-Intent [Електронний ресурс] // Cycle. — Режим доступу до ресурсу: https://cycle.js.org/model-view-intent.html

18. Architecting Your Compose UI [Електронний ресурс] // Google. — Режим доступу до ресурсу: https://developer.android.com/develop/ui/compose/architecture
19. TCA [Електронний ресурс] // PointFree. — Режим доступу до ресурсу: https://github.com/pointfreeco/swift-composable-architecture
20. Importing Objective-C into Swift [Електронний ресурс] // Apple. — Режим доступу до ресурсу: https://developer.apple.com/documentation/swift/importing-objective-c-into-swift
21. What's New in Kotlin 1.5.30 [Електронний ресурс] // JetBrains. — Режим доступу до ресурсу: https://kotlinlang.org/docs/whatsnew1530.html#experimental-interoperability-with-swift-5-5-async-await

ДОДАТКИ

Додаток А. Код файлу Effect.kt

```
fun interface Effect<out Action> {

   @NativeCoroutines
   fun run(): Flow<Action>

   companion object {
       fun none(): Effect<Nothing> = NoEffect
   }
}

object NoEffect : Effect<Nothing> {
   override fun run(): Flow<Nothing> = emptyFlow()
}

inline fun <Action, O> Effect<Action>.map(crossinline transform: (Action) -> O) =
Effect {
   run().map { transform(it) }
}

@OptIn(FlowPreview::class)
fun <Action> Effect<Action>.debounce(duration: Duration) = Effect {
run().debounce(duration) }

fun <Action> Effect<Action>.throttle(duration: Duration) = Effect {
   run().conflate()
       .onEach { delay(duration) }
}

infix fun <Action> Effect<Action>.mergeWithEffect(effect: Effect<Action>):
Effect<Action> = Effect {
   merge(run(), effect.run())
}

fun <Action> Effect.Companion.create(
   vararg actions: Action
): Effect<Action> = Effect { flowOf(*actions) }

fun <Action> Effect.Companion.create(
   flow: Flow<Action>
): Effect<Action> = Effect { flow }
```

```
fun <Action> Effect.Companion.create(
   block: suspend () -> Action
): Effect<Action> = Effect { block.asFlow() }

@OptIn(ExperimentalTypeInference::class)
fun <Action> Effect.Companion.create(
   @BuilderInference
   block: suspend ProducerScope<Action>.() -> Unit
): Effect<Action> = Effect { callbackFlow(block) }
data class CancellableEffect<Action>(
   val id: String,
   val effect: Effect<Action>,
   val cancelInFlight: Boolean = false
) : Effect<Action> by effect

fun <Action> Effect<Action>.cancelable(
   id: String,
   cancelInFlight: Boolean = false
) = CancellableEffect(
   id = id,
   effect = { run().cancellable() },
   cancelInFlight = cancelInFlight
)

inline fun <IAction, OAction> CancellableEffect<IAction>.map(
   crossinline transform: (IAction) -> OAction
): CancellableEffect<OAction> = CancellableEffect(
   id = id,
   cancelInFlight = cancelInFlight,
   effect = effect.map(transform)
)

fun <Action> CancellableEffect<Action>.debounce(
   duration: Duration
): CancellableEffect<Action> = copy(effect = effect.debounce(duration))

fun <Action> CancellableEffect<Action>.throttle(
   duration: Duration
): CancellableEffect<Action> = copy(effect = effect.throttle(duration))
```

