МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ
ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

Направление подготовки/специальность
09.03.01 Информатика и вычислительная техника

направленность (профиль)/специализация
«Технологии разработки программного обеспечения»

Выпускная квалификационная работа

“Разработка UI iOS-приложения на основе SwiftUI”

Обучающегося 4 курса
очной формы обучения
Стецук Максима Николаевича

Руководитель выпускной квалификационной работы:
кандидат физико-математических наук,
доцент кафедры ИТиЭО
Жуков Николай Николаевич

Санкт-Петербург
2025

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире мобильные приложения стали неотъемлемой частью повседневной жизни любого человека, ведь они предоставляют пользователям доступ к различной информации, развлечениям и услугам в удобном и быстро доступном формате. На данный момент времени, доля пользователей, предпочитающих использовать устройства на базе операционной системы iOS составляет 45% от общего числа пользователей смартфонов в Российской Федерации. Благодаря тому, что компания Apple поддерживает регулярное обновление своих устройств длительное время, количество пользователей использующих iPhone как основное устройство постоянно растет и спрос на качественные мобильные приложения под iOS, которые способны выполнять различные функции, включая общение, развлечения, покупки, финансы, а также образование и культурное развитие, которые становятся все более актуальными из-за возможности простого доступа пользователей к мобильным телефонам.

Несмотря на такое быстрое развитие мобильных устройств и приложений, огромное количество организаций, относящихся к различным сферам, до сих пор используют в качестве единственного способа связи с пользователем сайты, несмотря на потребность пользователей в оперативном доступе к их ресурсам, который может быть затруднен при использования мобильного телефона. На данный момент Apple и Google значительно упростили процесс создания нативных приложений под iOS и Android соответственно, благодаря значительному развитию языков программирования Swift и Kotlin, а также выпуску новых фреймворков для создания мобильных интерфейсов. К таким фреймворкам относится SwiftUI (для устройств на базе iOS), который представляют разработчикам возможность создавать адаптивные и интуитивно понятные пользователям интерфейсы применяя для этого декларативный стиль программирования, а не императивный. Примером фреймворка для разработки интерфейса с использованием императивного подхода является UIKit, который более

распространен в разработке приложений под iOS из-за количества предоставляемых возможностей и готовых решений. Но несмотря на преимущества которые он дает, UIKit является более сложным фреймворком как в освоении, так и в применении, ведь при его использовании разработчику необходимо учитывать все возможные состояния страницы или элемента и строго описывать данную пошагавшую логику отображения и взаимодействия. В то же время доступная функциональность SwiftUI постоянно обновляется, а его декларативный подход делает разработку небольших проектов более быстрой, благодаря гибкости, возможности переиспользования компонентов, а также самостоятельной обработке изменения состояния элементов. Данный фреймворк наиболее актуален для тех организаций, которым необходимо реализовать приложение под iOS с относительно простым функционалом, чтобы их пользователи могли получать доступ к ресурсам, как через браузер с компьютера, так и через мобильное приложение.

Особенностью мобильных приложений является необходимость в создании интерфейсов, которые будут интуитивно понятны пользователю, благодаря опыту, полученному благодаря использования иных приложений, а так же в необходимости создания адаптивного под различные устройства интерфейса. Это является немаловажным фактором при выборе декларативного подхода к написанию интерфейса, ведь он позволяет более простым способом реализовать элементы интерфейса и адаптировать их под устройства с различными размерами экранов.

В рамках данной работы будет рассмотрен фреймворк SwiftUI и его возможности при разработке мобильных приложений под iOS. Также будут рассмотрены основные подходы, применяемые при создании интерфейсов и важные методологии, которым необходимо следовать при реализации программных проектов. А также будет изучен процесс тестирования мобильных приложений написанных под операционную систему iOS.

Цель работы: Исследование возможностей фреймворка SwiftUI и разработка мобильного приложения для Центральной библиотечной системы Петроградского района на его основе.

Данная разработка выполняется на заказ, что доказывает ее актуальность и востребованность мобильных приложений для устройств на базе операционной системы iOS в современном мире.

Задачи работы:

- Анализ требований к интерфейсам мобильных приложений и подходов к их разработке;
- Исследование возможностей фреймворка SwiftUI при создании мобильных приложений под iOS;
- Проектирование архитектуры мобильного приложения;
- Разработка интерфейса и функциональности мобильного приложения в соответствии с современными практиками.

Объект исследования: Процесс разработки мобильного приложения под устройства на базе операционной системы iOS.

Предмет исследования: Современные инструменты и методологические подходы применяемые при разработке мобильных iOS-приложений.

В первой главе работы рассматриваются основные подходы к созданию мобильных приложений под iOS, особенности в разработке интерфейсов мобильных приложений, основные возможности SwiftUI и подходы к написанию универсального и поддерживаемого кода. Помимо принципов разработки в первой главе данной работы будут рассмотрены особенности тестирования мобильных приложений, которое необходимо на каждом этапе разработки. Во второй главе на практике рассматривается процесс разработки мобильного приложения для ЦБС, включающий в себя анализ требований, проектирование архитектуры проекта, реализацию компонентов интерфейса, их применение при сборке экранов приложения и реализацию взаимодействия приложения с сервером.

ГЛАВА 1. ОСОБЕННОСТИ РАЗРАБОТКИ iOS-ПРИЛОЖЕНИЙ

1.1 Подходы к созданию мобильных приложений

Разработка мобильных приложений является ключевым направлением развития современного рынка программного обеспечения. Существуют различные подходы, применяемые при разработке мобильных приложений, но наиболее распространенными на данный момент являются прогрессивные веб-приложения (PWA - Progressive Web Apps) и нативные мобильные приложения. Данные подходы к разработке приложений имеют свои особенности, преимущества и ограничения. В данной части работы будут рассмотрены оба подхода, с упором на разработку мобильных приложений под iOS, а также представлены конкретные преимущества нативных приложений над прогрессивными веб-приложениями при разработке под iOS.

Прежде чем приступать к непосредственному сравнению необходимо определить, что это за подходы и в чем их самое главное отличие:

- **PWA (Progressive Web App)** - это веб-приложение, разработанное с использованием веб-технологий, которое может работать как в браузере, так и «устанавливаться» на устройство без использования официального магазина приложений AppStore благодаря технологиям, которые позволяют маскировать такое приложение под нативное, при этом предоставляя возможность разработчикам использовать ряд возможностей системы, таких как оффлайн-режим, уведомления, добавление иконок на рабочий стол устройства и т.п. Но несмотря на эти возможности, основным ядром для PWA остается браузер.
- **Нативные приложения** - это приложения, разработанные специально для конкретной платформы, с использованием определенных технологий, которые предоставляет компания занимающаяся разработкой конкретной платформы (в области

разработки мобильных приложений под платформами понимают iOS и Android). В случае iOS к нативным относят приложения разработанные на языке Swift (или Objective-C) с применением фреймворков, которые предоставляет компания Apple. К таким фреймворкам относятся UIKit и SwiftUI. Данные приложения запускаются напрямую на устройстве и в качестве ядра используют операционную систему устройства, благодаря чему имеют полный доступ к возможностям, которые она предоставляет.

Одним из ключевых различий между PWA и нативными приложениями является доступ к системным функциям устройства. PWA ограничены тем функционалом, который доступен браузеру устройства, ведь как было сказано ранее, ядром таких приложений остается браузер, несмотря на схожесть с нативными мобильными приложениями. Таким образом многий функционал, который применяется в современных приложениях для них остается недоступным, например возможности Bluetooth, Face ID, который часто применяется для разблокировали приложения вместо пароля, а также функции связанные с системными возможностями и приложениями. В то же время нативные приложения написанные на Swift имеют прямой доступ ко всем API, которые предоставляет Apple.

PWA выполняют большую часть операций за счет возможностей браузера, что накладывает на них существенные ограничения, в том числе и по производительности. Данное ограничение напрямую влияет на графическую и логическую составляющие приложения, из-за чего отзывчивость интерфейса и скорость обработки действий значительно ниже чем у нативных приложений. Нативные приложения в свою очередь имеют высокую производительность, плавные анимации и быстрый отклик на действия пользователя. Это очень хорошо демонстрирует SwiftUI, который благодаря декларативному подходу позволяет использовать весь доступный ресурс устройства для отображения интерфейса и обработки действий пользователя.

На данный момент одним из ключевых факторов, исходя из которого у пользователей складывается впечатление о мобильном приложении, является его интерфейс. Как было сказано ранее в PWA интерфейс создается средствами веб-разработки, и, хотя может имитировать «нативный» стиль, всё равно ощущается менее органичным, что особенно заметно для пользователей устройств на базе iOS, так как в основе их интерфейса заложена философия Human Interface Design, в основе которой лежат принцип простоты интерфейса, использования компонентов с интуитивно понятным пользователю поведением, а также привычные анимации и отклик интерфейса. SwiftUI предлагает набор готовых компонентов, визуально и функционально соответствующих iOS-гайдлайнам, которые являются рекомендациями, основанными на философии Human Interface Design, что позволяет создавать естественные для пользователей iOS интерфейсы.

При разработке PWA возможности ограничены рамками браузера. Из-за этого расширение функционала часто требует обходных решений, чтобы обойти ограничения наложенные iOS. Это в свою очередь приводит к тому, что разработка сложной логики на JavaScript в PWA может быть менее структурированной и безопасной. В то же время инструменты, которые предоставляет Apple для разработки приложений позволяют делать систему более надежной и структурированной, что в свою очередь позволяет строить модульную архитектуру при разработке нативных мобильных приложений.

В данной работе рассматривается разработка приложений с помощью SwiftUI из-за чего нельзя не отметить, то насколько он ускоряет процесс разработки. Декларативный подход позволяет описывать интерфейс на основе состояния, при этом не описывая полную логику взаимодействия всех элементов, которая обрабатывается исходя из описания состояний отдельно взятых элементов. Данный подход значительно упрощает код, улучшая его читаемость и масштабируемость.

Таким образом, учитывая особенности разработки мобильных приложений под iOS, область их применения и требования которые

необходимо соблюдать, чтобы удовлетворить потребности пользователей, можно выделить следующие преимуществ нативного подхода при разработке мобильных приложений:

1. Полный доступ ко всем возможностям предоставляемых компанией Apple в собственном iOS API;
2. Высокая производительность и отзывчивость интерфейса;
3. Привычное пользователю визуальное оформление и соответствием гайдлайнам установленным компанией Apple;
4. Расширяемость и гибкость архитектуры;
5. Высокая скорость разработки небольших проектов благодаря SwiftUI.

Можно сказать что нативная разработка с использованием Swift и SwiftUI предоставляет значительно больше возможностей в сравнении с прогрессивными веб-приложениями, а так же способствует созданию более качественных мобильных приложений, как с функциональной стороны, так и со стороны взаимодействия пользователя с интерфейсом. Однако важно помнить, что при разработке программного продукта недостаточно выбрать наиболее актуальную технологию, важно следовать устоявшимся принципам, которые приняты в конкретной области разработки. Это особенно актуально в мобильной разработке, где каждая платформа имеет особый подход. Таким образом важно помнить, что при разработке интерфейса мобильного приложения под iOS важно придерживаться официальных гайдлайнов Apple (Human Interface Guidelines).

