Министерство науки и высшего образования Российской Федерации

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»

ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ

КАФЕДРА КОМПЬЮТЕРНЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

На правах рукописи

УДК 004.75

БУЛЬДИН ИЛЬЯ ДМИТРИЕВИЧ

СОЗДАНИЕ МОБИЛЬНОГО ПРИЛОЖЕНИЯ SIP-ТЕЛЕФОНИИ НИЯУ МИФИ

Выпускная квалификационная работа бакалавра

Направление подготовки 09.03.01 Информатика и вычислительная техника

|  |
| --- |
| Выпускная квалификационная работа защищена  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2023 г. |
| Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Секретарь ГЭК \_\_\_\_\_\_\_\_\_\_\_\_\_ |

г. Москва

2023

Студент-дипломник \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Бульдин И.Д. /

Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Романов Н.Н. /

Рецензент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Овчаренко Е.С. /

Заведующий кафедрой №12 \_\_\_\_\_\_\_\_\_\_\_\_\_\_ / Иванов М.А. /

**АННОТАЦИЯ**

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc122884539)

[1 Обзорная часть 7](#_Toc122884540)

[1.1 Описание языков программирования для разработки iOS приложений 7](#_Toc122884541)

[1.2 Использование императивного и декларативного подхода при разработке iOS приложений 9](#_Toc122884542)

[1.3 Выбор архитектурного паттерна 11](#_Toc122884543)

[2 Теоретическая часть 13](#_Toc122884544)

[2.1 SIP-телефония 13](#_Toc122884545)

[2.2 Анализ конкурентов и определение требований к системе 14](#_Toc122884546)

[**2.2.1 Анализ существующих решений от конкурентов** 14](#_Toc122884547)

[**2.2.2 Определение требований к мобильному приложению** 15](#_Toc122884548)

[2.3 Проектирование UX и UI системы 17](#_Toc122884549)

[**2.3.1 Экран набора номера** 17](#_Toc122884550)

[**2.3.2 Экран каталога абонентов и подразделений организации** 18](#_Toc122884551)

[**2.3.3 Экран недавних вызовов** 20](#_Toc122884552)

[**2.3.4 Экран профиля текущего SIP-аккаунта** 22](#_Toc122884553)

[**2.3.5 Экран управления учетными записями** 23](#_Toc122884554)

[**2.3.6 Экран авторизации** 25](#_Toc122884555)

[2.4 Архитектура мобильного приложения 27](#_Toc122884556)

[3 Программная реализация программного обеспечения мобильного iOS приложения для SIP-телефонии НИЯУ МИФИ 29](#_Toc122884557)

[3.1 Разработка сетевого слоя для взаимодействия с API НИЯУ МИФИ 29](#_Toc122884558)

[**3.1.1 Протокол API** 30](#_Toc122884559)

[**3.1.2 Перечисление MephiAPI** 30](#_Toc122884560)

[**3.1.3 Класс NetworkingService** 31](#_Toc122884561)

[3.2 Разработка слоя навигации с использованием паттерна Coordinator 32](#_Toc122884562)

[**3.2.1 Протоколы CoordinatorProtocol и FlowCoordinatorProtocol** 32](#_Toc122884563)

[**3.2.2 Класс ApplicationCoordinator и протокол MainCoordinatorProtocol** 32](#_Toc122884564)

# **ВВЕДЕНИЕ**

В наше время мобильные телефоны стали неотъемлемой частью нашей жизни, и возможность общения с помощью интернета стала доступна для большинства людей. SIP-телефония представляет собой технологию, которая позволяет осуществлять звонки через интернет.

Таким образом, актуальной научной задачей является изучение существующих решений в области создания мобильных приложений SIP-телефонии, анализ их функциональности и разработка мобильного приложения SIP-телефонии для операционной системы iOS, которое будет удовлетворять требованиям современной SIP-телефонии, учитывая специфику использования внутри НИЯУ МИФИ.

Целью работы является разработка мобильного приложения SIP-телефонии, которое будет обладать необходимым функционалом, удобным интерфейсом, высокой степенью надежности и предоставлять сотрудникам удобный инструмент для общения с помощью SIP-телефонии.

Для достижения поставленной цели необходимо решить следующие задачи:

* Изучение существующих решений в области мобильных приложений SIP-телефонии, анализ функциональности и возможностей этих приложений;
* Выбор технологий и инструментов, которые должны использоваться при разработке;
* Формулирование требований к разрабатываемому ПО;
* Проектирование UX и UI;
* Разработка архитектуры и интерфейса мобильного приложения, учитывая требования современной SIP-телефонии и особенности использования на мобильных устройствах с операционной системой iOS;
* Реализация функциональности мобильного приложения, обеспечивающей выполнение основных сформулированных требований;
* Тестирование разработанного ПО (локальное и с помощью службы бета-тестирования iOS приложений TestFlight)

Был проведен анализ существующих решений и разработана архитектура приложения. Для решения поставленной задачи были использованы современные технологии и инструменты, такие как Swift, UIKit, SnapKit и среда разработки Xcode, а также Abto VoIP SIP SDK для удобной работы с SIP-протоколом.

В рамках данной работы было разработано мобильное приложение, которое позволяет осуществлять звонки и общаться с помощью SIP-протокола. В приложении реализованы функции, такие как авторизация, каталог абонентов НИЯУ МИФИ, добавление контактов в раздел «Избранное», Multi Accounting, возможность осуществления звонков и т.д. Разработанное приложение прошло тестирование и отладку, что позволило убедиться в его стабильной работе и функциональности.

Разработка мобильного приложения SIP-телефонии для платформы iOS является актуальной и перспективной задачей. Одним из основных преимуществ использования SIP-телефонии является экономия средств на телефонные разговоры. Также SIP-телефония обеспечивает гибкость в настройке телефонной сети и возможность удаленной работы.

# **1 Обзорная часть**

## **1.1 Описание языков программирования для разработки iOS приложений**

Язык программирования является важным инструментом для разработки программного обеспечения, в том числе и мобильных приложений. При разработке iOS приложений существует несколько языков, которые могут быть использованы. В данной главе рассмотрены основные языки, которые можно использовать для разработки iOS приложений, их особенности, возможности и ограничения. Также приведено сравнение этих языков, обоснован выбор наиболее подходящего.

Для разработки iOS приложений можно использовать следующие языки программирования:

* Swift – язык программирования, созданный Apple для разработки iOS, macOS, watchOS и tvOS. Он является официальным языком разработки приложений для Apple и предлагает множество удобных функций и возможностей для разработки;
* Objective-C – язык программирования, который был ранее использован как основной язык разработки приложений для iOS, но сейчас уступает место Swift. Objective-C является объектно-ориентированным языком и использует динамическую типизацию;
* Kotlin/Native – язык программирования, разработанный компанией JetBrains, который позволяет написать кросс-платформенное приложение одной кодовой базой и скомпилировать его в нативный код для iOS;
* C++ – мощный язык программирования, который может использоваться для разработки iOS приложений с помощью таких инструментов как как Clang и LLVM. Однако, так как C++ является низкоуровневым языком, его использование требует большего опыта и знаний, чем более высокоуровневые языки, такие как Swift или Objective-C. В дополнение к этому, C++ поддерживается Apple неофициально, что может привести к нестабильной работе и проблемам с компиляцией;
* Другие языки, такие как Java и Python, не являются официально поддерживаемыми Apple и могут быть очень сложными для использования

Разработка кросс-платформенного приложения не рассматривается организацией. В целом, для нативной разработки iOS приложений рекомендуется использовать Swift или Objective-C.

Swift является более современным и удобным в использовании языком программирования, чем Objective-C. Он имеет синтаксис с высокой читабельностью, использует статическую типизацию, безопасен и предлагает больше современных функций и возможностей, чем Objective-C. Кроме того, Swift быстрее и эффективнее, чем Objective-C, благодаря использованию оптимизированных структур данных и алгоритмов. Поэтому для разработки мобильного приложения SIP-телефонии НИЯУ МИФИ был выбран язык программирования Swift.

## **1.2 Использование императивного и декларативного подхода при разработке iOS приложений**

UIKit и SwiftUI – это два разных фреймворка, используемых для создания пользовательского интерфейса в iOS-приложениях.

UIKit является стандартным фреймворком для разработки пользовательского интерфейса в iOS с момента выпуска первой версии iOS. Он предоставляет большое количество готовых элементов интерфейса, таких как кнопки, таблицы, коллекции и т.д., а также множество инструментов для управления их взаимодействием и анимацией. Однако UIKit требует большого количества кода для реализации даже простых интерфейсов, что может усложнить разработку и увеличить вероятность ошибок.

SwiftUI – это новый фреймворк, введенный Apple в 2019 году. Он предназначен для создания пользовательского интерфейса с помощью декларативного языка описания, который позволяет описывать интерфейс в терминах его состояний и поведения, а не в терминах низкоуровневых действий, которые нужно выполнить. Это позволяет создавать интерфейсы быстрее и проще, так как не нужно писать много кода для реализации отдельных элементов интерфейса.

В целом, оба фреймворка могут быть использованы для создания пользовательского интерфейса в iOS-приложениях, SwiftUI является более современным и удобным в использовании из-за его декларативного подхода и автоматической обработки изменений состояния. Однако предпочтение при разработке было отдано UIKit.

Далее описаны причины выбора императивного подхода для создания интерфейса iOS приложения:

* Более ранняя поддержка. UIKit уже был в iOS гораздо раньше, чем SwiftUI, поэтому он более широко распространен и лучше поддерживается;
* Больше контроля. UIKit предоставляет больше возможностей для настройки и контроля поведения элементов интерфейса. Это может быть полезно при создании таблиц с динамической высотой ячеек в зависимости от количества текста;
* Больше гибкости и возможностей. UIKit предоставляет больше функциональности и возможностей, чем SwiftUI, включая различные элементы интерфейса, анимации и инструменты управления контентом. Это может быть полезно при создании сложных интерфейсов с множеством индивидуальных элементов.

## **1.3 Выбор архитектурного паттерна**

Выбор архитектурного паттерна - важный шаг в разработке приложения для iOS. Архитектурный паттерн определяет структуру приложения и способ взаимодействия его различных частей. Правильный выбор архитектурного паттерна может упростить разработку, улучшить поддержку кода и повысить гибкость программного кода.

Существует несколько популярных архитектурных паттернов, которые можно использовать в приложениях для iOS, таких как MVC (Model-View-Controller), MVP (Model-View-Presenter), MVVM (Model-View-ViewModel) и VIPER. Каждый из этих паттернов имеет свои сильные и слабые стороны, и выбор паттерна зависит от конкретных требований и целей приложения.

Проведено сравнение различных архитектурных паттернов, их ключевые характеристики, чтобы повысить масштабируемость, поддерживаемость и тестируемость программного кода. Разница между MVC, MVP и MVVM несущественна. Controller «знает» и о модели, и о слое представления. Presenter «знает» о модели и зависит от слоя представления. ViewModel похожа на Presenter, но она отвязана от пользовательского интерфейса и слоя данных, потому что преобразует данные, полученные из модели в формат, удобный для слоя представления.

