## **РЕФЕРАТ**

Выпускная квалификационная работа бакалавра состоит из *00* страниц, *00* рисунков, *00* таблиц, *00* использованных источников, *00* приложений.

*КЛЮЧЕВЫЕ СЛОВА, КЛЮЧЕВЫЕ СЛОВА, КЛЮЧЕВЫЕ СЛОВА, КЛЮЧЕВЫЕ СЛОВА, КЛЮЧЕВЫЕ СЛОВА, КЛЮЧЕВЫЕ СЛОВА*

Объектом разработки в данной работе является *бла-бла-бла*.

Цель работы – *бла-бла бла-бла бла-бла-бла бла-бла-бла бла-бла бла-бла бла-бла-бла бла-бла-бла.*

Для достижения поставленной цели были проведены исследования *бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла.* Основное содержание работы состояло в разработке *бла-бла бла-бла бла-бла-бла. Бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла. Бла-бла бла-бла бла-бла-бла.*

Основными результатами работы, полученными в процессе разработки, являются *бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла.*

Данные результаты разработки предназначены для *бла-бла бла-бла бла-бла-бла. Бла-бла-бла. Бла-бла бла-бла бла-бла-бла.*

Применение/внедрение/использование *(выбрать нужное)* результатов данной работы позволяет *бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла. Бла-бла бла-бла бла-бла-бла. Бла-бла-бла.*

*Текст реферата должен отражать:*

* + *объект исследования или разработки;*
  + *цель работы;*
  + *методы или методологию проведения работы;*
  + *результаты работы и их новизну;*
  + *область применения результатов;*
  + *рекомендации по внедрению или итоги внедрения результатов;*
  + *экономическую эффективность или значимость работы;*
  + *прогнозные предположения о развитии объекта исследования.*

*Если отчет не содержит сведений по какой-либо из перечисленных структурных частей реферата, то в тексте реферата она опускается, при этом последовательность изложения сохраняется.*

*Оптимальный объем текста реферата – примерно 850 печатных знаков, но не более одной страницы машинописного текста.*

*В общее количество таблиц и рисунков не входят таблицы и рисунки приложений.*

Реферат убрать Стили подправить

Подправить страницы, содержание на 1 страницу

[РЕФЕРАТ 4](#_Toc132368445)

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 6](#_Toc132368446)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 7](#_Toc132368447)

[ВВЕДЕНИЕ 8](#_Toc132368448)

[1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ 10](#_Toc132368449)

[1.1 Обзор концепции “Backend Driven UI” 10](#_Toc132368450)

[1.2 Архитектурные подходы к созданию мобильных приложений 10](#_Toc132368451)

[1.3 Существующие решения “Backend Driven UI” 11](#_Toc132368452)

[2 НАЗВАНИЕ ВТОРОГО РАЗДЕЛА (ВТОРОЙ ГЛАВЫ), В КОТОРОМ ОПИСЫВАЕТСЯ РАЗРАБОТАННОЕ РЕШЕНИЕ 13](#_Toc132368453)

[2.1 Название первого подраздела (параграфа внутри главы), содержащего формальное (теоретическое) решение задачи или модель, описание алгоритма или подхода к решению задачи 13](#_Toc132368454)

[2.2 Название второго подраздела (параграфа внутри главы), обосновывающего стек используемых технологий 13](#_Toc132368455)

[2.3 Название третьего подраздела (параграфа внутри главы), содержащая описание программной разработки 13](#_Toc132368456)

[3 НАЗВАНИЕ ТРЕТЬЕГО РАЗДЕЛА (ТРЕТЬЕЙ ГЛАВЫ), В КОТОРОМ ПРЕДСТАВЛЯЮТСЯ РЕЗУЛЬТАТЫ РАБОТЫ 15](#_Toc132368457)

[3.1 Название первого подраздела (параграфа внутри главы), содержащего характеристику условий и места использования/применения разработки 15](#_Toc132368458)

[3.2 Название второго подраздела (параграфа внутри главы), представляющего результаты работы разработанного программного кода, экранные формы, формы генерируемых документов, дашборды и все остальное 15](#_Toc132368459)

[3.3 Название третьего подраздела (параграфа внутри главы), в котором приводятся технические характеристики разработанного решения и полученных результатов с соответствующими комментариями 16](#_Toc132368460)

[ЗАКЛЮЧЕНИЕ 17](#_Toc132368461)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 18](#_Toc132368462)

[ПРИЛОЖЕНИЕ А Наименование приложения 19](#_Toc132368463)

