УТВЕРЖДЁН

RU.17701729.06.05-01 81 01-1

A close up of a label

Description automatically generated

**Мобильное Приложение для Мотивации и Контроля Выполнения Привычек**

**Пояснительная записка**

**RU.17701729.06.05-01 81 01-1**

**Листов 33**

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 3](#_Toc195196888)

[1. ВВЕДЕНИЕ 5](#_Toc195196889)

[1.1 Наименование программы 5](#_Toc195196890)

[1.2 Документ, на основании которого ведётся разработка 5](#_Toc195196891)

[2. НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ 6](#_Toc195196892)

[2.1. Назначение программы 6](#_Toc195196893)

[2.1.1 Функциональное назначение 6](#_Toc195196894)

[2.1.2 Эксплуатационное назначение 6](#_Toc195196895)

[2.2 Краткая характеристика области применения 6](#_Toc195196896)

[3. ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ 7](#_Toc195196897)

[3.1. Постановка задачи на разработку программы 7](#_Toc195196898)

[3.2. Описание алгоритма функционирования программы 12](#_Toc195196899)

[3.2.1. Описание архитектуры приложения 12](#_Toc195196900)

[3.2.2. Описание сетевого слоя 14](#_Toc195196901)

[3.2.3. Описание реализации skeleton-loader’ов 17](#_Toc195196902)

[3.2.4. Описание созданных кастомных элементов интерфейса 18](#_Toc195196903)

[3.2.5. Описание созданных менеджеров данных 19](#_Toc195196904)

[3.2.6. Описание созданных автотестов (UNIT, UI) 20](#_Toc195196905)

[3.2.7. Описание кода и возможности сопровождения 21](#_Toc195196906)

[3.2.8 Описание надежности и актуальности подходов 22](#_Toc195196907)

[3.2.9 Описание работы с многопоточностью 22](#_Toc195196908)

[3.3. Описание и обоснование выбора метода организации входных и выходных данных 24](#_Toc195196909)

[3.3.1. Описание метода организации входных данных и выходных данных: 24](#_Toc195196910)

[3.3.2. Обоснование метода организации входных и выходных данных: 24](#_Toc195196911)

[3.4. Описание и обоснование выбора технических и программных средств 25](#_Toc195196912)

[3.4.1. Описание технических и программных средств 25](#_Toc195196913)

[Приложение можно запустить как на мобильном устройстве iPhone, так и на симуляторе мобильного устройства iPhone с операционной системой iOS 14.0 и выше. Мобильное устройство должно обладать следующими характеристиками: 25](#_Toc195196914)

[Минимальные требования: 25](#_Toc195196915)

[3.4.2. Обоснование выбора используемых технических и программных средств 26](#_Toc195196916)

[4. ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ 28](#_Toc195196917)

[4.1. Ориентировочная экономическая эффективность 28](#_Toc195196918)

[4.2. Предполагаемая потребность 28](#_Toc195196919)

[4.3. Экономические преимущества разработки по сравнению с отечественными и зарубежными аналогами 28](#_Toc195196920)

[5. ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ 29](#_Toc195196921)

[ПРИЛОЖЕНИЕ 1 31](#_Toc195196922)

[ПРИЛОЖЕНИЕ 2 32](#_Toc195196923)

# 1. ВВЕДЕНИЕ

## 1.1 Наименование программы

**Наименование темы разработки:** Мобильное Приложение для Мотивации и Контроля Выполнения Привычек.

**Наименование темы разработки на английском языке:** Mobile App for Motivation and Habit Tracking.

**Краткое наименование программы:** «Doordie».

## 1.2 Документ, на основании которого ведётся разработка

Основанием для разработки является учебный план подготовки бакалавров по направлениям 09.03.04 «Программная инженерия» и утвержденная академическим руководителем тема курсового проекта.

# 2. НАЗНАЧЕНИЕ И ОБЛАСТЬ ПРИМЕНЕНИЯ

## 2.1. Назначение программы

### 2.1.1 Функциональное назначение

В настоящее время все больше людей стремятся к личной эффективности и саморазвитию, для этого они ищут инструменты, которые помогают формировать полезные привычки и достигать поставленных целей. Мобильное приложение «Doordie» предлагает пользователям отслеживание выполнения своих привычек. Функциональные возможности приложения включают в себя: создание привычек; отслеживание выполнения привычек; визуализация прогресса выполнения; получение мотивационного сообщения во время выполнения привычек; отслеживание подряд идущих дней, когда были выполнены привычки. Благодаря этим инструментам пользователи смогут не только организовать свой день, но и получить дополнительную мотивацию для достижения следующих целей.

### 2.1.2 Эксплуатационное назначение

Мобильное приложение «Doordie» предназначено для индивидуального использования пользователями iOS-устройств в целях обеспечения пользователя инструментом для формирования, управления и анализа привычек.

Приложение рассчитано на использование в повседневной жизни, независимо от места и времени.  
 Приложение ориентировано на следующие сценарии:

* Организация утренней, дневной или вечерней рутины.
* Отслеживание выполнения привычек.
* Анализ достижений для повышения личной мотивации.

«Doordie» интегрируется в повседневную деятельность пользователей, целевая аудитория – как опытные, так и начинающие пользователи мобильных приложений любого возраста.

## 2.2 Краткая характеристика области применения

Мобильное приложение «Doordie» предназначено для индивидуального использования пользователями iOS-устройств с целью формирования, контроля и анализа выполнения привычек. Область применения включает в себя развитие полезных привычек, достижение долгосрочных целей и улучшение качества жизни через структурированный подход к ежедневным задачам

# 3. ТЕХНИЧЕСКИЕ ХАРАКТЕРИСТИКИ

## 3.1. Постановка задачи на разработку программы

Мобильное приложение должно предлагать пользователю следующий функционал:

