ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

СОДЕРЖАНИЕ

1 Постановка исследовательской задачи

Целью курсовой работы является проведение сравнительного анализа архитектурных подходов для создания мультиплатформенных приложений на Kotlin Multiplatform (KMP) с особым акцентом на интеграцию с нативной платформой iOS. В рамках исследования будут рассмотрены и сопоставлены как традиционные паттерны iOS-разработки (MVVM, Redux), так и современные KMP-ориентированные стеки (MVIKotlin с Decompose, MVVM из пакета Jetpack ViewModel). Основная задача — выявить наиболее эффективное и удобное решение для проекта, где iOS является одной из ключевых платформ.

Для достижения поставленной цели предполагается решить следующие задачи:

1. Проанализировать теоретические основы и практическое применение архитектурных паттернов MVVM и Redux в контексте нативной iOS-разработки [1].
2. Исследовать доступные архитектурные решения для Kotlin Multiplatform, уделив внимание подходам к управлению состоянием (MVI, MVVM) и навигации (Decompose, Jetpack Navigation) [2].
3. Определить ключевые критерии для сравнения подходов: сложность интеграции со SwiftUI, объем переиспользуемого кода, простота тестирования, количество шаблонного кода и общий опыт разработчика.
4. На основе теоретического анализа и сформулированных критериев выбрать и обосновать наиболее подходящий архитектурный стек для дальнейшей реализации.
5. Разработать прототип приложения, чтобы на практике проверить гипотезы и оценить выбранную архитектуру.
6. Сформулировать выводы и рекомендации по применению исследуемых подходов в проектах на KMP с фокусом на iOS.

Данная работа является актуальной, поскольку технология Kotlin Multiplatform, несмотря на растущую популярность, еще не имеет устоявшихся архитектурных стандартов, в особенности для реализации навигации и управления состоянием [3]. Это создает значительные трудности для разработчиков, особенно для тех, кто переходит с нативной iOS-разработки и оказывается перед выбором:

адаптировать привычные паттерны, такие как MVVM, или осваивать новые, KMP-специфичные библиотеки, например, MVIKotlin и Decompose. Отсутствие четких сравнительных анализов и практических рекомендаций усложняет принятие архитектурных решений и повышает риски при старте новых проектов. Таким образом, систематизация и сравнительный анализ ключевых архитектурных подходов для KMP-приложений с фокусом на iOS представляет собой важную практическую задачу.

2 Теоретический обзор

2.1 Нативные подходы в iOS разработке

2.1.1 Паттерн Model-View-ViewModel (MVVM)

Model-View-ViewModel (MVVM) — это архитектурный паттерн проектирования, который упрощает разделение разработки графического интерфейса пользователя (UI) от бизнес-логики или логики представления [4]. Он был создан как ответ на недостатки таких паттернов, как MVC и MVP, и получил широкое распространение в разработке под iOS с появлением фреймворков Combine и SwiftUI, которые обеспечивают нативную поддержку для связывания данных (data binding).

Основные компоненты паттерна:

— Model (Модель) — представляет данные и бизнес-логику приложения. Она не имеет прямого знания о `ViewModel` и `View`.

— View (Представление) — отвечает за отображение данных пользователю и передачу его действий в `ViewModel`. В SwiftUI `View` декларативно описывает UI и подписывается на изменения в ‘ViewModel‘.

— ViewModel (Модель представления) — выступает посредником между `Model` и `View`. Она получает данные из `Model`, обрабатывает их и предоставляет в формате, удобном для отображения в `View`. `ViewModel` ничего не знает о конкретной реализации `View`.

Ключевой особенностью MVVM является механизм связывания данных (Data Binding). В экосистеме Apple он реализуется через фреймворк Combine, который позволяет `View` автоматически обновляться при изменении данных в `ViewModel`.

Пример реализации на Swift и SwiftUI:

В качестве примера рассмотрим экран, отображающий информацию о пользователе.