# **ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ**

В настоящей выпускной квалификационной работе бакалавра применяют следующие термины с соответствующими определениями:

|  |  |
| --- | --- |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |
| *Термин* | *– определение* |

# **ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ**

В настоящей выпускной квалификационной работе бакалавра применяют следующие сокращения и обозначения:

|  |  |
| --- | --- |
| *Аббревиатура* | *– детальная расшифровка* |
| *Аббревиатура* | *– детальная расшифровка* |
| *Аббревиатура* | *– детальная расшифровка* |
| *Аббревиатура* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |
| *Обозначение* | *– детальная расшифровка* |

# **ВВЕДЕНИЕ**

Мобильная разработка – это длительный процесс создания мобильных приложений для смартфонов и планшетов. Чтобы создать удобное и интуитивно понятное приложение, разработчики должны уделить внимание множеству аспектов, одним из которых является разработка пользовательского интерфейса.

Разработка пользовательского интерфейса включает в себя создание и расположение элементов на экране, выбор цветовых схем, шрифтов, иконок и других графических элементов. Это может быть очень трудоемким процессом, поскольку разработчикам необходимо учитывать разные размеры экранов устройств, различные ориентации экранов и возможные варианты использования приложения. Однако тщательная работа над пользовательским интерфейсом может привлечь больше пользователей и обеспечить им комфортное взаимодействие с приложением.

Но работа над мобильным приложением не заканчивается после его выпуска. Как правило, разработчикам может потребоваться внести изменения, чтобы исправить ошибки, улучшить функциональность или привести дизайн в соответствие с новыми требованиями. Кроме того, существует необходимость в быстром реагировании на изменения на стороне сервера.

В таких случаях Backend Driven UI библиотека может стать незаменимым инструментом для разработчиков. Она позволяет создавать динамические интерфейсы, которые могут быть обновлены автоматически, когда изменяются данные на стороне сервера. Это облегчает разработку и обновление мобильных приложений и улучшает пользовательский опыт.

Кроме того, Backend Driven UI библиотека может быть использована для создания мультиплатформенных приложений, что дает возможность разработчикам повторно использовать код между различными платформами и ускорять процесс разработки. В целом, использование этой библиотеки может значительно упростить жизнь мобильным разработчикам и улучшить качество создаваемых ими приложений.

Цель исследования – разработать библиотеку, реализующую динамическое обновление верстки и навигации приложения с бекенда. По итогу должен получится фреймворк, который позволяет менять экраны, переходы и бизнес логику без обновления клиентского приложения. Кроме того, полученное решение должно быть удобным для использования разработчиками.

Для реализации соответствующей заявленной цели необходимо решить ряд задач:

1. выделить ключевые проблемы при анализе выдвигаемых требований;
2. сформировать области использования библиотеки;
3. выделение проблем существующих подходов;
4. определение самого удобного способа описания верстки, навигации и логики;
5. разработка библиотеки.

# **1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ**

## **1.1 Обзор концепции** **“Backend Driven UI”**

Backend Driven UI – это подход к проектированию пользовательского интерфейса, который предполагает, что все компоненты интерфейса приложения должны быть определены и управляемы на серверной стороне, а не на стороне клиента. Это позволяет создать гибкий и динамичный интерфейс, который можно изменять или дополнять без необходимости обновления всего приложения.

При реализации Backend Driven UI используются API-запросы для получения данных от сервера и дальнейшей обработки их в приложении. При этом сервер отвечает не только за данные, но и за структуру и логику приложения. Это позволяет сократить количество кода на стороне клиента и упростить его тестирование.

Кроме того, Backend Driven UI предоставляет большую свободу в дизайне интерфейса, так как все элементы могут быть изменены на серверной стороне. Это позволяет быстро и легко вносить изменения в интерфейс приложения без необходимости обновления самого приложения [1].

В целом, Backend Driven UI является хорошим инструменом для проектирования гибкого и эффективного интерфейса приложения, который может значительно сократить время разработки и улучшить пользовательский опыт.

## **1.2 Архитектурные подходы к созданию мобильных приложений**

Архитектурные подходы к созданию мобильных приложений имеют большое значение в разработке программного обеспечения, в том числе и для iOS-приложений. Правильно выбранный подход может обеспечить легкость поддержки и масштабирования приложения, уменьшить связность между его компонентами и улучшить его производительность.