Использование архитектурного паттерна MVС для приложения SIP-телефонии может быть хорошим выбором по нескольким причинам:

* Понятное и удобное разделение логики на изолированные компоненты. Это облегчает внесение изменений и поддержку приложения.
* Возможность переиспользования кода;
* Тестируемость. Разделение задач в MVC облегчает независимое тестирование различных частей приложения, что помогает обеспечить надежность и отсутствие ошибок в приложении;
* Поддерживаемость. Модель MVC облегчает обслуживание приложения с течением времени, поскольку изменения в модели или контроллере не требуют изменений в слое представления, и наоборот

В целом, архитектурный паттерн MVC может помочь сделать разработку приложения SIP-телефонии более эффективной и удобной, что может быть особенно важно для приложения, которое должно быть надежным и отзывчивым. Кроме того, MVC – это широко используемый и устоявшийся архитектурный паттерн, который знаком многим разработчикам, что может облегчить поиск ресурсов и поддерживаемость кода.

# **2 Теоретическая часть**

## **2.1 SIP-телефония**

Телефония с использованием Session Initial Protocol (SIP) – это метод совершения и приема телефонных звонков через Интернет с использованием широкополосного соединения и учетной записи SIP.

SIP – это стандартизированный протокол, используемый для инициирования, поддержания, изменения и завершения сеансов связи в реальном времени, включающих видео, голос, обмен сообщениями и другие виды связи между устройствами. SIP обычно используется в сочетании с другими протоколами, такими как RTP (Real-Time Transport Protocol) и RTCP (Real-Time Transport Control Protocol).

SIP использует сообщения, известные как SIP-requests и SIP-responses, для инициирования и управления сеансами связи. Эти сообщения отправляются между SIP-клиентами, которые представляют собой устройства или программы, использующие SIP, и SIP-серверами, которые представляют собой серверы, маршрутизирующие и управляющие SIP-запросами. SIP часто используется в системах VoIP (Voice Over IP) для установления и управления голосовыми и видеозвонками через Интернет.

Одним из основных преимуществ SIP-телефонии является то, что она позволяет пользователям совершать и принимать звонки из любого места, где есть широкополосное соединение. Это может быть особенно полезно для предприятий и организаций, сотрудники которых работают удаленно или в разных местах.

Существует множество различных приложений для SIP-софтфонов и аппаратных устройств, которые позволяют пользователям использовать преимущества SIP-телефонии. Далее в пункте 2.2 проведен анализ конкурентов и сформулированы требования для разрабатываемой системы.

## **2.2 Анализ конкурентов и определение требований к системе**

### **2.2.1 Анализ существующих решений от конкурентов**

Хотя существует множество мобильных приложений SIP-софтфонов для операционной системы iOS, они не всегда подходят для внутренних звонков. Это связано с тем, что такие приложения, как правило, предназначены для общего пользования, а не адаптированы к специфическим потребностям организации.

Прямым конкурентом является решение от компании Grandstream – GS Wave Lite. Wave Lite – это мобильное приложение, которое позволяет пользователям совершать и принимать телефонные звонки, используя широкополосное соединение и учетную запись SIP.

Одно из преимуществ Wave Lite заключается в том, что оно имеет широкий функционал и большое количество пользовательских настроек. Приложение включает такие функции, как переадресация вызовов, голосовая почта и текстовые сообщения. Еще одним преимуществом Wave Lite является поддержка HD аудио и видео звонков. На рисунке 2.2.1.1 представлен интерфейс мобильного приложения GS Wave Lite.

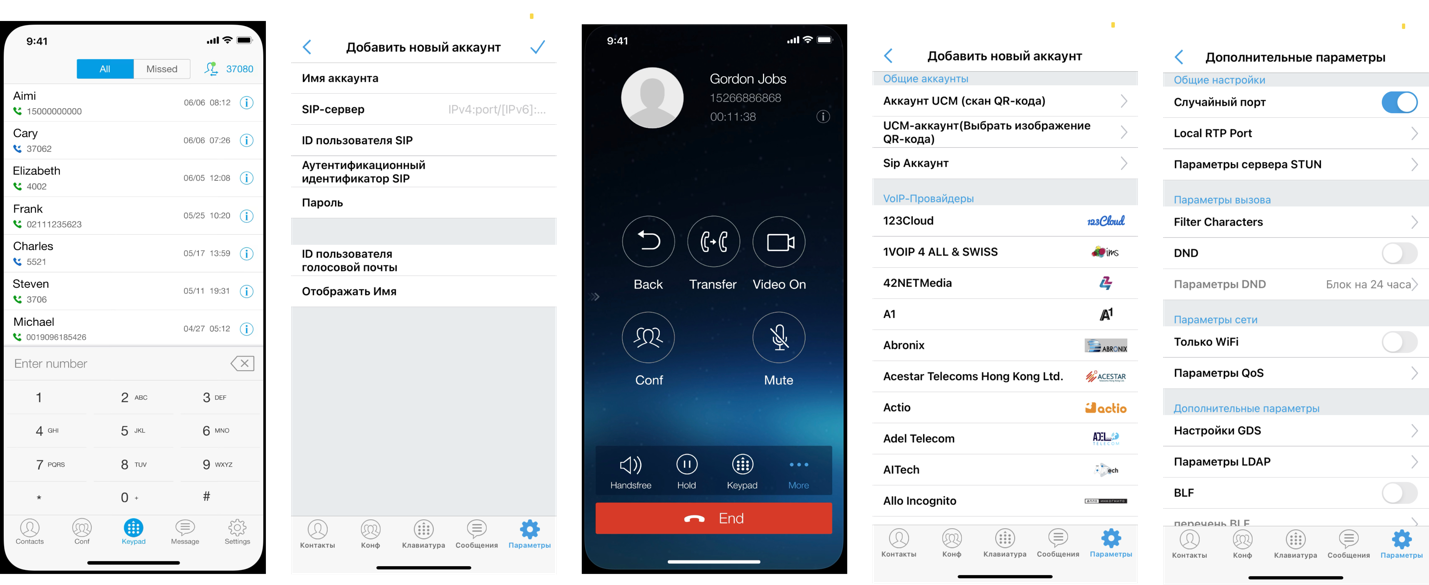


Рис. 2.2.1.1 - Интерфейс мобильного приложения GS Wave Lite

Однако есть и несколько недостатков использования Wave Lite. Одним из них является то, что это приложение стороннего производителя, и поэтому оно не так хорошо интегрируется с внутренними системами в организации, как пользовательское решение. Wave Lite не предусматривает решения одного из основных требований к системе – наличие каталога сотрудников организации, а из-за большого количества настроек процесс авторизации, например, может стать трудной задачей для пользователя.

Еще одним недостатком является то, что Wave Lite может не обеспечивать такой же уровень безопасности, как пользовательское решение.

Кроме того, НИЯУ МИФИ уже предоставляет собственное решение – VoIP MEPhI – для операционной системы Android. Наличие нескольких клиентов мобильного приложения от разных производителей может затруднить эффективное общение пользователей.

Учитывая преимущества и недостатки, для организации может быть лучше создать собственное мобильное приложение для SIP-телефонии. Это позволит адаптировать приложение к своим конкретным потребностям, обеспечить его хорошую интеграцию и предложить дополнительные функции и возможности, которых нет в готовых приложениях для SIP-софтфонов. В пункте 2.2.2 главы 2 определяются требования к разрабатываемому мобильному приложению

### **2.2.2 Определение требований к мобильному приложению**

Система должна отвечать следующим требованиям:

* Поддержка нескольких учетных записей. Приложение должно позволять пользователям настраивать и управлять несколькими SIP-аккаунтами, включая возможность переключения между ними по мере необходимости;
* Каталог абонентов и подразделений НИЯУ МИФИ. Приложение должно реализовывать функционал навигации и поиска по каталогу, а также кэширование ранее посещенных уровней вложенности каталога для доступа в ситуациях отсутствия сети;
* Каталог избранных пользователей. Приложение должно предоставлять пользователю возможность добавления абонентов в список «Избранное» для быстрого доступа к интересующим контактам;
* Недавние звонки. Система должна сохранять недавно совершенные звонки, а также отображать их в стандартном приложении «Телефон» на iOS устройствах;
* Высококачественные аудио звонки;
* Возможности настройки. Приложение должно позволять пользователям настраивать различные параметры приложения, например, доступность SIP-аккаунта для звонка;
* Простота использования. Приложение должно быть простым в использовании и иметь удобным интерфейсом, позволяющим пользователям быстро совершать и принимать звонки;
* Поддерживаемость. Исходный код программы должен соответствовать основным принципам разработки, таким как SOLID, KISS, DRY и т.д., а также быть задокументирован.

## **2.3 Проектирование UX и UI системы**

### **2.3.1 Экран набора номера**

Экран набора номера в мобильном приложении VoIP MEPhI предназначен для того, чтобы пользователи могли, набирая номера, легко совершать телефонные звонки.

С точки зрения UX (пользовательского опыта) экран набора номера спроектирован так, чтобы быть простым и интуитивно понятным в использовании. Он представляет пользователю клавиатуру, на которой отображаются цифры от 0 до 9, а также другие кнопки для специальных символов и действий (символы "\*" и "#", кнопка «Вызов» и кнопка «Стереть»). Клавиатура расположена в виде привычной сетки, которую легко понять и использовать.

С точки зрения пользовательского интерфейса экран набора номера использует чистые, современные элементы дизайна для создания визуально привлекательного интерфейса. Кнопки клавиатуры большие и легко нажимаются, они окружены большим количеством белого пространства, чтобы их было легко отличить друг от друга. На экране также имеется визуальная обратная связь при нажатии кнопки, а именно highlighting-состояние кнопки и звук, если телефон не находится в беззвучном режиме, при нажатии, чтобы помочь пользователям подтвердить, что их ввод был зарегистрирован.

При наборе номера размер шрифта уменьшается в зависимости от количества цифр.

Для набора номер был выбран именно такой подход, вместо набора с помощью компонента UITextField. Это связано с тем, что номер абонента имеет следующий вид:

<sip:20023@pbx.mephi.ru> (1)

sip – указание на то, что при звонке будет использован протокол SIP;

20023 – добавочный номер абонента в системе;

pbx.mephi.ru – домен

Таким образом, добавочный номер абонента всегда состоит только из цифр для домена НИЯУ МИФИ.

В целом, UX и UI экрана набора номера в приложении схож с дизайном в стандартном приложении "Телефон" на устройствах iOS для обеспечения простого и интуитивно понятного интерфейса для совершения телефонных звонков с акцентом на удобство использования и эстетику.

Интерфейс экрана набора номера представлен на рисунке 2.3.1.1.