1. Model

   Это простая структура данных, представляющая бизнес-сущность. Пример представлен на листинге 1

```
struct User {
    let name: String
    let age: Int
}
```

Листинг 1: Model: простая структура данных

2. ViewModel

   `ViewModel` инкапсулирует логику представления, получая данные из `Model` и подготавливая их для `View`, как представлено на листинге 2.

```
import Combine

class UserViewModel: ObservableObject {
    @Published var userName: String = "Загрузка..."
    @Published var isPremium: Bool = false
    private var user: User?

    func fetchUser() {
        // Имитация асинхронной загрузки
        DispatchQueue.main.asyncAfter(deadline: .now() + 1) {
            self.user = User(name: "Иван", age: 30)
            self.userName = self.user?.name ?? ""
            self.isPremium = (self.user?.age ?? 0) > 25
        }
    }
}
```

Листинг 2: ViewModel: Управляет логикой представления

3. View

   `View` декларативно описывает интерфейс и подписывается на изменения в `ViewModel`, как представлено в листинге 3.

```swift
import SwiftUI

struct UserView: View {
    @StateObject private var viewModel = UserViewModel()

    var body: some View {
        VStack(spacing: 16) {
            Text(viewModel.userName)
                .font(.title)
            if viewModel.isPremium {
                Text("Премиум-пользователь")
                    .foregroundColor(.green)
            }
        }
        .onAppear(perform: viewModel.fetchUser)
    }
}
```

Листинг 3: View: Отображает данные и взаимодействует с пользователем

Преимущества MVVM:

— Разделение логики: Четко отделяет логику представления (`ViewModel`) от пользовательского интерфейса (`View`), что делает код более структурированным и легким для понимания.

— Высокая тестируемость: Так как `ViewModel` не имеет прямой зависимости от UI-компонентов, ее логику можно легко покрыть автоматизированными юнит-тестами.

— Связывание данных (Data Binding): Нативная поддержка со стороны SwiftUI и Combine позволяет автоматически синхронизировать данные между `ViewModel` и `View`, что сокращает количество шаблонного кода для обновления UI.

Недостатки:

— Проблема масштабирования: На сложных экранах, где множество компонентов взаимодействуют друг с другом, `ViewModel` может стать слишком громоздкой (т.н. "Massive ViewModel"). Это затрудняет поддержку и навигацию по коду.

— Синхронизация состояния: При дроблении логики на несколько `ViewModel` для одного экрана возникает проблема синхронизации состояния и обмена данными между ними, что усложняет поток данных.
— Неявный поток данных: В отличие от однонаправленных потоков (как в Redux/MVI), логика обработки действий пользователя и изменения состояния может быть разбросана по разным методам `ViewModel`, что делает поток данных менее предсказуемым и сложным для отладки.

2.1.2 Паттерн Redux

Redux — это архитектурный паттерн и библиотека для управления состоянием приложения, основанная на принципах функционального программирования. Хотя он зародился в экосистеме JavaScript (React), его концепции были успешно адаптированы и для нативной iOS-разработки, часто с использованием таких библиотек, как ReSwift или The Composable Architecture (TCA), которая во многом следует идеям Redux.

Ключевые принципы Redux:

— Единый источник истины (Single Source of Truth): Состояние всего приложения хранится в одном объекте — `Store`. Это упрощает отладку и инспектирование состояния.
— Состояние только для чтения (State is read-only): Единственный способ изменить состояние — это отправить (dispatch) `Action` (действие), которое является простым объектом, описывающим намерение изменения.
— Изменения вносятся чистыми функциями: Для обработки `Action` и обновления состояния используются `Reducers` (редьюсеры) — чистые функции, которые принимают текущее состояние и `Action`, а затем возвращают новое состояние.

Пример реализации на Swift (концептуальный):

Рассмотрим простой счетчик.

1. State, Actions и Reducer

   Определим состояние, возможные действия и чистую функцию для их обработки, как представлено в листинге 4.

```
struct CounterState {
    var count: Int = 0
}

enum CounterAction {
    case increment
    case decrement
}

func counterReducer(state: inout CounterState, action: CounterAction) {
    switch action {
    case .increment: state.count += 1
    case .decrement: state.count -= 1
    }
}
```

Листинг 4: State, Actions и Reducer: Определение состояния и действий

2. Store

   `Store` управляет состоянием и является единственным источником истины, как показано в листинге 5.

```swift
import Combine

class Store<State, Action>: ObservableObject {
    @Published private(set) var state: State
    private var reducer: (inout State, Action) -> Void

    init(initialState: State, reducer: @escaping (inout State, Action)
    ↪ -> Void) {
        self.state = initialState
        self.reducer = reducer
    }

    func dispatch(_ action: Action) {
        reducer(&state, action)
    }
}
```

Листинг 5: Store: Управление состоянием

3. View

   `View` отображает состояние из `Store` и отправляет действия при взаимодействии, как представлено в листинге 6.

```swift
import SwiftUI

struct CounterView: View {
    @StateObject private var store = Store(
        initialState: CounterState(),
        reducer: counterReducer
    )

    var body: some View {
        VStack {
            Text("Count: \(store.state.count)")
            HStack {
                Button("Increment") { store.dispatch(.increment) }
                Button("Decrement") { store.dispatch(.decrement) }
            }
        }
    }
}
```

Листинг 6: View: Отображение состояния и отправка действий

Преимущества Redux:

— Предсказуемость: Однонаправленный поток данных делает логику изменения состояния строгой и легкой для отслеживания.