1.2 Особенности создания интерфейсов мобильных приложений

Как уже было сказано ранее, наиболее важной составляющей качественного приложения является его интерфейс, который должен быть удобным и интуитивно понятным пользователю. Особенно актуально это становится из-за роста конкуренции на рынке мобильных приложений, где одним из важнейших фактором определяющим успех приложения является пользовательский опыт.

Для того, чтобы интерфейс мобильного приложения соответствовал потребностям пользователей и принципам, заложенным в философию операционной системы и платформы под которые онон разрабатывается, необходимо четко следовать определенным правилам и рекомендациям, которые называют гайдлайнами (guidelines). Такой набор рекомендации, предоставляемый разработчикам под платформы, поддерживаемые компанией Apple называется Human Interface Guidelines (HIG). Он представляет собой перечень правил и рекомендаций по проектированию UI/UX приложения и затрагивает такие аспекты приложения, как визуальный стиль, навигация, адаптивность, интерактивность и многое другое.

Гайдлайны выполняют ряд важных функций, способствующих созданию качественных мобильных приложений:

- помогают поддерживать единый визуальный стиль приложения и системы;
- обеспечивают интуитивно понятное пользователю поведение интерфейса;
- помогают избежать критических ошибок в UI/UX;
- упрощают процесс адаптации интерфейса приложения под экраны различных размеров.

В основе философии, которую продвигает компания Apple в своих устройствах стоит человек и его опыт использования приложения. Из-за такого подхода, важную роль играет принцип единства всей системы.

Приложение не должно казаться пользователю чужеродным, оно должно вписываться в общий дизайн системы, а также использовать знакомые пользователю паттерны взаимодействия, как в навигации, так и в отклике элементов.

Наиболее важными принципами в философии Apple являются:

1. Clarity - интерфейс должен легко читаться, элементы должны быть понятными, а поведение элементов предсказуемым;
2. Deference - дизайн должен быть органичным и предоставлять пользователю возможность сосредоточиться на контенте;
3. Depth - переходы между экранами приложения должны создавать эффект наслоения элементов, визуализируя структуру с помощью анимаций, создающих пространственный эффект при взаимодействие с элементами интерфейса.

Наиболее важным требованием Apple является использование стандартных системных компонентов. В SwiftUI к таковым относятся TabView, NavigationView, ScrollView, Stack, Sheet, элементы отображения информации и многие другие. Использование системных компонентов делает интерфейс более стабильным и понятным пользователю.

Apple использует собственную библиотеку адаптированных шрифтов, которые позволяют обеспечить читаемость на различных устройствах, четкие границы между информационными блоками и адаптивность при изменении размера шрифта.

Гайдлайны Apple рекомендуют избегать излишней перегруженности интерфейса, в частности не рекомендуется применять излишне яркие цвета. При разработке мобильных приложений по iOS необходимо соблюдать принцип применения акцентных цветов для активных элементов интерфейса, использовать нейтральные цвета для фоновых компонентов, а также соблюдать четкие границы для создания иерархии в интерфейсе.

Помимо общих рекомендаций по структуре и визуальному оформлению интерфейса, Apple предлагает стандартизированные паттерны

навигации, использование анимаций и системных элементов управления. Все эти особенности будут подробно рассмотрены в следующих параграфах, при рассмотрении фреймворка SwiftUI, его преимуществ и основных компонентов.

Таким образом, соблюдение гайдлайнов Apple можно считать одной из ключевых составляющих качественной разработки мобильного приложения, поскольку оно обеспечивает высокий уровень пользовательского опыта, который является ключевым показателем качества приложения в целом.

1.3 Преимущества SwiftUI

Как уже было сказано ранее SwiftUI это фреймворк предоставляемый компанией Apple для разработки интерфейсов приложений под iOS, iPadOS, macOS, watchOS и tvOS. Данный фреймворк был представлен в 2019 году, а основной причиной его появления стала потребность разработчиков в более простом и гибком способе создания интерфейсов приложений под платформы Apple. Ранее уже было сказано, что в отличие от UIKit в основе которого лежит императивный подход, из-за чего необходимо большое количество стандартного кода, что сказывается на общей читаемости и расширяемости, SwiftUI предлагает более высокоуровневый подход. Он позволяет фокусироваться в первую очередь на интерфейсе и функционале, избегая лишней разработки компонентов более низкого уровня.

Основное отличие SwiftUI от UIKit это декларативный подход в описании интерфейса. Данная модель разработки позволяет минимизировать работу над логикой изменения интерфейса. Весь процесс изменения фреймворк SwiftUI обрабатывает самостоятельно, разработчику же необходимо описывать только возможные состояния конкретного элемента и то, что он должен из себя представлять в данном состоянии. Это значительно упрощает процесс проектирования интерфейса и его разработку, благодаря сокращению количества логики, которую необходимо описать при реализации переходов между экранами и изменений самих экранов в приложении.

Вместо ручного управления иерархией представлений элементов, их обновлением и пересозданием, как это необходимо делать при использовании UIKit, разработчику необходимо просто указывать зависимости между данными, на основе которых строится интерфейс, и их представлением, после чего SwiftUI самостоятельно отслеживает все изменения в данных и обновляет все зависимые элементы интерфейса, без необходимости в дополнительной работе с иерархией. Данный подход

позволяет сделать код более чистым и структурированным, а архитектуру проекта более простой.

При использовании SwiftUI для разработки простых мобильных приложений экономится значительное количество времени при реализации элементов, экранов и их взаимодействия между собой, ведь большую часть данной обработки забирает на себя фреймворк, в том числе обработку логики взаимодействия пользователя с приложением (нажатия, скролл и т.п.) и адаптивность.

Помимо обработки значительной части базовой логики, SwiftUI так же позволяет отслеживать вносимые в интерфейс изменения в реальном времени с помощью механизма Preview. Это значительно ускоряет весь процесс разработки, особенно на начальных этапах, когда необходимо вносить огромное количество небольших изменений и четко видеть то, как они влияют на интерфейс.

Очень важным достоинством SwiftUI является читаемость и четкая структурированность кода. Весь интерфейс строится на основе принципа переиспользования базовых или ранее реализованных компонентов. Каждый отдельный компонент имеет собственную логику, благодаря чему снижается общая загруженность кода отдельного экрана, так как он реализуется вызовом отдельных компонентов, а логику отображения и обновления при взаимодействии, как уже было сказано ранее, SwiftUI обрабатывает самостоятельно. Такой подход делает код понятным и облегчает процесс его обновления и масштабирования. Компонентный подход при разработке интерфейса мобильного приложения позволяет переиспользовать элементы, а реализация универсальных компонентов, позволяет легко модифицировать элементы интерфейса, логику их отображения и взаимодействия с другими элементами. При таком подходе к проектированию приложения, переиспользование и модификация отдельно взятого компонента не затрагивает другие его вызовы, что значительно сокращает количество дублирующегося кода и делает архитектуру проекта более надежной.

Ранее было выяснено, что основой для реализации качественного интерфейса, является следование гайдлайнам. Фреймворк SwiftUI с самого момента презентации позиционируется как инструмент, позволяющий создавать интерфейс соответствующий основным гайдлайнам Apple. По данной причине практически все элементы, анимации и переходы, которые предоставляет данный фреймворк полностью соответствуют поведению и отображению элементов самой системы, благодаря чему удовлетворяют рекомендациям, описанным в Human Interface Guidelines.

Помимо общей визуальной составляющей интерфейса приложения, неотъемлемым критерием качества является удобство навигации между экранами, ее интуитивность и простота позволяют предоставить пользователю наилучший опыт от использования приложения. Для выполнения данной задачи SwiftUI предлагает стандартизированный подход, который делает взаимодействие с интерфейсом интуитивно понятным.

В своих гайдлайнах Apple выделяет 4 ключевых вида сущностей навигации:

- Tab Bar (в SwiftUI определяется как TabView) - вариант навигации между основными, чаще всего независимыми, разделами приложения. В своих гайдлайнах Apple рекомендует использовать не более 5 основных разделов, причем каждый их них должен иметь собственный функционал, обособленный от остальных;
- Navigation Stack (в SwiftUI определяется как NavigationView) - позволяет пользователю последовательно переходить от экрана к экрану, при этом имея возможность вернуться на прошлый экран с помощью системной кнопки «Назад», как будто последовательно проваливаясь все глубже внутри приложения;
- Modals / Sheets - это временные экраны, которые могут отображаться поверх основного интерфейса приложения при заранее определенных условиях, при этом не нарушая последовательность навигации между основными экранами.

Особенностью визуальной составляющей такой сущности является ощущение наслоения, благодаря чему навигация кажется более интуитивной. Хороший пример применения такой сущности будет продемонстрирован при разработке интерфейса iOS-приложения;

- Contextual Menus - системные всплывающие меню, которые чаще всего вызываются удержанием элемента. Примером является меню с действием «Назад», которое показывается при удержании пальца на соответствующей кнопке. Использование таких сущностей позволяет существенно снизить загруженность интерфейса приложения.

Для реализации навигации SwiftUI имеет готовые элементы с собственной логикой, благодаря чему она автоматически соблюдает системные принципы и гайдлайны Apple.

С самого своего создания SwiftUI был задуман как фреймворк, который будет следовать всем техническим трендам, а также будет поддерживать все технологии, разрабатываемые компанией Apple. Как и более популярный UIKit, он способен взаимодействовать с реактивным фреймворком Combine, который необходим для управления потоками данных и их асинхронной обработки. Использование асинхронности в свою очередь позволяет реализовывать интерфейсы, которые будут оперативно реагировать на изменения, сохраняя при этом целостность интерфейса при загрузке или обновлении данных. Стоит отметить, что UIKit имеет более четкую иерархию всех представлений, что необходимо в крупных проектах, но проигрывает в скорости разработки при использовании в малых и средних проектах из-за необходимости более точечной настройки взаимодействия всех потоков данных и их строгой обработки.

Таким образом, SwiftUI представляет собой современный и эффективный инструмент для разработки интерфейсов под iOS. Он значительно упрощает процесс создания пользовательского интерфейса, повышает читаемость кода, а также снижает вероятность возникновения

критических ошибок. А декларативный подход, использование стандартизированных элементов, соблюдение гайдлайнов и поддержка современных инструментов, таких как Combine, делают SwiftUI лучшим инструментом для разработки в небольших проектов.

1.4 Основные компоненты SwiftUI в мобильной разработке

В отличие от UIKit, SwiftUI имеет продуманную систему стандартных компонентов (Views), которые могут комбинироваться и изменяться (с помощью модификаторов, которые называются modifiers) для выполнения конкретной задачи. Данные компоненты можно разделить на следующие категории: базовые элементы интерфейса, навигационные компоненты, элементы управления, контейнеры для компоновки интерфейса и модальные представления.

