Министерство образования Республики Беларусь

Учреждение образования «Белорусскией государственный университет информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Дисциплина «Объектно-ориентированное программирование»

| «К ЗАЩИТЕ ДОПУСТИТЬ» |
| --- |
| Руководитель курсовой работы  магистр техн. наук, ст. преподаватель |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А.П. Горбач |
| \_\_\_.\_\_\_. 20\_\_\_ |

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

на тему

**МОБИЛЬНОЕ ПРИЛОЖЕНИЕ ДЛЯ КОНТРОЛЯ ПРИЕМА ЛЕКАРСТВЕННЫХ СРЕДСТВ**

БГУИР КР 1-40 05 01 006 ПЗ

|  |  | Выполнил студент группы 914302  ВОРОБЕЙ Дарья Антоновна  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |
| --- | --- | --- |
|  |  | Курсовой проект представлен на проверку \_\_\_.\_\_\_. 20\_\_\_ |
|  |  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |

**РЕФЕРАТ**

БГУИР КР 1-40 05 01 006 ПЗ

МОБИЛЬНОЕ ПРИЛОЖЕНИЕ ДЛЯ КОНТРОЛЯ ПРИЁМА ЛЕКАРСТВЕННЫХ СРЕДСТВ / Д.А. Воробей. – Минск: БГУИР, 2021. – 70 с., чертежей(плакатов) – 4 л. формата A3

МОБИЛЬНАЯ РАЗРАБОТКА, IOS, SWIFT, SWIFTUI, VAPOR, SQLite, КЛИЕНТ-СЕРВЕР, ШАБЛОНЫ ПРОЕКТИРОВАНИЯ, XCODE, ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ .

Предмет: мобильное приложение для платформы IOS.

Объект: методы разработки клиент-серверного мобильного приложения на языке Swift.

Цель: разработка мобильного приложения для отслеживания приема лекарственных средств с выполнением функционала на базе шаблона проектирования Model-View-ViewModel с использованием языка Swift.

Методология проведения работы: в процессе решения поставленных задач использованы методики проектирования графического пользовательского интерфейса и алгоритмы, описывающие логику работы приложения в рамках клиент-серверной архитектуры.

Результаты работы: выполнен анализ литературных источников по теме; реализованы алгоритмы генерирования и корреляционного анализа сигналов; результаты корреляционного анализа представлены пользователю в форме, пригодной для последующей интерпретации.

Программа предназначена для отслеживания приема лекарственных средств пользователем в соответствии с введенными пользователем данными о медикаментах.

Область применения результатов: мобильная разработка, мобильные информационные системы.

СОДЕРЖАНИЕ

[В](#_fiy0bogh5trs)ведение…………………………………………………………………………...[6](#_fiy0bogh5trs)

[1 О](#_q3mgo7huf9qs)писание проекта……………………………………………………………....[8](#_q3mgo7huf9qs)

[1.1 Общие сведения о проекте………………………………………………..8](#_4m8oqty97d6o)

[1.2 Клиентская часть…………………………………………………………..8](#_1lic9mizzkdj)

[1.3 Серверная часть…………………………………………………………..15](#_6yug9ljdzh8i)

[2 О](#_pm38s7kf04a7)боснование выбора технологий…………………………………………….[19](#_pm38s7kf04a7)

[2.1Технологии программирования, используемые для решения поставленных задач…………………………………………………………..19](#_nmiumxnwefkf)

[2.2 Реализация объектно-ориентированных технологий программирования](#_cya6g1xgfogp) ………………………………………………………………………………....22

[3 И](#_hea331i6oaa0)нструментарий……………………………………………………………....[24](#_hea331i6oaa0)

[3.1 Обоснование используемых инструментов…………………………….24](#_vwwm87dz2gsl)

[3.2 Использование системы контроля версий GIT………………………....27](#_3yhkmiby2e9d)

[4 А](#_73stug72o7oh)рхитектурный шаблон проектирования MVVM…………………………..[29](#_73stug72o7oh)

[4.1 Разработка схемы алгоритма…………………………………………….29](#_66ppre7a1i1k)

[4.2 Разработка диаграммы последовательности…………………………....31](#_3slwcbtxfixp)

[4.3 Разработка диаграммы состояний……………………………………….33](#_k8iu6tcniuts)

[5 Ш](#_6n7awix7fb5x)аблон проектирования практических задач……………………………....[3](#_6n7awix7fb5x)6

[5.1 Порождающие паттерны………………………………………………....3](#_awtfg5pq4bcx)6

[5.2 Структурные паттерны…………………………………………………..38](#_a44til21ee3i)

[5.3 Поведенческие паттерны………………………………………………...40](#_tlldul52pknj)

[З](#_vbhmo0wa0z4k)аключение……………………………………………………………………….[4](#_vbhmo0wa0z4k)4

Список использованных источников …………………………………………..45

[П](#_8ei33hcdidn4)риложение А(обязательное) Проверка на антиплагиат……………………...[4](#_8ei33hcdidn4)6

[Приложение Б](#_9ofzl17m51w4)(обязательное) Листинг кода…………………………………....[4](#_9ofzl17m51w4)7

[Приложение B](#_e34kpqaodngo)(обязательное) Ведомость документов………………………...70

# ВВЕДЕНИЕ

В настоящее время на рынке мобильных приложений представлено множество продуктов, помогающих пользователям решать огромное количество повседневных задач. Особое внимание можно уделить приложениям медицинского назначения, к которым также относятся программы, помогающие пользователям планировать и контролировать прием лекарственных средств в соответствии с установленными дозировками и расписанием. Преимуществом подобных программным продуктов является наличие возможностей для усовершенствования и расширения мобильной системы на всем протяжении эксплуатации. Реализация подобных систем возможна в контексте приложения для операционной системы IOS.

Операционная система IOS была выпущена компанией Apple в 2007 году, и на сегодняшний день устройствами на базе данной операционной системы пользуются более миллиарда человек по всему миру, что создает множество возможностей на рынке мобильных приложений для рассматриваемой платформы. Нативными языками разработки для IOS являются ObjectiveC и Swift. ObjectiveC – компилируемый объектно-ориентированный язык программирования корпорации Apple, построенный на основе языка Си и парадигм Smalltalk, представленный компанией Apple в 1983 году. В 2014 году компания представила Swift— открытый мультипарадигмальный компилируемый язык программирования общего назначения, являющийся надстройкой над ObjectiveC и получивший широкое применение в среде мобильной разработки.

Для наилучшей работы приложения применяется взаимодействие с внешними программными продуктами и системами, такими как электронная почта, камера мобильного устройства, облачная база данных, библиотеки с открытым исходным кодом.

**Актуальность данной работы** состоит в разработке мобильного приложения, способного решать важную проблему в жизни людей, связанную со сложностью своевременного приема лекарственных средств. В связи с этим разработка приложения с удобным графическим пользовательским интерфейсом и широким функционалом для качественного выполнения своего предназначения, которое может быть использовано широким кругом пользователей, имеет очевидные перспективы практического применения как для самостоятельного использования, так и для внедрения в более сложные программные системы.

**Целью** курсовой работы является создание нативного IOS приложения для планирования и отслеживания приема лекарственных средств посредством использования современных технологий дизайна и программирования с применением программных средств. Функционал приложения призван соответствовать пожеланиям потенциальных пользователей и обеспечивать эффективное взаимодействие через пользовательский интерфейс.

**Задачей** данной курсовой работы является выполнение следующих этапов работы:

1 Изучение особенностей и проектирование архитектуры и интерфейса мобильного приложения для платформы IOS при помощи шаблона проектирования Model-View-ViewModel.

2 Реализация алгоритмов работы клиент-серверной архитектуры приложения.

3 Описание систем баз данных для авторизации и аутентификации пользователей и хранения данных о принимаемых каждым пользователем медикаментах в безопасной и приемлемой для обработки форме.

4 Изучение способов поиска и передачи данных между составными частями приложения.

Курсовой проект выполнен самостоятельно и проверен в системе «Антиплагиат». Процент оригинальности проекта составляет 97,86%. Скриншот результатов проверки работы приведен в Приложении А.

# 1 ОПИСАНИЕ ПРОЕКТА

## 1.1 Общие сведения о проекте

Разработанное приложение PillQuipo предназначено для поиска, добавления и настройки параметров употребляемых медикаментов с последующей возможностью редактирования и удаления записей, а также удобного отслеживания контроля приема лекарственных средств в соответствии с заданными инструкциями. Приложение имеет расширенный функционал, выраженный в надежной системе регистрации и авторизации, возможности осуществления различных способов поиска и добавления медикаментов и так далее. Дополнительно реализовано взаимодействие приложения с другими сервисами IOS-устройства, такими как Центр уведомлений и камера. В приложении реализуется клиент-серверная архитектура, описанная с использованием языка нативной разработки Swift.

## 1.2 Клиентская часть

Клиентская часть приложения используется в качестве основного средства для взаимодействия программного продукта с пользователем. Для качественной работы приложения необходимо реализовать не только понятный пользователю графический интерфейс, но и обеспечить высокую скорость и точность работы приложения, реакцию интерфейса на работу с данными со стороны пользователя и сервера. Также в рамках задач приложения необходимо обеспечить приложению права на взаимодействие с внешними службами устройства (центр уведомлений, календарь, камера) (рисунок 1.1):

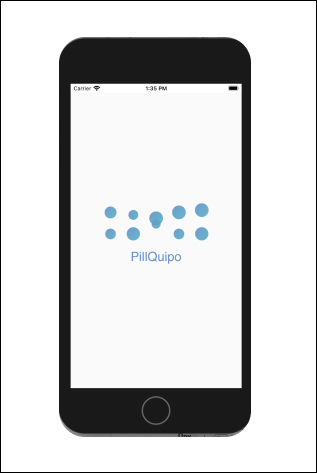


Рисунок 1.1 – Экран входа в приложение

В рамках фреймворка SwiftUI за запуск приложения на устройстве или симуляторе, а также выдачу и отслеживание изменений прав доступа приложения ко внешним службам отвечает комплекс функций, являющихся методами класса Delegate:

class Delegate: NSObject, UIApplicationDelegate{

func application(\_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {

FirebaseApp.configure()

registerForPushNotifications()

return true

}

func registerForPushNotifications() {

UNUserNotificationCenter.current()

.requestAuthorization(options: [.alert, .sound, .badge]) {(granted, error) in

print("Permission granted: \(granted)")

}

}

}

После использования данных методов можно переходить к остальным компонентам программы.

**1.2.1** Регистрация и авторизация**.**

Для начала работы в приложении необходимо выполнить вход в учетную запись или выполнить регистрацию в приложении. Основой механизма регистрации является использование технологий Firebase Authorization – платформы для хранения и управления учетными записями пользователей. В рамках платформы реализован полный функционал для входа в приложение и обработки исключительных ситуаций.

Для реализации авторизации используются объектно-ориентированные технологии. В качестве объектов выступают пользователи, содержащие соответствующие им атрибуты в качестве полей класса ModelData:

@Published var email = ""

@Published var password = ""

@Published var isSignUp = false

@Published var email\_SignUp = ""

@Published var password\_SignUp = ""

@Published var reEnterPassword = ""

@Published var isLinkSend = false

@Published var alert = false

@Published var alertMsg = ""

@Published var key = ""

@AppStorage("log\_Status") var status = false

@Published var isLoading = false

При входе в приложение пользователь может выполнить вход в учетную запись или, при необходимости, создать новую. При входе в существующий аккаунт пользователь вводит адрес электронной почты и соответствующий пароль, и при нахождении введенного сочетания в базе данных пользователю присваивается ключ и выполняется вход в аккаунт. В случае, если пользователь вводит некорректную информацию, на появляются уведомления, сообщающие об ошибке.

Для обработки ситуации, в которой пользователь забывает пароль, существует механизм восстановления доступа к учетной записи. При выборе опции восстановления пароля на введенный пользователем адрес электронной почты, соответствующий учетной записи в приложении, отправляется сообщение от имени приложения, содержащее ссылку для изменения пароля и соответствующие инструкции. При успешном изменении пароля пользователь может выполнить вход в учетную запись через приложение (рисунок 1.2).

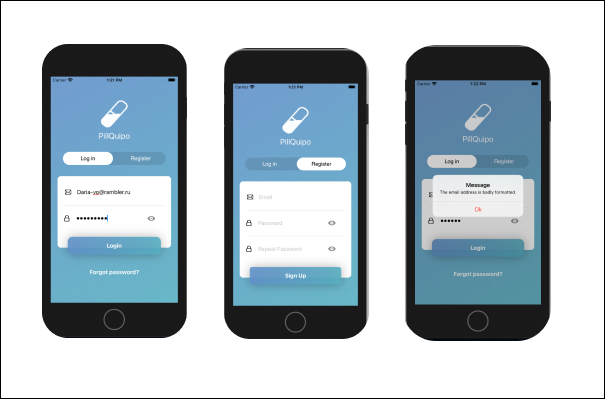


Рисунок 1.2 – Структура пользовательского интерфейса вкладок регистрации и авторизации

Если пользователь впервые использует приложение, ему необходимо выполнить регистрацию посредством ввода электронной почты и пароля. При некорректном вводе на экране появляются соответствующие ошибкам сообщения. Если регистрация проходит успешно, система управления учетными записями сохраняет данные пользователя в виде хэш-строки и для доступа к аккаунту генерирует уникальный ключ, который присваивается пользователю.

**1.2.2** Реализация добавления записейОдной из важнейших частей проекта является реализация функционала добавления медикаментов в список пользователя. В приложение это может быть реализовано тремя способами: поиск в базе данных, сканирование штрих-кода и ввод пользовательских значений.

При использования поиска в базе данных происходит поиск по полностью или частично введенной пользователем строке названия препарата в базе данных через программный интерфейс ресурса Federal Drug Agency. При обнаружении необходимого препарата пользователь при нажатии на соответствующий пункт осуществляет ввод дополнительных параметров посредством взаимодействия с компонентами интерфейса. Успешное добавление будет реализовано в случае заполнения всех полей соответствующими значениями и сохранено в виде соответствующей модели в локальной базе данных.

Для использования сканирования штрих-кода для поиска медикамента необходимо навести камеру на соответствующее изображение, в результате чего будет выполнен поиск в базе данных по соответствующему полю, и в случае успешного поиска будут получены данные о медикаменте. Далее пользователю необходимо ввести дополнительные параметры и подтвердить сохранение. При необходимости добавления пользовательских значений всех параметров медикамента необходимо в соответствующей форме выполнить ввод параметров через взаимодействие с интерфейсом (рисунок 1.3).

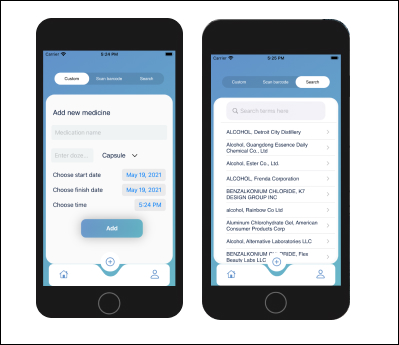


Рисунок 1.3 – Структура интерфейса вкладок добавления медикаментов

При корректном вводе будет выполнено сохранение записи в базу данных пользователя. В противном случае, обновление данных не произойдет и появится соответствующее сообщение об ошибке.

**1.2.3** Работа с записями и использование Core Data.

Core Data — [фреймворк](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) от компании Apple, встроенный в операционную систему IOS, MacOS, который позволяет разработчику взаимодействовать с базой данных. В приложении он используется для хранения и обработки данных авторизированного пользователя, полученных с сервера. Фреймворк реализует функционал записи, чтения(вывода), редактирования и удаления исчерпывающе описывающий возможности для работы с моделями базы данных (рисунок 1.4).

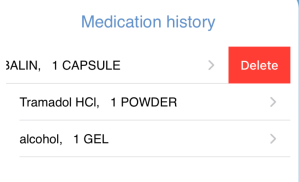


Рисунок 1.4 – Удаление медикамента из расписания

В приложении для реализации функционала CoreData создана модель Medication, объекты которой взаимодействуют с графическим интерфейсом и сервером через ViewModel, который реализует класс моделей Medication с присущими ему методами для работы с локальной и серверной базами данных. Инициализация внутри базы данных сущности, соответствующей модели Medication, выглядит следующим образом:

@Published var medications: [CoreDataModel] = []

let persistentContainer: NSPersistentContainer

init() {

persistentContainer = NSPersistentContainer(name: "MedicationViewModel")

persistentContainer.loadPersistentStores { (description, error) in

if let error = error {

fatalError("Core Data Store failed \(error.localizedDescription)")

}}}

После инициализации могут быть описаны остальные методы, обеспечивающие работу с данными модели. Для предоставления возможности использовать записи из базы данных внутри функции или представления необходимо выполнить следующие команды:

@Environment(\.managedObjectContext) var context

@FetchRequest(

entity: Medication.entity(),

sortDescriptors: [

NSSortDescriptor(keyPath: \Medication.generic\_name, ascending: true)]

) var medication: FetchedResults<Medication>

После выполнения описанных команд становится возможным использование Core Data в качестве локального хранилища данных пользователей для обеспечения доступа к необходимым данным с более высокой скоростью, а также для обеспечения возможности работы части функционала приложения в режиме оффлайн без необходимости обращения к серверу и хранящейся на нем базе данных.