— Централизованное состояние: Упрощает передачу данных между разными частями приложения без необходимости пробрасывать их через множество слоев.

— Отличные инструменты для отладки: Возможность "путешествия во времени"(time travel debugging), логирование всех действий.

Недостатки:

— Многословность (Boilerplate): Требует описания `State`, `Actions` и `Reducer`, что может быть избыточным для простых экранов.

— Сложен для понимания: Требует понимания принципов функционального программирования.

— Производительность: В очень больших приложениях с частыми обновлениями состояния могут возникнуть проблемы с производительностью, если не применять оптимизации.

2.2 Архитектурные подходы в Kotlin Multiplatform

2.2.1 Адаптация MVVM с Jetpack ViewModel

Стремление использовать привычный паттерн MVVM в KMP-проектах привело к появлению различных решений. Изначально `ViewModel` из Android Jetpack не была мультиплатформенной, что заставляло разработчиков создавать собственные реализации или использовать сторонние библиотеки (например, `moko-mvvm`).

Однако с недавнего времени Google официально поддерживает Jetpack ViewModel в Kotlin Multiplatform (`androidx.lifecycle:lifecycle-viewmodel-compose`) [**kvmodel**]. Это позволяет определять `ViewModel` в общем коде (`commonMain`), переиспользуя логику представления на всех платформах. На Android `ViewModel` автоматически интегрируется с жизненным циклом компонентов, а на iOS требует ручного создания и удержания.

Пример реализации:

1. ViewModel в общем коде (`commonMain`)

   `ViewModel` определяется в общем модуле и использует `StateFlow` для предоставления состояния, как показано в листинге 7.

```kotlin
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.launch

class GreetingViewModel : ViewModel() {
    private val _greetingText = MutableStateFlow("Загрузка...")
    val greetingText = _greetingText.asStateFlow()

    fun greet() {
        viewModelScope.launch {
            _greetingText.value = "Привет из KMP ViewModel!"
        }
    }
}
```

Листинг 7: ViewModel: Определение в общем коде

2. Интеграция с iOS и SwiftUI

   На стороне iOS для корректной интеграции со SwiftUI и подписки на `StateFlow` обычно создается класс-обертка, который является `ObservableObject`, как представлено в листинге 8.

```swift
import Foundation
import Combine
import shared

class GreetingIOSViewModel: ObservableObject {
    @Published var text: String = "..."

    private let kmpViewModel = GreetingViewModel()
    private var cancellable: AnyCancellable? = nil

    init() {
        // Установка подписки на StateFlow
        self.cancellable = kmpViewModel.greetingText
            .toPublisher()
            .receive(on: DispatchQueue.main)
            .sink { self.text = $0 }
    }

    func greet() {
        kmpViewModel.greet()
    }
}
```

Листинг 8: Интеграция с iOS: Класс-обертка для ViewModel

3. View в SwiftUI

   `View` работает уже с iOS-специфичной `ViewModel`, как показано в листинге 9.

```
import SwiftUI

struct ContentView: View {
    @StateObject private var viewModel = GreetingIOSViewModel()

    var body: some View {
        VStack {
            Text(viewModel.text)
            Button("Обновить", action: viewModel.greet)
        }
        .onAppear(perform: viewModel.greet)
    }
}
```

Листинг 9: View: Работа с iOS-специфичной ViewModel

Преимущества:

— Официальная поддержка: Подход поддерживается Google, что гарантирует его развитие и стабильность.

— Привычность для Android-разработчиков: Позволяет использовать знакомые концепции `ViewModel` и `viewModelScope`.

— Разделение логики: Сохраняется ключевое преимущество MVVM — отделение логики от UI.

Недостатки:

— Шаблонный код на iOS: Требуется создавать классы-обертки для каждой `ViewModel`, чтобы связать их со SwiftUI.