Додаток Б. Код файлу Reducer.kt

```
fun interface Reducer<State, Action> {

   fun reduce(state: State, action: Action): StateWithEffect<State, Action>
```

```
}

class EmptyReducer<State, Action> : Reducer<State, Action> {

   override fun reduce(
       state: State,
       action: Action
   ): StateWithEffect<State, Action> = state.withoutEffect()
}

class CombineReducer<State, Action>(
   private val reducers: List<Reducer<State, Action>>
) : Reducer<State, Action> {

   override fun reduce(state: State, action: Action): StateWithEffect<State,
Action> {
       return reducers.fold(state.withoutEffect()) { acc, reducer ->
           val (newState, effect) = reducer.reduce(acc.state, action)
           StateWithEffect(newState, acc.effect mergeWithEffect effect)
       }
   }
}

class OptionalReducer<State, Action, ChildState, ChildAction>(
   private val parentReducer: Reducer<State, Action>,
   private val childReducer: Reducer<ChildState, ChildAction>,
   private val toChildState: (State) -> ChildState?,
   private val toChildAction: (Action) -> ChildAction,
   private val updateParentState: (State, ChildState) -> State,
   private val toParentAction: (ChildAction) -> Action
) : Reducer<State, Action> {

   override fun reduce(state: State, action: Action): StateWithEffect<State,
Action> {
       val currentChildState = toChildState(state) ?: return
parentReducer.reduce(state, action)
       val currentChildAction = toChildAction(action)

       val (childState, childEffect) = childReducer.reduce(currentChildState,
currentChildAction)
       val (parentState, parentEffect) = parentReducer.reduce(state, action)

       return StateWithEffect(
           state = updateParentState(parentState, childState),
           effect = parentEffect mergeWithEffect childEffect.map(toParentAction)
       )
   }
```

```
}

data class StateWithEffect<State, Action>(
   val state: State,
   val effect: Effect<Action>
)

fun <State, Action> State.withoutEffect() = StateWithEffect<State, Action>(
   state = this,
   effect = Effect.none()
)

fun <State, Action> State.withSuspendEffect(
   id: String,
   cancelInFlight: Boolean = false,
   block: suspend () -> Action
) = StateWithEffect(
   state = this,
   effect = Effect.create(block).cancelable(id, cancelInFlight)
)

class ForEachReducer<State, Action, ElementState, ElementAction, ElementID>(
   private val reducer: Reducer<State, Action>,
   private val elementReducer: Reducer<ElementState, ElementAction>,
   private val toElementState: (State, ElementID) -> ElementState,
   private val toElementAction: (Action) -> Pair<ElementAction, ElementID>?,
   private val updateState: (State, ElementState, ElementID) -> State,
   private val toAction: (ElementAction, ElementID) -> Action,
) : Reducer<State, Action> {

   override fun reduce(state: State, action: Action): StateWithEffect<State,
Action> {
       val (elementAction, elementId) = toElementAction(action)
           ?: return reducer.reduce(state, action)

       val currentElementState = toElementState(state, elementId)
       val (elementState, elementEffect) = elementReducer.reduce(
           currentElementState,
           elementAction
       )

       val (parentState, parentEffect) = reducer.reduce(state, action)

       return StateWithEffect(
           state = updateState(parentState, elementState, elementId),
           effect = parentEffect mergeWithEffect elementEffect.map { toAction(it,
elementId) }
```

```
        )
    }
}

fun <State, Action, ElementState, ElementAction, ElementID> Reducer<State,
Action>.forEach(
    elementReducer: Reducer<ElementState, ElementAction>,
    toElementState: (State, ElementID) -> ElementState,
    toElementAction: (Action) -> Pair<ElementAction, ElementID>?,
    updateState: (State, ElementState, ElementID) -> State,
    toAction: (ElementAction, ElementID) -> Action
): Reducer<State, Action> = ForEachReducer(
    reducer = this,
    elementReducer = elementReducer,
    toElementState = toElementState,
    toElementAction = toElementAction,
    updateState = updateState,
    toAction = toAction
)

fun <State, Action, ElementState, ElementAction> Reducer<State, Action>.forEach(
    elementReducer: Reducer<ElementState, ElementAction>,
    toElementList: (State) -> List<ElementState>,
    toElementAction: (Action) -> Pair<ElementAction, Int>?,
    updateState: (State, ElementState, Int) -> State,
    toAction: (ElementAction, Int) -> Action
): Reducer<State, Action> = ForEachReducer(
    reducer = this,
    elementReducer = elementReducer,
    toElementState = { state, index -> toElementList(state)[index] },
    toElementAction = toElementAction,
    updateState = updateState,
    toAction = toAction
)

class ScopeReducer<State, Action, ChildState, ChildAction>(
    private val childReducer: Reducer<ChildState, ChildAction>,
    private val toChildState: (State) -> ChildState,
    private val toChildAction: (Action) -> ChildAction,
    private val updateState: (State, ChildState) -> State
) : Reducer<State, Action> {

    override fun reduce(state: State, action: Action): StateWithEffect<State,
Action> {
        val childAction = toChildAction(action)
        val childState = toChildState(state)
```

```
        childReducer.reduce(childState, childAction)

        return updateState(state, childState).withoutEffect()
    }
}
```

Додаток В. Код файлу Store.kt

```
interface Store<State, Action> {

    @NativeCoroutines
    val state: Flow<State>

    val currentState: State

    fun send(actions: List<Action>)

    fun send(action: Action) = send(listOf(action))

    fun <O> withState(block: (State) -> O): O = block(currentState)

    fun dispose()
}

class RootStore<State, Action>(
    initialState: State,
    private val reducer: Reducer<State, Action>,
    private val coroutineScope: CoroutineScope,
    private val exceptionHandler: ExceptionHandler
) : Store<State, Action> {

    private val mutableStateFlow = MutableStateFlow(initialState)

    override val state: Flow<State>
        get() = mutableStateFlow.asStateFlow()

    override val currentState: State
        get() = mutableStateFlow.value

    private val effectJobs = mutableMapOf<String, Job>()

    override fun send(actions: List<Action>) {
        coroutineScope.launch {
            val initial = currentState.withoutEffect<State, Action>()
            val (state, effect) = actions.fold(initial) { acc, action ->
```

```
                val (state, effect) = reducer.reduce(acc.state, action)
                StateWithEffect(state, acc.effect mergeWithEffect effect)
            }

            mutableStateFlow.value = state

            val effectJob = effect.run()
                .onEach { send(it) }
                .catch { exceptionHandler.handle(it) }
                .launchIn(coroutineScope)

            if (effect is CancellableEffect && effect.cancelInFlight) {
                effectJobs[effect.id]?.cancel()
                effectJobs.remove(effect.id)
                effectJobs[effect.id] = effectJob
            }
        }
    }

    override fun dispose() {
        coroutineScope.coroutineContext.cancelChildren()
    }
}

fun interface ExceptionHandler {
    fun handle(throwable: Throwable)
}

sealed class StoreException(override val message: String?): Exception() {
    class CancellationInFlightException(
        override val cause: Throwable?
    ): StoreException("Effect was cancelled on demand")

    class DisposeException: StoreException("Store was disposed")
}
```