Один из наиболее распространенных подходов к созданию мобильных приложений – это Model-View-Controller (MVC) [2]. В этом подходе приложение разделяется на три компонента: модель (Model), отвечающую за данные, представление (View), отображающее данные пользователю, и контроллер (Controller), который управляет взаимодействием между моделью и представлением. MVC используется в большинстве iOS-приложений и позволяет разделить логику приложения на легко поддерживаемые и масштабируемые компоненты.



Рисунок 1.1 – Схема архитектуры MVC

Однако существуют и другие подходы к созданию мобильных приложений. Например, Model-View-ViewModel (MVVM) – это развитие MVC, которое добавляет еще один компонент – модель представления (ViewModel). ViewModel связывает модель и представление, обрабатывает данные и управляет логикой приложения. Этот подход особенно полезен, когда необходимо реализовать реактивную логику, например, в рамках программирования на основе потоков. На рисунке 1.2 показано …

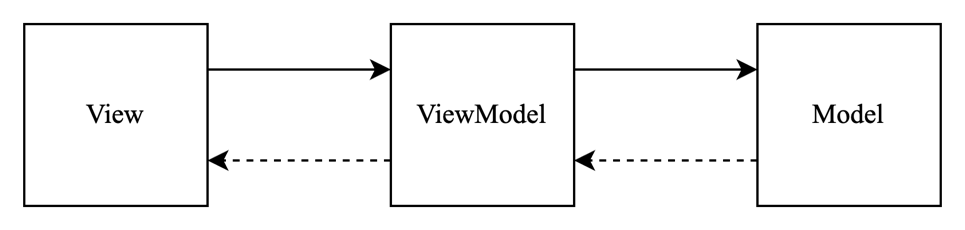


Рисунок 1.2 – Схема архитектуры MVVM

Еще одним популярным подходом является VIPER. В этом подходе приложение разделяется на пять компонентов: представление (View), презентер (Presenter), интерактор (Interactor), маршрутизатор (Router) и сущности (Entities). Представление отображает данные пользователю, презентер управляет логикой приложения и обрабатывает взаимодействие между пользователем и представлением, интерактор обеспечивает доступ к данным и бизнес-логике, маршрутизатор отвечает за навигацию в приложении, а сущности представляют данные и бизнес-логику. VIPER часто используется в коммерческой разработке и позволяет легко масштабировать, тестировать и поддерживать приложение.