**1.2.4** Центр уведомлений.

Одним из средств взаимодействия с пользователем в приложении является механизм по отправке уведомлений с напоминанием приема лекарств в соответствии с установленным временем приема медикаментов.

В рамках программы на языке Swift для подключения локальных уведомлений используется фреймворк UserNotifications. Первым шагом для любого приложения, поддерживающего уведомления, является запрос разрешения от пользователя. Как и в предыдущих версиях IOS, при использовании фреймворка UserNotifications обычно это делается, как только приложение запускается (рис. 1.5).

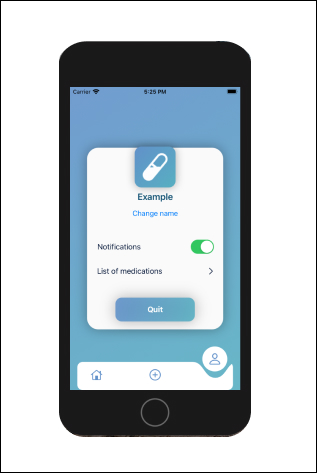


Рисунок 1.5 – Структура вкладки настроек аккаунта

Реализация добавления уведомления в расписание после установления его атрибутов выглядит следующим образом:

let content = UNMutableNotificationContent()

content.title = "PillQuipo"

content.subtitle = "Hey, it's time to take care of your health"

content.sound = UNNotificationSound.default

var dateComponents = DateComponents()

dateComponents.hour = hour

dateComponents.minute = minute

let trigger = UNCalendarNotificationTrigger(dateMatching: dateComponents, repeats: false)

let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: trigger)

localNotification.add(request)

При срабатывании уведомления, вне зависимости от того, открыто ли приложение на устройстве в данный момент, на экране появляется всплывающее уведомление (рисунок 1.6):

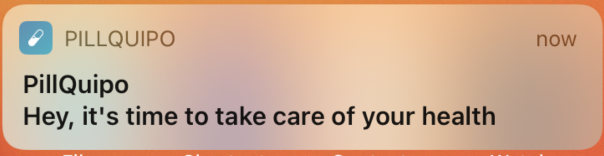


Рисунок 1.6 – Локальные уведомления

Описанные механизмы являются основными при определении функционала приложения. Все составные части приложения описаны с использованием объектно-реализованных технологий в рамках используемого стека технологий и среды разработки.

## 1.3 Серверная часть

Для обеспечения оптимального расхода памяти, обеспечения безопасности хранения и целостности данных и расширения функционала приложения реализована серверная часть на языке Swift, на которой храниться база данных и вся логика приложения, связанная со взаимодействием глобальной базы данных. Долгое время Swift оставался языком исключительно клиентской разработки, однако со временем появилась необходимость создания нативного средства для создания эффективной серверной части, так как современным приложениям необходимо хранить и обрабатывать всё больше данных, имеющих более сложную структуру.

**1.3.1** Программная реализация серверной части.

[Vapor](https://github.com/vapor/vapor) считается самым популярным с точки эффективности работы и количества плагинов. Он создан поверх фреймворка [Swift-nio](https://github.com/apple/swift-nio) от Apple, что делает его настоящим мощным инструментом. В отличие от Kitura и других фреймворков, которые не написаны исключительно на Swift, а скорее с использованием Node.js или других парсеров, Vapor отделяется от любой из этих зависимостей для предоставления парсера Swift и дает понятный и читаемый программный интерфейс.

Фреймворк Vapor предоставляет удобную файловую структуру для обеспечения удобства разработчиков при проектировании взаимодействия с базой данных. При описании данных используется технология объектно-ориентированного программирования, то есть используемые структуры данных представляются в виде классов с необходимыми атрибутами и расширениями, позволяющими использовать экземпляры класса для различных целей. Описание класса, представляющего собой концептуальную модель принимаемого пользователем медикамента, выглядит следующим образом:

final class Medication: Codable {

typealias Database = SQLiteDatabase

var id: Int?

var name: String

var type: String

var doze: String

var time: Date

var startDate: Date

var endDate: Date

init(name: String, type: String, doze: String, time: Date, startDate: Date, endDate: Date) {

self.name = name

self.type = type

self.doze = doze

self.time = time

self.startDate = startDate

self.endDate = endDate

}

}

extension Medication: SQLiteModel {}

extension Medication: Migration {}

extension Medication: Content {}

extension Medication: Parameter {}

Помимо определения атрибутов класса на данном этапе происходит инициализация переменных для упрощения процесса доступа при работе с ними. В качестве расширений используются интерфейсы SQLiteModel, Migration (для представления объектов класса как объектов базы данных), Content (для отправки и получения модели в формате JavaScript Object Notation) и Parameter (для обеспечения процессов маршрутизации).

Механизм передачи и обработки данных в фреймворке реализуется посредством взаимодействия контроллеров и функций-маршрутизаторов. Для описанного выше класса контроллер, обеспечивающий прием, вывод, редактирование и удаление записей в базе данных может выглядеть следующим образом:

func boot(router: Router) throws {

let medicationRoute = router.grouped("api", "medication")

medicationRoute.get(use: getAllHandler)

medicationRoute.get(Medication.parameter, use: getOneHandler)

medicationRoute.post(use: createHandler)

medicationRoute.put(Medication.parameter, use: updateHandler)

medicationRoute.delete(Medication.parameter, use: deleteHandler)

}

Для соединения контроллеров с маршрутизатором можно использовать следующие команды:

let medicationController = MedicationController()

try router.register(collection: medicationController)

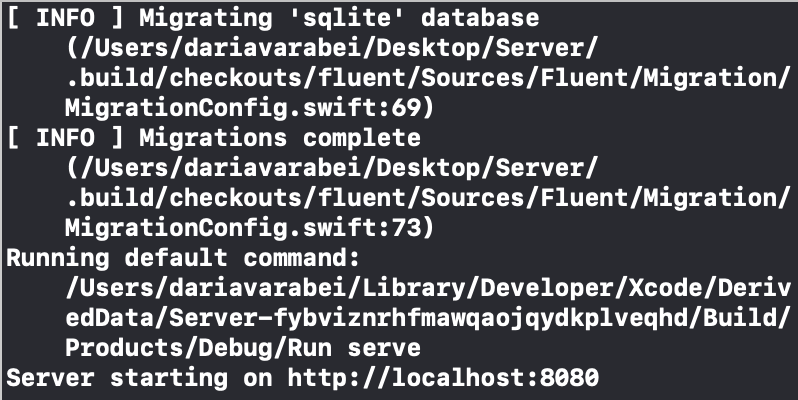


Рисунок 1.7 – Запуск сервера

В результате работы описанного механизма получаем серверное приложение, взаимодействующее с клиентской частью через интерфейс интернет протокола и способное обрабатывать запросы на вывод, добавление, редактирование и удаление записей из базы данных.

**1.3.2** База данных SQLite

База данных SQLiteиспользуется в качестве хранилища данных в серверной части приложения. Программный интерфейс Vapor обеспечивает взаимодействие со многими вендорами баз данных, и одним из лучших по многим параметрам является технология SQLite. SQLite — это встраиваемая кроссплатформенная база данных, которая поддерживает достаточно полный набор команд SQL, обеспечивающих ввод, обработку и хранение данных для работы с приложением.

Алгоритм создания файла базы данных, имеющей атрибуты, эквивалентные полям и типам данных, описанных в модели Medication для хранения моделей этого класса в качестве записей выглядит следующим образом:

services.register(router, as: Router.self)

let directoryConfig = DirectoryConfig.detect()

services.register(directoryConfig)

try services.register(FluentSQLiteProvider())

var databaseConfig = DatabasesConfig()

let db = try SQLiteDatabase(storage: .file(path:".../Server/manager.db"))

databaseConfig.add(database: db, as: .sqlite)

services.register(databaseConfig)

var migrationConfig = MigrationConfig()

migrationConfig.add(model: Medication.self, database: .sqlite)

services.register(migrationConfig)

Данный код сначала импортирует модуль SQLite. Затем создается новый класс со статической и публичной переменными. После чего происходит инициализация, при которой мы создаем файл с базой данных, соответствующей модели, отправляем запрос на установление соединения с базой данных и ее открытие. В случае успешного завершения процедуры сервер может начать свою работу, а в случае ошибки - выбросить соответствующее исключение и остановить работу приложения.

Механизм обработки запросов для работы с данными содержит в себе команды, заимствованные из SQL и имеющие схожий синтаксис. При получении запросов через при помощи ключевых слов серверное приложение определяет необходимое для выполнения действие и при помощи дополнительных параметров выполняет поставленную задачу. В качестве примера рассмотрим запрос на редактирование существующей в базе данных позиции, соответствующей модели Medication:

func updateHandler(\_ req: Request) throws -> Future<Medication> {

return try flatMap(to: Medication.self,

req.parameters.next(Medication.self), req.content.decode(Medication.self), where Medication.id == req.decoded.parameter(Medication.self (Medication.key))) { (medication, updatedMedication) in

medication.id = updatedMedication.id

medication.name = updatedMedication.name

medication.type = updatedMedication.type

medication.doze = updatedMedication.doze

medication.time = updatedMedication.time

medication.startDate = updatedMedication.startDate

medication.endDate = updatedMedication.endDate

return medication.save(on: req)

}

}

Для обмена данными между клиентом и сервером используется технология Representational state transfer. Для передачи от сервера к клиенту и наоборот поля передаваемых структур кодируются по правилам формата JavaScript Object Notation. Данный формат представляет собой простую строку, которая может быть использована в любом языке программирования.

# 2 ОБОСНОВАНИЕ ВЫБОРА ТЕХНОЛОГИЙ

## 2.1 Технологии программирования, используемые для решения

**поставленных задач**

Основой реализации функционала приложения является язык программирования Swift – мультипарадигмальный язык программирования, предназначенный для разработки приложений для IOS и macOS. Данный язык является инструментом нативной разработки, что позволяет ему сочетать в себе скорость выполнения, надежность, производительность и компактность кода.

Для описания пользовательского интерфейса и передачи данных в рамках клиентской части используется фреймворк SwiftUI, который позволяет проектировать и разрабатывать графический интерфейс с написанием меньшего количества кода, декларативным способом. Технология является новейшей в сфере IOS-разработки и предоставляет множество возможностей для описания графического представления интерфейса и обмена данными между составными частями приложения.

Использование Swift с целью разработки приложений имеет целый ряд преимуществ. Во-первых, благодаря постоянному развитию языка его скорость и производительность приближаются к данным показателям C++ и уже превышают показатели Objective-C. Во-вторых, благодаря простоте и понятности синтаксиса и пунктуации языка значительно уменьшается вероятность ошибок и сокращается время на написание и отладку программы.

Swift — более компактный язык программирования, чем ObjectiveC. Но, это вовсе не означает, что код простой. Иногда он очень даже сложный. Но в то же время, он обеспечивает преимущества, которыми не обладает ObjectiveC. Немаловажным является тот факт, что Swift является безопасным языком, который использует защиту от переполнений и опциональные типы данных, которые позволяют обрабатывать ситуации с null-значениями намного эффективнее.

Swift — единственный современный язык для разработки мобильных приложений в экосистеме Apple. Так же как и XCode — единственная официальная среда разработки для этого. Что с одной стороны ограничивает в выборе, с другой — вы имеете специализированный инструмент для работы.

С помощью MetalAPI (официальной библиотеки от Apple) делают игры и 3D графику. А библиотека ARKit позволяет создавать приложения дополненной реальности. В рамках фреймворка SwiftUI используются другие библиотеки для обеспечения дополнительного функционала работы с внешними службами и приложениями и контроля реализации основных требований языка и среды разработки.

Основной платформой для создания приложения является модуль Foundation, обеспечивающий базовый функционал приложения и предоставляющий базовые классы, такие как классы-оболочки и классы структуры данных. Важность модуля обусловлена тем, что Swift является объектно-ориентированным языком, в котором любой тип данных представлен на более низком уровне в качестве объекта одного из базовых классов, которые в свою очередь являются подклассами корневого класса NSObject.

Фреймворк AVFoundation используется для обработки аудиовизуальной информации в режиме реального времени, что предоставляет возможности для работы с камерой внутри приложения.Именно этот фреймворк выполняет поддержку работы с видео- и аудио-потоками, обеспечения обработки полученных данных и иного функционала (например, сканирование штрих-кодов и QR-кодов).

UserNotification представляет собой унифицированный, объектно-ориентированный способ работы с локальными и удаленными уведомлениями на платформах IOS, MacOS, tvOS и watchOS. Уведомления предоставляют возможность коммуникации с пользователем вне времени непосредственного использования приложения. При помощи UserNotification возможно работать как с локальными, так и с глобальными уведомлениями.

Фреймворк CoreData помогает интегрировать работу с нативной системой управления баз данных, которая используется в проекте в качестве локального хранилища для обеспечения более быстрого доступа к данным авторизированных пользователей в сочетании с богатым функционалом и высокой скоростью работы. Можно рассматривать Core Data, как оболочку над физическим реляционным хранилищем, представляющую данные в виде объектов, при этом сама Core Data не является базой данных.

Для обеспечения качества работы системы авторизации и аутентификации в приложении используется Firebase - облачная база данных, которая обеспечивает поддержку приложения, включая хранение данных и дополнительный функционал для регистрации и обработки исключительных ситуаций. При работе с приложением в целях безопасной регистрации и аутентификации применяется взаимодействие с сервисами электронной почты. Firebase предоставляет бэкенд, простой в использовании набор средств разработки и готовые библиотеки пользовательского интерфейса для реализации аутентификации пользователей в приложении. Он поддерживает аутентификацию как с помощью email и пароля, так и с помощью таких популярных поставщиков идентификации, как Google, Facebook, Twitter и GitHub.

Firebase поддерживает данные в формате JSON, и все подключенные к ней пользователи получают оперативные обновления после каждого изменения. Взаимодействие приложения и облачной системой управления базами данных обеспечивается посредством создания класса с соответствующими атрибутами и описания методов передачи данных в базу данных и обработке ситуаций, возникающих в процессе регистрации и ввода в аккаунт.

В серверной части основой проекта служит фреймворк Vapor 3 – современный server-side фреймворк, имеющий следующие особенности:

1 Статически типизированный (позволяет снизить потребляемую память и, соответственно, снизить стоимость создаваемого приложения).

2 Безопасность (обеспечивается за счет прямой поддержки со стороны операционной системы).

3 Скорость (является наиболее быстрым Swift-based фреймворком).

В отличие от Kitura и других фреймворков, которые не написаны исключительно на Swift, а скорее с использованием Node.js или других парсеров, Vapor отделяется от любой из этих зависимостей для предоставления парсера Swift и дает понятный и читаемый интерфейс. Vapor можно использовать для создания программных интерфейсов приложений, веб-приложений и приложений реального времени с помощью веб-сокетов. В дополнение к базовой структуре Vapor предоставляет функционал для объектно-реляционного отображения, язык шаблонов и пакеты для облегчения аутентификации и авторизации пользователей.

Vapor обеспечивает всестороннюю поддержку баз данных для поставщиков SQL, таких как MySQL и PostgreSQL, а также поставщиков NoSQL, таких как Redis и MongoDB. В связи с этим в качестве основной базы данных выбрана SQLite - компактная встраиваемая система управления базами данных.

Слово «встраиваемый» (embedded) означает, что SQLite не использует парадигму [клиент-сервер](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B8%D0%B5%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), то есть [движок](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%85%D1%80%D0%B0%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой [библиотеку](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), с которой программа компонуется, и движок становится составной частью программы.

Таким образом, в качестве протокола обмена используются вызовы функций интерфейса программного взаимодействия библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа. SQLite также используется в качестве локальной базы данных для обеспечения оффлайн доступа к некоторым данным в контексте фреймворка CoreData.