Благодаря декларативному подходу SwiftUI обеспечивает простую иерархию компонентов представлений (Views), которые могут комбинироваться и переиспользоваться при реализации новых кастомных View. Наиболее популярные базовые компоненты являются простейшими элементами, к которым относятся Text, Image, Spacer и многие другие. Такие элементы лежат в основе любого интерфейса и их главной задачей является отображение информации. Как понятно из названия компонента, Text позволяет выводить текстовую информацию с возможностью форматирования, включая изменение шрифта, его размера, цвета, стиля или даже направления при разработке интерфейса приложения LTR и RTL. Данный компонент является наиболее распространенным в любом мобильном приложении. Компонент Image необходим для отображения изображений. SwiftUI поддерживает работу с локальными изображениями размещенными в каталоге проекта, с системными символами SF Symbols, которые способны самостоятельно адаптировать собственные параметры под описанный интерфейс, а также с изображениями, которые приложение загружает из внешних источников, например в результате выполнения запроса к серверу. Применение модификаторов к изображениям позволяет исправить формат его отображения, добиться необходимой формы и размера для дальнейшего встраивая в интерфейс. К базовым элементам также относятся все компоненты, предназначенные для отрисовки геометрических

фигур, например Rectangle, Circle, RoundedRectangle и т.п. Чаще всего данные компоненты применяются при построении сетки интерфейса, которая необходима для динамического обновления информации на странице, если для этого недостаточно возможностей системы контейнеров, которую предоставляет SwiftUI.

Данная система контейнеров позволяет разработчику собирать интерфейс приложения из различных компонентов, при этом соблюдая четкую иерархию и последовательность элементов. Основными видами контейнеров являются VStack, HStack, ZStack, которые позволяют располагать дочерние элементы (находящиеся внутри Stack) по вертикали, горизонтали или в виде наложенного поверх других элемента соответственно. Возможность такого взаимодействия с компонентами позволяет реализовывать структурированный и адаптивный интерфейс, способный динамически реагировать на изменения происходящие на экране устройства. Помимо контейнеров типа Stack, SwiftUI также предоставляет простой инструмент для создания списков простых однотипных элементов с помощью контейнера List. Такие списки могут размещаться в более сложных элементах интерфейса, при этом сохраняя независимость с точки зрения возможностей взаимодействия. Примером этого является размещение List в ScrollView. Несмотря на то, что ScrollView имеет возможность прокрути, List сохраняет собственный механизм прокрути элементов, при этом также может быть прокручен вместе с остальным экраном при выполнении свайпа непосредственно по ScrollView в который он вложен.

Контейнер ScrollView также позволяет реализовать возможность прокрутки контента на экране в вертикальном и горизонтальном направлениях. Но в отличие от списков он может содержать в себе более сложные и разнообразные компоненты, тем самым позволяя реализовывать основные экраны приложения с возможностью скрола.

SwiftUI позволяет использовать системные компоненты для обработки пользовательских действий. К ним относятся переключатели (тоглы), кнопки,

выпадающие списки, текстовые поля и так далее. Важно отметить, что как и большинство других базовых компонентов, данные элементы обработки действий полностью согласуются с интерфейсом системы, что делает взаимодействие с ними интуитивно понятным для пользователей. TextField и SecureField используются для обработки ввода текстовой информации, Toggle позволяет реализовать включение и выключение конкретной функциональности на основе состояния конкретного переключателя, а Button необходим для добавления кнопок, способных выполнять какие-либо действия при нажатии. Например открытие страницы, модального окна, WebView или обработку введенной в TextField информации.

Важно помнить, что элементы управления связаны с состоянием и очень часто его необходимо четко отслеживать для реализации сложной логики, связанной с изменением интерфейса. Для этого используют свойства @State и @Binding при объявлении переменных. Использование переменных состояния позволяет создавать интерфейсы, которые мгновенно реагируют на действия пользователя. Данный инструмент очень часто используется как при обработке взаимодействия клиента и сервера, так и при реализации клиентской логики, когда весь функционал выполняется сугубо на устройстве пользователя, например при реализации фильтрации уже существующих в памяти устройства элементов по конкретному признаку.

Помимо компонентов для реализации визуальной составляющей экранов, SwiftUI предоставляет возможность простой реализации навигации. Для этого применяются специальные контейнеры, названия которых соответствуют сущностям навигации, описанным в прошлой главе. Нижняя панель навигации (TabBar) реализуется с помощью контейнеров TabView, которые позволяют создавать независимые разделы приложения. Вложенная навигация из основных разделов реализуется с помощью NavigationStack или NavigationView (вариант с более простой реализацией, но урезанным функционалом). Для показа временных элементов или страниц применяются Sheet и fullScreenCover. Они позволяют реализовать отображения

дополнительных представлений, которые временно заменяют или дополняют основной экран, что важно для сохранения четкой навигационной иерархии. А для реализации обработки вспомогательных действий применяется contextMenu, который создает специальной меню с действиями, которое вызывается лонгтапом по элементу, тем самым снижая общую загруженность интерфейса приложения.

Как было сказано ранее SwiftUI позволяет разработчику создавать переиспользуемые компоненты и модули, которые можно применять в разных частях приложения. Такой подход к реализации возможен благодаря гибкой настройке View-структур и возможности их модификации с помощью передаваемых параметров, а также модификаторов (padding, background и др.), которые позволяют изменять внешний вид компонентов, не затрагивая их логику,

Таким образом, SwiftUI имеет широкий инструментарий, с помощью которого можно реализовать, как самые простые компоненты интерфейса, например текстовые элементы, так и сложную многоуровневую архитектуру переходов между экранами. Каждый базовый компонент сочетается с системой iOS, что позволяет быстро создавать приложения, которые будут удовлетворять потребности конечного пользователя.

1.5 Управление данными и обновление интерфейса

При разработке мобильного приложения важно помнить, что любой интерфейс зависит от данных, которые в него передаются, из-за чего скорость их обработки существенно влияет на то, как быстро приложение будет реагировать на действия пользователя и обновлять интерфейс. Для разработки качественного приложения важно грамотно организовать процесс работы с данными, которые получает клиент. Как уже было рассказано ранее, SwiftUI использует декларативный подход к реализации UI, что позволяет существенно упростить процесс его обновления, благодаря возможностям реактивного фреймворка Combine, предназначенного для работы с потоками событий.

В данном случае основной особенностью SwiftUI является его возможность связывать состояние данных (в частности значения переменных состояния) с интерфейсом. В отличие от UIKit, где любое изменение в данных вызывает методы перерисовки интерфейса, в SwiftUI применяется механизм слежения за состоянием (подписка на события). Данный механизм значительно снижает риск возникновения рассинхронизации потоков данных, благодаря чему поддерживает стабильный UI. Однако данный механизм применим только для малых и средних проектов, из-за невозможности самостоятельного изменения процесса работы с данными на более низком уровне, что крайне необходимо в больших проектах, которые должны иметь более высокую производительность.

Combine это фреймворк, разрабатываемый компанией Apple для реализации обработки событий и асинхронного взаимодействия в простом функциональном стиле. Основным преимуществом данного фреймворка является его простота применения, даже при реализации обработки последовательностей событий при изменении данных. Как и SwiftUI, Combine предлагает декларативный подход к описанию потоков данных и их

обработке, тем самым позволяя упростить реализацию реактивности интерфейса.

Для связи данных с интерфейсом в SwiftUI используется протокол ObservableObject. Основным принципом данного протокола является наблюдение за изменением состояния. Каждый класс, который поддерживает данный протокол, может содержать в себе переменные, за которыми будет вестись наблюдение. Такие переменные отмечаются с помощью аннотации @Published, благодаря чему после изменения данных информируют все подписанные на них представления, а это в свою очередь является сигналом о необходимости обновить интерфейс. Можно сказать, что ObservableObject становится прослойкой между обработкой данных и интерфейсом. Использование аннотаций @StateObject и @ObservedObject упрощают процесс реализации связи между логикой и визуальной составляющей приложения, которая необходимо при разработке реактивного интерфейса.

Кроме работы с данными в рамках одного представления, Combine позволяет переиспользовать данные в нескольких местах благодаря @EnvironmentObject. Уместное использование данного инструмента, с четким осознанием необходимости реализации @ObservedObject или @EnvironmentObject помогает поддерживать структуру модульной архитектуры, при этом соблюдая принцип единой ответственности (Single responsibility principle).

Помимо реализации обработки данных необходимой для обновления интерфейса в реальном времени важно отметить, что SwiftUI поддерживает обобщенное программирование (generics). Использование дженериков позволяет разрабатывать универсальные компоненты, которые могут обрабатывать различные типы данных. Такой подход позволяет делать интерфейс более отзывчивым к изменениям, благодаря универсальности используемых компонентов, и большей устойчивости к ошибкам при обработке данных. Именно декларативный подход способствует реализации

универсальных компонентов в виде дженериков и их переиспользованию, благодаря возможности передачи различных параметров в такие структуры.

Применение Combine и обобщённых структур значительно упрощает процесс построения архитектуры в соответствии с современным подходом MVVM (Model-View-VIewModel). В контексте разработки приложения под iOS в роли VIewModel выступает протокол ObservableObject, дженерики выступают в роли Model, а View подписывается на изменения VIewModel и обновляется при изменениях.

Таким образом использование Combine и механизмов обработки данных, которые предоставляет SwiftUI позволяют разработчику:

- Автоматически синхронизировать интерфейс и обновление данных;
- Поддерживать гибкость и масштабируемость архитектуры;
- Минимизировать количество ошибок возникающих при изменении переменных состояния;
- Поддерживать асинхронность в работе приложения.

Это делает связку SwiftUI и Combine основой при создании мобильного приложения под iOS без использования UIKit, которое способно гарантировать стабильность работы и удобство интерфейса.

Выводы по первой главе

1. Проведено исследование существующих подходов к созданию мобильных приложений, на основе которого сделан вывод, что наиболее хорошим подходом при разработке мобильных приложений, является нативная разработка, осуществляемая при помощи инструментов, предназначенных для конкретной платформы, в качестве которой в рамках данной работы выступает iOS.

2. Проведен анализ основных особенностей создания интерфесов мобильных приложений. Были рассмотрены основные прицины, которым необходимо придерживаться при разработке интерфейса. Рассмотрены основные требования и рекомендации, которые предоставляет компания Apple в гайдлайнах, ориентированных на мобильные приложения. Данный анализ позволил определить наиболее подходящий инструмент, на основе которого выполнялась разработка приложения в 2 главе данной работы.

3. Рассмотрены основные преимущества фреймворка SwiftUI при разработке мобильных приложений. Проведено сравнение с UIKit, с упором на подход к написанию кода, отвечающего за визуальное представление интерфейса, взаимодействие между элементами внутри приложения и клиентской части с сервером.