— Отсутствие управления навигацией: Данный подход решает только проблему `ViewModel`, но не предлагает общего решения для навигации между экранами.

— Управление жизненным циклом на iOS: Необходимо вручную создавать и удерживать `ViewModel` на стороне iOS, в то время как на Android это происходит автоматически.

2.2.2 MVIKotlin с Decompose

MVIKotlin — это библиотека для Kotlin Multiplatform, реализующая архитектурный паттерн Model-View-Intent (MVI) [5]. В сочетании с Decompose — библиотекой для управления навигацией и жизненным циклом компонентов [6] — она предоставляет полноценное решение для создания мультиплатформенных приложений.

Ключевые концепции MVI:

— Model — состояние компонента

— View — UI-слой, отображающий состояние

— Intent — действия пользователя или системы

— Store — компонент, обрабатывающий Intent'ы и обновляющий состояние

Пример реализации:

1. Определение состояния и действий

   Определим состояние и возможные действия, как показано в листинге 10.

```
data class CounterState(
    val count: Int = 0
)

sealed class CounterIntent {
    object Increment : CounterIntent()
    object Decrement : CounterIntent()
}
```

Листинг 10: Определение состояния и действий

2. Создание Store

   Создадим Store для обработки действий и обновления состояния, как представлено в листинге 11.

```
import com.arkivanov.mvikotlin.core.store.Store
import com.arkivanov.mvikotlin.main.store.DefaultStoreFactory

class CounterStore(
    storeFactory: DefaultStoreFactory
) : Store<CounterIntent, CounterState, Nothing> by storeFactory.create(
    name = "CounterStore",
    initialState = CounterState(),
    reducer = { intent, state ->
        when (intent) {
            is CounterIntent.Increment -> state.copy(count = state.count
            ↪  + 1)
            is CounterIntent.Decrement -> state.copy(count = state.count
            ↪  - 1)
        }
    }
)
```

Листинг 11: Создание Store

3. Создание компонента с Decompose

   Создадим компонент, который будет управлять жизненным циклом и состоянием, как показано в листинге 12.

```
import com.arkivanov.decompose.ComponentContext
import com.arkivanov.essenty.lifecycle.doOnDestroy

class CounterComponent(
    componentContext: ComponentContext,
    private val storeFactory: DefaultStoreFactory
) : ComponentContext by componentContext {
    private val store = CounterStore(storeFactory)

    val state = store.state

    fun increment() {
        store.accept(CounterIntent.Increment)
    }

    fun decrement() {
        store.accept(CounterIntent.Decrement)
    }

    init {
        lifecycle.doOnDestroy {
            store.dispose()
        }
    }
}
```

Листинг 12: Создание компонента с Decompose

4. Интеграция с iOS и SwiftUI

   Для интеграции Decompose со SwiftUI возможно использовать универсальную обертку, которая превращает `Value` из мира Decompose в `ObservableObject` для мира SwiftUI. Это позволяет избежать написания повторяющегося кода для каждого экрана. Пример представлен в листингах 13, 14.

```swift
import SwiftUI
import shared

class ObservableValue<T>: ObservableObject {
    @Published var value: T
    private var cancellation: Cancellation?

    init(_ value: Value<T>) {
        self.value = value.value
        self.cancellation = value.subscribe { [weak self] in
            self?.value = $0
        }
    }

    deinit {
        cancellation?.cancel()
    }
}
```

Листинг 13: Универсальная обертка ObservableValue

```swift
struct CounterView: View {
    @StateObject private var model: ObservableValue<CounterState>

    private let component: CounterComponent

    init(_ component: CounterComponent) {
        self.component = component
        _model = StateObject(wrappedValue: ObservableValue(component.state))
    }

    var body: some View {
        VStack(spacing: 16) {
            Text("Count: \(model.value.count)")
                .font(.title)

            HStack(spacing: 24) {
                Button("Increment") { component.increment() }
                Button("Decrement") { component.decrement() }
            }
        }
    }
}
```

Листинг 14: SwiftUI View, использующее обертку

Преимущества MVIKotlin с Decompose:

— Единый подход к навигации: Decompose предоставляет унифицированное решение для управления навигацией на всех платформах.

— Предсказуемый поток данных: Однонаправленный поток данных делает поведение приложения более предсказуемым и тестируемым.

— Мультиплатформенность: Все компоненты определены в общем коде, что обеспечивает максимальное переиспользование.

— Управление жизненным циклом: Decompose автоматически управляет жизненным циклом компонентов на всех платформах.