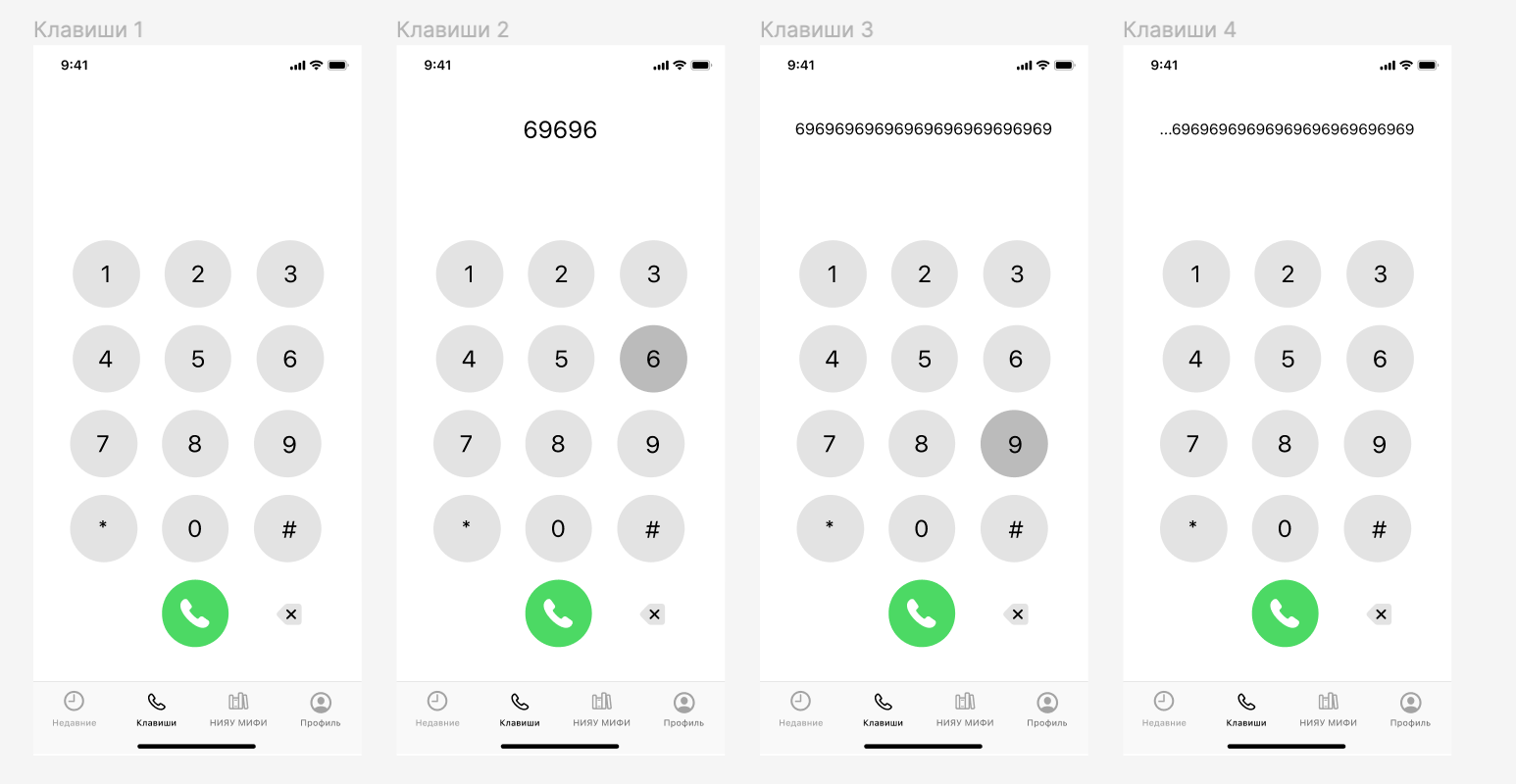


Рис. 2.3.1.1 - Интерфейс экрана набора номера

### **2.3.2 Экран каталога абонентов и подразделений организации**

Этот экран для мобильного приложения для разработан как справочник абонентов и подразделений университета. Он имеет минималистичный дизайн, который соответствует iOS HIG (Human Interface Guidelines).

В верхней части экрана находится поле ввода текста, которое позволяет пользователям мобильного приложения искать конкретное подразделение или абонента в каталоге. Результаты поиска отображаются в виде таблицы UITableView, где каждое подразделение или абонент представлены в виде ячейки UITableViewCell, при этом таблица поделена на секции, где первая секция – абоненты, вторая – подразделения, при этом на определенном уровне вложенности каталога может не быть одной из секций, тогда она не должна отображаться, чтобы не вводить пользователя в заблуждение.

Нажав на ячейку подразделения в таблице, пользователь переходит на следующий уровень вложенности справочника, что позволяет углубиться и просмотреть абонентов в интересующем подразделении. При нажатии на ячейку абонента отображается дополнительная информация о сотруднике, например, его должность и контактная информация. Ячейка абонента также включает кнопку вызова и кнопку добавления абонента в каталог избранных контактов текущего пользователя SIP-телефонии.

Высота ячеек для подразделений динамически увеличивается в зависимости от длины текста, что позволяет ячейкам правильно отображать длинные названия подразделений, не обрывая их. Интерфейс экрана каталога представлен на рисунке 2.3.2.1.

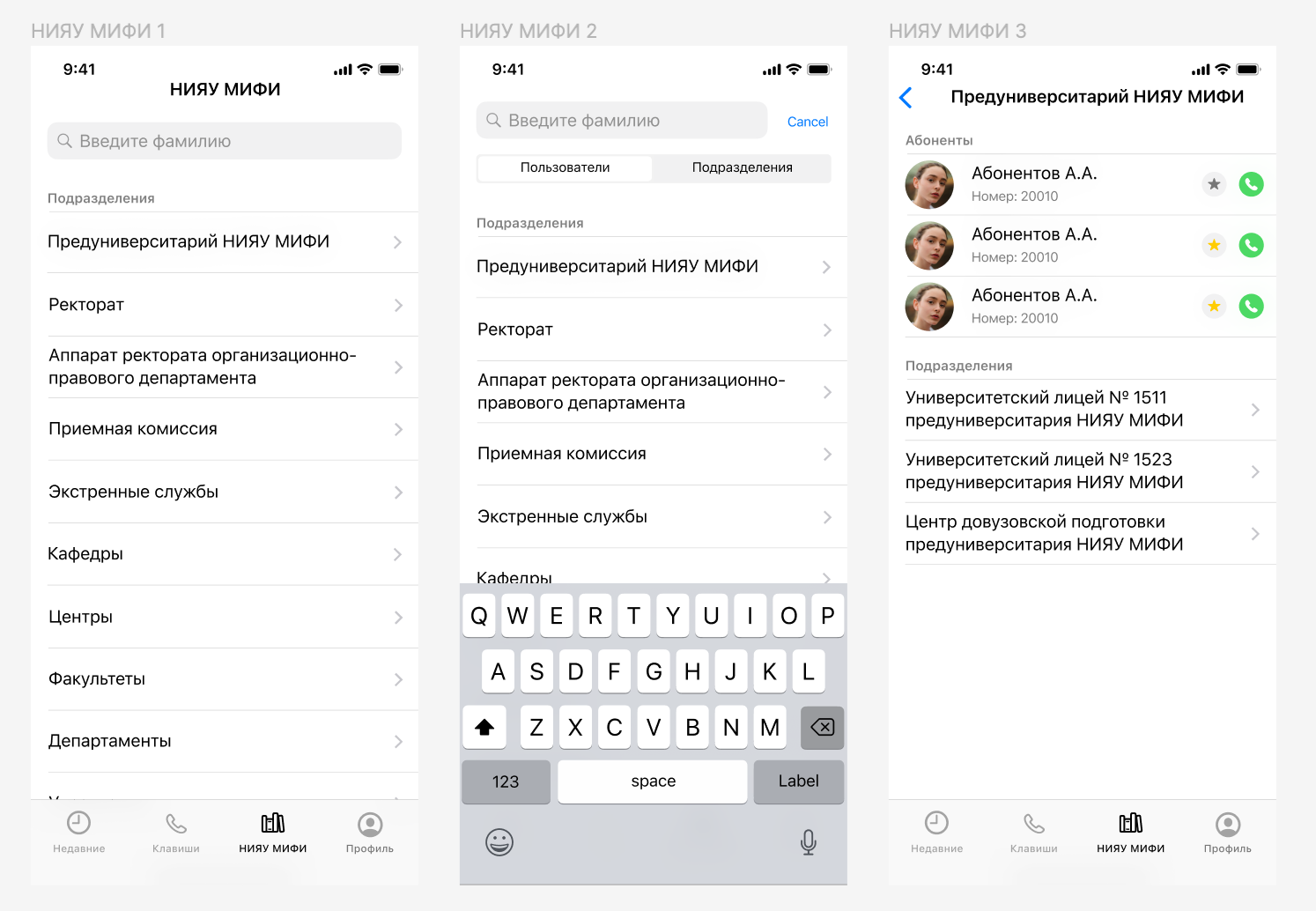


Рис. 2.3.2.1 - Интерфейс экрана каталога сотрудников

В целом, этот экран обеспечивает простой и интуитивно понятный интерфейс для доступа к справочнику абонентов и подразделений университета. Он позволяет пользователям легко искать и находить конкретных абонентов и подразделения, а также предоставляет возможности для просмотра дополнительной информации и связи с отдельными абонентами. Это является одним из ключевых преимуществ при разработке собственного решения SIP-софтфона для НИЯУ МИФИ. Минималистичный дизайн и динамическая высота ячеек обеспечивают визуальную привлекательность и простоту использования экрана.

### **2.3.3 Экран недавних вызовов**

Экран «Недавние» в мобильном приложении «VoIP MEPhI» предназначен для предоставления пользователям списка последних SIP-звонков, а также информации о каждом вызове и вариантов управления ими. На рисунке 2.3.3.1 представлен интерфейс экрана «Недавние».