4. Исследованы основные возможности SwiftUI, рассмотрены его основные компоненты и возможности взаимодействия с фреймворком для реализации асинхронной работы приложения. Это позволило определить подходы к проектированию архитектуры приложения, включая обработку данных и связь данных с представлениями интерфейса, которые были применены при разработке iOS-приложения, процесс которой рассматривается во второй части выпускной квалификационной работы.

ГЛАВА 2. Разработка iOS-приложения с использованием SwiftUI

В прошлой главе данной работы был проведен анализ требований к интерфейсам мобильных приложений, подходов к их разработке, возможностей связки фреймворков SiwiftUI и Combine, а также основных требований к дизайну мобильных iOS-приложений в соответствии с гайдлайнам Apple. В данной главе работы будет рассматриваться процесс разработки UI мобильного приложения под iOS, включающий в себя анализ технического задания, описание функциональности приложения, проектирование архитектуры, разработка интерфейса приложения, реализация модуля для работы с данными, подключение диплинков (Deeplink - это гиперссылка, которая перенаправляет пользователя в конкретный раздел приложения), а также разработка полноэкранной нотификации.

2.1 Функциональность мобильного приложения

Данная разработка выполняется на заказ Центральной библиотечной системы Петроградского района. После проведения обсуждения с представителем библиотеки, было составлено техническое задание, в котором был описан функционал, который необходим для реализации минимально жизнеспособного продукта (MVP - Minimal Viable Product). Данное техническое задание представлено в приложении 1.

Исходя из технического задания приложение должно включать в себя 3 основных раздела и подстраницы для данных разделов, отображение которых должно происходить на основе данных полученных с сервера. Кроме основных элементов интерфейса и страниц должна быть реализована обработка открытия приложения по диплинку, а также реализована возможность показа полноэкранной нотификации.

Основные разделы приложения

В ходе анализа технического задания были выделены три основных раздела приложения: библиотеки, мероприятия и читателю.

Раздел «Библиотеки» должен содержать демонстрировать пользователю список библиотек, относящихся к ЦБС Петроградского района. Там список библиотек должен представлять собой список карточек, каждая из которых содержит в себе краткую информацию о библиотеки, а также изображение в шапке карточки. Кроме визуального отображения карточки необходимо реализовать возможность перехода на страницу библиотеки после нажатия на карточку.

Раздел «Мероприятия» должен иметь функционал схожий с разделом «Библиотеки», в нем должен быть представлен список мероприятий, информация о каждом из которых должна отображаться в виде карточки, при нажатии на которую должна открываться страница конкретного мероприятия. Помимо данного функционала раздел мероприятий должен включать в себя возможность фильтрации по типу мероприятия, логика которой должны быть реализована на клиентской части приложения.

Третий раздел приложения должен предоставлять пользователю различные полезные функции, которые включают в себя возможность оставить отзыв о ЦБС, продлить книгу, а также посмотреть ответы на самые популярные вопросы. В отличие от разделов «Библиотеки» и «Мероприятия», раздел «Читателю» не содержит подстраниц, однако каждая из описанных функциональностей должна располагаться в отдельном подразделе, навигацию между которыми необходимо реализовать через NavBar (верхняя панель навигации).

Подраздел «Популярные вопросы» должен содержать в себе список популярных вопросов, ответы на которые должны показываться при нажатии на вопрос, подраздел «Оставить отзыв» должен содержать в себе кнопку при нажатии на которую должно открываться WebView с формой обратной связи, а «Продлить книгу» должен содержать кнопку для открытия WebView с

формой для продления книги, а также ответы на популярные вопросы по продлению книг.

Подстраницы приложения

При нажатии на карточку библиотеки или мероприятия должен происходить переход на страницу библиотеки или мероприятия соответственно. При переходе должна запрашиваться полная информация о библиотеки или мероприятии, которая будет отображаться на странице.

Страница мероприятия должна содержать изображение в шапке страницы, название мероприятия, описание мероприятия, дату и время проведения мероприятия, место проведения мероприятия, а также опциональную информацию о кураторе мероприятия. В самом низу страницы должна располагаться кнопка, при нажатии на которую должен открываться WebView, содержащий страницу регистрации на мероприятии.

Страница библиотеки также должна содержать изображение, название и место расположения. Однако в отличие от страницы мероприятия, описание библиотеки должно быть свернуто в заголовок и открываться только при нажатии, из-за его объема, который значительно увеличивает размер страницы. Кнопка в самом низу страницы также, как и на страницы мероприятия, должна открывать WebView, в котором должна открываться страница с более подробной информацией о библиотеки. Кроме этого, страница библиотеки должна содержать в себе 2 дополнительных элемента: подборку изображений библиотеки, а также подборку ближайших мероприятий, проводимых в данной библиотеке.

Логика фильтрации мероприятий должна быть реализована на клиенте. При открытии страницы библиотеки, помимо запроса подробной информации, должен запрашиваться список всех ближайших мероприятий, как при при открытии раздела «Мероприятия». После получения списка мероприятий, должна выполняться его фильтрации на основе идентификаторов библиотек, а затем отображаться подборка карточек

мероприятий содержащих сжатую информацию о конкретных мероприятиях. При нажатии на такую карточку должна открываться страница мероприятия, как при нажатии на карточку в разделе «Мероприятия».

Поддержка диплинков на экраны приложения

Как уже было сказано ранее Deeplink - это ссылка, ведущая на конкретный экран мобильного приложения. Данная функциональность крайне полезна при необходимости реализации рассылок, задачей которых является направить пользователя в приложение, или создании публикаций в мессенджерах или социальных сетях, направленных на увеличение количества пользователей мобильного приложения, а также общего количества посетителей библиотек и их мероприятий.

После изучения основного функционала приложения, а также основного списка экранов, был выделен список диплинков, которые должны быть поддержаны в рамках данной разработки:

- диплинк в раздел «Библиотеки»;
- диплинк в раздел «Мероприятия»;
- диплинк в раздел «Читателю»;
- диплинк на страницу мероприятия.

Данный набор диплинков необходим в рамках разработки MVP функционала приложения и позволяет перенаправлять пользователя в различные части приложения. Однако в ходе разработки необходимо тщательно продумать схему обработки диплинков, чтобы при дальнейшем расширении функциональности приложения, поддержка ссылок на новые экраны занимала минимальное количество времени и сил.

Полноэкранная нотификация

Данный вид нотификации чаще всего называют фуллскрином (fullscreen notification - это баннер с информацией, занимающий весь экран и показывающийся поверх основного контента). Как и диплинки, данный

инструмент используется для перенаправления пользователя на конкретный экран приложения. В рамках данного проекта, фуллскрин необходим для предоставления пользователю информации о наиболее интересных мероприятиях проводимых в библиотеках Петроградского района.

Данная нотификация должна содержать в себе заголовок с названием мероприятия, краткое текстовое описание мероприятия, фоновое изображение, а также две кнопки, одна из которых должна открывать страницу мероприятия из нотификации, а другая должна просто закрывать показанную нотификацию. Необходимо отметить, что в соответствии с гайдлайнами Apple такие нотификации вне зависимости от наличия кнопки закрытия нотификации должны содержать второй интуитивно понятный пользователю элемент для закрытия, чаще всего отображающийся в виде крестика в верхнем углу нотификации.

Как и отображение интерфейса, показ данной нотификации должен быть основан на данных, которые возвращает сервер на соответствующий запрос. Если данные пришли, то необходимо показывать пользователю нотификацию, если нет, то показ осуществляться не должен. А также, чтобы фуллскрин нотификации не мешали перенаправлению пользователей из сторонних ресурсов на конкретные экраны приложения, необходимо обрабатывать случай открытия приложения по диплинку и в таком случае не показывать нотификацию.

Результат анализа технического задания

После анализа технического задания был определен основной функционал приложения, который будет реализован в рамках данной работы. Определен набор основных экранов приложения, их взаимодействие и дополнительный функционал, а также определен подход к организации проекта. На основе данной информации была составлена диаграмма вариантов использования приложения (use-case diagram) представленная в приложении 2.

2.2 Архитектура проекта

При рассмотрении теории, посвященной созданию мобильных приложений был сделан акцент на необходимости проектирования качественной архитектуры проекта, которая позволит в дальнейшем поддерживать и расширять функциональность приложения. После того, как было проанализировано техническое задание, были выделены основные сущности, которые необходимы для качественной реализации мобильного iOS-приложения для центральной библиотечной системы. Иерархия модулей проекта, после декомпозиции, представлена на рисунке 2.1.

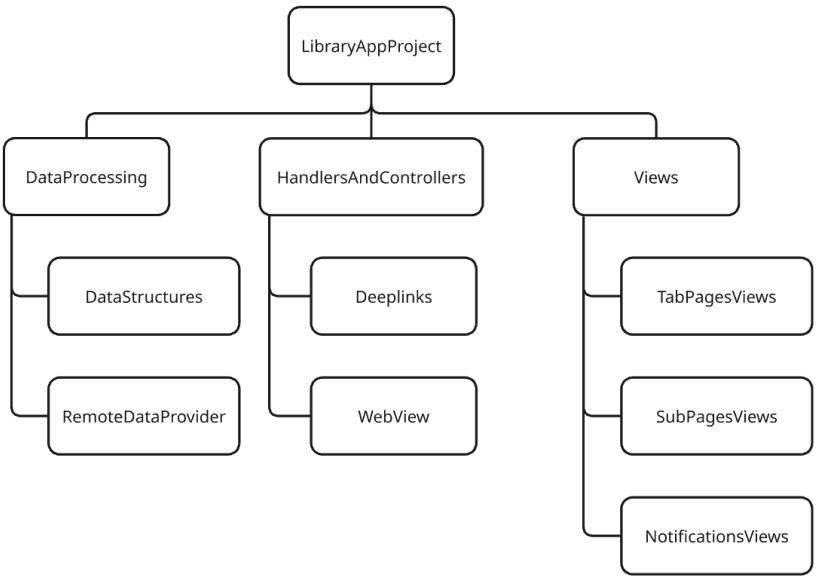

Рисунок 2.1 - Иерархия модулей приложения

Из схемы архитектуры видно, что приложений состоит из 3 основных модулей, каждый из которых выполняет определенную задачу:

- Модуль Views является основным модулем, необходимым для реализации интерфейса приложения, он содержит все представления экранов приложения, включая разделы, подстраницы, а также полноэкранную нотификацию.

- Модуль DataProcessing содержит всю реализацию обработки данных, на основе которых происходит отображение интерфейса;
- Модуль HandlersAndControllers содержит в себе реализации обработки диплинков и показа WebView, которое необходимо вызывать при нажатии на кнопки на определенных страницах.

Модуль представлений компонентов интерфейса

Как было описано ранее, данный модуль содержит в себе полную реализацию представлений интерфейса приложения. Он включает в себя визуальную составляющую основных разделов, которая реализуется в модуле TabPagesViews, визуальную составляющую подстраниц, которая реализуется в модуле SubPagesViews, а также визуальную составляющую фулскрина, которая реализуется в NotificationsViews.