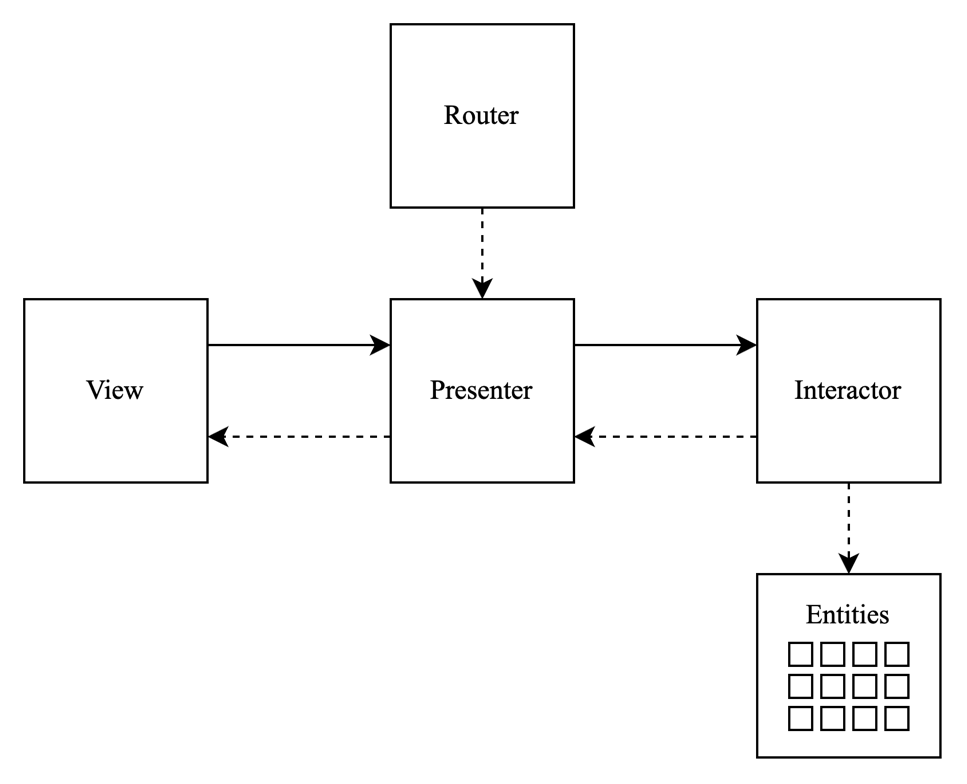


Рисунок 1.3 – схема архитектуры VIPER

Кроме того, стоит рассмотреть подход MVP. В MVP приложение разделяется на три основных компонента: модель (Model), представление (View) и презентер (Presenter). Как и в MVC, модель отвечает за данные, а представление – за отображение данных пользователю. Однако в отличие от MVC, где контроллер управляет взаимодействием между моделью и представлением, в MVP это делает презентер.



Рисунок 1.4 – схема архитектуры MVP

Презентер – это посредник между моделью и представлением. Он получает данные из модели и передает их в представление для отображения. Также презентер отслеживает действия пользователя в представлении и реагирует на них, изменяя состояние модели при необходимости. Это позволяет разделить логику приложения на более независимые компоненты и упростить тестирование.

В MVP нет явной модели представления, как в MVVM. Однако, презентер выполняет функции, аналогичные ViewModel в MVVM, такие как обработка данных и управление логикой приложения. Кроме того, в MVP также может использоваться подход программирования на основе потоков для реализации реактивной логики.

Еще один архитектурный подход – это Clean Architecture, который происходит из мира серверной разработки и был адаптирован для мобильных приложений. Он основан на принципах SOLID и DRY и позволяет создавать приложения, которые легко тестировать, масштабировать и сопровождать. В Clean Architecture приложение разбивается на несколько слоев, каждый из которых выполняет свою задачу. Наиболее важными слоями являются: Domain (доменный слой), который содержит бизнес-логику приложения, и Infrastructure (инфраструктурный слой), который отвечает за работу с сетью, базой данных и другими внешними системами.

Важно отметить, что выбор архитектуры зависит от конкретных задач и требований проекта. Некоторые подходы могут быть более подходящими для определенных типов приложений и задач, чем другие. Поэтому важно проанализировать требования к приложению и выбрать подход, который лучше всего соответствует этим требованиям.

## **Существующие решения “Backend Driven UI”**

В мире iOS-разработки существует не так много библиотек для реализации концепции Backend Driven UI с открытым доступом. Поскольку подобного рода библиотеки используются в крупных коммерческих проектах, каждая компания вкладывается в разработку собственных решений, удовлетворяя свои нужды. Однако существует ряд открытых проектов, рассмотрим основные из них:

DivKit – это библиотека от компании Яндекс, которая позволяет создавать динамические пользовательские интерфейсы на основе JSON-схем. DivKit позволяет разработчикам задавать структуру и поведение пользовательского интерфейса на стороне сервера, а затем использовать полученные данные для создания интерфейса на устройстве пользователя. Библиотека DivKit работает на основе шаблонов и позволяет быстро создавать и масштабировать пользовательские интерфейсы [3].

Библиотека JSONDrivenUI предоставляет набор классов и методов для создания пользовательского интерфейса на основе полученных данных от сервера. Одним из основных преимуществ этой библиотеки является ее легкость в использовании и гибкость.

ServerDrivenSwiftUI – это библиотека, предназначенная для создания пользовательского интерфейса на основе данных, полученных от сервера. Она использует SwiftUI для построения пользовательского интерфейса и позволяет разработчикам создавать динамические интерфейсы с минимальными усилиями. Эта библиотека предоставляет возможность создания пользовательского интерфейса с использованием простых и понятных конструкций.

Каждая из этих библиотек имеет свои особенности и преимущества, и выбор той или иной библиотеки будет зависеть от конкретных требований проекта.

# **2 РАЗРАБОТКА БИБЛИОТЕКИ ???**

## **2.1 Выбор технологий**

**2.1.1 Выбор языка программирования**

При создании библиотеки для iOS-приложений, мы должны правильно выбрать язык программирования. Ведь от правильного выбора языка зависит эффективность работы и максимальная производительность.

Существует несколько языков программирования, которые поддерживаются на iOS, но основными являются Objective-C и Swift. До появления Swift, основным языком для разработки iOS-приложений был Objective-C. Но сейчас многие разработчики предпочитают Swift из-за его современного синтаксиса и быстрой скорости выполнения.