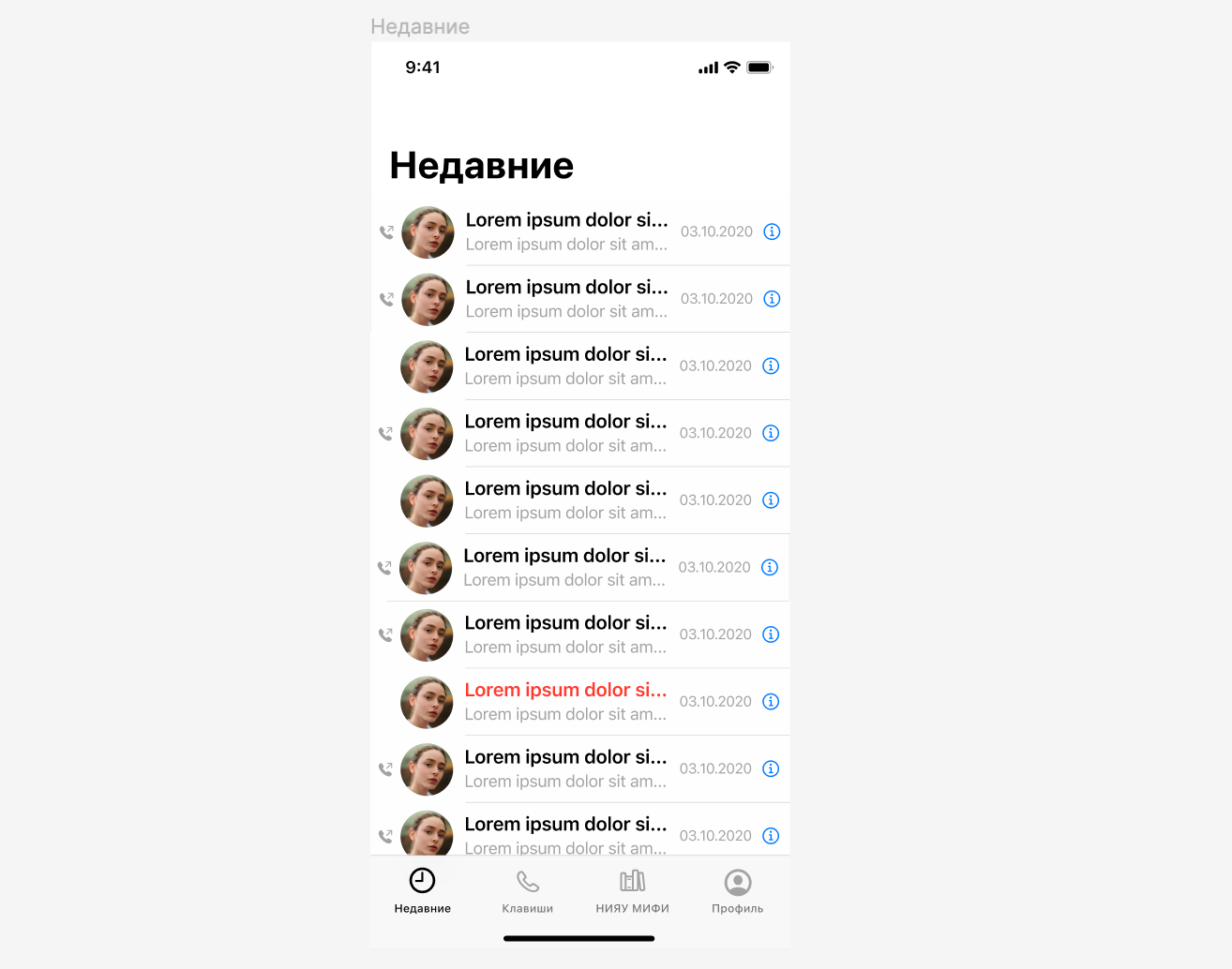


Рис. 2.3.3.1 - Интерфейс экрана недавних вызовов

С точки зрения UX, экран "Недавние" разработан для простого и интуитивно понятного использования. Он представляет пользователям список последних вызовов, причем каждый вызов представлен в виде ячейки в таблице с уникальным типом. Всего типов ячеек пять: исходящий звонок, входящий звонок, пропущенный звонок, исходящий отмененный звонок, входящий отмененный звонок. Интерфейс типов ячеек таблицы представлен на рисунке 2.3.3.2. Ячейки содержат информацию о звонке, например, дату звонка, имя звонившего, а также кнопку для получения более подробной информации о звонке.

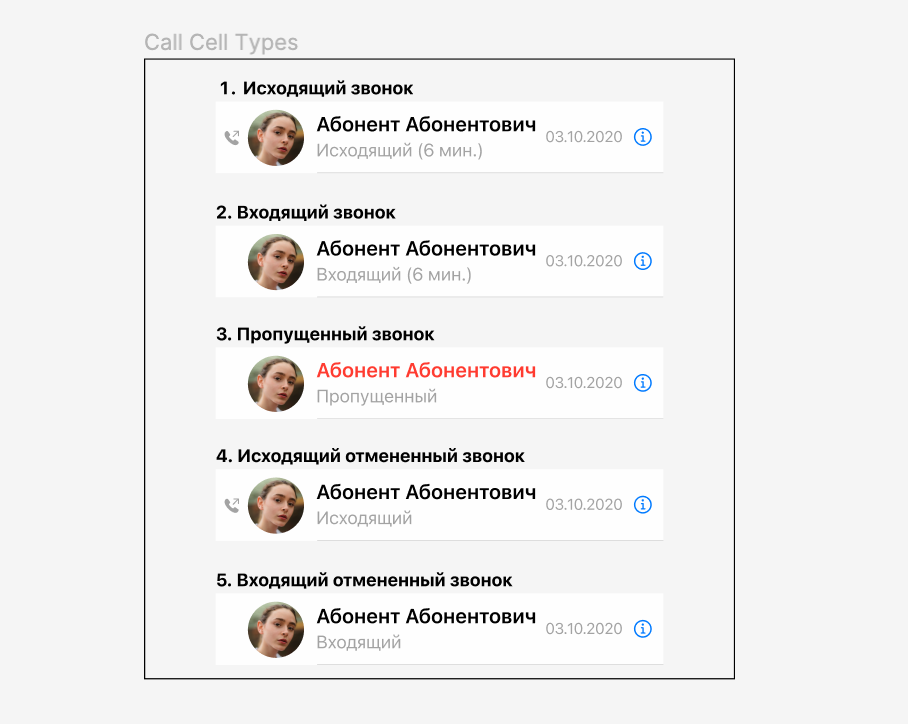


Рис. 2.3.3.2 - Интерфейс типов ячеек таблицы недавних вызовов

Экран «Недавние» использует чистые, современные элементы дизайна для создания визуально привлекательного интерфейса. Ячейки в таблице крупные и легко читаемые. На экране также присутствуют иконки и другие визуальные элементы, помогающие пользователям быстро понять статус каждого звонка.

В целом, UI и UX экрана «недавние» в приложении напоминает экран недавних вызовов из стандартного приложения «Телефон» на устройствах iOS и экран недавних вызовов в приложении «Telegram». Такой подход призван обеспечить простой и интуитивно понятный интерфейс для просмотра и управления последними телефонными звонками с акцентом на удобство использования и соответствие HIG.

### **2.3.4 Экран профиля текущего SIP-аккаунта**

Этот экран мобильного приложения предназначен для профиля SIP-абонента в системе НИЯУ МИФИ. Он предоставляет пользователям ряд инструментов и информации, связанной с их текущим SIP-аккаунтом и избранными контактами. На рисунке 2.3.4.1 представлен интерфейс экрана «Профиль» приложения VoIP MEPhI.

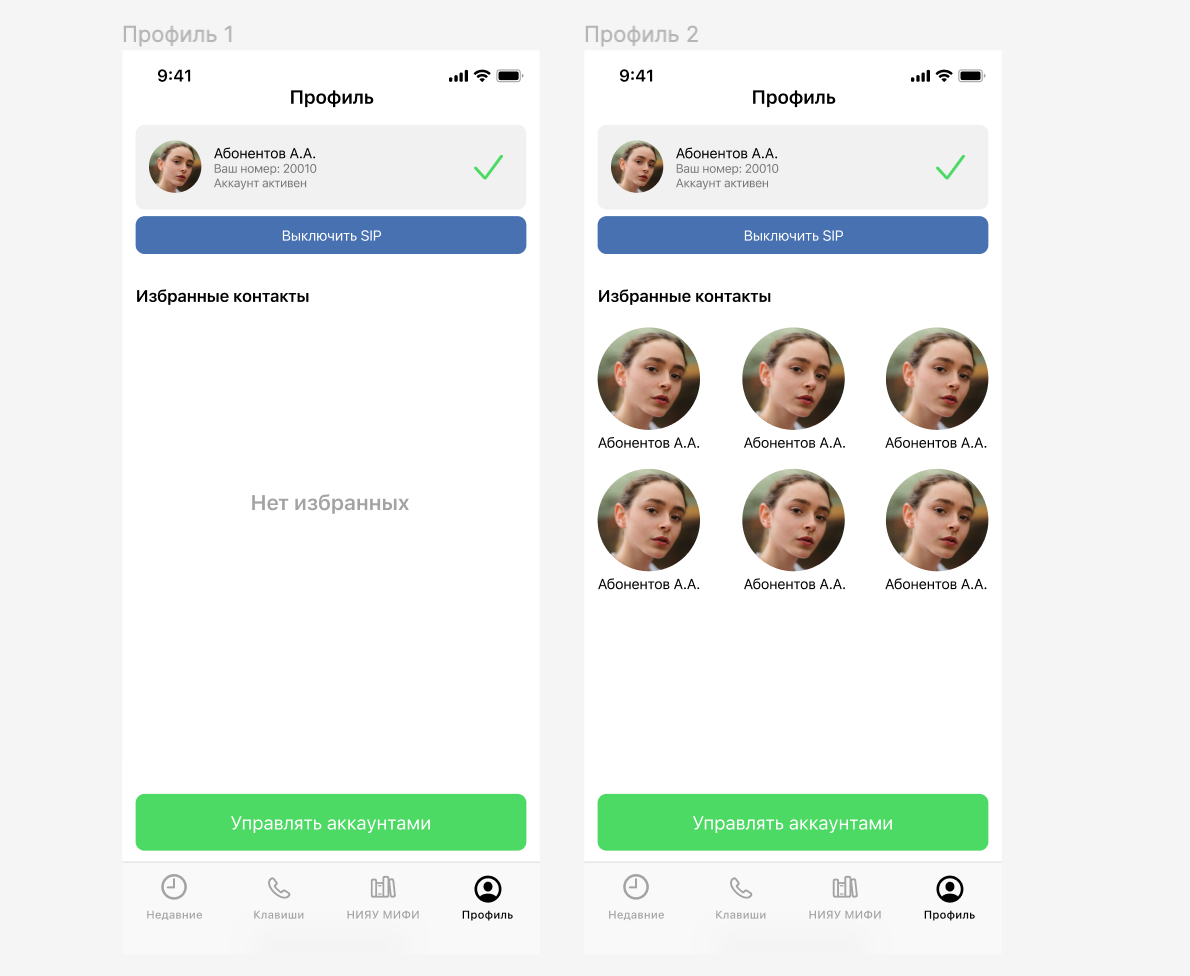


Рис. 2.3.4.1 - Интерфейс экрана профиля текущего SIP-аккаунта

В верхней части экрана находится карточка с информацией о текущем аккаунте. Эта карточка включает фотографию пользователя, имя, внутренний номер сотрудника и статус аккаунта. Статус аккаунта информирует пользователя о том, доступен ли он в данный момент для вызова другими пользователями. Пользователь может изменить статус своей учетной записи, нажав кнопку "Включить/выключить SIP", находящуюся под карточкой.

Ниже находится коллекция UICollectionView, в которой отображаются избранные контакты пользователя. Это коллекция позволяет пользователям быстро позвонить избранному контакту, нажав на его ячейку UICollectionViewCell коллекции. Если текущая учетная запись SIP не имеет избранных контактов, добавленных в каталог, отображается сообщение "Нет избранных".

Еще ниже на экране находится кнопка управления учетными записями, которая позволяет пользователю изменить текущую учетную запись SIP или выйти из нее.

В целом, UI и UX экрана «Профиль» разработаны таким образом, чтобы обеспечить простой и интуитивно понятный интерфейс для управления и взаимодействия с учетной записью SIP и избранными контактами с акцентом на удобство использования.