## 2.2 Реализация объектно-ориентированных технологий

**программирования**

Объектно-ориентированное программирование — [парадигма программирования](https://ru.wikipedia.org/wiki/%25D0%259F%25D0%25B0%25D1%2580%25D0%25B0%25D0%25B4%25D0%25B8%25D0%25B3%25D0%25BC%25D0%25B0_%25D0%25BF%25D1%2580%25D0%25BE%25D0%25B3%25D1%2580%25D0%25B0%25D0%25BC%25D0%25BC%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25B8%25D1%258F), основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования, то есть внутри одного класса могут существовать подклассы, в которых могут быть определены новые или переопределены имеющиеся атрибуты и методы, принадлежащие классу-родителю.

Так объектно-ориентированный подход позволяет использовать один и тот же программный код с разными данными. На основе классов создается множество объектов, у каждого из которых могут быть собственные значения полей. Нет необходимости вводить множество переменных, т. к. объекты получают в свое распоряжение индивидуальные пространства имен. В этом смысле объекты похожи на структуры данных. Объект можно представить как некую упаковку данных, к которой присоединены инструменты для их обработки – методы.

При написании программы применяются технологии объектно ориентированного программирования, основанные на представления программы как комплекса объектов различных классов и процессов взаимодействия между ними. Swift проектировался как объектно-ориентированный язык программирования на базе Objective-C. Это означает, что он построен с учетом следующих принципов:

1 Все данные в нем представляются объектами, являющимися представителями определенных классов.

2 Программу можно составить как набор взаимодействующих объектов, посылающих друг другу сообщения.

3 Каждый объект имеет собственную часть памяти и может состоять из других объектов.

4 Каждый объект имеет тип.

5 Все объекты одного типа могут принимать одни и те же сообщения (и выполнять одни и те же действия).

В связи с этим язык предполагает возможность применения всех основных принципов объектно-ориентированного программирования и создание пользовательских классов, объектов и методов, а также обеспечения объектно-реляционного взаимодействия. Исходя из цели и задач курсовой работы можно выделить следующие преимущества объектно-ориентированного подхода, выделяющиеся при написании программы:

1 Объектно-ориентированное программирование подразумевает повторное использование. Компьютерная программа написанная в форме объектов и классов может быть использована снова в других проектах без повторения кода.

2 Использование модулярного подхода в объектно-ориентированном программировании при грамотном использовании позволяет получить читаемый и гибкий код, более удобный в разработке, анализе и сопровождении.

3 В объектно-ориентированном программировании каждый [класс](https://python-scripts.com/data-class) имеет определенную задачу. Если ошибка возникнет в одной части кода, вы можете исправить ее локально, без необходимости вмешиваться в другие части кода.

4 Инкапсуляция данных вносит в реализацию дополнительный уровень безопасности в разрабатываемую программу с использованием объектно-ориентированного подхода.

5 Объектная декомпозиция уменьшает риск создания сложных систем программного обеспечения, так как она предполагает эволюционный путь развития системы на базе относительно небольших подсистем.

Большинство современных программно-математических сред поддерживают работу с составными элементами объектно-ориентированного программирования, так как он реализуется во множестве наиболее популярных языков разработки программных продуктов и требуют определенных условий от редакторов кода, компиляторов и другого инструментария.

При использовании языка Swift в интегрированной среде разработки Xcode реализуются концепции объектно-ориентированного программирования, все полученные в ходе выполнения программы данные представляются в виде объектов подклассов общего корневого класса NSObject, доступ к реализации методов которого может быть получен при

переходе в режим Assistant, в котором дается программное объяснение любого действия в программе на уровне языка Objective-C.

class CoreDataViewModel: ObservableObject{

@Published var medications: [CoreDataModel] = []

let persistentContainer: NSPersistentContainer

init() {

persistentContainer = NSPersistentContainer(name: "HelloCoreDataModel")

persistentContainer.loadPersistentStores { (description, error) in

if let error = error {

fatalError("Core Data Store failed \(error.localizedDescription)")

}

}

func getAllMovies() -> [Medication] {

let fetchRequest:

NSFetchRequest<Medication> = Medication.fetchRequest()

do {

return try persistentContainer.viewContext

.fetch(fetchRequest)

} catch {

return []

}

}

}

Объектно-ориентированная парадигма хорошо подходит для разработки сложных систем, так как подобные проекты следует начинать с высокоуровневого моделирования, на уровне которых уже все оперируется объектами. Соответственно, при переходе к этапу программной разработки, эти объекты намного проще будет переносить в код. При грамотном анализе и проектировании подобных систем объектно-ориентированный подход может значительно облегчить процесс сопровождения и модификации программного кода. Однако, использование данного подхода в относительно небольших системах не всегда оправдывает себя, так как может привести только к усложнению написания программы и увеличению времени ее исполнения.

# 3 ИНСТРУМЕНТАРИЙ

## 3.1 Обоснование используемых инструментов

Для разработки нативных приложений под операционную систему IOS, необходимо использовать среду разработки Xcode, поддерживающая разработку на таких языках, как Swift, ObjectiveC и С++. Xcode – инструментарий разработки приложений под Mac OS X и Apple iOS, разработанный компанией Apple. Поставляется бесплатно на дистрибутивном диске Mac OS X Install DVD вместе с операционной системой Mac OS X, но не устанавливается по умолчанию. Основным приложением пакета является встроенная среда разработки, которая называется Xcode. Помимо этого, пакет Xcode включает в себя большую часть документации разработчика от Apple и Interface Builder — приложение, использующееся для создания графических интерфейсов и проектирования взаимодействия с пользователем через компоненты интерфейса.

Для работы со средой разработки на его интерфейсе располагаются основные опции его функционала, позволяющие обеспечить полный цикл написания, структурирования и отладки написанной программы на языках Swift и ObjectiveC (рисунок 3.1):

1 Панель инструментов (англ. «toolbar») – самая верхняя часть окна, где расположены всевозможные настройки. Также там указано название нашей программы, ее состояние (например «компилируется») и количество ошибок (если они имеются).

2 Панель навигации (англ. «navigation tab») панель слева окна изначально показывает список файлов открытой программы, но благодаря переключателю (вверху панели) здесь также можно просмотреть структуру нашей программы, совершить поиск по ней, взглянуть есть ли ошибки, открыть дебаггер, найти точки остановки и т.д.

3 Редактор кода. Основная часть Xcode, в которой происходит написание кода программы.

4 Инспектор – верхняя правая панель. В нем можно посмотреть и изменить свойства текущего файла или содержание редактора кода.

5 Отладчик (англ. «debugger») Эта необходимая часть любой среды программирования находится в самом низу окна, если вкратце – выдает результат программы и показывает значения переменных.

6 Библиотека. В библиотеке располагаются различные группы элементов интерфейса.

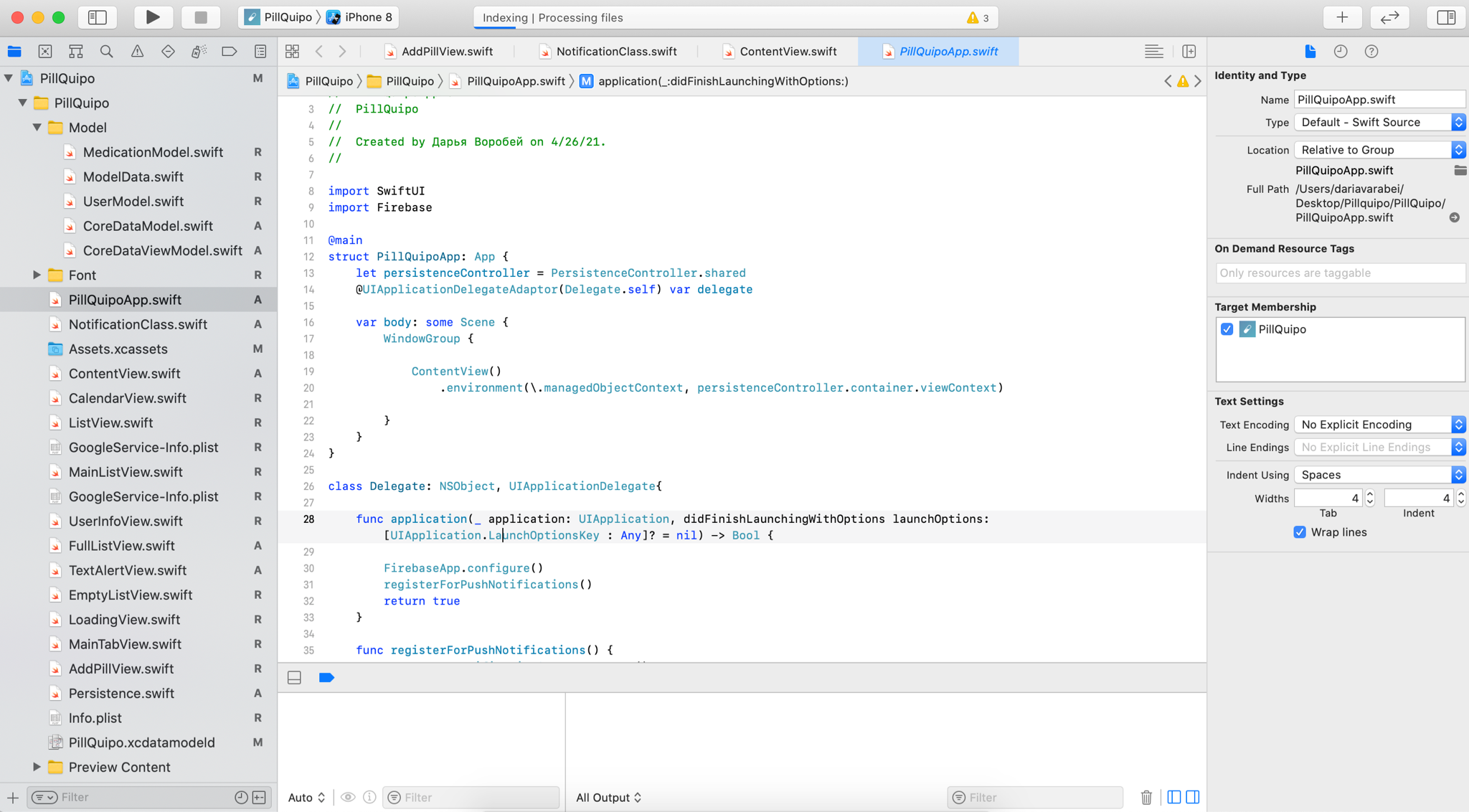


Рисунок 3.1 – Структура главного экрана Xcode

Xcode предоставляет множество возможностей для тестирования работы разработанных приложений. Для проверки работы программы на реальном устройстве не обязательно подключать внешнее IOS-устройство, а достаточно воспользоваться набором встроенных симуляторов. Набор включает в себя различные устройства компании Apple с различными версиями соответствующих им операционных систем. Представленные устройства с запущенными на них программами моделируют поведение существующих систем в режиме реального времени, позволяя также использовать внутренние приложения устройства(камера, календарь, центр уведомлений) и оценить работу программы без необходимости наличия устройства с операционной системой, на котором программное обеспечение может быть использовано.

При необходимости приложение может быть в короткие сроки загружено на внешнее устройство без необходимости установки дополнительных программ, используя исключительно устройство подключения с установленным соединением с аккаунтом разработчика приложения. При тестировании приложение будет копировать поведение реального продукта и запрашивать доступ ко внешним ресурсам и ресурсам устройства для исполнения предусмотренных функций.

Кроме написания программ интегрированная среда разработки Xcode позволяет создавать прикладные пакеты, библиотеки и модули для упрощения работы с некоторым рассматриваемым функционалом. Дополнительные модули, находящиеся в открытом доступе, могут быть также загружены с использованием команд и графического интерфейса Command Line Tools через соответствующие ссылки, а также подключены автоматически при помощи установленных программными модулями зависимостями из подключенных к приложению исполняемых файлов.

Менеджер пакетов – набор программного обеспечения, позволяющего управлять процессом установки, удаления, настройки и обновления различных компонентов программного обеспечения. Системы управления пакетами активно используются в различных дистрибутивах операционной системы Linux и других UNIX-подобных операционных системах (включая IOS) и являются важным компонентом при обеспечении разработки качественного программного обеспечения.

Одним из модулей, имеющих широкий функционал в рамках IOS-разработки, является Cocoapods, с помощью которого можно легко и быстро подключать различные инструменты, утилиты и библиотеки. CocoaPods разберется с зависимостями между библиотеками, которые вы используете, скачает их, создаст и будет поддерживать структуру проекта в надлежащем виде.

CocoaPods построен с Ruby, который устанавливается по умолчанию в версиях macOS. Для установки необходимого функционала необходимо использовать командную строку терминала, с помощью которой осуществляются загрузка, установка и подключение специальных файлов Gems, написанных на языке Ruby. Данные файлы также используются для построение соответствующих зависимостей из файла Podfile, который описывает необходимые для установки модули. Для установки Cocoapods необходимо иметь установленную утилиту Homebrew — менеджер пакетов для Mac OS X, который упрощает установку открытого программного обеспечения. Этот проект имеет открытый исходный код, также написанный на языке Ruby. К несомненным плюсам Homebrew можно отнести возможность удаленной установки программ с использованием протокола Secure Shell. Такая установка пройдет незаметно для пользователя и не будет отвлекать его от работы.

Обычно в пакетах содержатся уже скомпилированные версии программ. В Homebrew так же некоторые приложения распространяются в собранном виде, но все же стоит отметить, что большинство программ собирается из исходных кодов. Сценарий сборки и установки определяется так называемыми формулами, которые могут быть отредактированы в соответствии с личными потребностями, что обеспечивает высокую гибкость. В проекте Homebrew используется для подключения пакетов, обеспечивающих работу с облачной базой данных Firebase, а также применения другого менеджера пакетов – CocoaPods, который ответственен за импортирование других программных модулей.

## 3.2 Использование системы контроля версий GIT

Система контроля версий — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Система контроля версий появилась в 1980-х и до сих пор популярна как у разработчиков коммерческих продуктов, так и у open-source разработчиков. Система распространяется на условиях Открытого лицензионного соглашения и позволяет получать с сервера нужную версию проекта — «check-out» (извлечение), а затем пересылать обратно на сервер, «check-in» (возврат)*,* с внесенными изменениями.

Изначально система контроля версий была создана, чтобы избежать конфликта версий. Всем участникам для работы предоставлялась только самая последняя версия кода. Это была первая система контроля версий. Пользователю нужно было быстро фиксировать изменения в репозитории, пока другие не опередили его.

Такие системы наиболее широко используются при разработке программного обеспечения для хранения исходных кодов разрабатываемой программы. Однако они могут с успехом применяться и в других областях, в которых ведётся работа с большим количеством непрерывно изменяющихся электронных документов.

Git — распределённая система контроля версий. Система спроектирована как набор программ, специально разработанных с учетом их использования в сценариях. Это позволяет удобно создавать специализированные системы контроля версий на базе Git или пользовательские интерфейсы. Удаленный доступ к репозиториям Git обеспечивается git-доменом или сервером интернет-протокола обмена данными.

Использование системы контроля версий имеет множество преимуществ при разработке программного обеспечения, особенно когда работу над проектом осуществляет более одного человека. Git показывает очень высокую производительность в сравнении со множеством альтернатив. Это возможно благодаря оптимизации процедур фиксации коммитов, создания веток, слияния и сравнения предыдущих версий. Алгоритмы Git разработаны с учетом глубокого знания атрибутов, характерных для реальных деревьев файлов исходного кода, а также типичной динамики их изменений и последовательностей доступа.

Система Git рассчитана прежде всего на создание веток и использование тегов. Поэтому процедуры с участием веток и тегов (например, объединение и возврат к предыдущей версии) сохраняются в истории изменений. Не все системы управления версиями обладают настолько широкими возможностями отслеживания.

При работе с проектом используется GitHub — один из сервисов для использования системы контроля версий Git. Взаимодействие с сервисом происходит как через набор команд в консоли, так и с использованием встроенного функционала Command Line Tools в интегрированной среде разработки (рисунок 3.2).

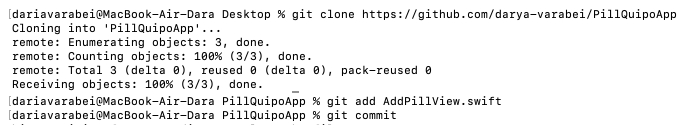


Рисунок 3.2 – Использование системы контроля версии Git

С их помощью происходит навигация между ветками проекта, подтверждение и отслеживание изменений кода и, при необходимости, возвращение к более ранней версии кода. Также система контроля версий обеспечивает сохранность кода для предотвращения его потери в случае, например, поломки компьютера или случайного удаления проекта с используемого персонального компьютера.