Swift предоставляет широкий набор инструментов для разработки iOS-приложений, которые значительно упрощают процесс создания приложений. Например, Playground – это инструмент, который позволяет экспериментировать и отлаживать код. А Xcode – это интегрированная среда разработки, которая используется для сборки и разработки приложений. Кроме того, для Swift существует множество библиотек и фреймворков, которые могут значительно ускорить разработку приложений.

Еще одной причиной выбора Swift является его функциональный подход. Swift поддерживает функции первого класса, замыкания, а также множество функциональных конструкций, таких как map, filter и reduce. Это делает код на Swift более выразительным и позволяет писать более чистый и модульный код [4].

В целом, выбор Swift для разработки библиотеки Backend Driven UI был обоснован его быстродействием, безопасностью, поддержкой функционального программирования и совместимостью с существующим кодом на Objective-C. Поэтому, мы можем быть уверены, что использование Swift в нашем проекте обеспечит наиболее эффективную и быструю работу нашей библиотеки.

Кроме Objective-C и Swift, также можно реализовать библиотеку на языке Kotlin с использованием KMP (Kotlin Multiplatform). Kotlin – это язык программирования, который был создан компанией JetBrains и считается официальным языком для разработки приложений на платформе Android. Однако, благодаря поддержке KMP, он также может быть использован для создания библиотек, которые могут быть запущены на различных платформах, включая iOS.

Одним из главных преимуществ использования Kotlin Multiplatform для разработки библиотеки Backend Driven UI является возможность повторного использования кода для Android приложений. Это позволяет значительно сократить время разработки и упростить поддержку библиотеки в будущем.

Однако, у этого подхода есть и свои недостатки. Во-первых, Kotlin Multiplatform все еще находится в стадии ранней разработки, поэтому его возможности могут быть ограничены. Во-вторых, поддержка KMP на iOS хуже, чем для Android.

В итоге, хотя использование Kotlin Multiplatform может иметь свои преимущества, учитывая цель разработки библиотеки исключительно для iOS, нативная библиотека на Swift все же остается предпочтительным выбором. Она обеспечивает наилучшую производительность, максимальную стабильность и совместимость с существующим кодом на Objective-C, что является критически важным фактором при разработке библиотек для iOS.

**2.1.2 Выбор библиотеки для отрисовки пользовательского интерфейса**

Для описания верстки приложений на языке Swift существуют две основные библиотеки: UIKit и SwiftUI.

UIKit является основной библиотекой для создания пользовательского интерфейса на iOS. Она была создана еще в 2007 году и с тех пор активно используется разработчиками. UIKit и предоставляет широкий набор инструментов для создания пользовательского интерфейса, таких как: кнопки, текстовые поля, таблицы, коллекции и многое другое [5-6].

SwiftUI, в свою очередь, является новой библиотекой, которая была представлена компанией Apple на конференции WWDC 2019. SwiftUI основывается на декларативном подходе и позволяет создавать интерфейс с помощью простых и интуитивных команд.

Можно выделить следующие плюсы UIKit:

Списки поправить

* широкий набор инструментов для создания пользовательского интерфейса,
* многолетний опыт использования и устоявшаяся практика,
* хорошая документация и большое количество примеров.

Однако недостатками являются:

* отсутствие декларативного подхода может приводить к увеличению количества кода,
* вероятное прекращение поддержки в течение текущего десятилетия.

При использовании SwiftUI имеем следующие положительные стороны:

* простой и интуитивный синтаксис,
* декларативный подход позволяет создавать интерфейс с помощью нескольких строк кода.

К минусам SwiftUI отнести:

* Недостаточную поддержку старых версий операционной системы,
* Недостаточную документацию и меньшее количество примеров, чем у UIKit.

Была выбрана библиотека UIKit, так как она обладает широким набором инструментов для создания пользовательского интерфейса, у нее больше опыта использования и устоявшаяся практика. Также декларативный подход, который присутствует в SwiftUI, не является ключевым для нашей библиотеки и мы можем достигнуть необходимого уровня гибкости и простоты кода с помощью UIKit.

2.1.3 Выбор архитектуры

В подразделе 1.2 мы рассмотрели различные архитектурные подходы для создания мобильных приложений, однако в нашем случае наиболее удачным для тестового приложения является MVP.

Основным преимуществом MVP является его модульность, простота и легкость в понимании. Кроме того, MVP облегчает тестирование, так как каждый компонент можно тестировать отдельно от других. Это позволяет создавать более стабильные и надежные приложения.

С другой стороны, MVP имеет и некоторые недостатки. Один из них заключается в том, что код может стать слишком сложным и раздутым, особенно если приложение становится больше и сложнее. Кроме того, разделение на три компонента может привести к тому, что разработка приложения может занять больше времени и требует большего количества кода, чем в других архитектурных подходах.