1. **Начальный экран:**
   1. Возможность перейти на экран входа;
   2. Возможность перейти на экран регистрации;
2. **Экран входа:**
   1. Возможность перейти на экран восстановления пароля;
   2. Возможность перейти на экран регистрации;
   3. Возможность ввести адрес электронной почты и пароль;
   4. Проверка существования аккаунта, привязанного к указанной почте;
   5. В случае, если адрес электронной почты привязан к аккаунту, но введен некорректный пароль, уведомить пользователя об этом;
   6. Возможность перейти к следующему шагу при нажатии кнопки «войти»;
3. **Экран восстановления пароля:**
   1. Возможность ввести адрес электронной почты;
   2. В случае, если адрес электронной почты не привязан к аккаунту, предложить пользователю зарегистрировать аккаунт;
   3. В случае, если адрес электронной почты привязан к аккаунту, уведомить пользователя об отправке последующих инструкций на указанную почту;
4. **Экраны регистрации:**
   1. Экран ввода почты:
      1. Возможность ввести адрес электронной почты;
      2. Проводить проверку соответствия введенного адреса электронной почты корректному формату при помощи *регулярных выражений* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1));
      3. В случае ввода адрес электронной почты, к которой уже привязан аккаунт, предложить пользователю восстановить пароль;
      4. В случае, если введенный адрес электронной почты введен в корректном формате и не привязан к какому-либо аккаунту, предоставить возможность перейти к следующему этапу;
   2. Экран ввода четырехзначного кода:
      1. Возможность ввести четыре отдельных цифры;
      2. Уведомлять пользователя о введенном некорректном коде;
      3. В случае ввода корректного вода направлять пользователя на следующий этап регистрации;
      4. Возможность повторно отправить код каждые шестьдесят секунд;
      5. При нажатии кнопки повторной отправки кода изменять корректный код;
   3. Экран ввода имени пользователя:
      1. Возможность ввести имя пользователя;
      2. Проводить проверку корректности введенного имени (длина не менее двух символов);
      3. В случае корректно введенного имени пользователя предоставить возможность перейти к следующему этапу;
   4. Экран ввода пароля:
      1. Возможность ввести пароль;
      2. Проводить проверку корректности введенного пароля (длина не менее восьми символов);
      3. В случае корректно введенного пароля предоставить возможность перейти к следующему этапу;
      4. Отображать информацию о проведении авторизации после нажатия на кнопку «зарегистрироваться»;
5. **Главный экран:**
   1. Отображение нижней панели навигации со следующими кнопками:
      1. Переход на экран аналитики;
      2. Переход на экран создания привычки;
      3. Переход на экран друзей;
      4. Переход на экран настроек;
   2. Отображение всех привычек пользователя и их статусом выполнения (выполнено/не выполнено);
   3. Возможность перейти на экран выполнения привычки по нажатию на привычку;
   4. Отображение мотивирующего сообщения, содержащего в себе процент выполненных привычек за текущий день, количество выполненных привычек и общее количество привычек, а также мотивирующий текст;
   5. Возможность выбрать часть дня, по которой будет происходить фильтрация привычек;
   6. Отображение ближайших дат в виде числа и дня недели;
   7. Отображение текущей даты;
   8. Отображение skeleton-loader’ов (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)) вместо привычек до тех пор, пока данные не загрузятся с сервера;
   9. Возможность перейти в профиль пользователя по нажатию на картинку пользователя;
   10. Возможность закрыть мотивирующее сообщение;
6. **Экран создания привычки:**
   1. Возможность ввести название привычки и мотивацию для выполнения привычки;
   2. Возможность выбрать цвет и картинку привычки;
   3. Возможность выбрать количество времени/подходов, необходимого для выполнения привычки;
   4. Возможность выбрать единиц измерения выполнения привычки;
   5. Возможность выбора частоты выполнения привычки;
   6. Возможность выбора части дня, в которую предполагается выполнять привычку;
   7. При условии введенного названия привычки предоставлять пользователю возможность создать привычку;
7. **Экран настроек:**
   1. Возможность перейти в профиль;
   2. Возможность сменить язык приложения;
   3. Возможность сменить цветовую тему приложения;
   4. Возможность перейти в раздел «помощь» для ознакомления с функционалом приложения;
   5. Возможность перейти в раздел «пользовательское соглашение»;
   6. Возможность перейти в раздел «информация о приложении»;
   7. Возможность перейти по ссылке в официальный телеграмм-канал приложения;
   8. Возможность совершить выход из аккаунта;
8. **Экран профиля:**
   1. Отображение имени текущего пользователя;
   2. Возможность выбрать раздел профиля: статистика или друзья;
   3. Отображение друзей в разделе «друзья»;
   4. Возможность перейти в профиль друга при нажатии на него;
   5. Отображение кнопки «добавить друга» в разделе «друзья», при нажатии на которую происходит переход на экран добавления друга;
   6. Отображение skeleton-loader’а вместо друзей до тех пор, пока данные о друзьях не загрузятся с сервера;
   7. Возможность удаления друзей при помощи свайпа (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)) в левую часть экрана;
9. **Экран выполнения привычки:**
   1. Возможность удалить привычку;
   2. Возможность перейти на экран изменения привычки;
   3. Отображение актуального времени выполнения привычки;
   4. Возможность запустить таймер;
   5. Возможность остановить таймер;
   6. Отображение имени выбранной привычки;
10. **Экран изменения привычки:**
    1. То же, что предлагает экран добавления привычки, но изначальные отображаемые данные должны соответствовать данным привычки;
11. **Экран друзей:**
    1. Отображение кнопки «добавить в друзья», при нажатии на которую происходит переход на экран добавления друга;
    2. Отображение друзей;
    3. Отображение skeleton-loader’а вместо друзей до тех пор, пока данные о друзьях не загрузятся с сервера;
    4. Возможность перейти в профиль друга при нажатии на него;
    5. Возможность удаления друзей при помощи свайпа в левую часть экрана;
12. **Экран профиля друга:**
    1. Отображение имени друга;
13. **Экран добавления пользователя в друзья:**
    1. Возможность ввести адрес электронной почты;
    2. Проводить проверку корректности введенного адреса электронной почты при помощи регулярного выражения;
    3. Вне зависимости от того, привязан ли к введенному адресу электронной почты аккаунт, уведомлять пользователя об успешном добавлении пользователя в друзья и возвращать пользователя на предыдущий экран;
14. **Экран аналитики:**
    1. Отображение столбчатой диаграммы с семью колонками, соответствующими семи последним дням (включая текущий), каждая из которых отображает количество выполненных привычек в тот или иной день;
    2. Отображение статистики для каждой привычки в виде семи последних дней, когда привычка была выполнена/не выполнена;
    3. Отображение skeleton-loader’а вместо аналитики по привычкам до тех пор, пока данные о друзьях не загрузятся с сервера;

## 3.2. Описание алгоритма функционирования программы

### 3.2.1. Описание архитектуры приложения

Выбор архитектуры приложения – одно из ключевых решений, которое во многом определяет удобство разработки, поддерживаемость и тестопригодность кода. В процессе изучения и сравнения различных архитектур (MVC [[14]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), MVVM [[15]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), MVP [[16]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), VIPER [[17]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), Clean Swift (VIP) [[18]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), TCA [[19]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ) и их модификации) я остановился на SVIP [[20]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), видоизменив ее, опираясь на статью «Оптимальный архитектурный шаблон iOS-приложения» [[20]](#_5._ИСТОЧНИКИ,_ИСПОЛЬЗОВАННЫЕ), и применил ее в проекте, используя *фреймворк UIKit* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)).