При дальнейшей разработке каждый модуль будет содержать в себе определенный набор компонентов, распределенных по каталогам, что позволит сделать структуру проекта более четкой, а реализацию интерфейса более гибкой, благодаря возможности переиспользования общих компонентов. TabPagesViews будет содержать в себе каталоги, необходимые для реализации общих компонентов разделов, общих компонентов карточек, которые будут использоваться для разделов «Библиотеки» и «Мероприятия», каталог с реализацией сборки карточек для соответствующих разделов в виде отдельных компонентов с помощью переиспользования общих компонентов, а также каталог с представлениями самих разделов в виде отдельных View структур.

Стоит отметить, что еще на этапе изучения технического задания было определено, что разделы «Библиотеки» и «Мероприятия» будут иметь общую реализацию отображения списка карточек на основе обобщенного программирования с использованием дженерик структур, что в свою очередь позволит снизить дублирование кода, необходимого для реализации данных разделов.

А также важно отметить, что реализация раздела «Читателю» будет вынесена в отдельный каталог из-за своей специфичность и необходимости создания новых компонентов, с отдельной иерархией представлений и визуальной составляющей.

Модуль обработки данных

В рамках данного модуля будет реализован весь основной функционал, направленный на обработку данных. Он будет включать в себя структуры данных, на основе которых будут реализованы запросы к серверу, обработка полученных ответов и передача обработанной информации в представления интерфейса, которые будут реализованы в модуле представлений компонентов интерфейса. Помимо структур данных в нем будет реализован так называемый обработчик данных (Remote Data provider), которые будет выполнять все функции связанные с отправкой запросов, обработкой ответов, маршрутизацией и выбором окружения, в рамках которого запускается приложения. Для лучшего понимания функционала данного модуля необходимо подробнее рассмотреть его отдельные составляющие.

Для реализации взаимодействия с сервером будет реализован отдельный модуль работы с сетевыми запросами, который будет включать в себя выбор окружения (выбор базового URL к которому приложение будет отправлять запросы), хранение всех конечных URL к которым приложение может отправлять запросы (такие конечные маршруты называются Endpoints), а также строгую типизацию на основе структур данных для каждого отдельного Endpoint, которая будет определять правила взаимодействия и обработки ответов сервера.

Помимо общего модуля обработки, будут реализованы отдельные классы загрузки данных для каждого Endpoint, с которой будет взаимодействовать клиентское приложение. Исходя из результатов анализа всего было выделено 5 основных точек взаимодействия: список библиотек (libraries), список мероприятий (events), отдельная библиотека (library/{id}),

отдельное мероприятие (event/{id}) и полноэкранная нотификация (notification). Такое разделение на отдельные классы с собственными методами и инициализацией будет необходимо при дальнейшей реализации связи модуля обработки данных с представлениями визуальной составляющей интерфейса, с помощью вызова методов классов и сохранения промежуточных результатов внутри переменных состояния, которые инициализируются непосредственно в классах и благодаря методам @Published, будут иметь возможность отдавать сигнал об изменении данных и необходимости обновить интерфейс.

Модуль обработки дополнительной функциональности

Данный модуль будет именоваться как HandlersAndControllers, основным предназначением данного модуля будет являться обработка диплинков и реализация показа WebView.

Для обработки диплинков указанных в техническом задании будет необходимо реализовать 2 основных обработчика: первый будет обрабатывать диплинки на разделы приложения, которые включают «Библиотеки», «Мероприятия» и «Читателю», а второй будет обрабатывать диплинк на страницу мероприятия, реализуя последовательный переход в раздел мероприятий, а затем на страницу мероприятия.

Обработка показа WebView реализуется с помощью реализации контроллера и самого контейнера, который передается в этот контроллер. Исходя из этого и принципов проектирования архитектуры проектов, принятых в iOS разработке необходимо будет реализовать 2 отдельных файла, один из которых будет отвечать за вызов системных методов отображения веб окон поверх контента нативного приложения, а второй будет необходим для реализации обертки данного веб окна. Данная обертка является важной составляющей, необходимой для показа WebView в нативном интерфейсе в виде поднятой шторки.

Таким образом после проведения анализа технического задания было проведено планирование архитектуры проекта, выделены основные составляющие интерфейса приложения, взаимодействие между которыми было рассмотрено при описании функциональности приложения, указаны основные основные модули, необходимые для реализации функциональности обрабатываемой на уровне системы, а также полностью описан принцип работы модуля обработки данных.

Исходя из этого были выделены основные этапы на которые будет разделен весь процесс разработки:

- Разработка интерфейса;
- Реализация модуля обработки данных;
- Обработка диплинков в приложении;
- Реализация универсальной фуллскрин нотификации.

Данное разделение процесса разработки на этапы позволило оценить время, необходимое на разработку каждой отдельной составляющей приложения и приступить к реализации интерфейса приложения и его функциональности, соблюдая архитектуру, описанную в данном параграфе.

2.3 Разработка интерфейса

В прошлой главе было проведено проектирование архитектуры приложения, в соответствии с принципами, которые применяются в мобильной разработке iOS-приложений, а так же выделены основные этапы разработки, каждый из которых сфокусирован на отдельной функциональности и логике приложения.

В данном параграфе будет рассмотрен процесс разработки интерфейса приложения, универсальных компонентов, а также сборка экранов приложения из реализованных компонентов. Будут рассмотрены основные элементы, принцип их работы, а также представлены примеры готовых экранов приложения. Но, как и в любом другом проекте, первым шагом разработки стало создание репозитория на GitHub для возможности контроля версий, в процессе реализации новых модулей приложения. Ссылка на репозитория с кодом проекта представлена в приложении 3. Важно отметить, что в рамках разработки данного программного продукта, был выбран подход, при котором каждый компонент выполняет одну единственную задачу и выносится в отдельный файл, как отдельная сущность, это значительно упрощает работу с кодом и повышает его читаемость и расширяемость.

Разработка основных компонентов интерфейса

В ходе разбора теории по теме разработки мобильных приложений, было определено, что для разработки расширяемого и поддерживаемого приложения необходимо создавать универсальные компоненты, реализующие элементы интерфейса, что позволяет избегать дублирования кода и делает его более чистым и читаемым.

Первым шагом разработки интерфейса стало создание основных компонентов интерфейса и подготовка шаблонов структур для будущих компонентов. Все реализованные заглуши имеют схожий вид, благодаря

использованию декларативного подхода к написанию кода при использовании SwiftUI. Пример шаблона структуры раздела со списком библиотек представлен на листиге 2.1.

```
struct LibrariesView: View {
    var body: some View {
        NavigationView {
            …
        }
    }
}
```

Листинг 2.1 - Пример заготовки структуры раздела

После создания шаблонов структур основных страниц, а также создание каталогов для хранения будущих элементов, заготовка для модуля представлений интерфейса была полностью реализована. Структура полученного модуля представлена на рисунке 2.2.

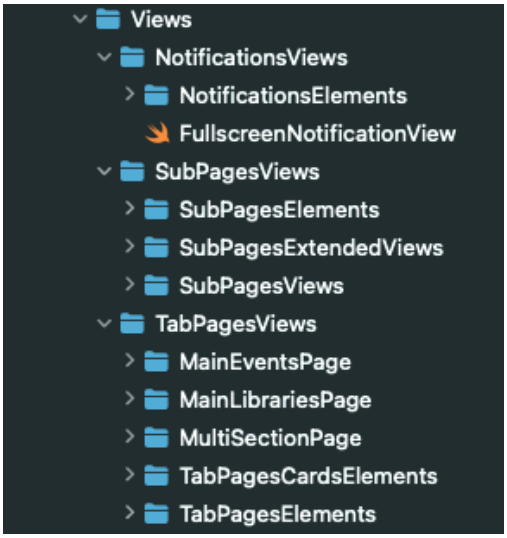

Рисунок 2.2 - Структура модуля представлений интерфейса

После того как была реализована заготовка структуры модуля для реализации интерфейса, началась работа над отдельными компонентами интерфейса. В основе разработки таких компонентов лежал принцип единственной ответственности, благодаря чему все компоненты, которые

могут быть переиспользованы на различных экранах, были реализованы в виде параметризованных структур, которые получают на вход указанный список параметров исходя их которых и происходит отображение конкретного элемента. Пример реализации кода компонента для отображения информации с иконкой в едином горизонтальном контейнере представлен на листинге 2.2.

```
struct CardInfoRow: View {
    var systemImageName: String
    var text: String
    var customColor: Color
    var customFont: Font
    var horizontalPadding: CGFloat = 25
    var body: some View {
        HStack {
            Image(systemName: systemImageName)
                .foregroundColor(customColor)
            Text(text)
                .font(customFont)
                .foregroundColor(customColor)
                .multilineTextAlignment(.leading)
        }
        .padding(.horizontal, horizontalPadding)
        .padding(.bottom, 1)
    }
}
```

Листинг 2.2 - Универсальный контейнер с информацией

Исходя из представленного кода видно, что данная структура имеет 5 основных параметров: systemImageName, text, customColor, customFont, horizontalPadding. Данные параметры необходимы для того чтобы при вызове структуры в другом компоненте, передавать в нее изображение, текст, цвет текста, шрифт текста, а также горизонтальный отступ, который может не передаваться, благодаря тому что имеет базовое значение 25, указанное в самой структуре. Такая реализация компонента позволяет использовать его несколько раз, не дублируя код, а вызывая как функцию внутри другой структуры, что будет продемонстрировано при реализации табов и подстраниц приложения.

SwiftUI имеет встроенную систему динамического Preview, что позволяет видеть влияние вносимых изменений в код на отображение конкретного элемента. Пример вызова Preview в коде представлен на листиге 2.3, а на рисунке 2.3 представлено отображение универсального горизонтального контейнера с информацией на мобильном устройстве в интерфейсе XCode.

```
#Preview("CardInfoRow Preview") {
    CardInfoRow(
        systemImageName: "clock",
        text: "25 мая 19:00",
        customColor: customColor,
        customFont: customFont
    )
}
```

Листинг 2.3 - Пример вызова динамического Preview

Рисунок 2.3 - Пример отображения элемента с помощью Preview

Аналогичным образом были реализованы остальные компоненты, необходимые для реализации карточек мероприятий и библиотек, которые будут использоваться для создания списков библиотек и мероприятий в соответствующих разделах. А также были созданы компоненты, которые в дальнейшем будут использованы при верстке страниц библиотеки и

мероприятия. Пример набора компонентов для карточек, которые необходимы для разделов «Библиотеки» и «Мероприятия» представлен на рисунке 2.4.