Ссылка на репозиторий, содержащий проект: https://github.com/darya-varabei/PillQuipoApp

# 4 АРХИТЕКТУРНЫЙ ШАБЛОН ПРОЕКТИРОВАНИЯ

# MVVM

Шаблон проектирования MVVM (Model - View - ViewModel) является наиболее применяемым при разработке программ в фреймворке SwiftUI. Использование данного шаблона позволяет обойти некоторые ограничения MVC, и в то же время применить его сильные стороны. Основной целью применения шаблона является разделение модели и ее представление в пользовательском интерфейсе, что позволяет осуществлять отдельную разработку графического пользовательского интерфейса и логики работы с данными, а также упрощает сопровождение и обновление программного кода.

## 4.1 Разработка схемы алгоритма

Схема алгоритма позволяет наглядно представить способ работы приложения в графическом виде. Полная схема алгоритма работы программы представлена в приложении.

Алгоритм работы приложения начинается с проверки, авторизирован ли пользователь в приложении на данном устройстве. В случае отсутствия авторизации пользователю предлагается войти в учетную запись, используя адрес электронной почты и пароль, или создать новый аккаунт, введя адрес электронной почты и пароль, и далее перейдя по ссылке, пришедшей от имени приложения на электронную почту, тем самым подтвердив создание аккаунта (рисунок 4.1):

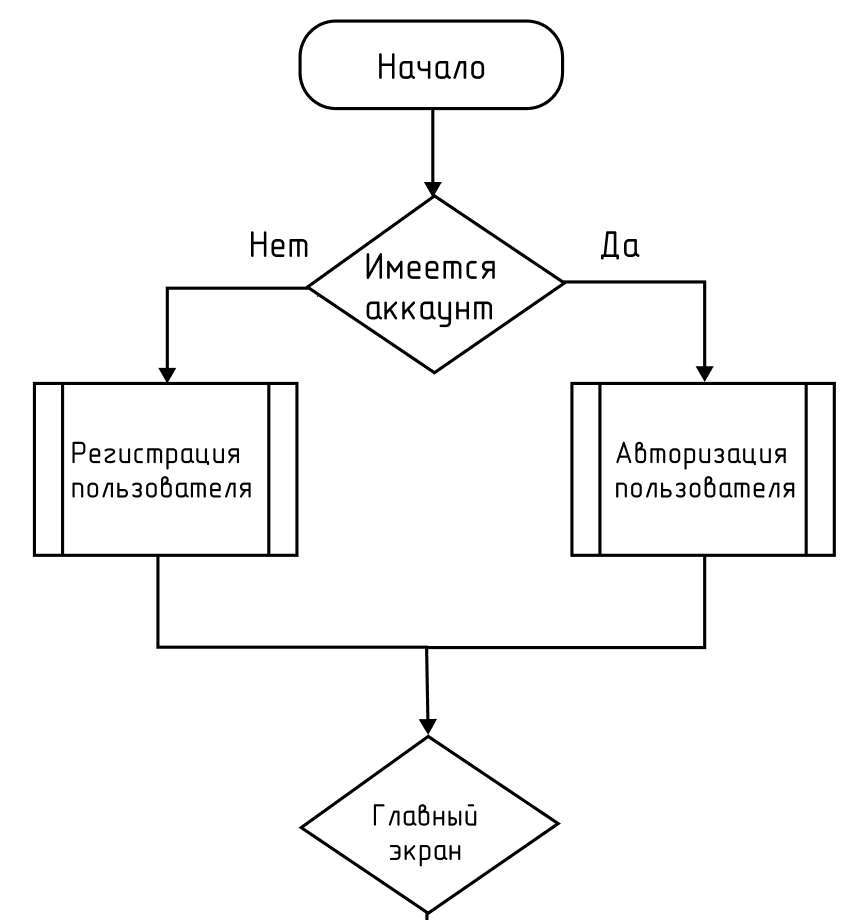


Рисунок 4.1 – Схема алгоритма входа в приложение, регистрации и авторизации

После успешного входа в учетную запись пользователь переходит на главный экран, на который при помощи запросов в локальную базу данных при нажатии на даты в календаре выводится список рекомендованных к принятию препаратов на этот день (рисунок 4.2):

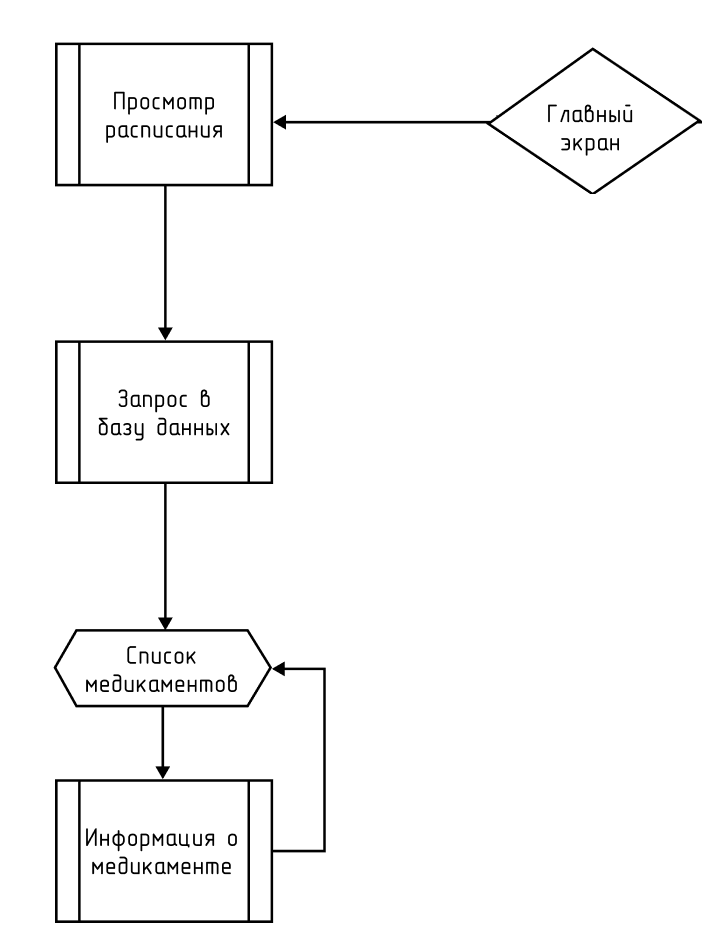


Рисунок 4.2 – Схема алгоритма работы главной страницы

Главной страницы через главное меню можно также перейти во вкладки с настройками аккаунта и добавлением новых препаратов в список. Функционал пользовательских настроек включает в себя полный список внесенных в расписание препаратов, в котором предоставляется возможность редактирования информации и удаления из расписания (рисунок 4.3):

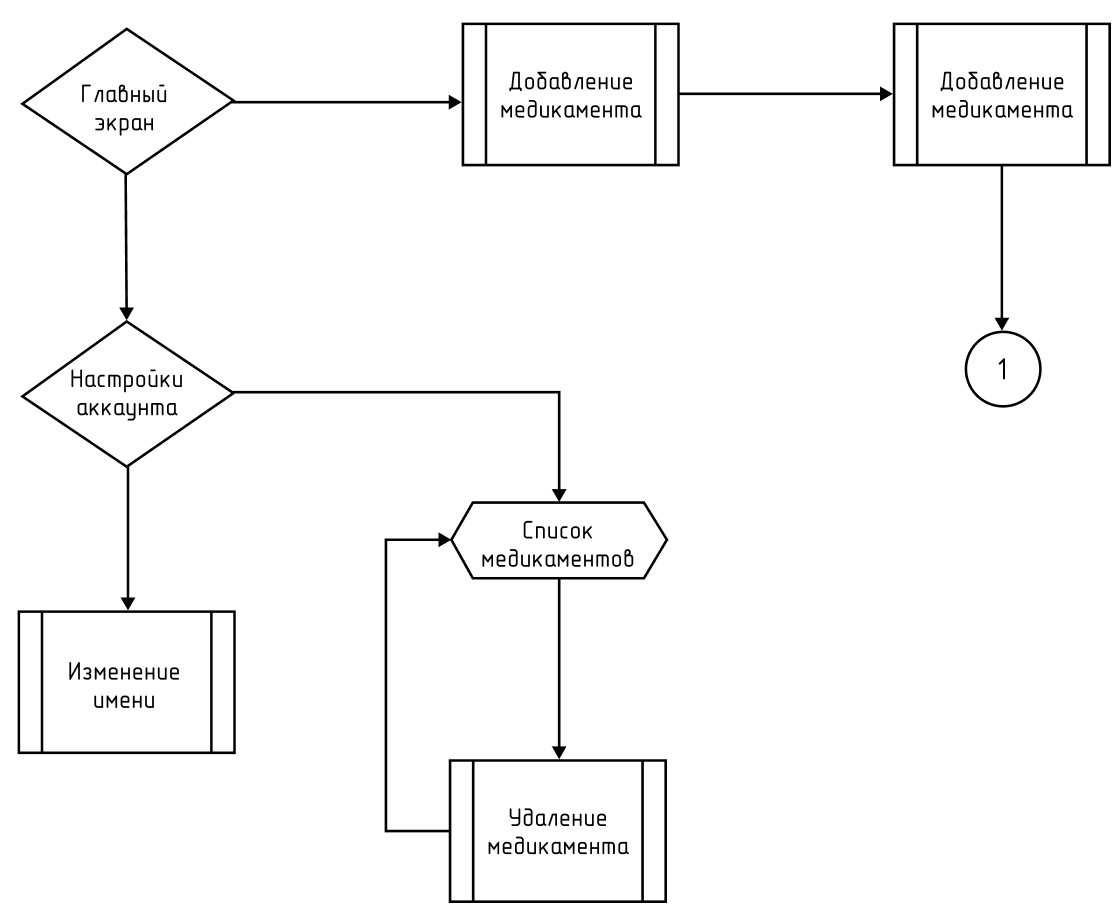


Рисунок 4.3 – Схема алгоритма работы со вкладками на главном экране

Вкладка добавления медикаментов содержит в себе опции добавления препарата через список с программного интерфейса Federal Drug Agency, добавление через сканирование штрих-кода или QR-кода, а также через самостоятельный ввод всех параметров. После успешного добавления медикамента любым из описанных способов происходит сохранение в базе данных посредством методов, описанных в классе Medication (рисунок 4.4):

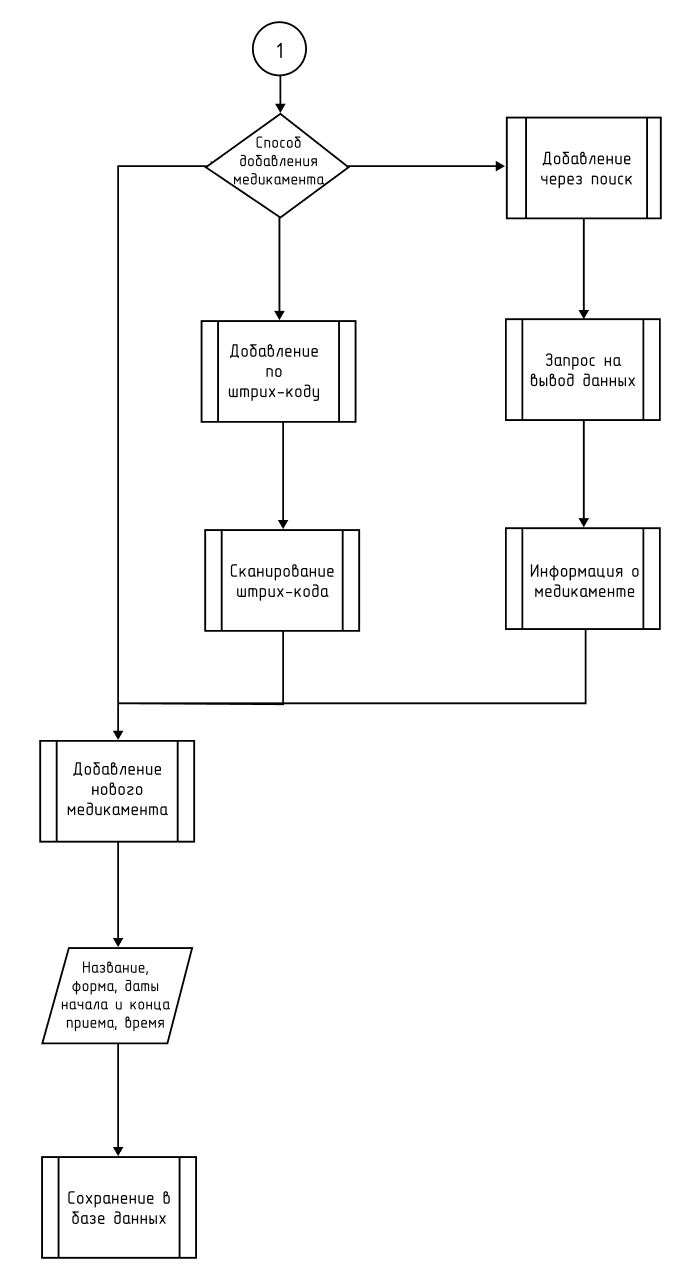


Рисунок 4.4 – Схема алгоритма добавления медикамента в список

Блок-схема алгоритма наглядно показывает последовательность срабатывания функций при различных вариантах использования программы.

## 4.2 Разработка диаграммы последовательности

Диаграмма последовательности позволяет определить не только последовательность действий в ходе работы с программой, но и время, в которое происходит каждое из событий. Диаграмма, представленная в курсовой работе, содержит 5 объектов, взаимодействующих между собой. Полная диаграмма последовательности представлена в приложении.

В диаграмме описан вариант взаимодействия с пользователем при выполнении GET-запроса. При выполнении запроса компонент View при получении запроса вызывается соответствующий классу метод getAllMedications(), который служит для вывода всех соответствующих данному пользователю записей о медикаментах. При выполнении инструкций метода база данных присылает сообщение об успешном выполнении запроса, экземплярам класса присваиваются значения из записей базы данных и для вывода на экран устройства вызывается функция, формирующая View каждой записи (рисунок 4.5):

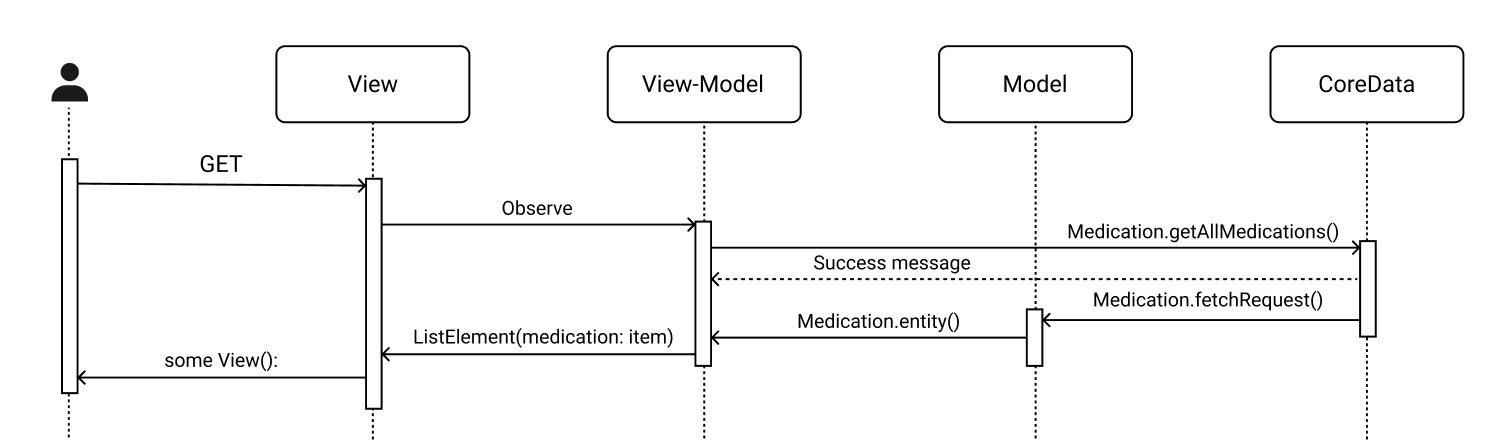


Рисунок 4.5 – Диаграмма последовательности для варианта взаимодействия приложения с пользователем (GET-запрос)

Следующим вариантом взаимодействия является обработка пользовательского POST запроса. View-Model ожидает соответствующего сообщения от пользовательского интерфейса, а затем вызывает соответствующий метод класса, который сохраняет запись в базе данных. После успешного сохранения записи и обновления базы данных, View-Model получает соответствующее сообщение, которое запускает механизм вывода сохраненной записи в главном списке после сообщения пользователю (рисунок 4.6):

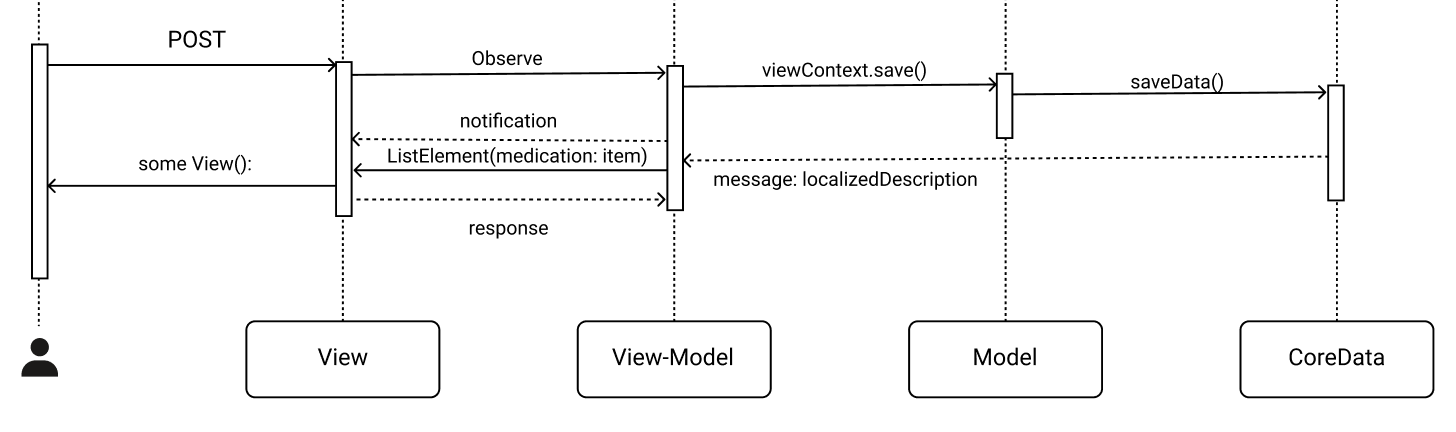


Рисунок 4.6 – Диаграмма последовательности для варианта взаимодействия приложения с пользователем (POST-запрос)

Представленная диаграмма последовательностей показывает наиболее типичные последовательности действий при работе с программой.