Прежде чем принять решение, было важно определить, какие именно критерии и задачи наиболее критичны для моего проекта. Таким образом, были выявлены следующие требования:

* + - Крайне важно было добиться разделения ответственности между слоями приложения. Чем яснее структурирован код, тем легче его поддерживать и расширять, а также появляется возможность тестирования. Таким образом было выявлено требование хорошей декомпозиции.
    - Архитектура должна быть достаточно понятной, чтобы в ней было легко ориентироваться, но при этом не должна быть чрезмерно сложной. Таким образом, было выявлено требование удобство разработки в одиночку.
    - Написание тестов – первостепенная задача, позволяющая понимать, что расширяя функционал – не ломается написанный ранее. Также это возможность быть уверенным в том, что программа имеет именно то поведение, которое закладывает в нее разработчик. Таким образом было выявлено требование хорошей тестируемости приложения.
    - Так как приложение использует UIKit, то хотелось найти подход, который хорошо подходит к этому фреймворку, не заставляя переходить на SwiftUI. Таким образом было выявлено требование ориентации на UIKit.

Именно SVIP совмещает в себе все вышеназванные преимущества, так как она не только забирает все лучшее у VIP-подобных архитектур, но и не является достаточно легкой в понимании и поддержке при разработке в одиночку.

Классическая SVIP архитектура предполагает разделение на следующие четыре модуля: ViewController, Interactor, Presenter и Worker. При этом существует отдельный файл, представляющий модели «общения» между слоями. ViewController связан с Interactor при помощи протокольной связи. Interactor связан с Presenter при помощи протокольной связи. При этом View связан с Presenter при помощи непосредственной связи.

ViewController отвечает за отображение интерфейса и обработку пользовательских действий. Содержит ссылку на интерактор, которому передает события пользователя.

Interactor содержит всю бизнес-логику, обрабатывает данные и взаимодействует с воркерами для выполнения сетевых запросов.

Presenter форматирует данные, полученные от Interactor’а, в формат, пригодный для отображения в ViewController.

Worker выполняет вспомогательные задачи, но в основном выполняет сетевые запросы. В основном помогает разгрузить интерактор, выделив из него тяжеловесную логику.

Для инициализации и связывания этих компонентов создается конфигуратор, который создает экземпляры каждого компонента и устанавливает между ними необходимые связи. Таким образом, каждый экран приложения требует семь файлов. Но я пришел к выводу, что эффективнее написать общий сетевой слой, который позволяет выполнять запросы в сеть в едином формате. Подробнее об этом написано в пункте 3.2.2. Это позволило уменьшить количество файлов, а также улучшить читаемость кода, так как логика обращения в сеть не дублируется в каждом воркере, а инкапсулируется в отдельном классе, экземпляр которого создает интерактор при необходимости. Классическая SVIP архитектура имеет следующую схему:

A diagram of a network

Description automatically generated

Рис. 1: Схема классической SVIP архитектуры

Причины отказа от каждой из архитектур:

* MVC: классический подход, рекомендуемый Apple. Но с увеличением функционала ViewController становится *божественным объектом* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)), где тесно переплетается бизнес-логика, отображение и навигация. Кроме того, такую архитектуру невозможно тестировать.
* MVP: в отличие от MVC успешно выносит логику в Presenter, но остается вопрос где хранить состояние, что остается на усмотрение разработчика. Нет единой концепции, легко запутаться.
* MVVM: отличная архитектура для разработки на SwiftUI, но с UIKit требует дополнительных слоев.
* VIPER: одна из лучших архитектур, однако для среднего по размерам проекта выглядит громоздко, требует много протоколов.
* Clean Swift (VIP): Слишком формализована, что избыточно в проекте.
* TCA: как и MVVM – ориентирован на SwiftUI.

В общей сложности было написано более десяти тысяч строк кода.

### 3.2.2. Описание сетевого слоя

В условиях современного мобильного приложения часто возникает необходимость в качественной и масштабируемой реализации сетевого слоя. Задача состояла в том, чтобы создать единый модуль отправки запросов к серверу, который умеет работать со всеми типами HTTP-запросов. Это требовалось для снижения дублирования кода, повышения тестируемости и соответствия принципам проектирования. Использование отдельных воркеров для каждого экрана хоть и является стандартной практикой, но приводит к дублированию кода и усложнению написания unit-тестов, поскольку каждый компонент имел жесткую связь с низкоуровневыми сетевыми запросами.

Таким образом, существовало несколько вариантов решения проблемы:

1. Классический подход с воркерами для каждого экрана, где каждая бизнес-логика имела собственного воркера, отвечающего за формирование запроса, обработку ответа и декодирования данных. Но при росте количества экранов быстро возникают дублирования и трудности с поддержкой;
2. Унифицированный API сервис (выбранный мной), который представляет из себя интерфейс для отправки запросов в едином формате, при этом поддерживая все HTTP-методы. Этот сервис внедряется в интерактор через *dependency injection* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)), что позволяет интегрировать его как абстракцию через протокол.
3. Гибридный подход, совмещающий два вышеуказанных подхода. Главный недостаток – усложнение архитектуры и потеря консистентности кода, ведь не все Interactor’ы требуют особой реализации сетевого слоя.

Из всех вариантов мной был выбран второй – унифицированный API-сервис.