### **2.3.5 Экран управления учетными записями**

Этот экран приложения предназначен для того, чтобы пользователи могли управлять своими учетными записями SIP. Он появляется как модальный экран со стилем modelPresentationStyle и отображает таблицу доступных учетных записей. На рисунке 2.3.5.1 представлен интерфейс экрана аккаунтами.

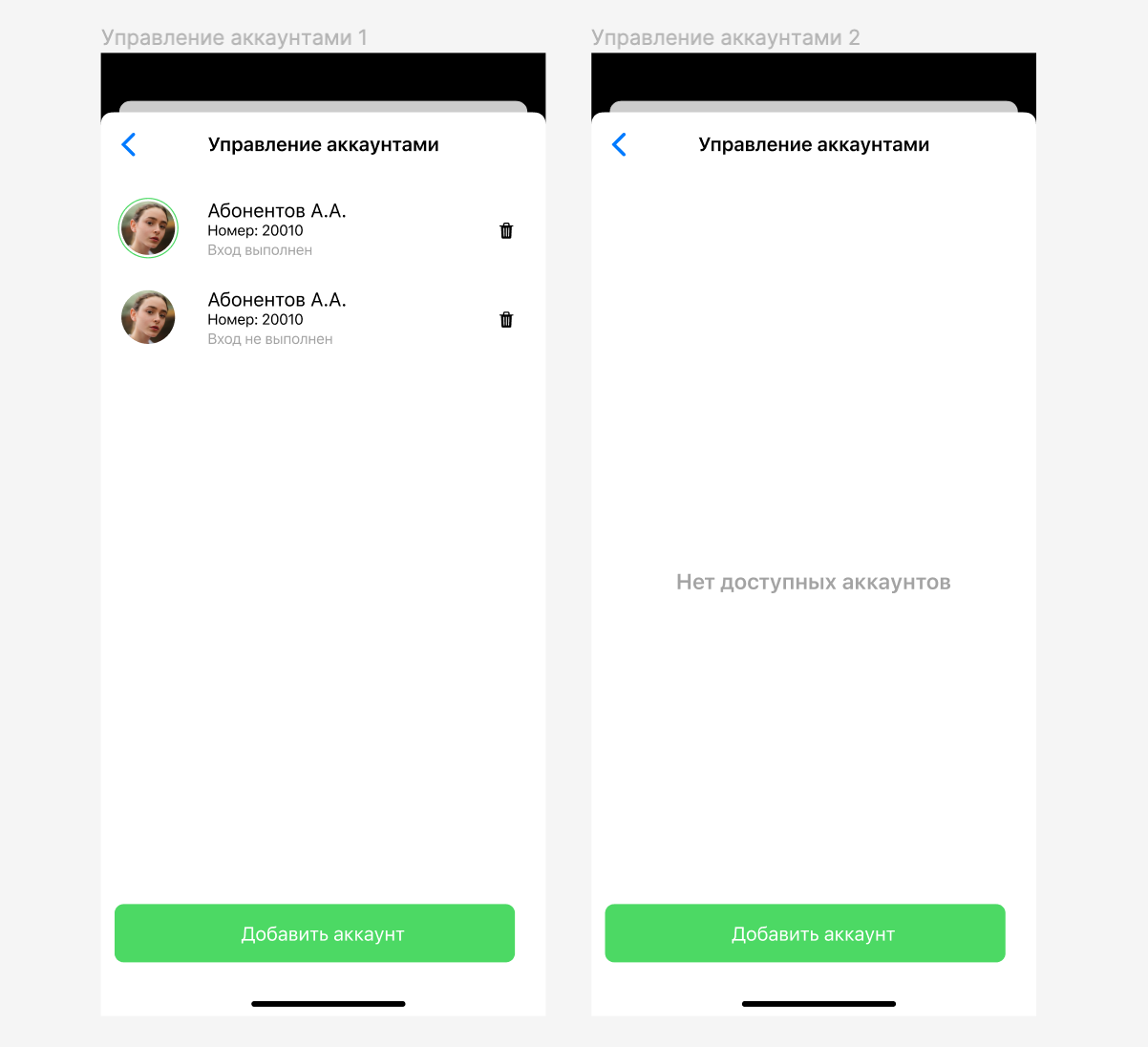


Рис. 2.3.5.1 - Интерфейс экрана управления SIP-аккаунтами

В ячейках таблицы отображается фотография пользователя, его имя, добавочный номер и статус учетной записи. Статус учетной записи информирует пользователя о том, вошел ли он в данный момент в систему для конкретного аккаунта или нет. В каждой ячейке также имеется иконка корзины, который позволяет пользователю выйти из учетной записи.

Когда пользователь нажимает на учетную запись, в которую не выполнен вход, появляется UIAlertController, требующий подтверждения для изменения учетной записи. Когда пользователь нажимает на значок корзины для учетной записи, в которую он вошел, появляется UIAlertController, запращивающий подтверждение для выхода, после выхода из текущего аккаунта он автоматически входит в другую доступную учетную запись при ее наличии. Когда пользователь нажимает на значок корзины для учетной записи, в которую не выполнен вход, появляется UIAlertController с просьбой подтверждения выхода, статус входа сохраняется для текущей учетной записи.

Под таблицей доступных аккаунтов находится кнопка "Добавить аккаунт", которая ведет на экран авторизации.

С точки зрения UX экран предоставляет пользователям четкое и организованное представление функционала доступных SIP-аккаунтов. UIAlertController’ы, которые появляются, когда пользователь пытается изменить или выйти из учетной записи, обеспечивают дополнительный уровень безопасности и гарантируют, что пользователь осознает последствия своих действий. Это важно, если пользователь ждет звонок на SIP-аккаунт А, не давая возможности случайно выполнить вход в SIP-аккаунт Б.

В целом, UI и UX этого экрана разработаны для обеспечения удобного и понятного интерфейса для управления учетными записями SIP с акцентом безопасность.

### **2.3.6 Экран авторизации**

Этот экран приложения, представляет собой авторизацию пользователя и имеет два поля ввода данных: для логина и пароля. На рисунке 2.3.6.1 представлен интерфейс экрана авторизации пользователя.

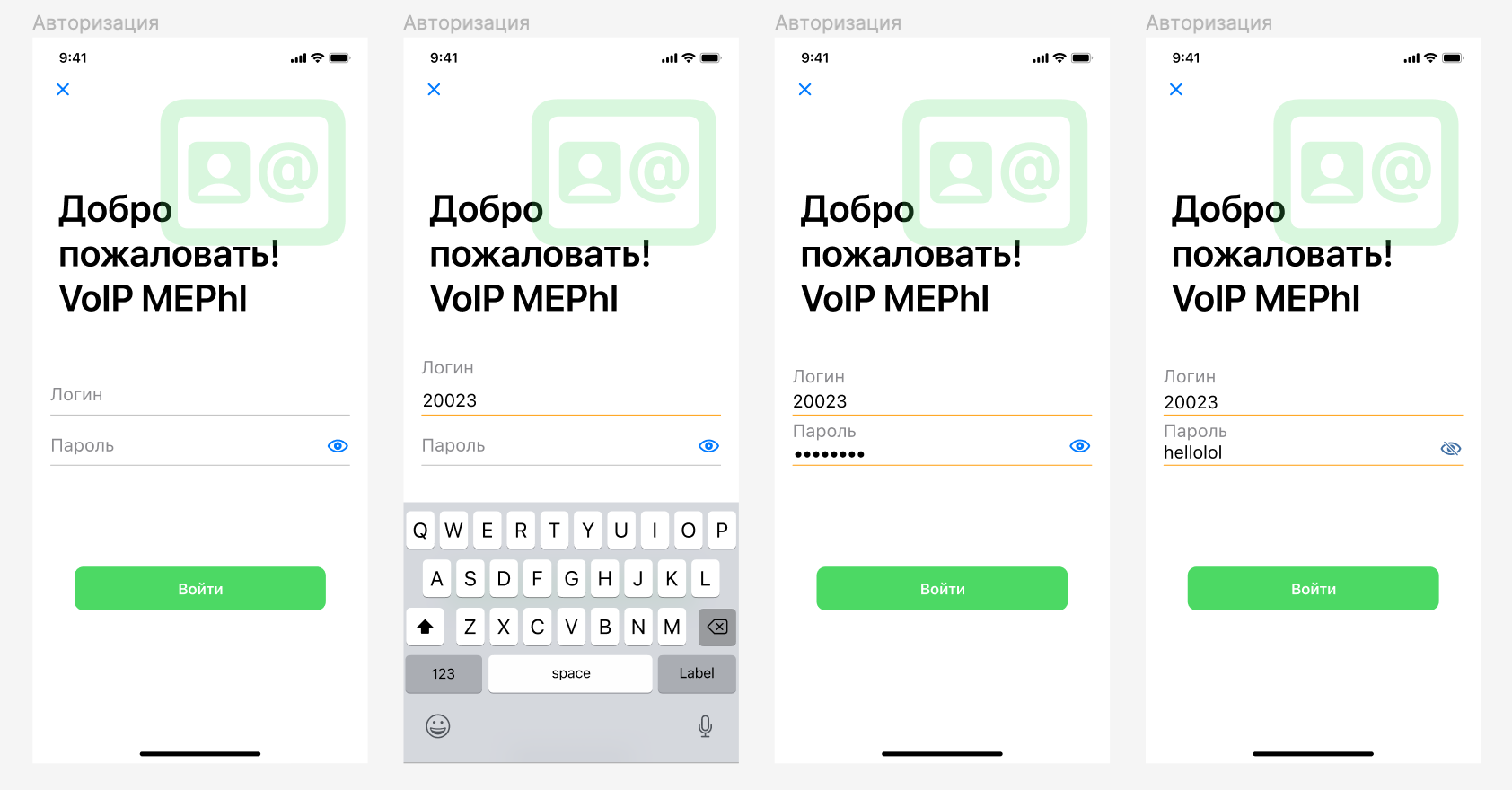


Рис. 2.3.6.1 - Интерфейс экрана авторизации

Когда пользователь вводит свой логин или пароль, поле ввода визуально изменяется, указывая на то, что оно заполнено.

Поле пароля отображается в защищенном виде, но есть возможность раскрыть его текстовое представление, нажав на иконку глаза.

Под полями ввода данных находится кнопка "Войти", которая инициирует проверку логина и пароля. Логин и пароль не должны быть пустыми, логин должен состоять только из цифр, а вход в аккаунт по указанному логину не должен быть осуществлен. Если валидация не удалась, появляется предупреждение, информирующее пользователя о проблеме. Если проверка прошла успешно, пользователь входит в систему и появляется оповещение об успешном или неудачном входе. Вход может быть выполнен неуспешно по ряду причин, например, неправильный логин или пароль, отсутствие сети и другие.

В целом, UI и UX этого экрана разработаны для интуитивно понятного знакомого пользователю интерфейса для входа в учетную запись SIP.

## **2.4 Архитектура мобильного приложения**

В качестве архитектуры приложения была выбрана архитектура MVC+Coordinator. На рисунке 2.4.1 представлена схема MVC (Model-View-Controller).