## 4.3 Разработка диаграммы состояний

Диаграмма состояний описывает поток управления из одного состояния в другое. Состояния определяются как состояние, при котором объект существует, и он изменяется при запуске какого-либо события. Наиболее важной целью диаграммы состояний является моделирование времени жизни объекта от создания до завершения. Полная диаграмма состояний представлена в приложении.

При прохождении регистрации и авторизации происходит работа с объектами учетных записей пользователей. При входе в приложение не авторизированный пользователь может войти в существующий аккаунт либо создать новый. Если пользователь не помнит пароль от аккаунта, он может сбросить его и ввести новый через переход по ссылке на электронной почте. Подтверждение регистрации нового аккаунта также необходимо производить через электронный почтовый ящик (рисунок 4.7):

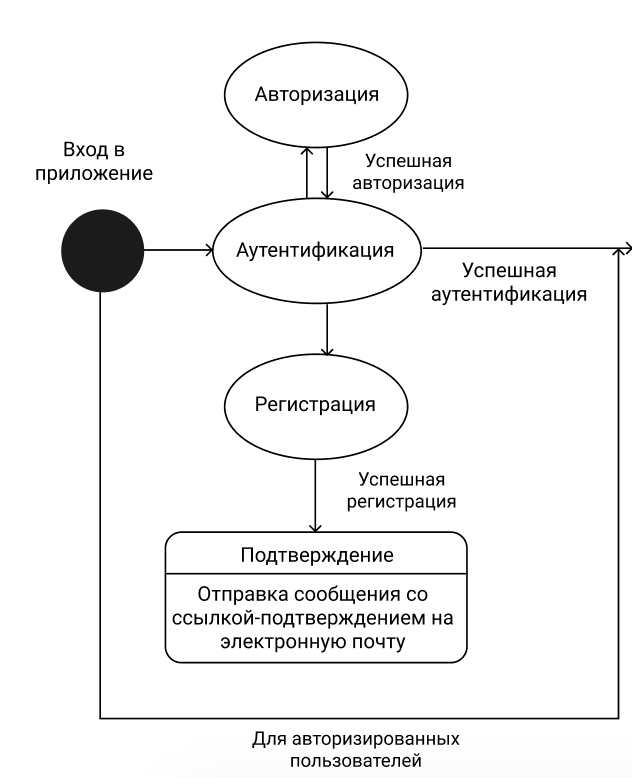


Рисунок 4.7 – Диаграмма состояния при прохождении регистрации и авторизации

При работе с главным экраном и экранами добавления медикаментов происходит обработка объектов из класса Medications, содержащего описание принимаемых медикаментов и методов для работы с объектами класса и их представления на графическом пользовательском интерфейсе. Также после авторизации появляется возможность получать уведомления от приложения после получения подтверждения от пользователя (рисунок 4.8):

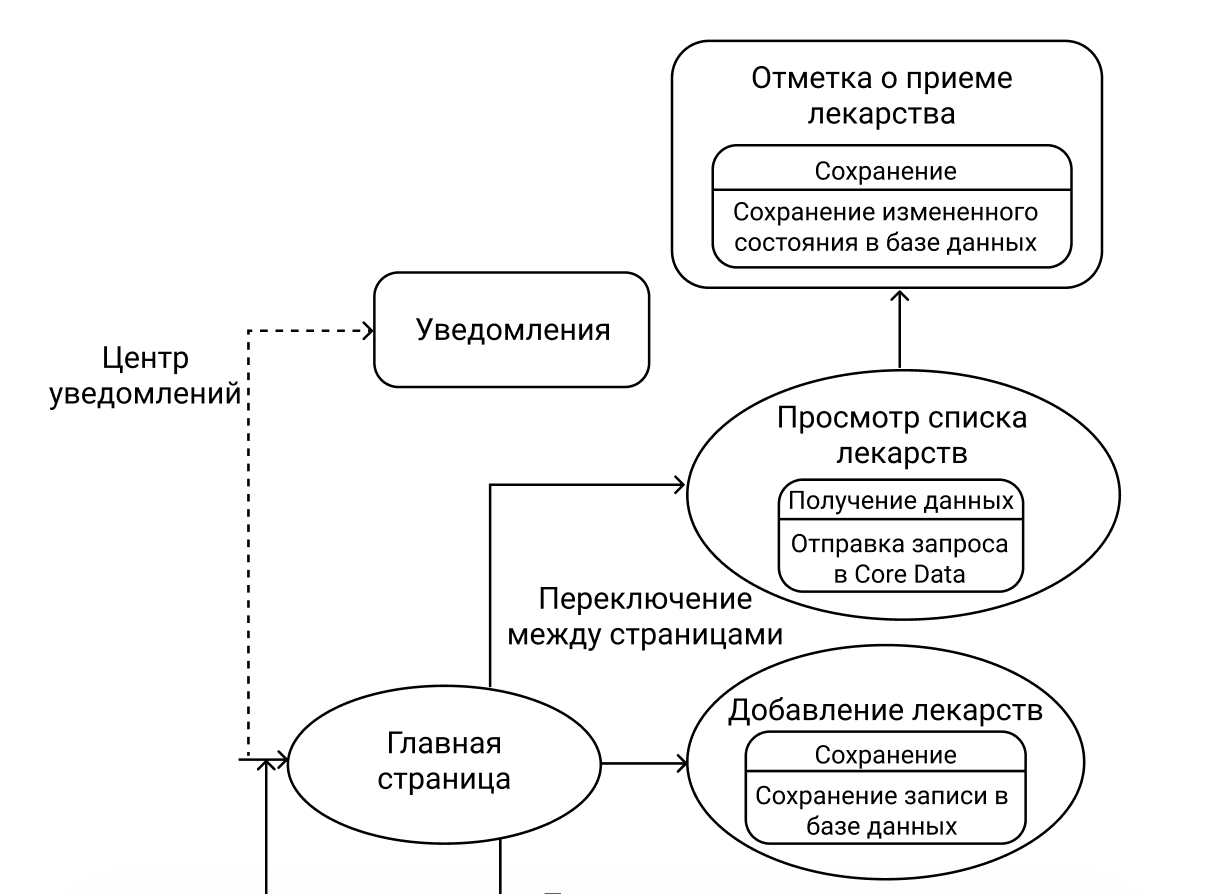


Рисунок 4.8 – Диаграмма состояния при работе на вкладках главного экрана и добавлении препарата

При работе во вкладке страницы пользователя происходит взаимодействие как с объектом класса User, так и Medication. Во вкладке можно выполнить редактирование и удаление внесенных в расписание записей. Для работы с центром уведомлений реализован контроллер для разрешения и запрета отправки уведомлений на экран. Редактирование расписания локальных уведомлений происходит при добавлении, редактировании и удалении медикаментов из расписания. Также используется функционал для изменения имени пользователя и выполнением выхода из учетной записи (рисунок 4.9):

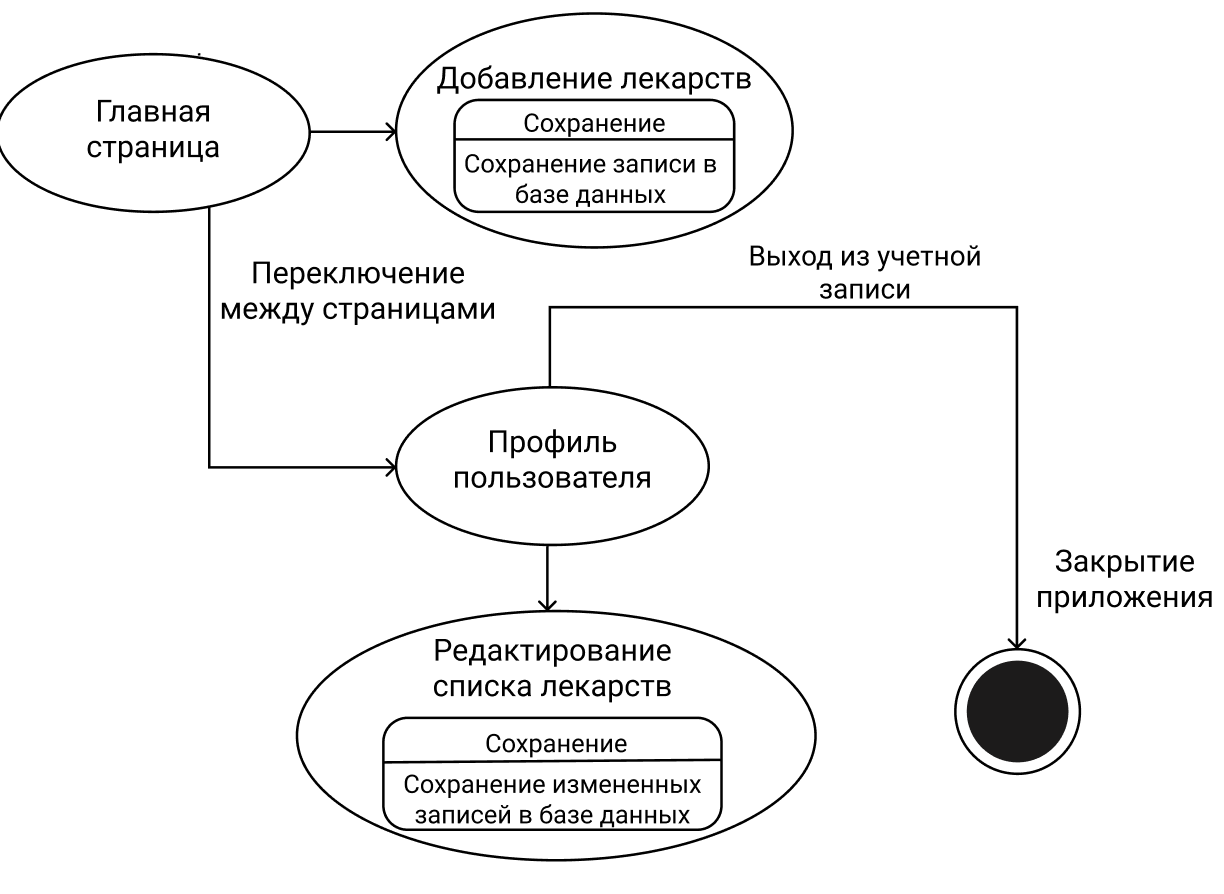


Рисунок 4.9 – Диаграмма состояния при реализации работы с профилем пользователя и выходом из программы

Реализованный набор графического материала позволяет получить наилучшее представление о процессах, происходящих внутри приложения при различных вариантах использования на разных этапах выполнения поставленных задач.

# 5 ШАБЛОН ПРОЕКТИРОВАНИЯ ПРАКТИЧЕСКИХ

# ЗАДАЧ

Паттерн(шаблон) проектирования — это часто встречающееся решение определенной проблемы при проектировании архитектуры программ. Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это четкий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах. При проектировании практических задач при разработке программного обеспечения на языке Swift может применяться широкий круг паттернов, многие из которых различаются по своему предназначению и образуют 3 группы:

1 Порождающие паттерны (отвечают за создание объектов без внесения в программу лишних зависимостей).

2 Структурные паттерны (отвечают за построение связей между созданными объектами).

3 Поведенческие паттерны (отвечают за эффективность и безопасность взаимодействия между объектами).

## 5.1 Порождающие паттерны

Одним из часто встречающих порождающих паттернов является Singleton – шаблон, при использовании которого объект существует в системе исключительно в единственном экземпляре и к нему предоставляется глобальная точка доступа. Singleton скрывает от клиентов все способы создания нового объекта, кроме специального метода. Этот метод либо создаёт объект, либо отдаёт существующий объект, если он уже был создан. В отличие от глобальных переменных, Singleton гарантирует, что никакой другой код не заменит созданный экземпляр класса, поэтому вы всегда уверены в наличии лишь одного объекта-одиночки (рисунок 5.1).

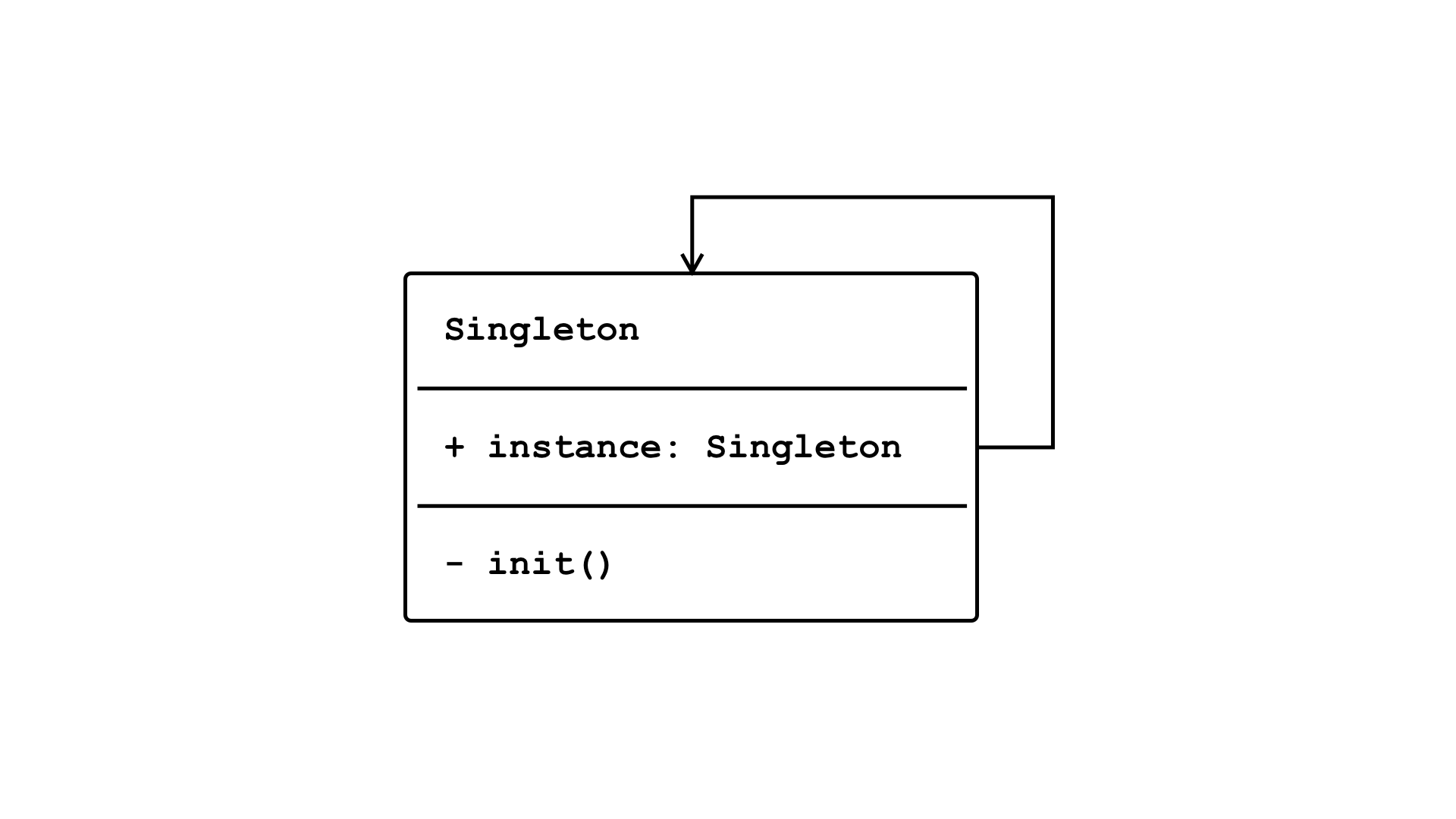


Рисунок 5.1 – Паттерн проектирования Singleton [16]

Еще одним популярным паттерном является Factory Method, который позволяет переложить обязанности создания специфических объектов на подклассы родительских классов не зная, объект какого класса в итоге будет создан. Объекты созданные фабричным методом – схожи, потому как у них один и тот же родительский объект. Потому, если локализировать создание таких объектов, то можно добавлять новые типы, не меняя при это код который использует фабричный метод (рисунок 5.2).