Сравнив MVP с другими популярными архитектурами, такими как MVC и VIPER, можно отметить, что MVP является более простым и понятным подходом. MVC, например, требует более сложного взаимодействия между компонентами, что может затруднить разработку и тестирование приложения. В то же время, VIPER является более сложным и расширяемым архитектурным подходом, но он также требует большего количества кода и времени для разработки.

В итоге, учитывая то, что наше тестовое приложение относится к категории небольших приложений, MVP является наилучшим выбором для него. Этот подход позволит быстро и легко разрабатывать приложение, при этом обеспечивая его стабильность и надежность.

## **2.2 Проектирование архитектуры библиотеки**

Одной из ключевых задач при разработке библиотеки является правильная архитектура, которая позволит в дальнейшем легко поддерживать и развивать ее. В данном подразделе мы рассмотрим структуру json файла конфигурации, структуру элементов пользовательского интерфейса и основные классы библиотеки.

2.2.1 Структура файла конфигурации

Файл конфигурации, является важным элементом в процессе создания приложения. Перед тем, как мы начнем глубже разбираться в структуре данного файла, давайте определимся, что такое json.

JSON – это формат данных, основанный на текстовом представлении объектов, предназначенных для обмена данными между различными программами. Он является удобным и универсальным способом передачи информации, поскольку данные в формате json могут быть легко прочитаны как человеком, так и программным обеспечением.

Итак, json файл конфигурации будет содержать массив структур, которые описывают каждый экран приложения – BDUIScreen. Каждый экран имеет уникальный идентификатор (id), по которому можно производить переход на него. Кроме того, экран содержит корневой элемент типа BDUIView, который является главным элементом интерфейса соответствующего экрана.

2.2.2 Структура элементов пользовательского интерфейса

Интерфейс является одним из важнейших компонентов, который определяет визуальное взаимодействие между пользователем и приложением. Интерфейс должен быть интуитивно понятным, легко управляемым и приятным в использовании.

Существует несколько способов описания положения элементов интерфейса на экране. Один из способов – это явное указание координат прямоугольника, содержащего наш элемент. Однако, данный подход не является оптимальным, поскольку при изменении размеров экрана или разрешения устройства, элементы могут располагаться неправильно и некорректно отображаться на экране.

Второй способ – это указание якорных точек, которые в последствие будут использованы в системе AutoLayout, которая посредством решения линейных уравнений вычислит нужные координаты. Этот метод более гибкий, чем предыдущий, но требует множества различных якорей и более сложной логики. Итоговое количество якорей должно быть минимальным, также недопустимо дублирование информации посредством якорей разного типа – получим конфликт при итоговом решении системы уравнений координат.

Третий метод – использование контейнеров, которые с помощью своей внутренней логики представления будут размещать элементы. Этот метод является наиболее гибким и удобным для использования, поскольку позволяет делегировать расположение элементов контейнерам (горизонтальным, вертикальным) и задавая параметры отображения получать нужную верстку.

В реализации нашей библиотеки мы преследуем цель упростить создание приложений и предоставить разработчикам инструменты для быстрого и удобного создания элементов пользовательского интерфейса. Поэтому удобнее всего использовать третий метод –контейнеры. Они были придуманы как упрощение AutoLayout и используются повсеместно вместе с ним. Таким образом, наш элемент BDUIView будет содержать не только информацию о своем внешнем виде (стиль, наполнение), но и будет хранить в себе дочерние элементы, которые будут отрисовываться согласно типу элемента.

Также базовый класс BDUIView будет содержать в себе общую логику для всех элементов, такую как настройка своих размеров и внешнего вида.

При создании любого элемента пользовательского интерфейса необходимо задать его размеры. Их можно задать как явно, указав ширину и высоту, так и неявно, ограничив размер элемента относительно родительского контейнера. В первом случае размеры элемента будут фиксированными, а во втором – зависеть от размеров родительского контейнера и настроек отображения.

Настройка внешнего вида элемента – это задание его стиля, цвета, текстуры и т.д. В зависимости от типа элемента, эти настройки могут быть различными. Например, для кнопки мы можем задать стиль кнопки в обычном и фокусном состояниях и т.д., а для контейнера – фоновый цвет, отступы между дочерними элементами и т.д.