У такого подхода есть ряд преимуществ:

1. Централизованный подход к работе с сетью:
   1. Весь код, связанный с формированием URL, настройкой заголовков, кодированием/декодированием данных и обработкой ошибок располагается в одном файле. Это помогает избежать дублирования кода, так как не создается однотипная логика обращения в сеть.
   2. Все HTTP-запросы проходят через одну реализацию, что обеспечивает консистентное поведение и единообразную обработку ошибок.
2. Соблюдение принципов SOLID:
   1. Сервис внедряется в Interactor через инициализатор, к которому имеет доступ файл-конфигуратор. Это гарантирует, что Interactor зависит не от конкретной реализации, а от абстракции (протокола). Это облегчает подмену реализации для проведения unit-тестирования.
   2. Бизнес-логика и логика обращения в сеть четко разделены, что удовлетворяет принципу SRP.
3. Повышение тестируемости:
   1. При использовании протокола для определения контракта реализации сетевого слоя, можно подменить реальную реализацию на *Mock* или *Stub* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)), что упрощает написание unit-тестов для Interactor’ов.
   2. Каждый Interactor не содержит жестких зависимостей от конкретной реализации ни сетевого слоя, ни Presenter’а, что позволяет полностью изолировать тестируемый модуль от внешних факторов.
4. Упрощение поддержки и масштабируемости:
   1. Одним движением можно повлиять на все сетевые взаимодействия в приложении: смена способа обработки ошибок, возможность изменить способ кодирования/декодирования данных, добавить новый функционал и т.д.). Если бы вместо этого у каждого экрана был свой воркер – это потребовало бы изменять каждый воркер, что вызывает трудности по мере написания кода.
   2. Впоследствии можно с легкостью реализовать дополнительные методы или расширения внутри одного класса.
5. Улучшение архитектуры:
   1. Единый API-сервис позволяет повысить читаемость кода и снижает вероятность ошибок, связанных с несовместимостью различных реализаций. Программист точно знает какую логику ожидать от каждого обращения в сеть.
   2. Уменьшается объем ненужного кода, что облегчает рефакторинг и сопровождение проекта.
6. Решение потенциальных проблемы:
   1. Если для некоторых экранов требуется особая логика взаимодействия с сетью – это можно решить несколькими путями. Наименее подходящий – расширить функционал API-сервиса. Второй – создать новый API-сервис, унаследовать его от первого и расширить функционал. Третий – добавить привычный воркер и работать с ним. Второй вариант является наилучшим, так как сохраняет абсолютно все преимущества API-сервиса и не нарушает *LSP* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)).
   2. Появление слишком широкой функциональности можно решить путем декомпозиции каждого модуля на логические подсекции. На данный момент проблема решена путем инкапсулирования логики обращения в сеть приватным методом, в то время как его вызывают все остальные публичные методы API-сервиса для отправки или получения данных.
7. Использование обобщений:
   1. Все методы API-сервиса умеют работать с обобщениями, что позволяет

В проекте реализация сетевого слоя находится в файле Networking, где протокол APIServiceProtocol определяет основные способы отправки и получения данных, а APIService реализует их, добавляя инкапсулированную логику отправки данных.

1. Реализация отдельных методов для GET и остальных запросов:
   1. API сервис предлагает метод get для GET запросов и метод send для всех остальных типов. Разница в том, что в get невозможно передать тело запроса, при отсутствии такой логики могут происходить непредвиденные ошибки, когда пользователь пытается отправить GET запрос с телом.

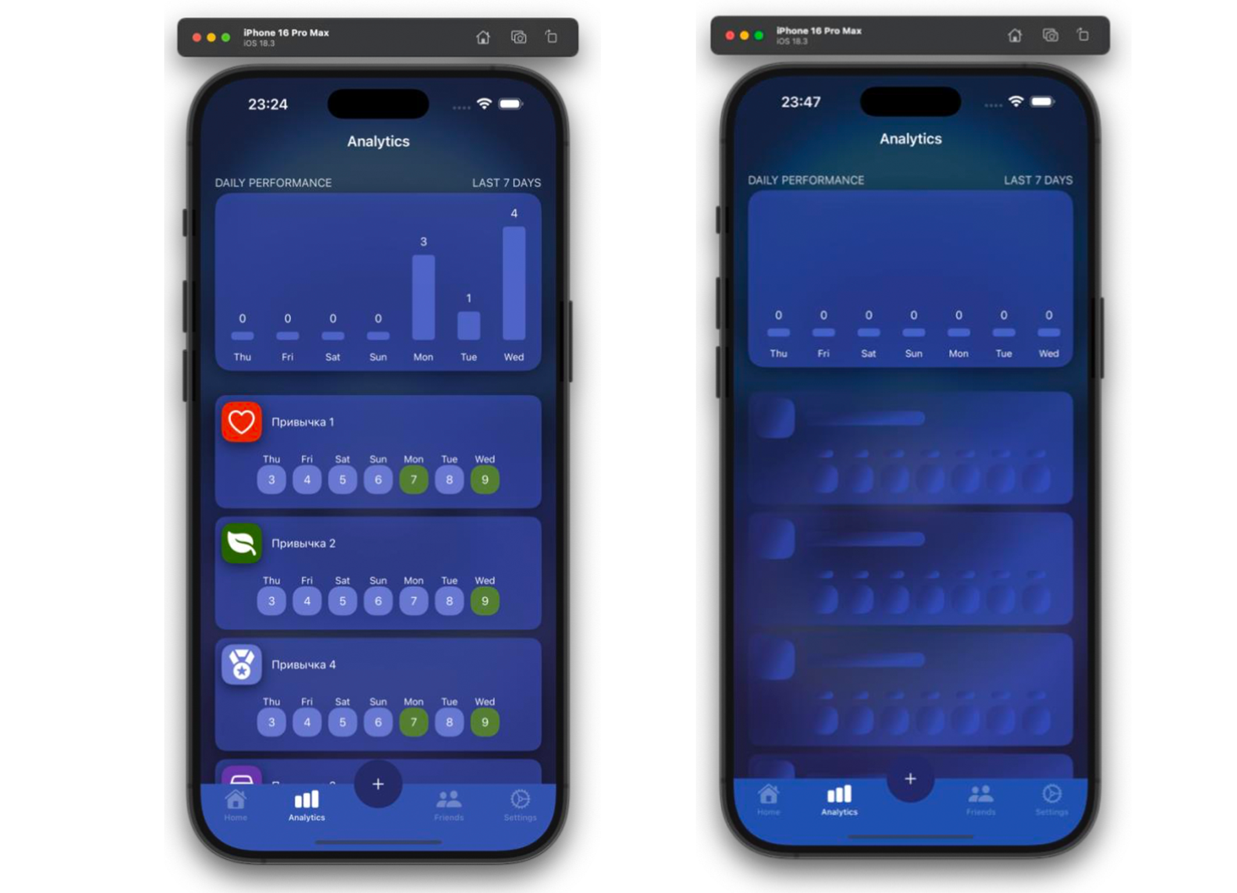
Сравнение с классическим подходом (воркерами):

1. Преимущества:
   1. Унификация кода;
   2. Поддерживаемость и гибкость;
   3. Улучшенная тестируемость;
   4. Соблюдение DIP;
2. Недостатки:
   1. Потенциальное усложнение и разрастание по мере увеличения функциональных требований к программе;
   2. Единый компонент «точки отказа», т.е. если в сетевом слое что-то ломается – это отражается на всех экранах. Однако тестирование позволит решить эту проблему.