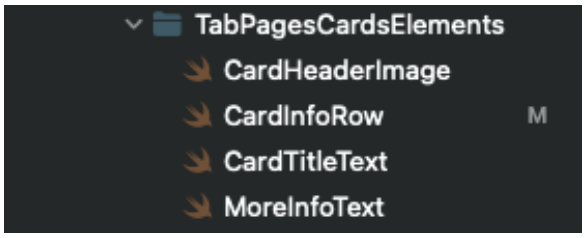

Рисунок 2.4 - Общие компоненты карточек библиотек и мероприятий

Верстка экранов приложения

После разработки простых компонентов интерфейсов приложения, была начата работа по верстке экранов для разделов «Библиотеки» и «Мероприятия», к которым относятся как сами страницы разделов, так и их подстраницы. Для поддержки универсальности кода разделов был применен подход обобщенного программирования и реализован дженерик, предназначенный для отображения списка карточек в обоих разделах. Пример реализованного дженерика приведен на листинге 2.4.

```
struct GenericNavigationListView<DataType: Identifiable,
DestinationView: View, RowView: View>: View {
    let items: [DataType]
    let destination: (DataType) -> DestinationView
    let row: (DataType) -> RowView
    var body: some View {
        VStack {
            ForEach(items) { item in
                NavigationLink(destination: destination(item)) {
                    row(item)
                }
            }
        }
        .padding(.bottom, 15)
    }
}
```

Листинг 2.4 - Дженерик для отображения списка карточек

Данный подход позволил избавиться от дублирования кода для двух разделов, благодаря тому, что данная структура способна принимать различные структуры данных, что необходимо из-за различий в структурах, на основе которых будет реализована обработка данных списка библиотек и списка мероприятий. После того, как данная структура была реализована, она была применена в структурах разделов. Пример вызова данной структуры показан на листинге 2.5.

```
GenericNavigationListView(
    items: libraries,
        destination: { library in
            LoadingDetailView(
                loader: LibraryDetailLoader(),
                loadAction: { $0.loadDetailLibraryData(libId:
library.libId) },
                dataExtractor: { $0.libraryDetailData },
                content: { LibraryDetailView(libraryData: $0) }
            )
        },
    row: { library in
        LibraryCardView(library: library)
    }
)
```

Листинг 2.5 - Пример вызова дженерика в разделе «Библиотеки»

В качестве параметров, которые были в нее переданы выступили структуры данных списков мероприятий библиотек, структуры View куда необходима была навигация при нажатии на карточку, а также сама структура карточки, в которую подставляются данные. После этого в структуру раздела мероприятий был добавлен вызов компонента, отвечающего за фильтрацию мероприятий по их типу. Таким образом 2 основных раздела были реализованы. На рисунках 2.5 и 2.6 представлены примеры данных разделов.

Рисунок 2.5 - Раздел «Библиотеки»

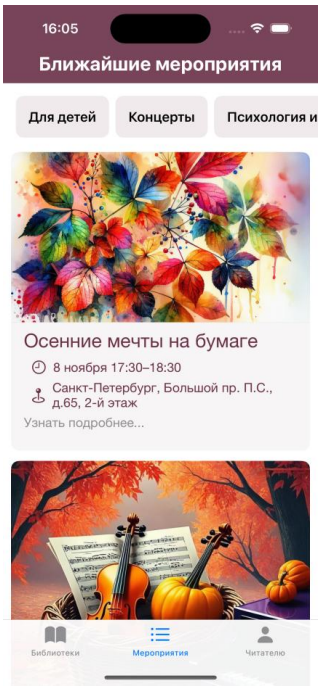

Рисунок 2.6 - Раздел «Мероприятия»

После верстки экранов данных разделов, началась работа над страницами отдельных мероприятий и библиотек. Благодаря элементам, которые были подготовлены при разработке основных компонентов, данный процесс заключался в сборе воедино компонентов, в ранее подготовленных структурах. На рисунке 2.7 представлен набор компонентов, использованных при верстке страниц.

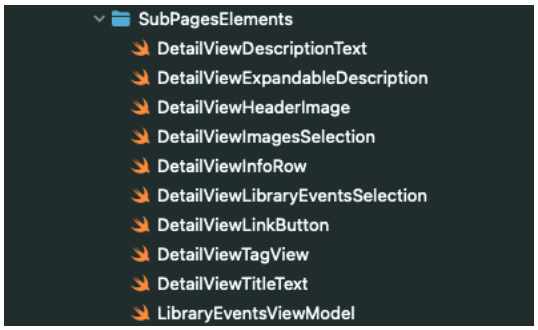

Рисунок 2.7 - Общие компоненты подстраниц

Кроме представления информации на данных страницах, необходимо было предоставить пользователю возможность просматривать дополнительную информацию, представленную на сайте библиотеки, а также

выполнять регистрацию на мероприятие прямо внутри приложения. В рамках реализации MVP функционала было решено реализовать это с помощью открытия WebView при нажатии на кнопку внизу страницы. Для этого были реализованы контейнер для WebView и контроллер, отвечающий за показ самого контента сайта на основе библиотеки WebKit. Код контроллера WebView представлен на листинге 2.6. Пример отображения реализованного WebView представлен на рисунке 2.8.

```
struct WebViewController: UIViewRepresentable {
    let url: URL
    func makeUIView(context: Context) -> WKWebView {
        let webView = WKWebView()
        let request = URLRequest(url: url)
        webView.load(request)
        return webView
    }
    func updateUIView(_ webView: WKWebView, context: Context) {}
}
```

Листинг 2.6 - Контроллер WebView

Рисунок 2.8 - Пример WebView

После реализации возможности показа WebView данная функция была добавлена на страницы мероприятия и библиотеки. Для реализации показа и обновления состояния экранов была применена переменная состояния, которая информирует интерфейс о текущем состоянии WebView. При нажатии на кнопку значение переменной состояния меняется и отображается модальное окно, с помощью вызова модифаера sheet, который позволяет показывать временное View поверх основного контента. Пример использования данной конструкции для в структуре страницы мероприятия приведен на листинге 2.7.

```
.sheet(isPresented: $showWebView) {
    if let link = libEventData.eventLink, let url = URL(string: link) {
        WebViewContainer(url: url, isPresented: $showWebView)
    }
}
```

Листинг 2.7 - Пример обработки логики показа модального окна

После добавления данного метода и остальных элементов, был также реализован клиентский функционал, который позволяет фильтровать список всех мероприятий и отображать на странице библиотеки подборку карточек мероприятий данной библиотеки. Для этого было реализовано представление самой урезанной карточки мероприятия, а также реализован механизм фильтрации списка мероприятий. Представление данной подборки было реализовано с помощью отдельной структуры, что позволит в дальнейшем свободно ее модифицировать или переиспользовать на других экранах.

Данная доработка являлась последней на этапе разработки страниц библиотеки и мероприятия, примеры отображения на основе данных взятых из кода представлены на рисунках 2.9 и 2.10 соответственно.

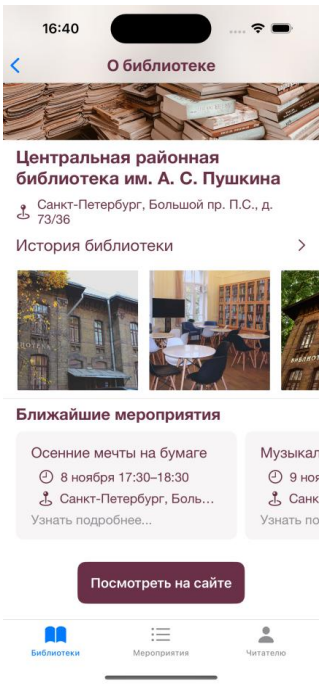

Рисунок 2.9 - Страница библиотеки

Рисунок 2.10 - Страница мероприятия

Разработка раздела с мультивыбором

После разработки экранов для разделов «Библиотеки» и «Мероприятия» началась работа над разделом «Читателю». В отличие от других разделов, реализованных в виде списков карточек, данный раздел должен был иметь верхнюю навигационную панель, которая необходима для переключения между его экранами. Так как в рамках данной работы должны быть реализованы 3 основных экрана данного раздела, а также поддержана возможность его расширения, был разработан механизм, которые позволяет отображать в верхней навигационной панели кнопки, при нажатии на которые будет выполняться перерисовка экрана. Пример кода данного механизма представлен на листинге 2.8.

```
VStack {
    if selectedTab == 0 {
        ExtendBookSectionView()
    } else if selectedTab == 1 {
        FeedbackSectionView()
    } else if selectedTab == 2 {
        QuestionsSectionView()
    }
}
.toolbar {
```

```
                ToolbarItem(placement: .navigationBarLeading) {
                    ScrollView(.horizontal, showsIndicators: false) {
                        HStack {
                            Image("logoCbs")
                                .resizable()
                                .scaledToFit()
                                .frame(width: 40, height: 40)
                            SectionTabButton(
                                title: "Продлить книгу",
                                index: 0,
                                selected: $selectedTab
                            )
                            SectionTabButton(
                                title: "Оставить отзыв",
                                index: 1,
                                selected: $selectedTab
                            )
                            SectionTabButton(
                                title: "Популярные вопросы",
                                index: 2,
                                selected: $selectedTab
                            )
                        }
                    }
                }
            }
            .toolbarBackground(customColor2, for: .navigationBar)
            .toolbarBackground(.visible, for: .navigationBar)
```

Листинг 2.8 - Механизм переключения между экранами

После реализации механизма переключения между экранами, были разработаны компоненты экранов, в частности элемент для отображения популярных вопрос, при нажатии на который происходит показ скрытого текста с использованием анимации. Данный элемент используется на экранах «Продлить книгу» и «Популярные вопросы». Реализация анимации расхлопа представлена на листинге 2.9.

```
DispatchQueue.main.asyncAfter(deadline: .now() + 0.1) {
    withAnimation(.easeInOut(duration: 0.25)) {
        isVisible = true
    }
}
```

Листинг 2.9 - Реализация анимации расхлопа

Как только все компоненты экранов были реализованы, была проведена сборка экранов из готовых компонентов, примеры отображения готовых экранов представлены на рисунках 2.12-1.14.

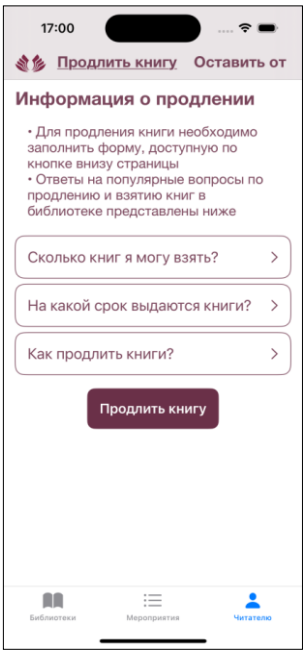

Рисунок 2.12 - Экран «Продлить книгу»

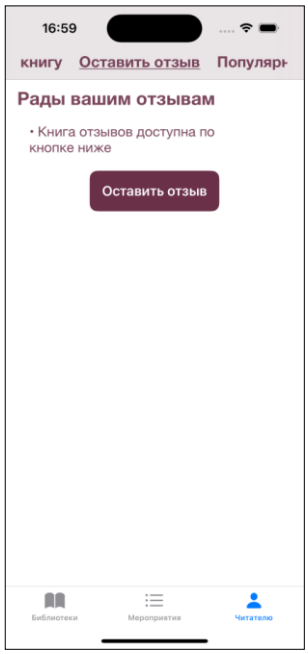

Рисунок 2.13 - Экран «Оставить отзыв»

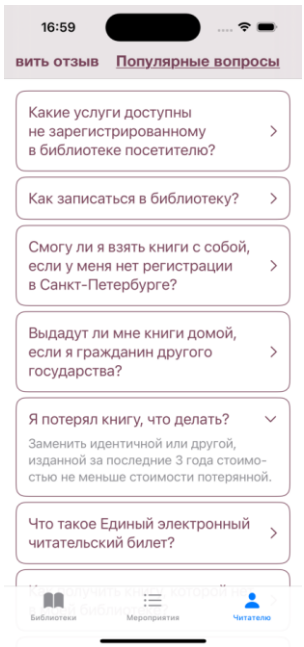

Рисунок 2.14 - Экран «Популярные вопросы»

На данном этапе основная часть разработки интерфейса iOS-приложения по заказу центральной библиотечной системы Петроградского района была завершена. Следующим этапом разработки в рамках данной работы стала разработка модуля обработки данных, которая будет рассмотрена в следующем параграфе.