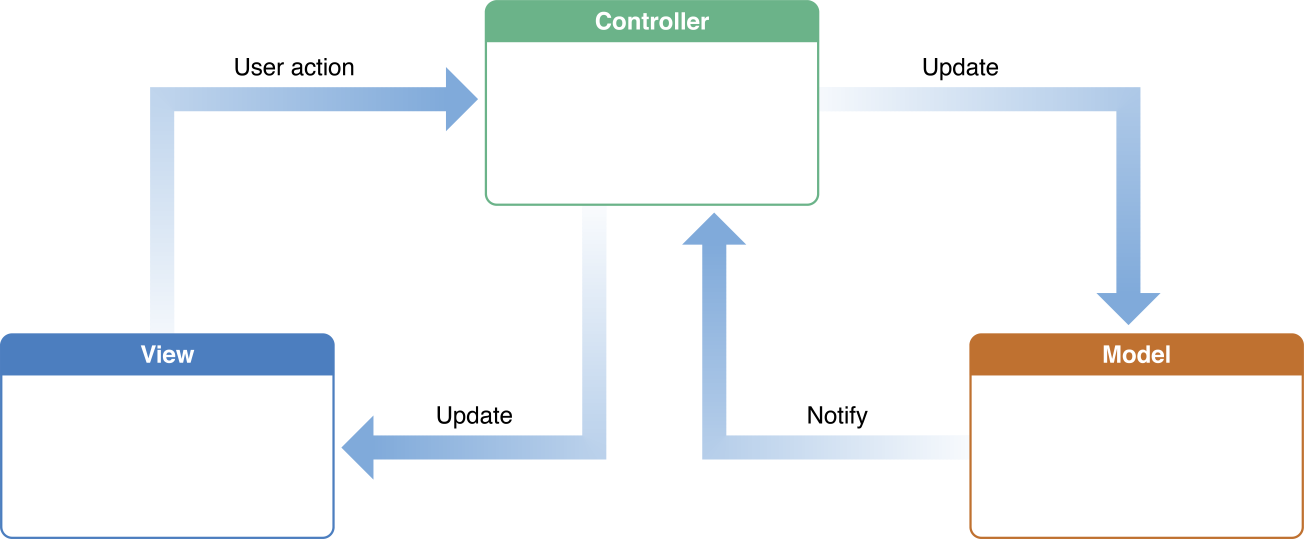


Рисунок 2.4.1 – Схематическое изображение архитектуры MVC

Достоинства выбранной архитектуры:

* Архитектура является понятной для программиста, поддержка кода не будет требовать много времени на понимание работы с архитектурной частью приложения;
* Равномерная нагрузка на Model, View и Controller;
* Переиспользуемость и расширяемость программного кода

В рамках мобильного приложения VoIP MEPhI, в котором не требуется реализация большого количества экранов, MVC является удобным вариантом с точки зрения скорости разработки и дальнейшей поддержки кода.

Для инкапсуляции логики навигации в мобильном приложении используется паттерн Coordinator. Вместо прямого вызова UIViewController’а из другого UIViewController’а используется отдельная изолированная и невидимая для разных Flow сущность. На рисунке 2.2.2 представлена упрощенная схема работы с паттерном Coordinator.

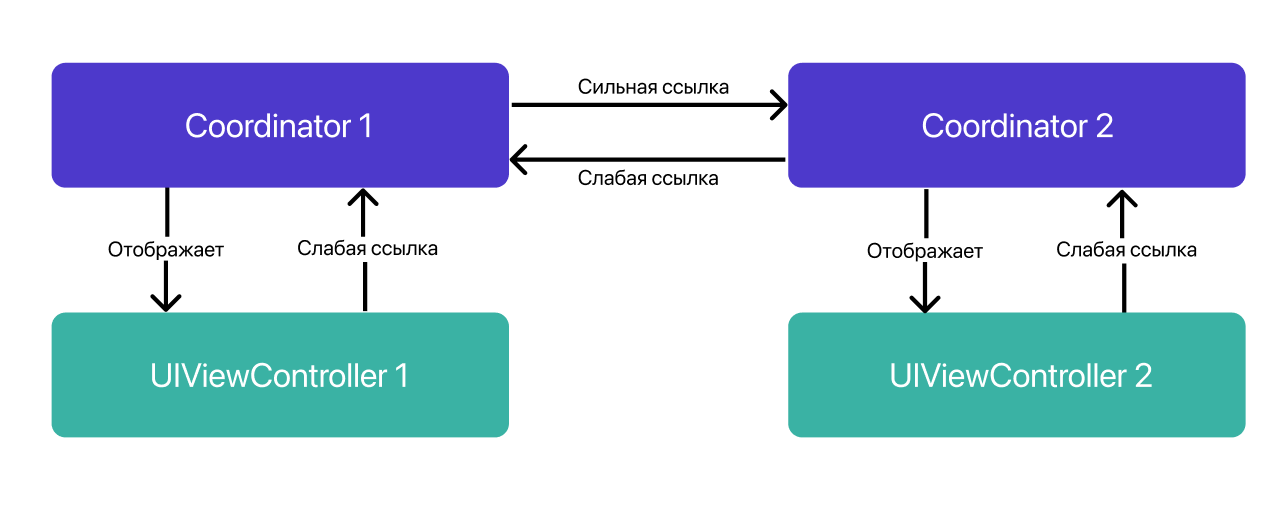


Рисунок 2.4.2 – Схематическое изображение паттерна Coordinator

Из схемы видно, что существует «родительский» Coordinator 1 и «дочерний» – Coordinator 2. В Swift управление памятью происходит посредством механизма ARC (Automatic Reference Counting), т.е. объект удаляется из памяти, если количество сильных ссылок на него равно нулю. Поэтому существуют и слабые ссылки, чтобы избежать ситуации «Reference cycle», когда объекты ссылаются друг на друга по кругу и не дают друг другу удалиться из памяти.

# **3 Программная реализация программного обеспечения мобильного iOS приложения для SIP-телефонии НИЯУ МИФИ**

## **3.1 Разработка сетевого слоя для взаимодействия с API НИЯУ МИФИ**

Реализация качественного сетевого слоя для работы с API в разрабатываемом приложении важна по ряду причин.

Во-первых, хорошо спроектированный сетевой уровень имеет решающее значение для обеспечения эффективного взаимодействия приложения с API. Это особенно важно, когда приложение работает с приходится часто делать запросы к API. В случае VoIP MEPhI требует частого обращения к API университета, например, при навигации по каталогу абонентов и подразделений в рамках новой сессии работы приложения. Надежный сетевой слой может гарантировать, что приложение сможет справиться с этими требованиями, не испытывая проблем с производительностью или сбоев.

Во-вторых, качественный сетевой уровень может помочь повысить поддерживаемость программного кода. При дальнейшей поддержке кода важно реализовать наглядную и гибкую систему, разделенную на изолированные сущности, внесение изменений в которые не повредит работе кода в другой части приложения.

На рисунке 3.1.1 представлена схема разработанного сетевого слоя мобильного приложения.

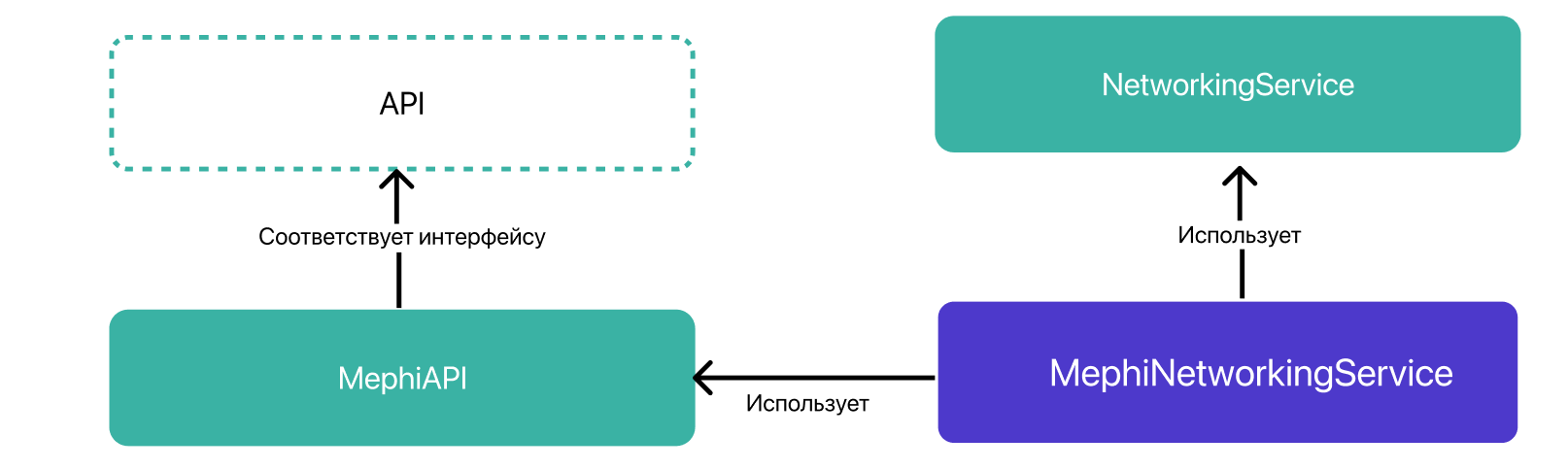


Рисунок 3.1.1 – Схематическое изображение работы сетевого слоя приложения

### **3.1.1 Протокол API**

Реализован протокол под названием "API", который определяет набор требований для взаимодействия с API. Протокол определяет пять свойств: "scheme", "baseURLofAPI", "pathOfAPI", "parametersOfAPI" и "httpAPImethod".

* **`scheme`**: представляет протокол, используемый для доступа к API, и определяется перечислением под названием "RequestScheme". Перечисление "RequestScheme" имеет два варианта: "http" и "https", которые соответствуют двум наиболее распространенным протоколам, используемым для доступа к API;
* **`baseURLofAPI`**: представляет базовый URL API;
* **`pathOfAPI`**: представляет конкретный путь в API, к которому приложение хочет получить доступ.
* **`parametersOfAPI`:** представляет любые дополнительные параметры, которые необходимо передать API при выполнении запроса.
* **`httpAPImethod`:** представляет метод HTTP, который должен быть использован при выполнении запроса к API. Он определяется перечислением под названием "HttpMethodOfAPI", которое имеет пять вариантов: "GET", "POST", "DELETE", "PATCH" и "PUT". Эти случаи соответствуют пяти наиболее часто используемым методам HTTP для взаимодействия с API.

Исходный код представлен в Приложении А.

### **3.1.2 Перечисление MephiAPI**

Код определяет перечисление под названием "MephiAPI", которое соответствует протоколу "API". Перечисление "MephiAPI" имеет пять случаев: "catalogInfo", "findUsers", "findUnits", "getPhoto" и "getUserInfo". Каждый случай представляет собой отдельный endpoint в сервисе MephiAPI.

Перечисление "MephiAPI" также включает статическую константу "key", которая представляет собой строку, используемую в качестве ключа API при выполнении запросов к endpoint’ам MephiAPI.