### 3.2.3. Описание реализации skeleton-loader’ов

Мобильное приложение должно быть отзывчивым и «общаться» с пользователем, реагируя на все действия, иначе пользователь может остаться недовольным опытом использования. Один из таких примеров – отображение skeleton-loader’ов вместо некоторых данных до тех пор, пока реальные данные не будут в состоянии отобразиться (например, пока не пришел ответ от сервера). Иными словами, они сигнализируют пользователю о том, что данные находятся в процессе загрузки. Благодаря им создается «живость» интерфейса, что снижает восприятие задержек интерфейса у пользователя. В результате создается впечатление быстрой и отзывчивой работы приложения. К достоинствам skeleton-loader’ов можно отнести фиксирование пространства будущего контента, что обеспечивает стабильный макет приложения и снижает вероятность скачков при смене состоянии ячейки после загрузки данных.

В моем приложении это выглядит следующим образом:



Подобные «мерцающие» заглушки присутствуют на всех экранах, где происходит загрузка данных с сервера, включая экран привычек и экран друзей (как основной, так и в профиле). После загрузки данных skeleton-loader’ы заменяются на реальные данные.

В основе анимации лежит кастомный (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)) элемент интерфейса (далее шиммер), унаследованный от UIView. Шиммер имеет два метода: startShimmer и stopShimmer, которые запускают и останавливают анимацию соответственно. При желании можно изменить цвет, скорость, угол наклона анимации, а также задать все остальные внешние параметры. Использование шиммера как основу всех skeleton-loader’ов позволяет добиться консистентности в интерфейсе. Внутри шиммера реализована градиентная настройка цвета при помощи констант. Стоит отметить, что шиммер использует встроенный CAGradientLayer и CABasicAnimation. Такой подход задействует аппаратное ускорение анимаций, что позволяет добиться максимальной плавности анимации даже на устройстве с ограниченными ресурсами. Это минимизирует нагрузку на главный поток приложения, поддерживая отзывчивость всех остальных элементов интерфейса.

### 3.2.4. Описание созданных кастомных элементов интерфейса

В процессе создания приложения так или иначе появлялась потребность в создании собственных элементов интерфейса, которые имеют уникальное поведение, которое требуется заложить в компонент. И такие компоненты могут требоваться в разных частях приложения, и чтобы не дописывать новую логику поверх существующих компонентов в каждом таком месте, в приложение было добавлено несколько уникальных компонентов интерфейса, имеющих уникальное поведение.

EmailCodeTextField: компонент для ввода данных с клавиатуры, основанный на встроенном компоненте UITextField. Используется на экране ввода кода с почты. Основное отличие заключается в том, что этот компонент работает в паре с такими же компонентами следующим образом: если пользовательский курсор (фокус) находится в начале текста, и он нажимает кнопку «backspace» (назад), то текстовое поле очищается и фокус пользователя переносится на предыдущую ячейку (при наличии); а также переводит фокус на следующее поле ввода текста после ввода любой цифры (другие символы ввести нельзя). Этот компонент позволяет добиться максимально комфортного ввода четырехзначного кода.



ExtendedHitButton: компонент, наследующий функционал стандартной кнопки UIButton, за тем исключением, что расширяет область касания на заданную константу во все стороны. Этот эффект добивается путем переопределения метода point(inside:with:) и использования отрицательных отступов. Это актуально для маленьких кнопок, чей размер нельзя увеличивать по эстетическим соображения дизайнера проекта, но которые слишком маленькие для комфортного нажатия пользователем. Такая практика используется повсеместно практически в каждом мобильном приложении.

PasswordTextField: компонент для ввода пароля, наследующий функционал встроенного UIPasswordTextField, за исключением доработки поведения скрытия пароля. При смене состояния отображения пароля (скрытый/не скрытый) компонент гарантирует корректное отображение пароля (замена точек на настоящие символы). Компонент создан в целях улучшения пользовательского опыта, так как встроенный компонент для ввода пароля стирает весь введенный пароль после открытия ранее введенного пароля, что приводит к негативной реакции со стороны пользователя.

### 3.2.5. Описание созданных менеджеров данных

Также в архитектуре приложения немаловажную роль играют менеджеры данных – специальные компоненты, отвечающие за работу с данными. Одним из наиболее эффективных методов реализации менеджеров является паттерн Singleton (одиночка), который гарантирует единственность существования экземпляра класса и глобальный уровень доступа без необходимости создавать экземпляр класса «на месте».

В моем приложении реализовано несколько менеджеров для работы с различными данными:

MotivationsManager: менеджер отвечает за генерацию мотивационных сообщений в зависимости от переданного процента выполнения привычек.

DateManager: менеджер отвечает за предоставление даты и времени в подходящем для отображения формате, а также отвечает за локализацию времени в приложении, генерацию дат и другую связанную с этим функиональность. Главное преимущество: единообразность возвращаемых дат, что гарантирует корректное сравнение дат как внутри приложения (клиента), так и внутри базы данных на сервере.

LocalizationManager: менеджер, отвечающий за локализацию строк в приложении, выбирая подходящий языковой пакет в приложении в зависимости от системных настроек. Упрощает поддержку приложения на нескольких языках, так как при входе приложение знает какой язык установлен у пользователя и позволяет отобразить корректный язык, а также помочь DateManager’у выбрать подходящий часовой пояс.

UserDefaultsManager: менеджер для работы с пользовательскими настройками и данными. Инкапсулирует логику сохранения, чистки и чтения данных из локального хранилища.

CoreManager: менеджер для работы с CoreData, обеспечивающий единую точку доступа к персистентному хранилищу приложения. Позволяет избежать дублирования кода, а также совмещает в себе вышеназванные преимущества других менеджеров: позволяет сохранять, читать и очищать данные, а также упрощает настройку персистентного хранилища приложения.