Дочерние классы BDUIView могут дополнять базовую логику настройки, в зависимости от их типа. Например, для контейнера необходимо будет задать логику расположения дочерних элементов. Для этого мы можем использовать систему координат, где каждый элемент имеет свои координаты относительно родительского контейнера. Координаты могут быть явно заданы или рассчитаны автоматически с помощью системы AutoLayout.

Для элемента типа текст необходимо задать настройку отображаемого текстового стиля. В зависимости от настроек, текст может быть выровнен по левому краю, по центру или по правому краю. Также можно задать шрифт, размер шрифта, цвет текста и т.д.

Таким образом, каждый элемент UI в нашей библиотеке имеет свою уникальную логику настройки и отображения. Однако, общая логика определена в базовом классе BDUIView, что позволяет как уменьшить количество кода самого компонента, так и унифицировать представление компонента в файле конфигурации

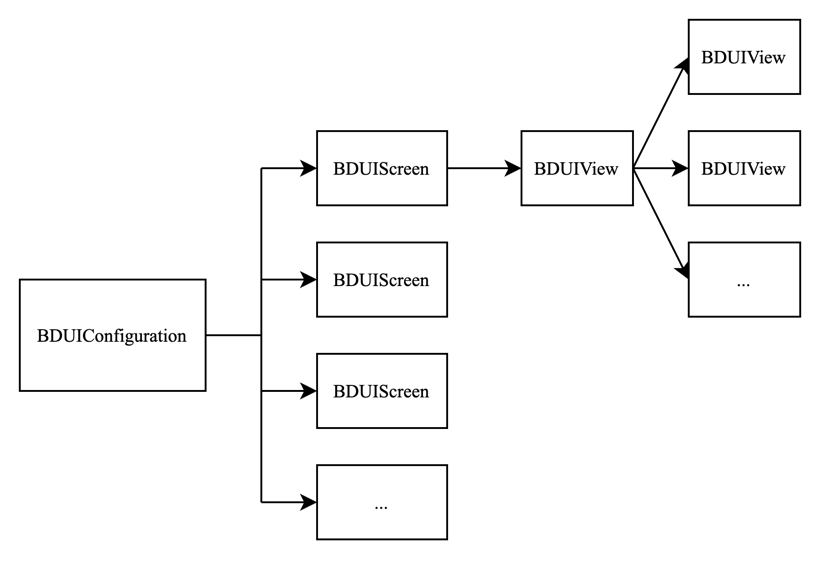


Рисунок 2.1 – структура модели конфигурации приложения

2.2.3 Структура основных классов библиотеки

Рассмотрим структуру главных классов нашей библиотеки и их взаимодействие между собой.

Для парсинга полученной конфигурации элементов пользовательского интерфейса мы будем использовать отдельный объект BDUIParser. Он будет отвечать за разбор конфигурационных файлов и создание объектов, соответствующих каждому элементу пользовательского интерфейса.



Рисунок 2.1 – Cхема работы BDUIParser

Для настройки и создания объектов пользовательского интерфейса мы будем использовать паттерн Factory. Класс BDUIViewFactory будет отвечать за конфигурацию и создание объектов UI-элементов. Он позволит нам легко создавать объекты с различными настройками и параметрами, а также обеспечит единый интерфейс для работы с ними.

Одним из главных классов нашей библиотеки будет BDUIScreensController. Он будет инкапсулировать в себе всю логику работы с экранами, включая навигацию между экранами и передачу данных между модулями приложения. Это позволит разделить логику работы с экранами на отдельный уровень и упростить разработку приложений, особенно для более сложных проектов. Также он будет хранить в себе конфигурацию и фабрику, таким образом по необходимости отрисовки определенного экрана, контроллер будет запрашивать модель экрана у BDUIConfiguration, затем с помощью BDUIFactory сконструирует интерфейс экрана и передаст его в модуль экрана.