Перечисление "MephiAPI" реализует пять необходимых свойств протокола "API": "scheme", "baseURLofAPI", "pathOfAPI", "parametersOfAPI" и "httpAPImethod". Реализация этих свойств варьируется в зависимости от случая перечисления.

В целом, этот код определяет перечисление, которое может быть использовано для указания деталей запроса API к endpoint’ам MephiAPI, таких как протокол, используемый для доступа к API, базовый URL API, конкретная конечная точка API, к которой приложение хочет получить доступ, любые дополнительные query-параметры, которые необходимо передать API, и метод HTTP, который должен быть использован при выполнении запроса.

Исходный код представлен в приложении А.

### **3.1.3 Класс NetworkingService**

Этот класс сетевого сервиса, который используется для выполнения API-запросов к серверу. Класс NetworkService определяет три метода: buildURLComponents(\_:), url(of:) и request(\_:completion:).

Метод **`buildURLComponents(\_:)`** – это приватный вспомогательный метод, который создает объект компонентов URL на основе информации, содержащейся в сущности, реализующей протокол API.

Метод **`url(of:)`** возвращает объект URL на основе информации из сущности, реализующей протокол API.

Метод **`request(\_:completion:)`** – это основной метод класса NetworkService, который используется для выполнения API-запроса. В качестве параметров он принимает конечную точку API и сбегающее замыкание. Сначала он формирует URL-запрос на основе информации в протоколе API, затем создает задачу с URL-запросом. Когда задача завершается, она либо вызывает замыкание с декодированными данными ответа, либо с ошибкой, если она возникла.

Исходный код представлен в приложении А.

## **3.2 Разработка слоя навигации с использованием паттерна Coordinator**

В пункте 2.4 Приведено теоретическое обоснование выбора паттерна Coordinator для инкапсуляции логики навигации в приложении.

### **3.2.1 Протоколы CoordinatorProtocol и FlowCoordinatorProtocol**

Определены протоколы для координации навигации в приложении. FlowCoordinatorProtocol определяет свойство для родительского координатора, а CoordinatorProtocol добавляет свойство для корневого UIViewController’a или его наследника и методы для запуска и завершения навигации. CoordinatorProtocol также включает расширение, которое добавляет вычисляемое свойство для контроллера навигации, а также метод для сброса уровня вложенности навигации к корневому контроллеру. Эти протоколы и методы могут быть реализованы и использованы для управления навигацией между различными экранами и разделами приложения.

Исходный код представлен в Приложении Б.

### **3.2.2 Класс ApplicationCoordinator и протокол MainCoordinatorProtocol**

Определен класс ApplicationCoordinator, который используется для управления основным Flow навигации приложения на основе UITabBarItem’ов. Он соответствует протоколу MainCoordinatorProtocol, который в свою очередь соответствует протоколу CoordinatorProtocol, описанному в пункте 3.2.1, и определяет несколько дочерних координаторов для управления различными частями приложения. У класса ApplicationCoordinator также есть свойство rootViewController, которое представляет собой UIViewController, представляющий корневой контроллер приложения. Метод start() используется для настройки интерфейса TabBar’а приложения.

Исходный код приведен в приложении Б.

## **3.3 Разработка сервиса управления учетными записями**

Определен класс "AvailableAccountsService", который реализует протокол под названием "AvailableAccountsServiceProtocol". Этот класс отвечает за управление списком аккаунтов и отслеживает текущий активный аккаунт.

Класс реализует методы для добавления, удаления и изменения текущей учетной записи, а также для регистрации SIP-аккаунта с данными новой учетной записи. Класс также имеет сущность, реализующую паттерн Observer, который прослушивает уведомления "AbtoPhoneEvent" и отвечает обновлением данных текущей учетной записи в конфигурации SIP-телефонии.

Программный код представлен в Приложении В.

### **3.3.1 Авторизация новой учетной записи**

На рисунке 3.3.1.1 представлена блок-схема алгоритма авторизации новой учетной записи.

При валидации логина и пароля используется метод, определенный в класс SignInViewController, isLogPassOk(login: String, password: String) -> Bool.

Метод проверяет валидность логина и пароля, не пустые ли логин и пароль, состоит ли логин только из цифр и не произведен ли уже вход для учетной записи с введенным логиком. Если одно из этих условий не выполняется, выводится сообщение об ошибке и возвращается false. Метод также включает визуальную обратную связь, например, анимацию ошибок текстового поля и звуковое оповещение, чтобы проинформировать пользователя о возникших ошибках.

Исходный код представлен в приложении В.

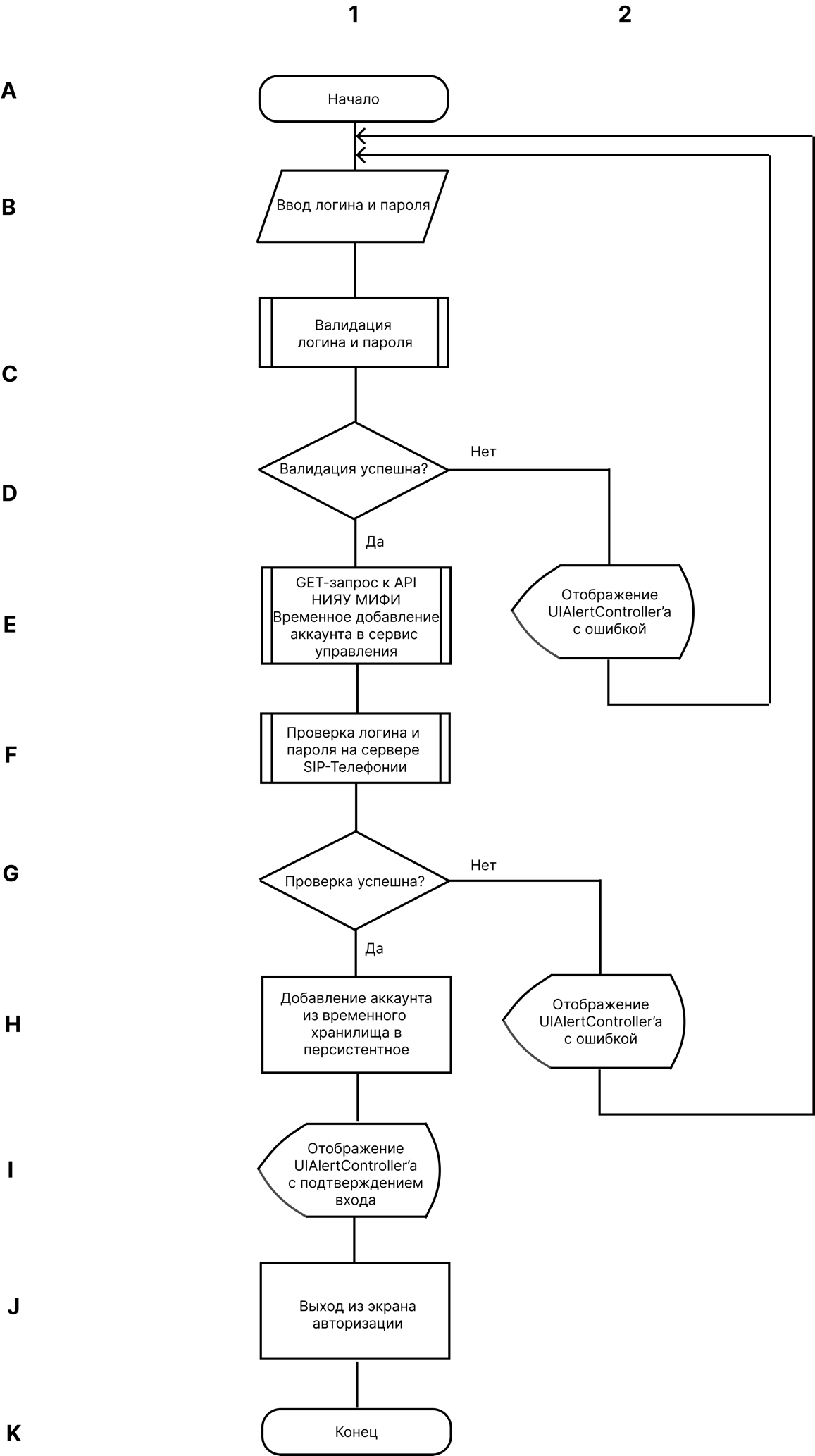


Рисунок 3.1.1 – Блок-схема алгоритма авторизации пользователя

# **4 Тестирование**

# **5 Технико-экономические показатели**

# **ЗАКЛЮЧЕНИЕ**

**ПРИЛОЖЕНИЕ А. Листинг кода, демонстрирующий работу сетевого слоя мобильного приложения**

**А1 – API.Swift**

**protocol** API {

**var** scheme: RequestScheme { **get** }

**var** baseURLofAPI: String { **get** }

**var** pathOfAPI: String { **get** }

**var** parametersOfAPI: [URLQueryItem] { **get** }

**var** httpAPImethod: HttpMethodOfAPI { **get** }

}

**enum** RequestScheme: String {

**case** http

**case** https

}

**enum** HttpMethodOfAPI: String {

**case** GET

**case** POST

**case** DELETE

**case** PATCH

**case** PUT

}

**А2 – MephiAPI.Swift**

**enum** MephiAPI: API {

**static** **let** key = "*some\_api\_key*"

**case** catalogInfo(filterCode: String)

**case** findUsers(query: String)

**case** findUnits(query: String)

**case** getPhoto(guid: String)

**case** getUserInfo(phone: String)

**var** scheme: RequestScheme {

**switch** **self** {

**default**:

**return** .https

}

}

**var** baseURLofAPI: String {

**switch** **self** {

**default**:

**return** "sd.mephi.ru"

}

}

**var** pathOfAPI: String {

**switch** **self** {

**case** .catalogInfo:

**return** "/api/6/get\_units\_mobile\_catalog.json"

**case** .findUsers:

**return** "/api/6/get\_subscribers\_mobile.json"

**case** .findUnits:

**return** "/api/6/get\_units\_mobile\_find.json"

**case** .getPhoto:

**return** "/api/6/get\_photo\_mobile.jpg"

**case** .getUserInfo:

**return** "/api/6/get\_displayname.json"

}

}

**var** parametersOfAPI: [URLQueryItem] {

**switch** **self** {

**case** .catalogInfo(**let** filterCode):

**let** params = [

URLQueryItem(name: "api\_key", value: MephiAPI.key),

URLQueryItem(name: "filter\_code\_str", value: filterCode),

]

**return** params

**case** .findUsers(query: **let** query):

**let** params = [

URLQueryItem(name: "api\_key", value: MephiAPI.key),

URLQueryItem(name: "filter\_lastname", value: "LIKE|%\(query)%"),

]

**return** params

**case** .findUnits(query: **let** query):

**let** params = [

URLQueryItem(name: "api\_key", value: MephiAPI.key),

URLQueryItem(name: "filter\_fullname", value: "LIKE|%\(query)%"),

]