Недостатки:

— Сложность интеграции: Требует более глубокого понимания архитектуры и дополнительных настроек по сравнению с MVVM.

— Объем шаблонного кода: Необходимость определения состояний, интентов и редьюсеров для каждого компонента может привести к увеличению объема кода.
— Необходимо обучение: Разработчикам, привыкшим к традиционным паттернам, может потребоваться время для освоения MVI и Decompose.

3 Детали реализации

В рамках практической части работы было разработано мультиплатформенное приложение на Kotlin Multiplatform, демонстрирующее получение, отображение и навигацию по данным, полученным из открытого API.

3.1 Функциональность

Приложение реализует базовый сценарий взаимодействия с удаленным сервером и предоставляет следующий функционал:

1. Отображение списка персонажей: При запуске приложения открывается главный экран, на котором в виде списка отображаются персонажи, загруженные из Star Wars API (SWAPI). Реализована пагинация для постепенной подгрузки данных при прокрутке списка (см. рисунок 1).
2. Просмотр детальной информации: При нажатии на элемент списка происходит переход на экран с подробной информацией о выбранном персонаже (см. рисунок 2).
3. Обработка состояний: В приложении реализована обработка состояний загрузки (рисунок 3) и ошибок (рисунок 4) как для списка, так и для экрана с детальной информацией, что обеспечивает корректное отображение UI в различных сценариях.

3.2 Структура проекта

Проект имеет стандартную для Kotlin Multiplatform архитектуру и состоит из двух основных модулей:

— `shared`: Общий модуль, содержащий бизнес-логику, логику представления, сетевой слой и доменные модели. Код в этом модуле компилируется как для Android, так и для iOS.

— `iosApp`: Модуль, специфичный для платформы iOS, который содержит UI-слой, реализованный на SwiftUI, и обеспечивает интеграцию с общим кодом из модуля `shared`.

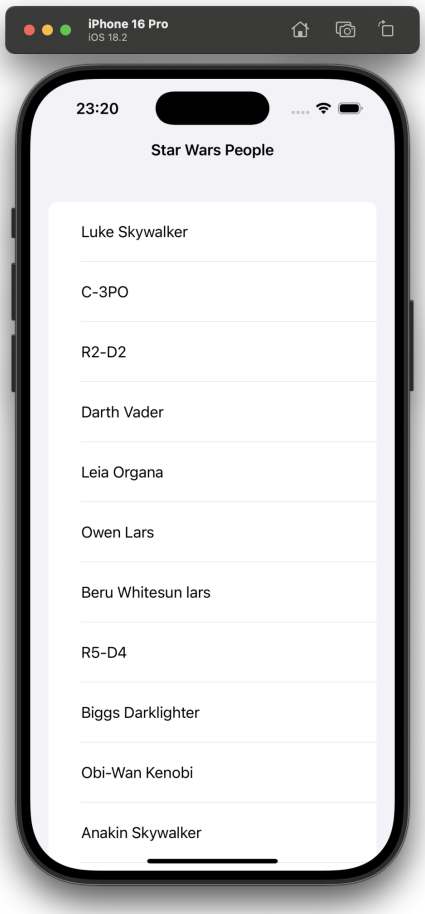

Рис. 1 — Экран со списком персонажей

— `composeApp`: Модуль, отвечающий за пользовательский интерфейс на Jetpack Compose. Используется для приложений для платформ Android и Desktop, в данной курсовой работе не рассматривается.

Модуль `shared` внутри разделен на три логических слоя, что соответствует принципам чистой архитектуры:

— `data`: Отвечает за получение данных из сети (с помощью Ktor) и их преобразование.

— `domain`: Содержит бизнес-сущности (модели) и интерфейсы репозиториев.

— `presentation`: Реализует логику представления с использованием MVIKotlin и управляет навигацией с помощью Decompose.