2.4 Реализация модуля обработки данных

В теоретической части данной работы была подробно рассмотрена важность качественно реализованного процесс обработки данных, а в параграфе посвященном архитектуре разрабатываемого приложения была разработана структура данного модуля, которая соответствует современному подходу, который применяется при реализации iOS-приложений.

Данный этап разработки делится на 2 основных этапа:

- Разработка модуля для маршрутизации запросов и их стандартизации;
- Реализация классов для работы с данными.

Важно отметить, что до начала разработки модуля обработки данных было принято решение о необходимости реализации отдельного класса для каждого экрана, что существенно упростит дальнейший процесс их подключения к нужным представлениям интерфейса.

Разработка модуля обработки маршрутов

Первым шагом на данном этапе стало создание заготовок 3 основных компонентов, которые необходимы для реализации обработки маршрутов, а именно: Environment, APIEndpoint и APIManager.

Environment определяет базовый URL к которому будет обращаться приложение, при отправке запроса на сервер. Данный компонент позволит упростить процесс смены базового URL, так как при необходимости будет достаточно заменить его в одном месте, а не в различных частях приложения. Такой подход сделает модуль болееадаптивным, а также позволит в будущем поддерживать несколько окружений.

APIEndpoint представляет собой Enum, в котором перечислены основные пути, по которым приложение выполняет запросы. На листинге 2.10 представлен фрагмент реализованного Enum с одним путем.

```
enum APIEndpoint {
    static var events: URL {
        URL(string: "\(Environment.baseURL)/events")!
    }
    …
}
```

Листинг 2.10 - Пример APIEndpoint

После того как был описан APIEndpoint, началась разработка непосредственно APIManager, основной задачей которого является стандартизация реализации запросов к серверу, получение данных и их декодирование в соответствии с указанной структурой данных, которую приложение ожидает получить. Для выполнения данной операции был реализован универсальный метод, представленный на листинге 2.11.

```
private func fetch<T: Decodable>(
    url: URL,
    type: T.Type,
    completion: @escaping (Result<T, Error>) -> Void) {
    URLSession.shared.dataTask(with: url) { data, _, error in
        if let error = error {
            completion(.failure(error))
            return
        }
        guard let data = data else {
            completion(.failure(NSError(domain: "No data", code: 0)))
            return
        }
        do {
            let decoded = try JSONDecoder().decode(T.self, from: data)
            completion(.success(decoded))
        } catch {
            completion(.failure(error))
        }
    }.resume()
}
```

Листинг 2.11 - Метод для выполнения запросов и декодирования ответов

После реализации данного метода были реализованы вспомогательны публичные методы, с помощью вызова универсального метода, описанного выше, которые будут необходимы при реализации классов для работы с данными. На листинге 2.12 представлен код метода, который будет применен при реализации класса для работы с данными списка библиотек.

```
func fetchLibraries(completion: @escaping (Result<[Library], Error>) -
> Void) {
    fetch(url: APIEndpoint.libraries, type: LibraryResponse.self) {
result in
        completion(result.map { $0.libraries })
    }
}
```

Листинг 2.12 - Публичный метод загрузки данных для списка библиотек

После реализации аналогичных методов для остальных экранов, разработка модуля обработки маршрутов была завершена.

Реализация классов для работы данными

Как было описано в теоретической части работы, для обеспечения реактивной работы интерфейс необходимо применение реактивного фреймворка Combine, предназначенного для реализации работы с асинхронным процессами, в частности с загрузкой данных. Помимо применения данного фреймворка необходимо использовать переменные публишеры (@Published), на основе информации об изменении которых будет происходить отображение интерфейса.

Первым шагом реализации стало создание шаблонов для каждого класса, каждый из которых был обозначен как ObservableObject, после этого были реализованы методы для асинхронной загрузки данных с использованием публичных методов, реализованных на прошлом этапе, после чего были добавлены переменные с аннотацией @Published и добавлен вызов метода загрузки данных при инициализации данных классов с помощью метода init. Класс для работы с данными страницы мероприятия представлен на листинге 2.13.

```
class EventDetailLoader: ObservableObject {
    @Published var eventDetailData: LibEventData? = nil
    func loadDetailEventData(eventId: Int) {
        APIManager.shared.fetchEventDetail(eventId: eventId) { result in
            DispatchQueue.main.async {
                switch result {
                case .success(let event):
                    self.eventDetailData = event
                case .failure(let error):
```

```
                    print("Ошибка загрузки мероприятия по ID \(eventId):
\(error)")
                    self.eventDetailData = nil
                }
            }
        }
    }
}
```

Листинг 2.13 - Класс для работы с данными страницы мероприятия

После реализации, использование данных указанных в коде для отображения экранов было заменено вызовов методов данных классов, что позволило приложению выполнять запросы к серверу. На данном шаге реализация модуля обработки была завершена. На рисунке 2.15 представлен успешно выполненный запрос за информацией о списке мероприятий. Для проверки корректности использовался сниффер траффика Charles Proxy, который позволяет видеть запросы, которые выполняет мобильное приложение.

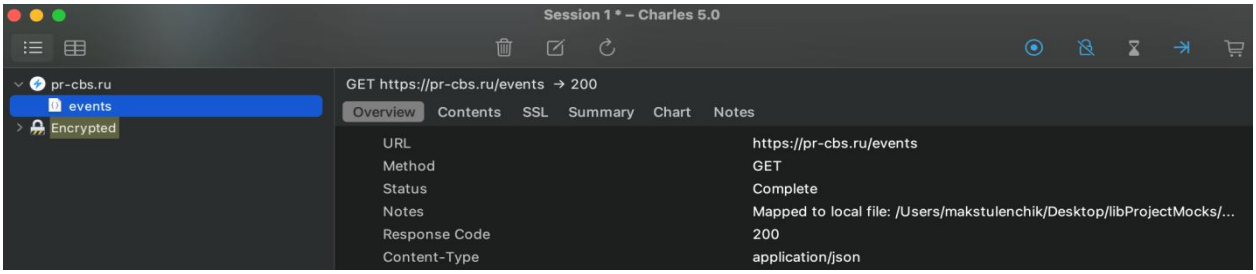

Рисунок 2.15 - Успешно выполненный запрос списка мероприятий

2.5 Обработка диплинков в приложении

Как уже было сказано ранее, диплинк - это ссылка, ведущая на конкретный экран мобильного приложения. Пример обработки базового диплинка для перехода в приложение представлен на рисунке 2.16.

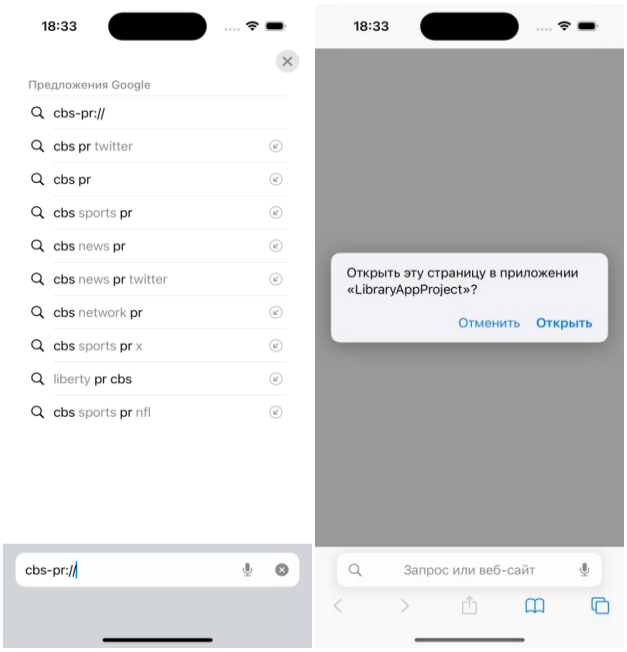

Рисунок 2.16 - Принцип работы диплинков

Реализация таких ссылок стала следующим шагом при разработке iOS-приложения в рамках данной работы. На рисунке 2.16 представлен базовый диплинк, для реализации обработки которого необходимо выполнить настройку info.plist. Для этого в настройках проекта для таргета соответствующего вашему приложению необходимо указать URL схему, которая настраивается в разделе URL Types. Пример настройки URL схемы представленной на рисунке 2.16 (cbs-pr://) представлен на рисунке 2.17.

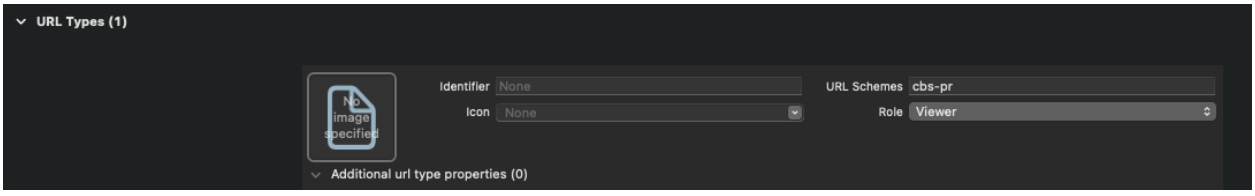

Рисунок 2.17 - Настройка URL схемы.

После того как была настроена базовая URL схема был создан Enum со списков обозначений основных табов приложения, которые будут применятся в качестве путей диплинка и указываться в качестве параметра url.host, используемого при обработке диплинков.

После этого были реализованы два обработчика, один из которых предназначен для обработки диплинков на табы приложения и реализуется с помощью механизма switch, позволяющего указывать несколько возможных вариантов обработки функции в зависимости от переданного значения (улучшенный вариант структуры if else), а второй предназначен для обработки диплинка на страницу мероприятия вида «cbs-pr://events/{eventId}». Код данного обработчика представлен на листинге 2.14.

```
struct LibEventDeepLinkHandler {
    static func handleEventDeepLink(url: URL) -> Int? {
        guard url.scheme == "cbs-pr", url.host == "events" else {
return nil }
        let components = url.pathComponents.filter { !$0.isEmpty }
        if components.count > 1, let eventId = Int(components[1]) {
            return eventId
        }
        return nil
    }
}
```

Листинг 2.14 - Обработчик диплинка на страницу мероприятия

После реализации данных методов, был добавлен их вызов в основном файле проекта (ContentView) в модификаторе «.onOpenURL», что позволило корректно открывать выбранные экраны приложения по ссылке. На этом была успешно завершена реализация обработки диплинков, входящих в MVP функционал разрабатываемого приложения.