**return** params

**case** .getPhoto(guid: **let** guid):

**let** params = [

URLQueryItem(name: "api\_key", value: MephiAPI.key),

URLQueryItem(name: "phone", value: guid),

]

**return** params

**case** .getUserInfo(phone: **let** phone):

**let** params = [

URLQueryItem(name: "api\_key", value: MephiAPI.key),

URLQueryItem(name: "line", value: phone),

]

**return** params

}

}

**var** httpAPImethod: HttpMethodOfAPI {

**switch** **self** {

**default**:

**return** .GET

}

}

}

**A3 – NetworkingService.swift**

**final** **class** NetworkService {

**private** **class** **func** buildURLComponents(\_ endpoint: API) -> URLComponents {

**var** components = URLComponents()

components.scheme = endpoint.scheme.rawValue

components.host = endpoint.baseURLofAPI

components.path = endpoint.pathOfAPI

components.queryItems = endpoint.parametersOfAPI

**return** components

}

**static** **func** url(of endpoint: API) -> URL? {

**let** components = buildURLComponents(endpoint)

**return** components.url

}

**class** **func** request<T: Decodable>(endpoint: API, completion: **@escaping** (Result<T, Error>) -> Void) {

**let** components = buildURLComponents(endpoint)

**guard** **let** url = components.url **else** {

**return**

}

**var** urlRequest = URLRequest(url: url)

urlRequest.httpMethod = endpoint.httpAPImethod.rawValue

**let** session = URLSession(configuration: .default)

**let** dataTask = session.dataTask(with: urlRequest) { data, response, error **in**

**if** **let** error = error {

completion(.failure(error))

**return**

}

**guard** **let** data = data, response != **nil** **else** {

**return**

}

**do** {

**let** decodedObject = **try** JSONDecoder().decode(T.**self**, from: data)

completion(.success(decodedObject))

} **catch** {

completion(.failure(error))

}

}

dataTask.resume()

}

}

**ПРИЛОЖЕНИЕ Б. Листинг кода, демонстрирующий работу паттерна Coordinator**

**Б1 – CoordinatorProtocol.swift**

**protocol** FlowCoordinatorProtocol: AnyObject {

**var** parentCoordinator: MainCoordinatorProtocol? { **get** **set** }

}

**protocol** CoordinatorProtocol: FlowCoordinatorProtocol {

**var** rootViewController: UIViewController { **get** **set** }

**func** start() -> UIViewController

**@discardableResult** **func** resetToRoot() -> **Self**

}

**extension** CoordinatorProtocol {

**var** navigationRootViewController: UINavigationController? {

**get** {

(rootViewController **as**? UINavigationController)

}

}

**func** resetToRoot() -> **Self** {

navigationRootViewController?.popToRootViewController(animated: **false**)

**return** **self**

}

}

**Б2 – ApplicationCoordinator.swift**

**protocol** MainCoordinatorProtocol: CoordinatorProtocol {

**var** recentCoordinator: RecentCoordinatorProtocol { **get** }

**var** dialingCoordinator: DialingCoordinatorProtocol { **get** }

**var** catalogCoordinator: CatalogCoordinatorProtocol { **get** }

**var** profileCoordinator: ProfileCoordinatorProtocol { **get** }

}

**class** ApplicationCoordinator: MainCoordinatorProtocol {

**var** parentCoordinator: MainCoordinatorProtocol?

**lazy** **var** recentCoordinator: RecentCoordinatorProtocol = RecentCoordinator()

**lazy** **var** dialingCoordinator: DialingCoordinatorProtocol = DialingCoordinator()

**lazy** **var** catalogCoordinator: CatalogCoordinatorProtocol = CatalogCoordinator()

**lazy** **var** profileCoordinator: ProfileCoordinatorProtocol = ProfileCoordinator()

**lazy** **var** rootViewController: UIViewController = UITabBarController()

**func** start() -> UIViewController {

**let** recentVC = recentCoordinator.start()

recentCoordinator.parentCoordinator = **self**

recentVC.tabBarItem = UITabBarItem(title: "Недавние", image: .init(systemName: "clock"), tag: 0)

**let** dialingVC = dialingCoordinator.start()

dialingCoordinator.parentCoordinator = **self**

dialingVC.tabBarItem = UITabBarItem(title: "Клавиши", image: .init(systemName: "phone"), tag: 1)

**let** catalogVC = catalogCoordinator.start()

catalogCoordinator.parentCoordinator = **self**

catalogVC.tabBarItem = UITabBarItem(title: "Каталог", image: .init(systemName: "books.vertical"), tag: 2)

**let** profileVC = profileCoordinator.start()

profileCoordinator.parentCoordinator = **self**

profileVC.tabBarItem = UITabBarItem(title: "Профиль", image: .init(systemName: "person.crop.circle"), tag: 3)

**let** tabBarController = rootViewController **as**! UITabBarController

**let** tabBar = tabBarController.tabBar

tabBar.tintColor = .yinyang

tabBar.backgroundColor = .systemBackground

tabBarController.viewControllers = [recentVC, dialingVC, catalogVC, profileVC]

**return** rootViewController

}

}

**ПРИЛОЖЕНИЕ В. Сервис управления учетными записями.**

**AvailableAccountsService.swift**

**protocol** AvailableAccountsServiceProtocol {

**var** currentAccount: Account? { **get** }

**var** accounts: [Account] { **get** }

**var** newAccount: Account { **get** }

**func** setNewAccount(\_ account: Account)

**func** addAccount(\_ account: Account)

**func** removeAccount(\_ account: Account)

**func** changeCurrentAccount(\_ account: Account)

**func** abtoRegisterPhone(\_ account: Account)

}

**final** **class** AvailableAccountsService: AvailableAccountsServiceProtocol {

**static** **let** shared: AvailableAccountsServiceProtocol = AvailableAccountsService()

**private** **static** **var** accountsUserDefaultsKey: String { .init(describing: **Self**.**self**) }

**private** **static** **var** currentAccountUserDefaultsKey: String {

**return** "\(String(describing: **Self**.**self**)).currentAccount"

}

**private** **init**() {

NotificationCenter.default.addObserver(

**self**,

selector:**#selector**(onRegisterNotification(\_:)),

name: .AbtoPhoneEvent,

object: **nil**)

**do** {

**let** accs = **try** UserDefaults.standard.take(objectType: [Account].**self**, forKey: AvailableAccountsService.accountsUserDefaultsKey) ?? []

accounts = accs.sorted {

**switch** ($0.isActive, $1.isActive) {

**case** (**true**, false): **return** **true**

**case** (**false**, true): **return** **false**

**default**: **return** **false**

}

}

} **catch** {

print("Error decoding userdefautls")

}

**do** {

currentAccount = **try** UserDefaults.standard.take(objectType: Account.**self**, forKey: AvailableAccountsService.currentAccountUserDefaultsKey)

} **catch** {

print("Error decoding userdefautls")

}

}

**private**(**set**) **var** accounts: [Account] = [] {

**didSet** {

NotificationCenter.default.post(name: .AvailableAccountsChanged, object: **nil**)

updateAccountsUserDefaults()

}

}

**private**(**set**) **var** currentAccount: Account? {

**didSet** {

**guard** **let** abtoPhone = AppDelegate.shared.providerDelegate?.abtoPhone **else** {

**return**

}

**let** config = abtoPhone.config

config.regUser = currentAccount?.login

config.regPassword = currentAccount?.password

config.regDomain = AppConstants.domain

config.regExpirationTime = 300

config.save(toUserDefaults: currentAccount?.configUserDefaultsKey ?? "noAccount")

abtoPhone.finalizeConfiguration()

NotificationCenter.default.post(name: .CurrentAccountChanged, object: **nil**)

updateCurrentAccountUserDefaults()

}

}

**private**(**set**) **var** newAccount: Account = .init()

**func** addAccount(\_ account: Account) {

accounts.append(account)

}

**func** setNewAccount(\_ account: Account) {

newAccount = account

}

**func** removeAccount(\_ account: Account) {

accounts.removeAll { $0.login == account.login }

**if** **let** currentAccount = currentAccount {

**if** currentAccount.login == account.login {

**if** **let** \_ = accounts.last {

changeCurrentAccount(accounts.last!)

} **else** {

**self**.currentAccount = **nil**

}

}

}

}

**func** changeCurrentAccount(\_ account: Account) {

**let** accs = accounts.map { acc **in**

**if** acc.login == account.login {

**let** activeAcc = acc.makeActiveCopy()

**self**.currentAccount = activeAcc

**return** activeAcc

} **else** {

**return** acc.makeInactiveCopy()

}

}.sorted {

**switch** ($0.isActive, $1.isActive) {

**case** (**true**, false): **return** **true**

**case** (**false**, true): **return** **false**

**default**: **return** **false**

}

}

**self**.accounts = accs

}

**private** **func** updateAccountsUserDefaults() {

**do** {

**try** UserDefaults.standard.place(object: accounts, forKey: AvailableAccountsService.accountsUserDefaultsKey)

} **catch** {

print("Error placing accounts")

}

}

**private** **func** updateCurrentAccountUserDefaults() {

**do** {

**try** UserDefaults.standard.place(object: currentAccount, forKey: AvailableAccountsService.currentAccountUserDefaultsKey)

} **catch** {

print("Error placing accounts")

}

}

**func** abtoRegisterPhone(\_ account: Account) {

**guard** **let** abtoPhone = AppDelegate.shared.providerDelegate?.abtoPhone **else** { **return** }

**let** config = abtoPhone.config

config.regUser = account.login

config.regPassword = account.password

config.regDomain = AppConstants.domain

config.regExpirationTime = 300

abtoPhone.finalizeConfiguration()

}

**@objc** **func** onRegisterNotification(\_ notification: Notification) {

**guard** **let** event = notification.object **as**? AbtoPhoneEvent,

event.isSigningInEvent **else** {

**return**

}

**switch** event {

**case** .registerSuccess(\_):

print("onConnectionStatusNotification reg success")

**guard** newAccount.login != "" **else** { **return** }

**let** postDict: [String : SignInEvent] = [

"SignInEvent" : SignInEvent.signInSuccess

]

NotificationCenter.**default**.post(name: .SignInEvent, object: **nil**, userInfo: postDict)

addAccount(newAccount)

changeCurrentAccount(newAccount)

newAccount = .init()

**case** **let** .registerFailed(\_, code, status):

print("onConnectionStatusNotification reg failed: \(code) - \(status)")

**let** postDict: [String : SignInEvent] = [

"SignInEvent" : SignInEvent.signInFailed(code: code, status: status)

]

NotificationCenter.**default**.post(name: .SignInEvent, object: **nil**, userInfo: postDict)

**if** AppConstants.isMyService {

DispatchQueue.main.async() {

AppDelegate.shared.abtoService.abtoPhone.unregister()

}

} **else** {

DispatchQueue.main.async() {

AppDelegate.shared.providerDelegate?.abtoPhone.unregister()

}

}

**case** .unregister(\_):

print("onConnectionStatusNotification unregister")

**case** **let** .remoteAlerting(\_, code):

print("onConnectionStatusNotification alerting: \(code)")

**default**:

**break**

}

}

}