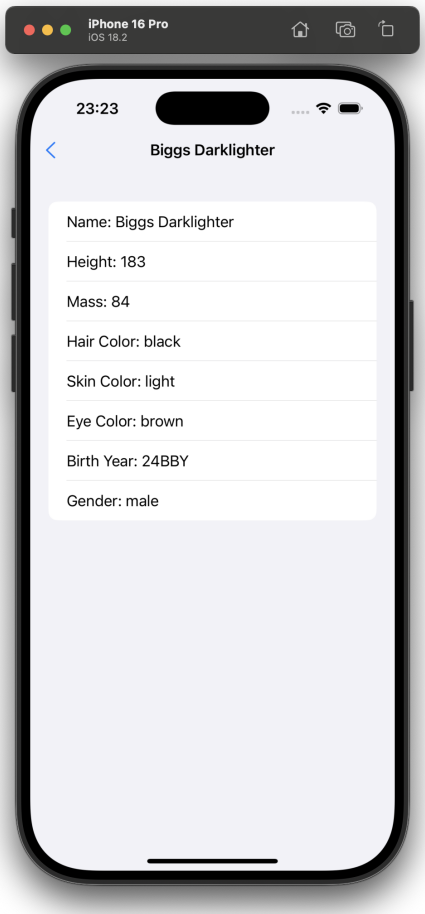

Рис. 2 — Экран с подробной информацией о персонаже

3.3 Используемые технологии и библиотеки

Для реализации проекта был выбран следующий стек технологий:

— **Kotlin Multiplatform**: Основа для создания переиспользуемого кода между iOS и Android.

— **Decompose**: Библиотека для мультиплатформенной навигации и управления жизненным циклом компонентов. Она позволила реализовать всю навигационную логику в общем модуле `shared`.

— **MVIKotlin**: Реализация паттерна MVI для управления состоянием. Использовалась для создания предсказуемого и тестируемого потока данных.

— **Ktor** [7]: Асинхронный сетевой фреймворк для выполнения HTTP-запросов к API.

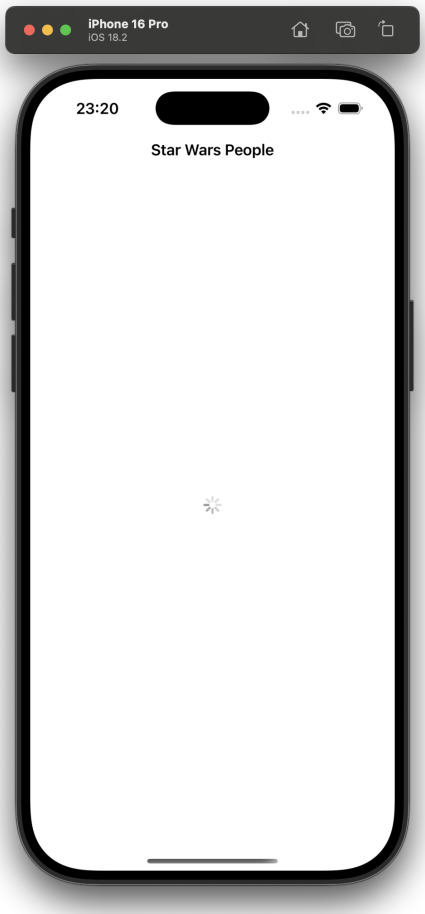

Рис. 3 — Экран в состоянии загрузки

— **Kotlinx.serialization**: Библиотека для сериализации и десериализации данных в формате JSON.
— **SwiftUI**: Декларативный фреймворк для построения UI на платформе iOS.

3.4 Сетевое взаимодействие и загрузка данных

Для выполнения сетевых запросов в общем модуле `shared` используется мультиплатформенная библиотека Ktor [7]. Она позволяет один раз описать логику взаимодействия с API и переиспользовать ее на всех платформах. Настройка клиента Ktor происходит в общем коде, где устанавливаются базовый URL и настраивается сериализация JSON с помощью `kotlinx.serialization`, как показано в листинге 15.