### 3.2.6. Описание созданных автотестов (UNIT, UI)

Хорошим тоном любого приложения является наличие автотестов. Эти тесты не только гарантируют корректную логику выполнения программы, но и позволяют изменять и расширять функционал со своевременным обнаружением поломки старой логики. Также тесты дают уверенность при рефакторинге кода, ведь можно сразу же узнать, что что-то сломалось.

В приложении реализовано полное покрытие unit-тестами всех слоев бизнес-логики каждого экрана, а также покрытие тестами сетевого слоя (API-сервиса).

Необходимость покрытия тестами Interactor’а вызвана тем, что это один из ключевых компонентов каждого экрана, выполняющий зачастую нетривиальную логику, в которой легко запутаться. Тестирование Interactor’а позволяет добиться уверенности в стабильности сценариев. Например, покрытый тестами HomeInteractor позволяет сказать, что во всех случаях ответа от сервера приложение корректно обрабатывает сценарий и предпринимает соответствующие действия. Кроме того, так как Interactor являются связующим звеном View и Presenter, тесты гарантируют, что запрос на переход на другой экран действительно передается от первого второму. Выбранная архитектура SVIP подходит для написания тестов. Всю реальную логику представления, форматирования и запросов в сеть можно закрыть заглушками, что изолирует бизнес-логику от всего остального.

Кроме того, покрыты UI-тестами некоторые пользовательские сценарии. Отдельно стоит отметить три теста: возможность войти в аккаунт, возможность выйти из аккаунта и возможность создания привычки. UI-тесты позволяют проверить как пользователь взаимодействует с интерфейсом, симулируя переходы и нажатия. Таким образом, UI-тесты гарантируют возможность выполнения тех или иных функций приложения.

После добавления новой логики и запуска автотестов, которые успешно проходятся, программист может быть полностью уверен, что новая логика не сломала старую, и что пользователь не будет испытывать негативный опыт взаимодействия с приложением.

### 3.2.7. Описание кода и возможности сопровождения

Весь исходный код программы написан согласно лучшим практикам разработки iOS-приложений на фреймворке UIKit. Каждый экран имеет собственное приватное перечисление констант, что не только улучшает читаемость кода, но и выносит все визуальные составляющие в одно место, позволяя при необходимости изменить интерфейс быстро перейти к необходимому компоненту. Кроме того, весь код приложения имеет как обычные комментарии, так и MARK’и, отделяющие блоки классов на соответствующие разделы. Также весь код имеет «говорящие названия»: как классы, так и методы с переменными, что упрощает понимание кода программисту, который хочет изменить поведение приложения.

A screenshot of a computer program

Description automatically generated

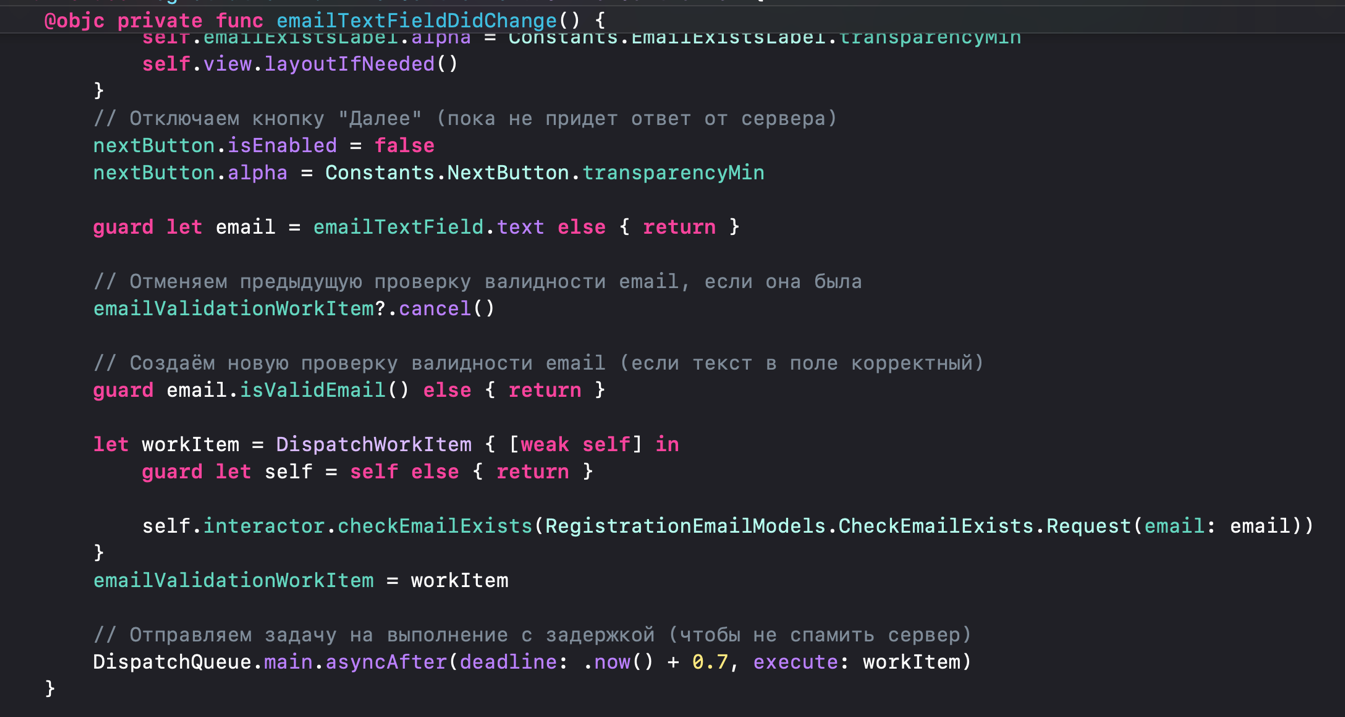
### 3.2.8 Описание надежности и актуальности подходов

В приложении используются современные средства свифт: безопасные опционалы, типобезопасный Codable, отлов ошибок через Result. Благодаря последнему сетевой слой гарантирует, что он либо вернет ответ от сервера, либо подробную ошибку. Архитектура позволила покрыть тестами всю бизнес-логику, что минимизирует возможность появления непредвиденного поведения. Асинхронные вызовы выполняются потокобезопасно.