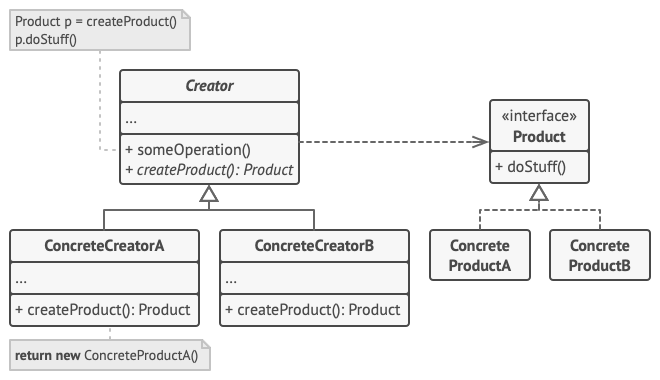


Рисунок 5.2 – Паттерн проектирования Factory Method [16]

Реализация паттерна Abstract Factory в некотором виде похожа на реализацию Factory Method. Основным отличием является то, что абстрактная фабрика поддерживает создание независимых объектов, относящихся к разным семействам, в то время как обычная фабрика порождает объекты исключительно одного и того же типа. Это достигается редактированием внутренности метода, ответственного за порождение новых объектов (рисунок 5.3).

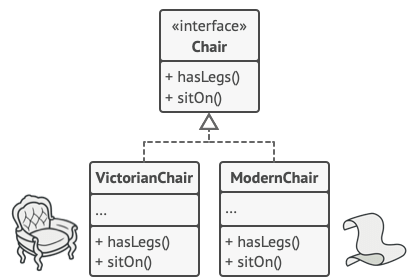


Рисунок 5.3 – Паттерн проектирования Abstract Factory Method [16]

Решением для создания сложных объектов, при создании которых создаются другие дочерние объекты, является использование паттерна Builder. В языке Swift паттерн состоит из двух компонент – Builder и Director. Builder занимается именно построение объекта, а Director знает какой Builder использовать чтобы выдать необходимый продукт. Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги. Чтобы создать объект, вам нужно поочередно вызывать методы строителя. Причем не нужно запускать все шаги, а только те, что нужны для производства объекта определенной конфигурации (рисунок 5.4).

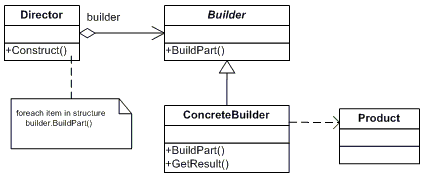


Рисунок 5.4 – Паттерн проектирования Builder [16]

Иногда при работе программы нам необходимо получить точную копию некоторого объекта. Для этого можно использовать паттерн Prototype. Реализация этого паттерна в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как Swift разрешает доступ к приватным полям любого объекта текущего класса (рисунок 5.5).

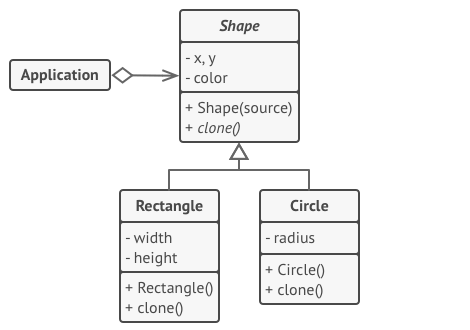


Рисунок 5.5 – Паттерн проектирования Prototype [16]

Рассмотренные порождающие паттерны проектирования являются наиболее часто встречающимися в языке Swift.

## 5.2 Структурные паттерны

Adapter - структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту. Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки (рисунок 5.6).

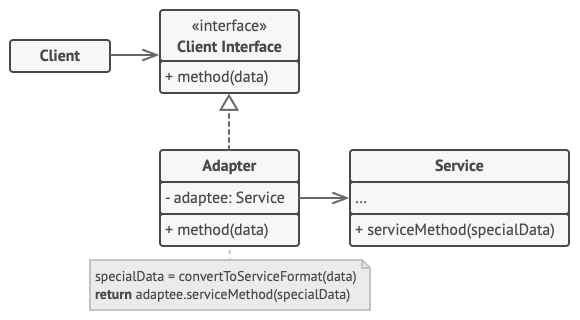


Рисунок 5.6 – Паттерн проектирования Adapter [16]

Паттерн Bridge имеет явную связь с концепцией объектно-ориентированного программирования, так как его основной задачей является отделение абстракции от реализации с целью увеличения гибкости кода. Таким образом, вы сможете изменять графический интерфейс приложения, не трогая низкоуровневый код работы с операционной системой. И наоборот, вы сможете добавлять поддержку новых операционных систем, создавая подклассы реализации, без необходимости менять классы графического интерфейса (рисунок 5.7).

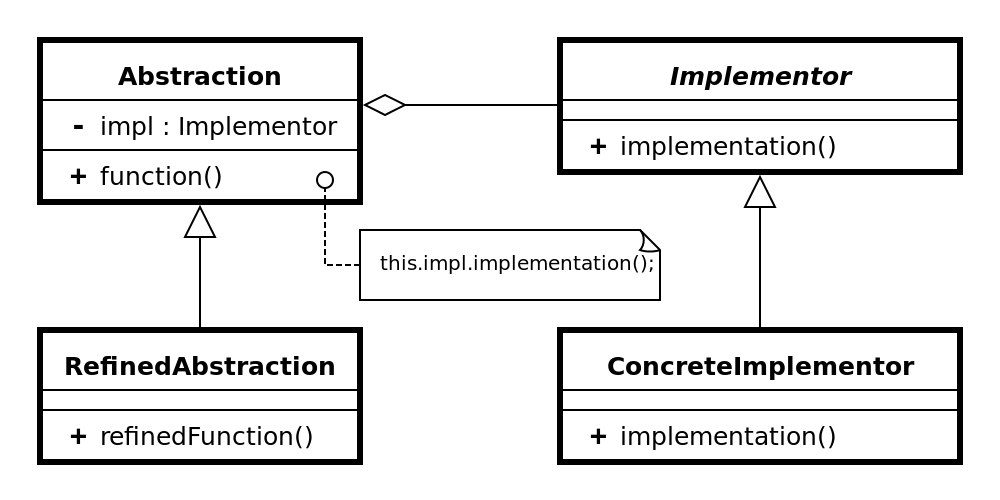


Рисунок 5.7 – Паттерн проектирования Bridge [16]

Для организации работы с древовидными структурами используется паттерн компоновщик. Паттерн Компоновщик предлагает хранить в составных объектах ссылки на другие простые или составные объекты. Те, в свою очередь, тоже могут хранить свои вложенные объекты и так далее. В итоге вы можете строить сложную древовидную структуру данных, используя всего две основные разновидности объектов (рисунок 5.8).

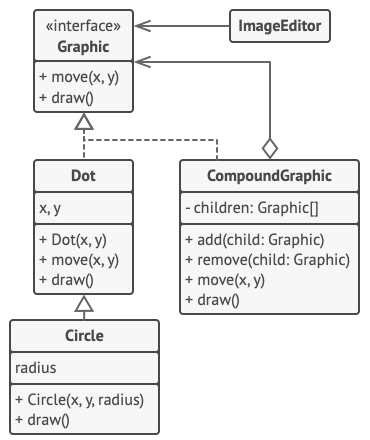


Рисунок 5.8 – Паттерн проектирования Composite [16]

Целью использования паттерна Decorator является добавление нового функционала к уже существующему объекту, при этом не задевая и не меняя других объектов. При реализации паттерна целевой объект помещается в другой объект-обертку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё (рисунок 5.9).

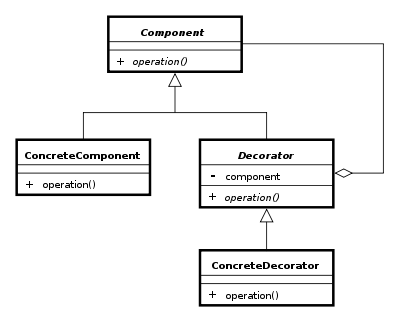


Рисунок 5.9 – Паттерн проектирования Decorator [16]

Структурные паттерны особенно полезны при организации работы нескольких независимых библиотек.

## 5.3 Поведенческие паттерны

Медиатор – паттерн которые определяет внутри себя объект, в котором реализуется взаимодействие между некоторым количеством объектов. При этом эти объекты, могут даже не знать про существования друг друга, потому взаимодействий реализованых в медиаторе может быть огромное количество. Компоненты не должны общаться друг с другом напрямую. Если в компоненте происходит важное событие, он должен известить своего посредника, а тот сам решит — касается ли событие других компонентов, и стоит ли их оповещать. При этом компонент-отправитель не знает кто обработает его запрос, а компонент-получатель не знает кто его прислал (рисунок 5.10).

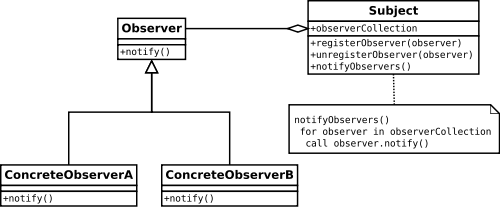


Рисунок 5.10 – Паттерн проектирования Mediator [16]

Когда у нас есть сложный объект, в котором содержится большое количество различных элементов, и мы хотим выполнять различные операции в зависимости от типа этих элиментов, возникает необходимость использовать паттерн Visitor для повышения надежности кода. Паттерн визитер – позволяет вынести из наших объектов логику, которая относится к этим объектам, в отдельный клас, что позволяет нам легко изменять / добавлять алгоритмы, при этом не меняя логику самого класа. Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя (рисунок 5.11).

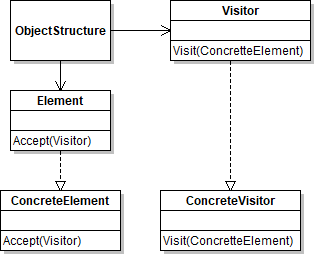


Рисунок 5.11 – Паттерн проектирования Visitor [16]

Паттерн стратегия (Strategy) основывается на предположении, что в различных ситуациях один и тот же объект может вести себя по-разному. В связи с этим появляется необходимость определить семейство алгоритмов, которые могут взаимозаменяться. Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию. Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста (рисунок 5.12).

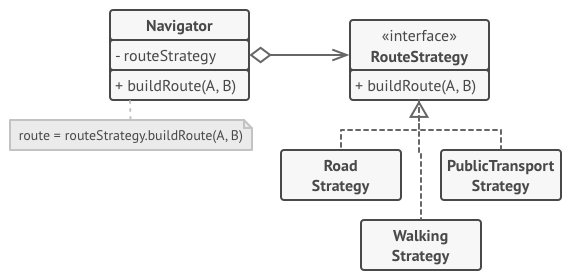


Рисунок 5.12 – Паттерн проектирования Strategy[16]

Популярным методом для решения множества задач является использование Template Pattern (Шаблонный паттерн). Данный паттерн предлагает разработчику разбить алгоритм на последовательность некоторых шагов, затем описать последовательность этих шагов в отдельном методе класса и запускать работу со свойствами класса через вызов только одного метода (рисунок 5.13).

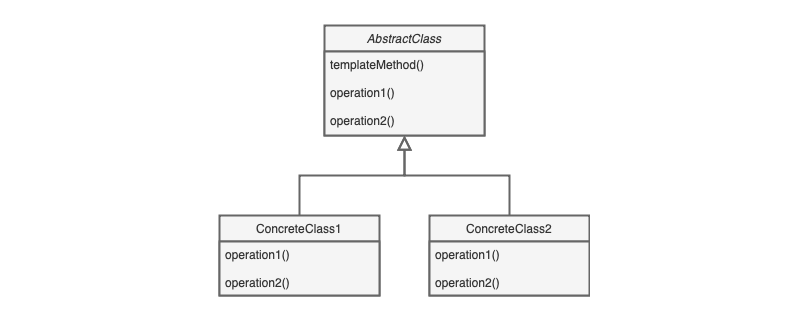


Рисунок 5.13 – Паттерн проектирования Template [16]

Chain of responsibility (Цепочка обязанностей) — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи. Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В данном случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы (рисунок 5.14).

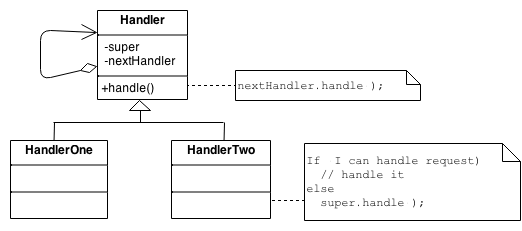


Рисунок 5.14 – Паттерн проектирования Chain of responsibility[16]

Рассмотренные паттерны являются наиболее популярными при разработке приложений и информационных систем на языке Swift.

# 

# ЗАКЛЮЧЕНИЕ

Разработка мобильных приложений под операционную систему IOS является перспективным направлением на рынке информационных технологий за счет наличия полностью нативного стека технологий для работы над приложениями на всех стадиях разработки в сочетании с высокой производительностью и доступностью необходимого инструментария и технологий.

Для реализации необходимого функционала приложения использованы возможности средств разработки на языке Swift в контексте фреймворков Vapor и SwiftUI. Рассмотренные технологии обеспечили создание графического пользовательского интерфейса и механизма передачи данных внутри клиентской части и между клиентской и серверной частью. Также реализовано использование внутренних служб мобильного устройства, таких как сканирование биометрических данных и центр локальных уведомлений с последующей обработкой полученных данных в целях реализации поставленных задач.

Разработанное мобильное приложение помогает контролировать прием медикаментов пользователем. Благодаря использованию нативных технологий разработки достигается высокий уровень скорости, безопасности и производительности приложения. Приложение рассчитано на широкий круг пользователей, принимающих лекарственные средства как временно, так и на постоянной основе, для предотвращения несвоевременного употребления лекарственных средств.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гриб, А.С. Преимущества и недостатки архитектуры MVC при разработке IOS-приложения / А.С. Гриб, Д.В. Проходский // Студенческий вестник: электрон. научн. журн. 2019. Ч. 7, № 47- (97). – С. 30 – 32.

2. Гриб, А.С. Преимущества и недостатки архитектуры MVP при разработке IOS-приложения / А.С. Гриб, Д.В. Проходский // Студенческий вестник: электрон. научн. журн. 2019. Ч. 7, № 47- (97). – С. 27 – 29.

3. Официальный ресурс производителя [Электронный ресурс]. – Режим доступа:<https://developer.apple.com>.- Время доступа: 09.05.2021

4. Swift. Разработка приложений в среде Xcode для iPhone и iPad с использованием iOS SDK. - М.: Вильямс, 2015. - 816 c.

5. Apple Developer Network [Электронный ресурс] / ARKit– Режим доступа: <https://developer.apple.com/arkit>.- Время доступа: 04.05.2021

6. Марк, Д. Swift: разработка приложений в среде Xcode для iPhone и iPad с использованием iOs SDK / Д. Марк. - М.: Вильямс И.Д., 2015. - 816 c.

7. Погружение в паттерны проектирования [Электронный ресурс]. – Режим доступа:<https://refactoring.guru/ru/design-patterns/factory-method>

8. Дэйв, М. iOS 5 SDK. Разработка приложений для iPhone, iPad и iPod touch / М. Дэйв, Н. Джек. - М.: Вильямс, 2012. - 672 c.