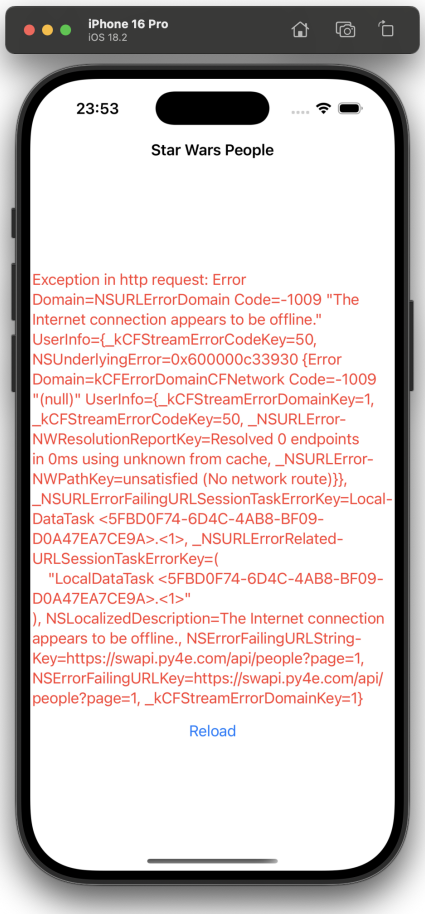

Рис. 4 — Экран в состоянии ошибки

```kotlin
// shared/data/network/HttpClient.kt
private val httpClient = HttpClient {
    install(ContentNegotiation) {
        json(Json {
            prettyPrint = true
            isLenient = true
            ignoreUnknownKeys = true
        })
    }
}
```

Листинг 15: Настройка HTTP-клиента Ktor

Загрузка данных инкапсулирована в `Store`-компонентах (MVIKotlin), таких как `ListStore` и `DetailsStore`. Они инициируют асинхронную загрузку данных в `Bootstrapper` или `Executor`, используя корутины.

3.5 Управление состоянием и пагинация

Управление состоянием экранов реализовано с помощью MVIKotlin. Для каждого экрана (`ListComponent`, `DetailsComponent`) создан свой `Store`, который отвечает за обработку действий пользователя (`Intent`), выполнение бизнес-логики (`Executor`) и обновление состояния (`Reducer`).

Пагинация реализована на экране списка. Когда пользователь доходит до последнего видимого элемента, UI-слой на стороне iOS отправляет в `ListComponent` намерение `onLoadNextPageClicked`, как показано в листинге 16.

```
// iosApp/ListView.swift
List {
    ForEach(model.items, id: \.url) { item in
        // ...
        .onAppear {
            if item == model.items.last {
                component.onLoadNextPageClicked()
            }
        }
    }
}
```

Листинг 16: Инициирование пагинации в SwiftUI

Это намерение обрабатывается в `ListStore`, который, в свою очередь, запускает загрузку следующей страницы данных, как показано в листинге 17.

```kotlin
// shared/presentation/list/ListStore.kt
private inner class ExecutorImpl : CoroutineExecutor<ListStore.Intent,
    Action, ListStore.State, Msg, Nothing>() {
    private var currentPage = 1

    override fun executeIntent(intent: ListStore.Intent) {
        when(intent) {
            is ListStore.Intent.LoadNextPage -> {
                if (!state().isLastPage && !state().isLoading) {
                    loadPage(currentPage + 1)
                }
            }
            // ...
        }
    }
}
```

Листинг 17: Обработка пагинации в ListStore

3.6 Обработка ошибок и интеграция с SwiftUI

В случае ошибки при выполнении сетевого запроса `Store` перехватывает исключение и сохраняет сообщение об ошибке в своем состоянии, как показано в листинге 18.

```kotlin
// shared/presentation/list/ListStore.kt
private fun loadPage(page: Int) {
    scope.launch(SupervisorJob()) {
        dispatch(Msg.Loading(true))
        peopleRepository.getPeople(page)
            .onSuccess {
                // ...
            }
            .onFailure {
                dispatch(Msg.Error(it.message ?: "Unknown Error"))
            }
    }
}
```

Листинг 18: Обработка ошибок сетевого запроса

Состояние из `Store` (включая ошибку) передается в `Component`, а затем в UI-слой iOS через обертку `ObservableValue`. SwiftUI-представление подписывается на изменения этого объекта и декларативно отображает либо данные, либо индикатор загрузки, либо сообщение об ошибке с кнопкой для повторной попытки (листинг 19). Примеры состояний представлены на рисунках 1, 2, 3, 4.

```swift
// iosApp/ListView.swift
ZStack {
    if let error = model.error, !model.isLoading {
        VStack(spacing: 16) {
            Text(error)
                .foregroundColor(.red)
            Button("Reload", action: component.onReloadClicked)
        }
    } else {
        // ... отображение списка
    }

    if model.items.isEmpty, model.isLoading {
        ProgressView()
    }
}
```

Листинг 19: Отображение состояния ошибки в SwiftUI

Этот подход позволяет полностью инкапсулировать логику загрузки данных и обработки ошибок в общем `shared`-модуле, оставляя для iOS-приложения только задачу отображения готового состояния.

4 Анализ и выводы

На основе практической реализации и теоретического обзора был проведен анализ выбранного архитектурного подхода — связки MVIKotlin и Decompose — в контексте iOS-ориентированного проекта на Kotlin Multiplatform [3].