### 3.2.9 Описание работы с многопоточностью

Многопоточность и работа с асинхронными функциями – распространенная проблема в программировании. Практически каждое мобильное приложение умеет обращаться в сеть, что нельзя реализовать без использования многопоточности – в ином случае интерфейс пользователя будет блокироваться до момента ответа от сервера, что в худших случаях может привести к непредвиденному завершению работы, так как встроенный watchdog в iOS принудительно завершает приложения, которые не отзываются (поток заблокирован, так как ждет ответа от сервера). Проанализировав возможные инструменты многопоточности в Swift я остановился на GCD (grand central dispatch). Кроме него Swift предлагает такие инструменты, как: NSThread, Operation(Queue), а также свежий Swift Concurrency. NSThread является низкоуровневым и не рекомендуется для использования в качестве основного средства многопоточности в Swift. Operation и OperationQueue являются оберткой над GCD, добавляя ненужную моему проекту логику. Swift Concurrency – самый свежий подход к многопоточности в Swift. Он предлагает множество преимуществ: начиная Structured Concurrency, заканчивая отсутствием блокирования потока на момент ожидания ответа (поток, пока дожидается ответа, может взять на выполнение новую задачу), но несмотря на это он имеет много сложностей в реализации, хоть и имеет привычный синтаксис async/await. Мой выбор в сторону GCD был обусловлен его фундаментальностью, а также наличием множества обучающих материалов.   
 При выполнении асинхронных операций могут возникнуть retain cycle – процесс, не позволяющий автоматически освободить занятую память от объектов, которые более не требуются приложению. При использовании GCD были учтены возможности появления retain cycle, поэтому во всех замыканиях, которые захватывают объект по ссылке через self был добавлен блок [weak self], гарантирующий, что замыкание забирает объект по слабой ссылке (которая не учитывается при чистке памяти). На примере загрузки данных с сервера это можно описать так: если экран закрыт до прихода ответа от сервера, замыкание не удержит устаревший Interactor или ViewController и освободит память. Кроме того, все тяжелые задачи были вынесены в фоновый поток для сохранения плавности и отзывчивости интерфейса, при этом обновление интерфейса происходит на главном потоке.

Одним из ключевых достоинств многопоточности в приложении является *debounce* (см. [Перечень терминов](#_ПРИЛОЖЕНИЕ_1)). Лучше всего его можно проследить на экране RegistrationEmailCodeViewController, где debounce защищает сервер от отправки большого количества однотипных и бессмысленных запросов. Когда пользователь начинает вводить почту – на каждое изменение символа срабатывает метод emailTextFieldDidChange. В нем реализовано ожидание: предыдущая задача отправки почты на сервер отменяется и создается новая с задержкой выполнения 0.7 с, которая задается через DispatchQueue.main.asyncAfter(deadline:execution:). Таким образом, запрос на сервер отправляется спустя незаметное для пользователя время, тем не менее убирая ненужные запросы (пока пользователь не ввел почту полностью). Подобный прием часто используется в поисковых строках браузеров. Описанный механизм сильно уменьшает количество запросов: вместо десятка запросов отправляются лишь единицы. Дополнительно, перед отправкой запроса на сервер при помощи регулярных выражений проверяется корректность адреса почты на соответствие стандартам. Это предотвращает возможность перехода на следующий этап при вводе некорректного адреса электронной почты, а также дополнительно снижает нагрузку на сервер.



## 3.3. Описание и обоснование выбора метода организации входных и выходных данных

### 3.3.1. Описание метода организации входных данных и выходных данных:

Входными данными приложения является информация, которую пользователь передает посредством взаимодействия с интерфейсом: адрес электронной почты, четырехзначный код, имя, пароль, название привычки, мотивация к выполнению привычки, цвет иконки привычки, картинка привычки, количество времени, необходимого для выполнения привычки, единицы измерения выполнения привычки, частота привычки и часть дня, в которую планируется выполнять привычку. А также адрес электронной почты пользователя для добавления последнего в друзья.

Выходными данными приложения является информация, отображаемая на интерфейсе.

### 3.3.2. Обоснование метода организации входных и выходных данных:

Выбор метода организации входных и выходных данных был сделан на основании общих принципах удобства использования мобильных приложений.

## 3.4. Описание и обоснование выбора технических и программных средств

### 3.4.1. Описание технических и программных средств

### Приложение можно запустить как на мобильном устройстве iPhone, так и на симуляторе мобильного устройства iPhone с операционной системой iOS 14.0 и выше. Мобильное устройство должно обладать следующими характеристиками:

### Минимальные требования:

* Операционная система iOS версии 14.0 и выше;
* Процессор Apple A11 Bionic и выше;
* Оперативная память не менее 3 Гб;
* Не менее 500 Мб свободного пространства на устройстве;
* Поддержка технологии не менее 4G/5G или Wi-Fi
* Бесперебойный доступ к сети интернет скоростью не менее 0.2 Мбит/сек
* Разрешение экрана не менее 1792 x 828 5.8"

Рекомендуемые требования:

* Операционная система iOS версии 14.0 и выше;
* Процессор Apple A13 Bionic и выше;
* Оперативная память 4 Гб;
* Не менее 500 Мб свободного пространства на устройстве;
* Поддержка технологии 4G/5G или Wi-Fi
* Бесперебойный доступ к сети интернет скоростью не менее 0.2 Мбит/сек
* Разрешение экрана 2426×1125 6.1"

Для работы приложения требуется запустить серверную часть приложения на любом удобном сервере. Сервер должен обладать следующими характеристиками:

Минимальные требования:

* 8 ГБ оперативной памяти (ОЗУ) или больше
* 32 ГБ постоянной памяти (ПЗУ) или больше
* Беспрерывный доступ к сети интернет со скоростью не менее 2 МБ/c

Рекомендуемые требования:

* 32 ГБ оперативной памяти (ОЗУ) или больше
* 128 ГБ постоянной памяти (ПЗУ) или больше
* Беспрерывный доступ к сети интернет со скоростью не менее 10 МБ/с

Используемые программные средства: Swift, UIKit, Python, FastAPI, PostgreSQL, Docker, а также среды разработки xcode и pycharm.

### 3.4.2. Обоснование выбора используемых технических и программных средств

Выбор технических средств был обусловлен анализом использования памяти устройства при помощи инструментов, предоставляемых средой разработки xcode.

Выбор программных средств обусловлен следующим:

Выбор языка программирования Swift обусловлен тем, что Swift является единственным языком, предоставляемым для нативной разработки приложений на iOS.