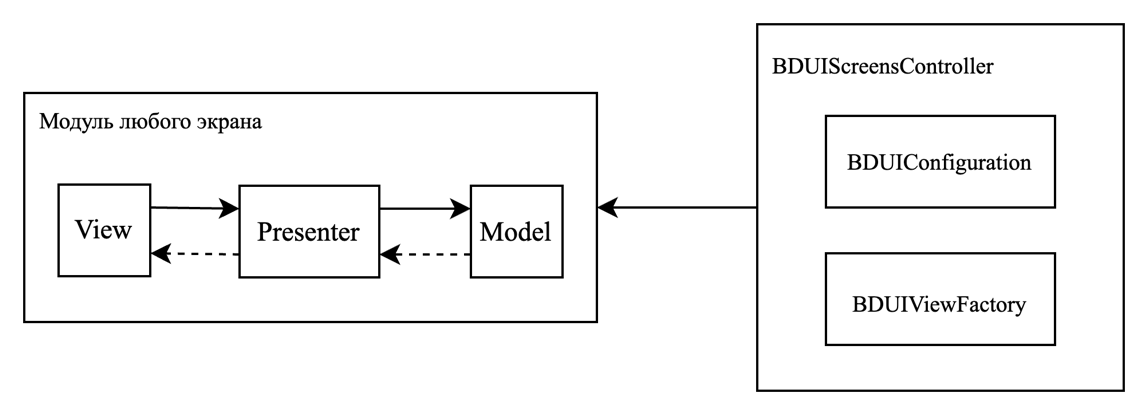


Рисунок 2.1 – схема работы BDUIScreensController

Для объединения всех классов в единую структуру можно использовать либо Dependency Injection, либо можно создать отдельный класс BDUIManager, который будет отвечать за сборку всех компонентов и предоставлять единый интерфейс для работы с ними. Такой подход позволит упростить управление всеми компонентами библиотеки и обеспечить более гибкую настройку.

Важно отметить, что использование хорошо структурированных классов в нашей библиотеке поможет нам создавать более гибкие и масштабируемые приложения, упростить их разработку и обеспечить высокую степень переиспользуемости кода*.*

# **ЗАКЛЮЧЕНИЕ**

В результате выполнения выпускной квалификационной работы бакалавра были выполнены следующие задачи:

1. проведен анализ существующих решений в сфере Backend Driven UI для мобильных приложений,
2. выявлены основные проблемы и потребности пользователей данных библиотек,
3. спроектирован удобный способ описания верстки,
4. разработана обозначенная библиотека.

Разработанная библиотека позволяет легко отрисовывать интерфейсы приложений целиком полагаясь на конфигурацию из бекенда. Также на бекенд были вынесены логика перемещения между экранами. Таким образом, реализованное решение идеально подходит для реализации небольших экранов, содержащих дополнительную информацию или простую логику использования, что упрощает их поддерживание и создание.

Также библиотека позволяет обновлять верстку экранов без выпуска новых версий использующих ее приложений. Это позволяет оперативней доводить новый функционал до потребителя, без необходимости прохождения долгого процесса обзора приложения магазином приложений.

Помимо возможности менять верстку также имеется возможность менять пользовательский путь. При необходимости можно ограничивать доступ пользователей к тем или иным экранам, элементам. Это может пригодиться, например, в случае выявления критического бага, чтобы оставить падение приложения.

Библиотека реализована модульно, что позволяет без особого труда расширять ее, наполняя новыми элементами пользовательского интерфейса, что позволит расширить границы ее применения.

В данной работе была рассмотрена в основном клиентская часть реализации концепции Backend Driven UI. В дальнейшем можно улучшить серверную часть библиотеки, как минимум настроив версионирование конфигурационных файлов для конкретных версий приложений, как максимум добавив обработку действий пользователя на бекенд, таким образом можно менять логику работу приложения посредством изменения бекенд кода, а не кода конфигурационного файла.

Однако, наше решение имеет ряд общих для подобных библиотек недостатков. Использование Backend Driven UI в мобильных приложениях усложняет их тестирование, что может повлечь к увеличению числа багов и усложнению их поиска и устранения, поскольку кроме кода мобильного приложения нужно анализировать конфигурационный файл.

В целом, получившиеся решение решает поставленные перед ним задачи и позволяет уже сейчас начать переводить часть пользовательского интерфейса или целые экраны на Backend Driven UI.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Доклад «Неправильный путь в Backend Driven UI» – URL: <https://www.youtube.com/watch?v=X5306RAU5S4&ab_channel=Разработка> (дата обращения 20.04.2023).
2. Cocoa Core Competencies Model-View-Controller – URL: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html (дата обращения 10.02.2023).
3. Документация DivKit – URL: https://divkit.tech/ru/doc/ (дата обращения 14.03.2023).
4. Swift Documentation, Functions – URL: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/functions/ (дата обращения 12.03.2023).
5. UIKit documentation – URL: https://developer.apple.com/documentation/uikit (дата обращения 11.03.2023).
6. Доклад Яндекс Go готовит Backend Driven UI – URL: https://www.youtube.com/watch?v=Pmx6Hs1nyvg (дата обращения 19.04.2023).