9. SwiftUI 1.0 by Stanford [Электронный ресурс] / – Режим доступа:<https://web.stanford.edu/class/cs193p/cgi-bin/drupal/>.- Время доступа: 01.05.2021 - 11:32

10. Android Jetpack [Electronic Resource] / Google Developers . - Режим доступа: https://developer.android.com/jetpack/ . - Время доступа: 10.05.2021.

11. Joseph, J. Bambara SQL Server® Developer's Guide / Joseph J. Bambara, Paul R. Allen. - Москва: Мир, 2016. - 235 c.

12. Минаси М. Графический интерфейс пользователя: секреты проектирования. - М.: Мир, 2010. - 453 с.

13. Мэтт Нойбург. Программирование для iOS 7. Основы Objective-C, Xcode и Cocoa. iOS 7 Programming Fundamentals: Objective-C, Cocoa, and Xcode Basics. — М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1), 2014. — 384 с.

14. Основы программирования на языке Objective-C для iOS : учеб. пособие / А.В. Кузин, Е.В. Чумакова. — М. : ИНФРА-М, 2019. — 118 с. Режим доступа: <http://znanium.com/catalog/product/1019936>

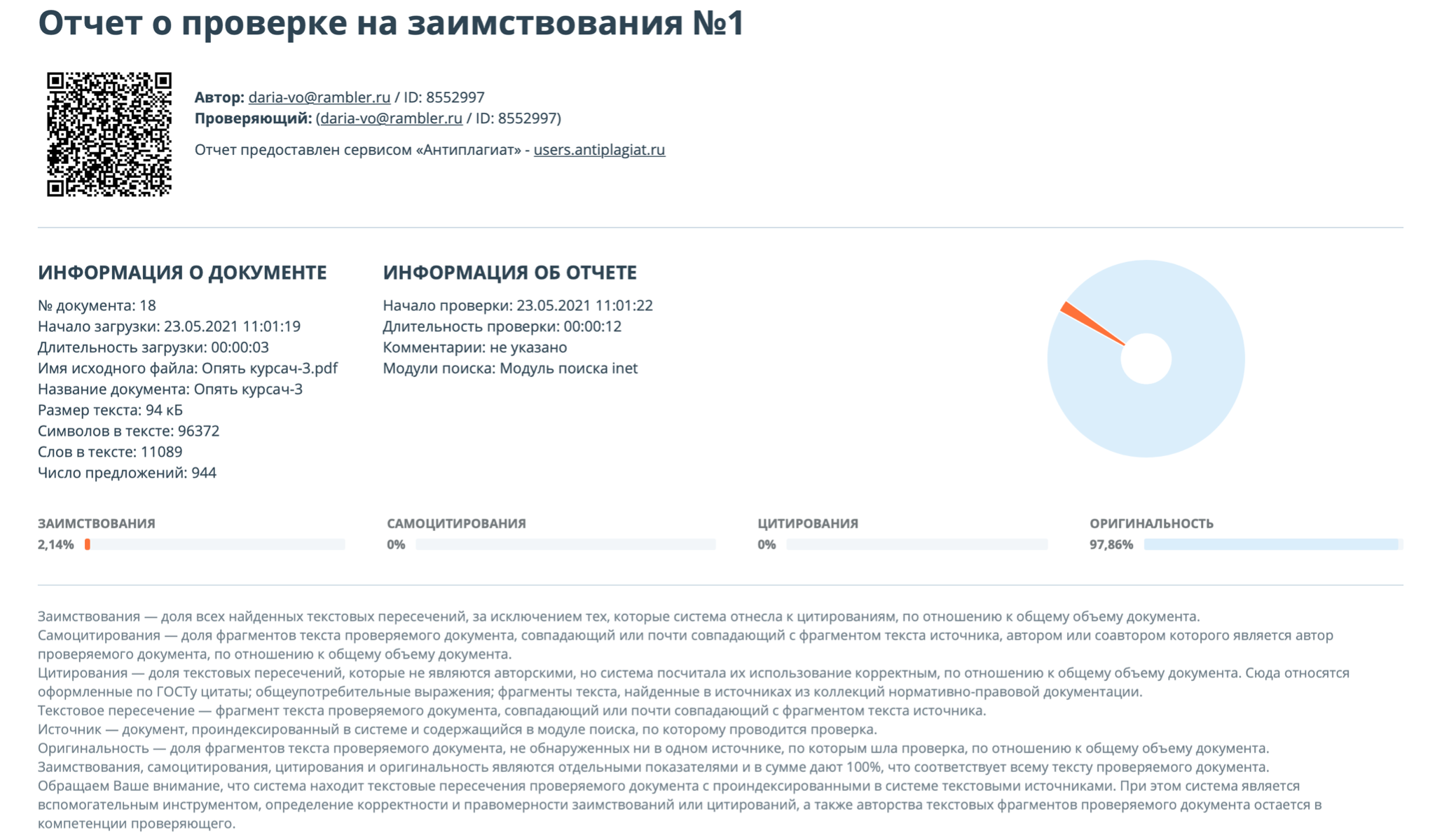
15. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. -- СПб: Питер, 2001. -- 368 с.: ил.

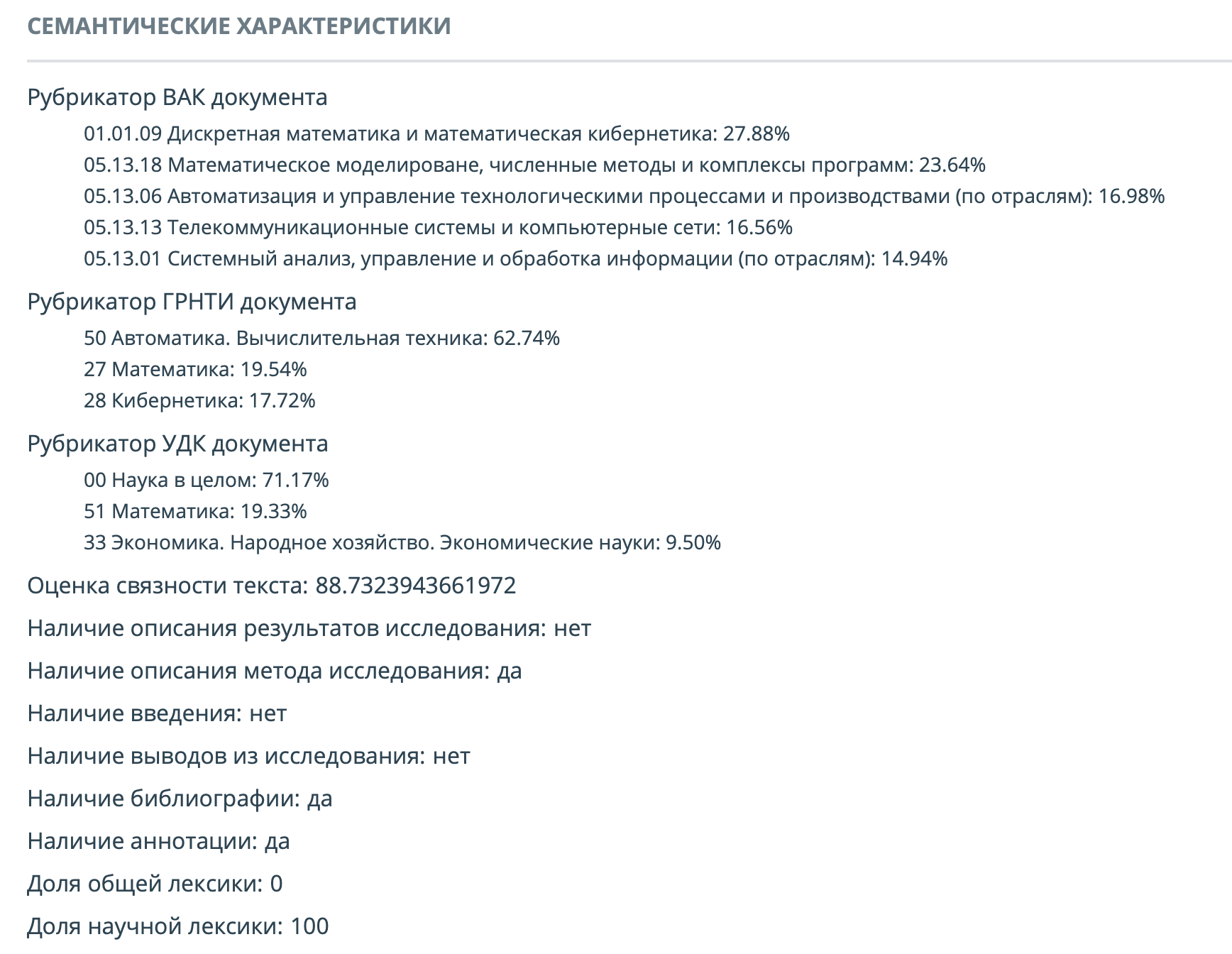
16. RefactoringGuru [Электронный ресурс] / – Режим доступа: <https://refactoring.guru/ru>. - Время доступа: 12.05.2021 - 19:08

# ПРИЛОЖЕНИЕ А

# (обязательное)

# Проверка на антиплагиат





# ПРИЛОЖЕНИЕ Б

**(обязательное)**

**Листинг кода**

/\*MedicationModel.swift\*/

import Foundation

import SwiftUI

class MedicationModel: ObservableObject, Codable{

enum CodingKeys: CodingKey{

case product\_ndc, generic\_name, labeler\_name, dosage\_form, userId, start\_date, finish\_date, doze, time, state

}

@Published var product\_ndc = ""

@Published var generic\_name = ""

@Published var labeler\_name = ""

@Published var dosage\_form = ""

@Published var userId = ""

@Published var start\_date = Date()

@Published var finish\_date = Date()

@Published var doze = ""

@Published var time = Date()

@Published var state = false

init() { }

required init(from decoder: Decoder) throws {

let container = try decoder.container(keyedBy: CodingKeys.self)

product\_ndc = try container.decode(String.self, forKey: .product\_ndc)

generic\_name = try container.decode(String.self, forKey: .generic\_name)

labeler\_name = try container.decode(String.self, forKey: .labeler\_name)

dosage\_form = try container.decode(String.self, forKey: .dosage\_form)

userId = try container.decode(String.self, forKey: .userId)

start\_date = try container.decode(Date.self, forKey: .start\_date)

finish\_date = try container.decode(Date.self, forKey: .finish\_date)

doze = try container.decode(String.self, forKey: .doze)

time = try container.decode(Date.self, forKey: .time)

state = try container.decode(Bool.self, forKey: .state)

}

func encode(to encoder: Encoder) throws {

var container = encoder.container(keyedBy: CodingKeys.self)

try container.encode(product\_ndc, forKey: .product\_ndc)

try container.encode(generic\_name, forKey: .generic\_name)

try container.encode(labeler\_name, forKey: .labeler\_name)

try container.encode(dosage\_form, forKey: .dosage\_form)

try container.encode(userId, forKey: .userId)

try container.encode(start\_date, forKey: .start\_date)

try container.encode(finish\_date, forKey: .finish\_date)

try container.encode(doze, forKey: .doze)

try container.encode(time, forKey: .time)

try container.encode(state, forKey: .state)

}

func postMethod() {

guard let url = URL(string: "https://reqres.in/api/pillquipo/api") else {

print("Error: cannot create URL")

return

}

guard let jsonData = try? JSONEncoder().encode(self) else {

print("Error: Trying to convert model to JSON data")

return

}

var request = URLRequest(url: url)

request.httpMethod = "POST"

request.setValue("application/json", forHTTPHeaderField: "Content-Type")

request.setValue("application/json", forHTTPHeaderField: "Accept") //

request.httpBody = jsonData

URLSession.shared.dataTask(with: request) { data, response, error in

guard error == nil else {

print("Error: error calling POST")

print(error!)

return

}

guard let data = data else {

print("Error: Did not receive data")

return

}

guard let response = response as? HTTPURLResponse, (200 ..< 299) ~= response.statusCode else {

print("Error: HTTP request failed")

return

}

do {

guard let jsonObject = try JSONSerialization.jsonObject(with: data) as? [String: Any] else {

print("Error: Cannot convert data to JSON object")

return

}

guard let prettyJsonData = try? JSONSerialization.data(withJSONObject: jsonObject, options: .prettyPrinted) else {

print("Error: Cannot convert JSON object to Pretty JSON data")

return

}

guard let prettyPrintedJson = String(data: prettyJsonData, encoding: .utf8) else {

print("Error: Couldn't print JSON in String")

return

}

print(prettyPrintedJson)

} catch {

print("Error: Trying to convert JSON data to string")

return

}

}.resume()

}

func getMethod() {

guard let url = URL(string: "https://reqres.in/api/pillquipo/api") else {

print("Error: cannot create URL")

return

}

var request = URLRequest(url: url)

request.httpMethod = "GET"

URLSession.shared.dataTask(with: request) { data, response, error in

guard error == nil else {

print("Error: error calling GET")

print(error!)

return

}

guard let data = data else {

print("Error: Did not receive data")

return

}

guard let response = response as? HTTPURLResponse, (200 ..< 299) ~= response.statusCode else {

print("Error: HTTP request failed")

return

}

do {

guard let jsonObject = try JSONSerialization.jsonObject(with: data) as? [String: Any] else {

print("Error: Cannot convert data to JSON object")

return

}

guard let prettyJsonData = try? JSONSerialization.data(withJSONObject: jsonObject, options: .prettyPrinted) else {

print("Error: Cannot convert JSON object to Pretty JSON data")

return

}

guard let prettyPrintedJson = String(data: prettyJsonData, encoding: .utf8) else {

print("Error: Could print JSON in String")

return

}

print(prettyPrintedJson)

} catch {

print("Error: Trying to convert JSON data to string")

return

}

}.resume()

}

func putMethod() {

guard let url = URL(string: "https://reqres.in/api/pillquipo/api2") else {

print("Error: cannot create URL")

return

}

guard let jsonData = try? JSONEncoder().encode(self) else {

print("Error: Trying to convert model to JSON data")

return

}

var request = URLRequest(url: url)

request.httpMethod = "PUT"

request.setValue("application/json", forHTTPHeaderField: "Content-Type")

request.httpBody = jsonData

URLSession.shared.dataTask(with: request) { data, response, error in

guard error == nil else {

print("Error: error calling PUT")

print(error!)

return

}

guard let data = data else {

print("Error: Did not receive data")

return

}

guard let response = response as? HTTPURLResponse, (200 ..< 299) ~= response.statusCode else {

print("Error: HTTP request failed")

return

}

do {

guard let jsonObject = try JSONSerialization.jsonObject(with: data) as? [String: Any] else {

print("Error: Cannot convert data to JSON object")

return

}

guard let prettyJsonData = try? JSONSerialization.data(withJSONObject: jsonObject, options: .prettyPrinted) else {

print("Error: Cannot convert JSON object to Pretty JSON data")

return

}

guard let prettyPrintedJson = String(data: prettyJsonData, encoding: .utf8) else {

print("Error: Could print JSON in String")

return

}

print(prettyPrintedJson)

} catch {

print("Error: Trying to convert JSON data to string")

return

}

}.resume()

}

func deleteMethod() {

guard let url = URL(string: "https://reqres.in/api/pillquipo/api1") else {

print("Error: cannot create URL")

return

}

var request = URLRequest(url: url)

request.httpMethod = "DELETE"

URLSession.shared.dataTask(with: request) { data, response, error in

guard error == nil else {

print("Error: error calling DELETE")

print(error!)

return

}

guard let data = data else {

print("Error: Did not receive data")

return

}

guard let response = response as? HTTPURLResponse, (200 ..< 299) ~= response.statusCode else {

print("Error: HTTP request failed")

return

}

do {

guard let jsonObject = try JSONSerialization.jsonObject(with: data) as? [String: Any] else {

print("Error: Cannot convert data to JSON")

return

}

guard let prettyJsonData = try? JSONSerialization.data(withJSONObject: jsonObject, options: .prettyPrinted) else {

print("Error: Cannot convert JSON object to Pretty JSON data")

return

}

guard let prettyPrintedJson = String(data: prettyJsonData, encoding: .utf8) else {

print("Error: Could print JSON in String")

return

}

print(prettyPrintedJson)

} catch {

print("Error: Trying to convert JSON data to string")

return

}

}.resume()

}

}

/\*ModelData.swift\*/

import Foundation

import SwiftUI

import Firebase

class ModelData: ObservableObject{

@Published var email = ""

@Published var password = ""

@Published var isSignUp = false

@Published var email\_SignUp = ""

@Published var password\_SignUp = ""

@Published var reEnterPassword = ""

@Published var isLinkSend = false

@Published var alert = false

@Published var alertMsg = ""

@Published var key = ""