Выбора фреймворка – одна из главных задач при разработке мобильного приложения. И выбор фреймворка UIKit для разработки приложения был обусловен его значительными преимуществами перед альтернативным фреймворком SwiftUI, потому что в приложении присутствуют сложные анимации. UIKit имеет следующий ряд достоинств:

* Предоставляет комплексные инструменты для реализации и детального управления анимациями. Главные особенности: поддержка явных транзакций и переходов (UIViewControllerAnimatedTransitioning и UIViewControllerTransitioningDelegate), позволяющие управлять анимациями перехода между экранами; UIKit обладает огромным спектром обработки пользовательских жестов при помощи UIGestureRecognizer и его производных классов, а также имеет низкоуровневые методы обработки событий (touchesBegan, touchesMoves, touchesEnded).
* UIKit является фундаментальным фреймворком для iOS разработки. Это фреймворк до сих пор активно используется, несмотря на новинку в лице SwiftUI. Это также объясняется тем, что SwiftUI является оберткой над UIKit, а значит практически все, что можно сделать на SwiftUI, можно сделать и на UIKit, но при этом иметь гораздо больше возможностей. Также UIKit имеет гораздо больше обучающих материалов и официальную документацию от Apple.

Выбор языка программирования Python и фреймворка FastAPI обусловлен тем, Python является простым и удобным языком для написания серверной части приложения, предоставляющий огромный выбор фреймворков, среди которых был выбран FastAPI, преимуществами которого является следующее:

1. Возможность интеграции с pydantic для обеспечения корректности данных;
2. Обрабатывает тысячи запросов одновременно;
3. Автоматически генерирует swagger-документацию;
4. Входит в десятку самых быстрых API фреймворков языка Python;
5. Разделяет логику на независимые сервисы;

Выбор платформы контейнеризации Docker обусловлен следующим:

1. Изоляция окружения, что предотвращает конфликты версий;
2. Оркестрация через Docker Compose помогает легко добавлять новый функционал;
3. Каждый новый контейнер вне зависимости от платформы гарантирует индентичность работы;

Выбор базы данных PostgreSQL обусловлен ее быстродействием, большим количеством обучающих материалов, функциональностью и возможностью запустить как отдельный контейнер при помощи Docker, изолируя базу данных от внешних факторов.

# 4. ОЖИДАЕМЫЕ ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

## 4.1. Ориентировочная экономическая эффективность

В рамках данного проекта расчет экономической эффективности не предусмотрен.

## 4.2. Предполагаемая потребность

Мобильное приложение «Doordie» предназначено для пользователей, заинтересованных в отслеживании и формировании привычек. Целевая аудитория – заинтересованные в личном развитии люди вне зависимости от возраста.

## 4.3. Экономические преимущества разработки по сравнению с отечественными и зарубежными аналогами

«Doordie» имеет достаточный основной и дополнительный функционал, чтобы конкурировать на рынке похожих приложений. Сравнение мобильного приложения «Doordie» с аналогами представлено в документе «Функциональный анализ конкурентов и анализ бизнес- процессов» от 25.03.25 (см. [Приложение 2](#_ПРИЛОЖЕНИЕ_2))

# 5. ИСТОЧНИКИ, ИСПОЛЬЗОВАННЫЕ ПРИ РАЗРАБОТКЕ

1. ГОСТ 19.101-77: Виды программ и программных документов. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

2. ГОСТ 19.102-77: Стадии разработки. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

3. ГОСТ 19.103-77: Обозначения программ и программных документов. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

4. ГОСТ 19.104-78: Основные надписи. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

5. ГОСТ 19.105-78: Общие требования к программным документам. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

6. ГОСТ 19.106-78: Требования к программным документам, выполненным печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

7. ГОСТ 19.201-78: Техническое задание. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

8. ГОСТ 19.301-79: Программа и методика испытаний. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

9. ГОСТ 19.401-78: Текст программы. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

10. ГОСТ 19.404-79: Пояснительная записка. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

11. ГОСТ 19.505-79: Руководство оператора. Требования к содержанию и оформлению. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

12. ГОСТ 19.603-78: Общие правила внесения изменений. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

13. ГОСТ 19.604-78: Правила внесения изменений в программные документы, выполненные печатным способом. // Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.

14. «MVC». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: <https://ru.wikipedia.org/wiki/Model-View-Controller>

15. «MVVM». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: <https://ru.wikipedia.org/wiki/Model-View-ViewModel>

16. «MVP». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: https://ru.wikipedia.org/wiki/Model-View-Presenter

17. «VIPER». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: <https://medium.com/@pinarkocak/understanding-viper-pattern-619fa9a0b1f1>

18. «Clean Swift( VIP)». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: https://www.netguru.com/blog/clean-swift-ios-architecture-pattern

19. «TCA». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: https://github.com/pointfreeco/swift-composable-architecture

20. «SVIP». Просмотрено: 20 декабря 2024 г. [Онлайн]. Доступно на: https://habr.com/ru/companies/wildberries/articles/798275/

# ПРИЛОЖЕНИЕ 1

**ПЕРЕЧЕНЬ ТЕРМИНОВ**

**Регулярное выражение -** формальный язык, используемый в компьютерных программах, работающих с текстом, для поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов.

**Skeleton-loader** - шаблон проектирования пользовательского интерфейса, используемый для улучшения взаимодействия с пользователем во время загрузки содержимого в веб и мобильных приложениях.

**Свайп** - специальный жест, когда вы кладете палец на экран смартфона или планшета и ведете его в нужном направлении по экрану.

**Фреймворк -** программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

**UIKit** – среда разработки и приложений и набор инструментов графического пользовательского интерфейса от Apple Inc.

**Mock** – техника в программировании, при которой создаются искусственные объекты или функции, которые имитируют реальные.

**Stub** - объекты, также называемые заглушками, которые возвращают заранее определенные значения на определенные входные данные.

**LSP** – (от англ. Liskov Substituion Principle) - функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом

**Dependency** **Injection** - процесс предоставления внешней зависимости программному компоненту.

**Кастомный** - индивидуализация продукции под заказы конкретных потребителей путём внесения конструктивных или дизайнерских изменений

**Debounce** - Прием debounce гарантирует, что функция не будет вызвана снова, пока не пройдет определенное количество времени без ее вызова

# ПРИЛОЖЕНИЕ 2

**Функциональный анализ конкурентов и анализ бизнес-процессов**

A table with text on it

Description automatically generated

A table with text on it

Description automatically generated



A blank sheet of paper

Description automatically generated with medium confidence