2.6 Реализация универсальной фуллскрин нотификации

Последней функциональность, которую необходимо было добавить в рамках технического задания, являлась полноэкранная нотификация. Процесс разработки данной функциональности состоял из 2 этапов:

- Верстка визуальной части нотификации;
- Реализация логики показа нотификации.

После разработки отдельных компонентов нотификации, было сверстано ее полное визуальное представление, Пример верстки нотификации представлен на рисунке 2.18.

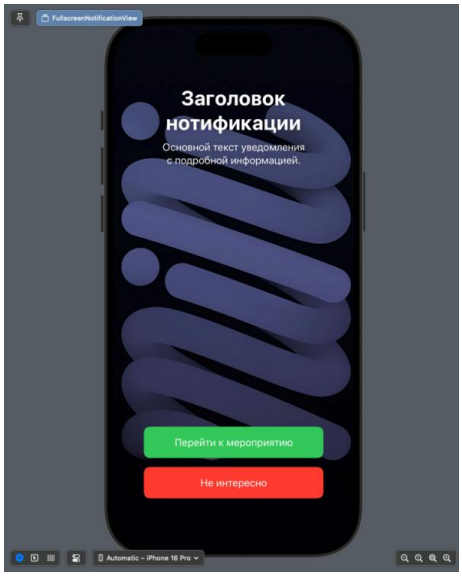

Рисунок 2.18 - Пример верстки нотификации

После того как верстка была выполнена, был реализован функционал связанный с показом нотификации и переходами при обработке нажатий на кнопки. Для реализации перехода на страницу мероприятия был использован механизм показа информации поверх текущего контента fullScreenCover, который использует NavigationStack для показа страницы мероприятия со стандартными элементами управления поверх нотификации. Помимо показа страницы мероприятия, был реализован механизм закрытия страницы события и нотификации при тапе по кнопке назад, который основан на

изменении значения переменной состояния. Код реализованного механизма обработки перехода и закрытия нотификации представлен на листинге 2.15.

```
.fullScreenCover(isPresented: $isEventPresented) {
    NavigationStack {
        LoadingDetailView(
            loader: EventDetailLoader(),
            loadAction: { $0.loadDetailEventData(eventId:
notification.eventsId) },
            dataExtractor: { $0.eventDetailData },
            content: { LibEventDetailView(libEventData: $0) })
            .toolbar {
                ToolbarItem(placement: .navigationBarLeading) {
                    Button(action: {
                        self.isEventPresented = false
                        self.isPresented = false }) {
                        Image(systemName: "chevron.left")
                            .imageScale(.medium)
                            .font(.title)
                    }
                }
            }
    }
}
```

Листинг 2.15 - Механизм реализации перехода на страницу мероприятия

После этого были добавлены новый маршрут и обработчик запросов в модуль обработки данных, а также реализован класс для работы с данными нотификации. Как только данный класс был реализован, в основной файл ContentView были добавлены переменные состояния, необходимые для остлеживания показа нотификации и наличия данных, необходимых для ее показа. После этого был использован модификатор onRecieve для отслеживания текущего состояния переменной хранящей данные о нотификации. В случае успешного получения данных, происходит изменение значения переменной показа нотификации, после чего выполняется условие и нотификация отображается поверх основного контента.

Таким образом, был реализован механизм показа нотификации, необходимый для информирования пользователя о наиболее важных мероприятиях, проводимых в библиотеках, относящихся к ЦБС Петроградского района, а разработка приложения была успешно завершена.

Выводы по второй главе

1. Проведен анализ технического задания на разработку мобильного приложения, исходя из которого, составлен перечень основных функций, элементов интерфейса и экранов приложения. Кроме текстового описания, для четкой визуализации возможностей разработанного приложения составлена UML диаграмма вариантов использования приложения.

2. Продумана и описана архитектура проекта и его составляющих, определены основные модули приложения, описано назначение каждого модуля и рассмотрены их составляющие. Помимо общей структуры проекта определены основные подходы к реализации отдельных модулей приложения, включая разработку интерфейса и обработку данных.

3. Разработаны основные компоненты интерфейса приложения, с соблюдением принципа единственной ответственности. Реализованы основные экраны приложения с применением переиспольуемых компонентов и дженерик структур, реализованных с применением принципов обобщенного программирования.

4. Разработан модуль для получения и обработки данных, примененный для реализации клиент-серверного взаимодействия, на основе которого реализовано отображение элементов интерфейса и представление информации пользователю.

5. Реализованы дополнительные функции приложения, включая обработку гиперссылок на различные экраны и показ полноэкранных нотификаций, предназначенные для привлечения пользователей на проводимые в ЦБС мероприятия.

ЗАКЛЮЧЕНИЕ

Цель данной работы заключалась в исследовании возможностей фреймворка SwiftUI и разработке мобильного приложения для Центральной библиотечной системы Петроградского района на его основе.

Данная разработка выполнялась на заказ, и для решения поставленных задач было выполнено следующее:

1. Проведен анализ требований к интерфейсам современных мобильных приложений, а также рассмотрены основные подходы, применяемые при их разработке, на основе которого было принято решение в качестве основного подхода выбрать нативную разработку с использованием SwiftUI, позволяющую четко следовать требованиям к мобильным приложениям, описанным в гайдлайнах.
2. Проведен анализ основных возможностей SwiftUI при разработке мобильных приложений. Были рассмотрены основные компоненты, используемые при разработке интерфейсов iOS-приложений, рассмотрены основные принципы, лежащие в основе реализации реактивного интерфейса, а также фреймворк Combine, применяемые для асинхронной обработки данных.
3. Проведен анализ технического задания, составлен список функциональности, включенной в MVP функционал приложения, на основе которого составлена диаграмма вариантов использования, дающая полное представление о функциональных возможностях приложения. Анализ технического задания и выделенный набор функций позволил провести проектирование архитектуры мобильного приложения, описать основные модули, их функции и варианты взаимодействия.
4. Разработан интерфейс мобильного приложения и клиентская логика, отвечающая за обработку взаимодействия клиентской части

приложения с сервером и визуализацию представлений элементов приложения, на основе полученных данных.

Таким образом, поставленные задачи были полностью решены, разработанный программный продукт передан заказчику, а цель выпускной квалификационной работы достигнута.

СПИСОК ЛИТЕРАТУРЫ

1. Amer, E. Swift Apprentice: Fundamentals (First Edition): Beginning Programming in Swift / E. Amer, A. Gallagher, M. Galloway. — First Edition. — San Francisco : Kodeco, 2016. — 384 с.
2. Apple Developer Documentation. — Текст : электронный // Apple Developer : [сайт]. — URL: https://developer.apple.com/documentation/ (дата обращения: 19.05.2025).
3. Fabian, R. Data-Oriented Design: Software Engineering for Limited Resources and Short Schedules / R. Fabian. — 1-st ed. — London : Richard Fabian, 2018. — 308 с.
4. Human Interface Guidelines. — Текст : электронный // Apple Developer : [сайт]. — URL: https://developer.apple.com/design/human-interface-guidelines (дата обращения: 10.05.2025).
5. Mathias, M. Swift Programming: The Big Nerd Ranch Guide / M. Mathias, J. Gallagher, M. Ward. — 3-rd ed. — Boston : Pearson, 2020. — 496 с.
6. Patterns of Enterprise Application Architecture / M. Fowler, D. Rice, M. Foemmel [и др.]. — 1st ed. — Boston : Addison-Wesley Professional, 2002. — 533 с.
7. Swift. Самое полное руководство по разработке в примерах от сообщества Stack Overflow. — 1-е изд. — Москва : АСТ, 2025. — 480 с. — Текст : непосредственный.
8. Акулич, М. Успешная разработка и запуск мобильных приложений / М. Акулич. — : Издательские решения, 2019. — 52 с. — Текст : непосредственный.
9. Гатин, Р. Р. ПРИМЕНЕНИЕ ПАТТЕРНА ПРОЕКТИРОВАНИЯ MVVM В РАЗРАБОТКЕ СОВРЕМЕННЫХ ПРОГРАММНЫХ РЕШЕНИЙ / Р. Р. Гатин. — Текст : непосредственный // Опыт и проблемы реформирования системы менеджмента на современном предприятии: тактика и стратегия . — Пенза : Пензенский государственный аграрный университет, 2023. — С. 95-99.

10. Грей, Э. В. Swift. Карманный справочник. Программирование в среде iOS и OS X / Э. В. Грей. — 2-е изд. — Москва : Вильямс, 2016. — 288 с. — Текст : непосредственный.
11. Интерфейс. Основы проектирования взаимодействия / А. Купер, Р. Рейман, Д. Кронин, К. Носсел. — 4-е изд. — Санкт-Петербург : Питер, 2022. — 720 с. — Текст : непосредственный.
12. Казанский, А. А. Разработка приложений на Swift и SwiftUI с нуля / А. А. Казанский. — 2-е изд. — Петербург : БХВ, 2022. — 416 с. — Текст : непосредственный.
13. Льюис, Шон Нативная разработка мобильных приложений. Перекрестный справочник для iOS и Android / Шон Льюис, Майк Данн. — 1-е изд. — : ДМК-Пресс, 2020. — 376 с. — Текст : непосредственный.
14. Маккарти, Мэтт Нативная разработка мобильных приложений. Перекрестный справочник для iOS и Android / Мэтт Маккарти. — М. : МИФ, 2018. — 368 с. — Текст : непосредственный.
15. Маскри, М. Swift 3: Разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK / М. Маскри, К. Топли, Д. Марк. — 3-е изд. — Москва : Вильямс, 2017. — 896 с. — Текст : непосредственный.
16. Мэннинг, Д. Head First. Изучаем Swift / Д. Мэннинг, П. Баттфилд-Эддисон. — 1-е изд. — Санкт-Петербург : Питер, 2022. — 400 с. — Текст : непосредственный.
17. Норман, Д. А. Дизайн привычных вещей / Д. А. Норман. — 2-е изд. — Москва : МИФ, 2018. — 380 с. — Текст : непосредственный.
18. Степанов, А. А. От математики к обобщенному программированию / А. А. Степанов, Д. Э. Роуз. — Москва : ДМК-Пресс, 2016. — 263 с. — Текст : непосредственный.
19. Усов, В. Swift. Основы разработки приложений под iOS, iPadOS и macOS / В. Усов. — 6-е изд. — : Питер, 2023. — 544 с. — Текст : непосредственный.
20. Харазян, А. А. Язык Swift / А. А. Харазян. — : БХВ, 2016. — 176 с. — Текст : непосредственный.