4.1 Сильные стороны

Выбранный стек продемонстрировал ряд значительных преимуществ, ключевых для мультиплатформенной разработки:

— Максимальное переиспользование кода. Вся бизнес-логика, логика представления (управление состоянием) и даже навигация были вынесены в общий модуль `shared`. Это сокращает дублирование кода и упрощает синхронизацию версий приложения для разных платформ.

— Масштабируемость. Компонентная модель Decompose позволяет легко добавлять новые экраны и потоки, изолируя их логику. Такой подход предотвращает разрастание классов и смешивание ответственности, в отличие от потенциальной проблемы «Massive ViewModel» в классическом MVVM.

— Нативная производительность. Поскольку UI-слой для каждой платформы остается полностью нативным (SwiftUI для iOS), приложение сохраняет высокую производительность и отзывчивость, свойственную нативным решениям. Общая логика компилируется в нативный код, не создавая дополнительных издержек.

— Платформонезависимая отладка. Бизнес-логику и логику представления можно разрабатывать, тестировать и отлаживать в общем модуле, используя, например, desktop-запуск. Это позволяет вести основную часть разработки, не имея под рукой iOS-устройства или MacBook, что значительно ускоряет и упрощает процесс.

— Предсказуемость и тестируемость. Паттерн MVI обеспечивает строгий однонаправленный поток данных, что делает поведение приложения предсказуемым, а отладку — прозрачной. Логика обновления состояния инкапсулирована в чистых функциях-редьюсерах, которые легко покрываются юнит-тестами.

4.2 Слабые стороны

Несмотря на весомые преимущества, у подхода есть и недостатки:

— Высокий порог входа. Для эффективного использования стека MVIKotlin + Decompose разработчикам, особенно привыкшим к классическому MVVM, требуется время на освоение новых концепций и библиотек.

— Увеличенный объем шаблонного кода. Реализация каждого экрана требует описания `State`, `Intent` и `Store`, что для простых случаев может показаться избыточным по сравнению с лаконичностью MVVM.

— Интеграционный слой. Для связи общего кода с нативным UI (SwiftUI) требуется создание дополнительного слоя-адаптера, что добавляет сложности, хотя и решается универсальными обертками.

5 Заключение

Выбранный архитектурный стек MVIKotlin + Decompose является мощным и стратегически верным решением для средних и крупных KMP-проектов, где важны долгосрочная поддержка, масштабируемость, максимальное переиспользование кода и строгая архитектура. Он идеально подходит для команд, готовых инвестировать время в изучение подхода, чтобы получить предсказуемую и легко тестируемую кодовую базу [2].

Для небольших проектов или прототипов, где скорость разработки является ключевым фактором, а команда не знакома с MVI, более прагматичным выбором может стать адаптация MVVM. Однако в таком случае вопросы навигации и межэкранного взаимодействия придется либо решать отдельно для каждой платформы, что снизит объем переиспользуемого кода, либо написать собственное решение `ViewModel` поверх Decompose, не используя MVIKotlin, что потребует от разработки изучения данного фреймворка.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Olaoye A. Basics of iOS Application Development //. — 02.2022. — С. 1—23. — DOI: 10.1007/978-1-4842-8023-2_1.
2. Nurpeissov Y. Master Kotlin Multiplatform with Decompose and MVI // Medium. — 2025. — URL: https://medium.com/@yeldar.nurpeissov/master-kotlin-multiplatform-with-decompose-and-mvi-part4-e61dac75d5f9.
3. Mikhalchenkov M. MVIKotlin in Practice: A Modern Architecture framework for Android and KMP // Medium. — 2025. — URL: https://medium.com/@mikhaltchenkov/mvikotlin-in-practice-a-modern-architecture-framework-for-android-and-kmp-ca68e58be94b.
4. Fuksa M., Speth S., Becker S. MVVM Revisited: Exploring Design Variants of the Model-View-ViewModel Pattern. — 2025. — arXiv: 2504.18191 [cs.SE].
5. Ivanov A. MVIKotlin – Extendable MVI framework for Kotlin Multiplatform. — URL: https://arkivanov.github.io/MVIKotlin/. [Электронный ресурс].
6. Ivanov A. Decompose – Kotlin Multiplatform lifecycle-aware business logic components. — URL: https://arkivanov.github.io/Decompose/. [Электронный ресурс].
7. Ktor framework [Электронный ресурс]. — URL: https://ktor.io.