@AppStorage("log\_Status") var status = false

@Published var isLoading = false

func resetPassword(){

let alert = UIAlertController(title: "Reset", message: "Enter your email to reset you password", preferredStyle: .alert)

alert.addTextField{ (password) in

password.placeholder = "Email"

}

let proceed = UIAlertAction(title: "Reset", style: .default){ (\_) in

if alert.textFields![0].text! != ""{

withAnimation{

self.alertMsg = "Password reset link has been sent successfully"

}

Auth.auth().sendPasswordReset(withEmail: alert.textFields![0].text!){ (err) in

withAnimation{

self.isLoading.toggle()

}

if err != nil{

self.alertMsg = err!.localizedDescription

self.alert.toggle()

return

}

self.alertMsg = "Password reset link has been sent successfully"

self.alert.toggle()

}

}

}

let cancel = UIAlertAction(title: "Cancel", style: .destructive, handler: nil)

alert.addAction(cancel)

alert.addAction(proceed)

UIApplication.shared.windows.first?.rootViewController?.present(alert, animated: true)

}

func login(){

if email == "" || password == ""{

self.alertMsg = "Fill the content properly"

self.alert.toggle()

return

}

withAnimation{

self.isLoading.toggle()

}

Auth.auth().signIn(withEmail: email, password: password) { (result, err) in

withAnimation{

self.isLoading.toggle()

}

if err != nil{

self.alertMsg = err!.localizedDescription

self.alert.toggle()

return

}

let user = Auth.auth().currentUser

if !user!.isEmailVerified{

self.alertMsg = "Verify your email address"

self.alert.toggle()

try! Auth.auth().signOut()

return

}

withAnimation{

userKey = self.email

self.status = true

print(userKey)

}

}

}

func signUp(){

if email\_SignUp == "" || password\_SignUp == "" || reEnterPassword == ""{

self.alertMsg = "Fill the content properly"

self.alert.toggle()

isLoading = false

return

}

if password\_SignUp != reEnterPassword{

self.alertMsg = "Password doesn't match"

self.alert.toggle()

isLoading = false

return

}

withAnimation{

self.isLoading.toggle()

}

Auth.auth().createUser(withEmail: email\_SignUp, password: password\_SignUp){ (result, err) in

withAnimation{

self.isLoading

}

if err != nil{

self.alertMsg = err!.localizedDescription

self.alert.toggle()

return

}

result?.user.sendEmailVerification(completion: {(err) in

if err != nil{

self.alertMsg = err!.localizedDescription

self.alert.toggle()

return

}

self.alertMsg = "Verify your email address"

self.alert.toggle()

})

}

}

func logOut(){

try! Auth.auth().signOut()

withAnimation{

self.status = false

}

email = ""

userKey = ""

password = ""

email\_SignUp = ""

password\_SignUp = ""

reEnterPassword = ""

}

}

/\*CoreDataModel.swift\*/

import Foundation

import SwiftUI

struct CoreDataModel: Decodable, Hashable{

var product\_ndc: String

var generic\_name: String

var dosage\_form: String

var userId: String

var start\_date: Date

var finish\_date: Date

var doze: String

var time: Date

var state: Bool

}

/\*CoreDataViewModel.swift\*/

import Foundation

import SwiftUI

import CoreData

class CoreDataViewModel: ObservableObject{

@Published var medications: [CoreDataModel] = []

let persistentContainer: NSPersistentContainer

init() {

persistentContainer = NSPersistentContainer(name: "HelloCoreDataModel")

persistentContainer.loadPersistentStores { (description, error) in

if let error = error {

fatalError("Core Data Store failed \(error.localizedDescription)")

}

}

}

func saveData(context: NSManagedObjectContext){

medications.forEach{ (data) in

let entity = Medication(context: context)

entity.dosage\_form = data.dosage\_form

entity.doze = data.doze

entity.finish\_date = data.finish\_date

entity.generic\_name = data.generic\_name

entity.product\_ndc = data.dosage\_form

entity.start\_date = data.start\_date

entity.state = data.state

entity.time = data.time

entity.userId = data.dosage\_form

}

do{

try context.save()

print("success")

}

catch{

print(error.localizedDescription)

}

}

func updateMovie() {

do {

try persistentContainer.viewContext.save()

} catch {

persistentContainer.viewContext.rollback()

}

}

func deleteMovie(medication: Medication) {

persistentContainer.viewContext.delete(medication)

do {

try persistentContainer.viewContext.save()

} catch {

persistentContainer.viewContext.rollback()

print("Failed to save context \(error)")

}

}

func getAllMovies() -> [Medication] {

let fetchRequest: NSFetchRequest<Medication> = Medication.fetchRequest()

do {

return try persistentContainer.viewContext.fetch(fetchRequest)

} catch {

return []

}

}

func fetchData(context: NSManagedObjectContext){

let url = URL(string: "https://reqres.in/api/pillquipo/api")

var request = URLRequest(url: URL(string: url)!)

request.addValue("swiftui2.0", forHTTPHeaderField: "field")

let session = URLSession(configuration: .default)

session.dataTask(with: request) { (data, res, \_) in

guard let jsonData = data else{return}

let response = res as! HTTPURLResponse

if response.statusCode == 404{

print("API error")

return

}

do{

let medication = try JSONDecoder().decode([CoreDataModel].self, from: jsonData)

DispatchQueue.main.async{

self.medications = medication

self.saveData(context: context)

}

}

catch{

print(error.localizedDescription)

}

}

.resume()

}

}

/\*MainListView.swift\*/

import SwiftUI

struct MainListView: View {

@Environment(\.managedObjectContext) private var viewContext

@State var countPositionsToday = false

@State var countPositions = 0

@Environment(\.managedObjectContext) var context

@State var selectedDate = dates[0]

@State var ifContent = false

@State var counter = 0

@FetchRequest(

entity: Medication.entity(),

sortDescriptors: [

NSSortDescriptor(keyPath: \Medication.time, ascending: true)

],

predicate: NSPredicate(format: "userId == %@", userKey)

) var medication: FetchedResults<Medication>

var body: some View{

ZStack{

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(0.9), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

VStack(){

if medication.isEmpty{

EmptyListView()

.padding(.bottom, 40)

}

else{

ScrollView(.vertical, showsIndicators: false) {

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(0.9), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

VStack(alignment: .center) {

ForEach(medication) { item in

let isNotEarlier = Calendar.current.compare(selectedDate, to: item.start\_date!, toGranularity: .hour)

let isNotLatter = Calendar.current.compare(selectedDate, to: item.finish\_date!, toGranularity: .hour)

if ((isNotEarlier == .orderedDescending || isNotEarlier == .orderedSame) && (isNotLatter == .orderedAscending || isNotLatter == .orderedSame)){

let today = Calendar.current.compare(selectedDate, to: Date(), toGranularity: .hour)

if today == .orderedSame {

ListElementToday(medication: item)

}

else{

ListElement(medication: item)

}

}

}

}

}

}

}.padding(.top, 160)

.padding(.bottom, 80)

VStack(){

ZStack(alignment: Alignment(horizontal: .center, vertical: .top)){

Rectangle()

.frame(width: 375, height: 160)

.foregroundColor(Color(ColorsSaved.almostWhite))

.background(Color(ColorsSaved.almostWhite))

.shadow(radius: 10)

VStack {

ScrollView(.horizontal, showsIndicators: false) {

HStack {

ForEach(dates, id: \.self) { date in

DayView(date: date, selectedDate: self.$selectedDate)

}

}

}

.padding(.bottom, 5)

SelectedDayView(selectedDate: $selectedDate)

}.padding(.top, 30)

.padding(.horizontal, 25)

}

}.padding(.bottom, 525)

}

.animation(.easeInOut)

}

func predic(){

let datePredicate = NSPredicate(format: "start\_date > %@", self.selectedDate as CVarArg)

print(datePredicate)

}

}

/\*UserInfoView.swift\*/

struct UserInfoView: View {

@ObservedObject var textBindingManager = TextBindingManager(limit: 20)

@StateObject var modeldata = ModelData()

@AppStorage(Settings.notifications) var notifications: Bool = false

@State var notificationsOn = true

@State var IsNameChange = false

@State var showList = false

@State private var showingAlert = false

static let gradientStart = Color(ColorsSaved.gradientBlue)

static let gradientEnd = Color(ColorsSaved.gradientTorquise)

var body: some View {

ZStack{

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(0.9), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

ZStack(){

NavigationLink("", destination: Detail(show: self.$notificationsOn), isActive: self.$notificationsOn)

RoundedRectangle(cornerRadius: 25.0)

.frame(width: 300, height: 400, alignment: .center)

.foregroundColor(Color(ColorsSaved.almostWhite))

.shadow(radius: 15)

ZStack{

VStack(){

ZStack{

RoundedRectangle(cornerRadius: 15.0)

.fill(LinearGradient(

gradient: .init(colors: [Self.gradientStart, Self.gradientEnd]),

startPoint: .topLeading,

endPoint: .bottomTrailing

))

.frame(width: 90, height: 90, alignment: .top)

.shadow(radius: 10 )

Image("logo")

.frame(width: 60, height: 60, alignment: .center)

}

Text(textBindingManager.text)

.font(.system(size:20))

.foregroundColor(Color(ColorsSaved.textTorquise))

.bold()

.padding(.horizontal, 50)

.padding(.bottom, 7)

Text("Change name")

.foregroundColor(Color(ColorsSaved.gradientBlue))

.font(.system(size:16))

.onTapGesture {

self.showingAlert.toggle()

print(userKey)

defineName()

}.padding(.bottom, 20)

HStack{

Toggle(isOn: $notificationsOn) {

Text("Notifications")

.foregroundColor(Color(ColorsSaved.textBlue))

}.onChange(of: notificationsOn, perform: { value in

notification = notificationsOn

})

}

.padding(.vertical, 20)

.padding(.horizontal, 60)

Button(action: {

self.showList = true

}) {

HStack(spacing: 100){

Text("List of medications")

.foregroundColor(Color(ColorsSaved.textBlue))

Image(systemName: "chevron.right")

.foregroundColor(Color(ColorsSaved.textBlue))

}

}.sheet(isPresented: $showList) {

FullListView()

}

Button(action:modeldata.logOut, label:{

Text("Quit")

.foregroundColor(.white)

.fontWeight(.bold)

.padding(.vertical)

.frame(width: UIScreen.main.bounds.width - 200)

}).background(LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(0.9), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing))

.cornerRadius(14)

.offset(y: 40)

.padding(.bottom, -40)

.shadow(radius: 15)

}.padding(.bottom,100)

}

if self.showingAlert{

AlertControlView(textString: $textBindingManager.text,

showAlert: $showingAlert,

title: "Change username",

message: "Type in a textfield below")

}

}

}

}

func defineName(){

username = textBindingManager.text

}

/\*AddPillView.swift\*/

import SwiftUI

import CodeScanner

import AVFoundation

import UserNotifications

enum ActiveAlert {

case first, second

}

struct Responses: Codable{

var results: [Resul]

}

struct Resul: Codable, Hashable {

var product\_ndc: String

var generic\_name: String

var labeler\_name: String

var dosage\_form: String

}

struct AddPillView: View {

@State var index = 0

@State var isListed = false

@State var result = MedicationModel()

@State private var isShowingScanner = false

@State var text = ""

var body: some View {

ZStack{

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(1), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

VStack{

HStack(){

Button(action: {

withAnimation(.spring(response: 0.8, dampingFraction: 0.5, blendDuration: 0.5)){

self.index=0

}

}){

Text("Custom")

.font(Font.custom("Karla-Bold", size: 12))

.padding(.vertical, 10)

.frame(width: (UIScreen.main.bounds.width - 80) / 3)

.foregroundColor(self.index == 0 ? .black: .white)

}.background(self.index == 0 ? Color.white: Color.clear)

.clipShape(Capsule())

Button(action: {

withAnimation(.spring(response: 0.8, dampingFraction: 0.5, blendDuration: 0.5)){

self.index = 1

self.isShowingScanner = true

}

}){

Text("Scan barcode")

.font(Font.custom("Karla-Bold", size: 12))

.padding(.vertical, 10)

.frame(width: (UIScreen.main.bounds.width - 80) / 3)

.foregroundColor(self.index == 1 ? .black: .white)

}.background(self.index == 1 ? Color.white : Color.clear)

.clipShape(Capsule())

Button(action: {

withAnimation(.spring(response: 0.8, dampingFraction: 0.5, blendDuration: 0.5)){

self.index=2

}

}){

Text("Search")

.font(Font.custom("Karla-Bold", size: 12))

.padding(.vertical, 10)

.frame(width: (UIScreen.main.bounds.width - 80) / 3)

.foregroundColor(self.index == 2 ? .black: .white)

}.background(self.index == 2 ? Color.white: Color.clear)

.clipShape(Capsule())

}.background(Color.black.opacity(0.1))

.clipShape(Capsule())

.padding(.bottom, 20)

if self.index == 0 {

addCustomPill(name: result.generic\_name, dosage: result.dosage\_form, dosage\_form: "Capsule", start\_date:Date(), finish\_date:Date())

} else if self.index == 1 {

addByScanner()

.frame(width: 375, height: 480, alignment: .center)

}

else {

addBySearch(text: text)

}

if isListed{

MainTabView()

}

}

}

}

}

struct addCustomPill: View{

@Environment(\.managedObjectContext) private var viewContext

@State var selected = 0

@State private var activeAlert: ActiveAlert = .first

@State private var showAlert = false

@StateObject var medication = MedicationModel()

@State var medications: [Medication] = [Medication]()

@State var name: String

@State var dosage: String

@State var dosage\_form: String

@State var start\_date: Date

@State var finish\_date: Date

let lightGreyColor = Color(red: 239.0/255.0, green: 243.0/255.0, blue: 244.0/255.0, opacity: 1.0)

@State var doze = 1

var types: [String] = ["Capsule", "Tablet", "Injection", "Gel", "Concentrate", "Liquid", "Aerosol", "Cream", "Spray", "Suspension", "Other"]

@State var selection: String = "Capsule"

var dateFormatter: DateFormatter {

let formatter = DateFormatter()

formatter.dateStyle = .long

return formatter

}

var body: some View{

ZStack{

ZStack{

RoundedRectangle(cornerRadius: 25.0)

.foregroundColor(Color(ColorsSaved.almostWhite))

.frame(width: 360, height: 480, alignment: .center)

VStack(alignment: .leading){

Text("Add new medicine")

.font(Font.custom("Karla-Bold", size: 20))

.foregroundColor(Color(ColorsSaved.textBlue))

.padding(.leading, 20)

TextField("Medication name", text: $name)

.padding(.all, 10)

.foregroundColor(Color(ColorsSaved.textBlue))

.background(lightGreyColor)

.cornerRadius(5.0)

.padding()

HStack{

TextField("Enter doze...", text: $medication.doze)

.frame(width: 100, height: 20)

.padding(.all, 10)

.foregroundColor(Color(ColorsSaved.textBlue))

.background(lightGreyColor)

.cornerRadius(5.0)

HStack(spacing: 0){

Picker(self.dosage\_form, selection: self.$dosage\_form) {

ForEach(types, id: \.self) { word in

Text(word)

.underline()

.font(Font.custom("Karla-Bold", size: 10))

.tag(word)

.frame(width: 100)

}

}.foregroundColor(Color(ColorsSaved.textBlue))

.frame(width: 100)

.font(Font.custom("Karla-Bold", size: 18))

.pickerStyle(MenuPickerStyle())

Image(systemName:"chevron.down")

}

}.padding(.horizontal, 15)

.padding(.bottom, 10)

DatePicker("Choose start date",selection: $medication.start\_date, displayedComponents: .date)

.foregroundColor(Color(ColorsSaved.textBlue))

.padding(.horizontal, 20)

DatePicker("Choose finish date",selection: $medication.finish\_date, displayedComponents: .date)

.foregroundColor(Color(ColorsSaved.textBlue))

.padding(.horizontal, 20)

DatePicker("Choose time",selection: $medication.time, displayedComponents: .hourAndMinute)

.foregroundColor(Color(ColorsSaved.textBlue))

.padding(.horizontal, 20)

Button(action: {

if name == "" || medication.doze == "" || medication.start\_date >= medication.finish\_date{

self.activeAlert = .first

}

else{

self.activeAlert = .second

medication.generic\_name = name

let medicationData = Medication(context: viewContext)

medicationData.generic\_name = name

medicationData.dosage\_form = dosage\_form

medicationData.doze = medication.doze

medicationData.start\_date = medication.start\_date

medicationData.finish\_date = medication.finish\_date

medicationData.state = false

medicationData.time = medication.time

medicationData.userId = userKey

let calendar = Calendar.current

let hour = calendar.component(.hour, from: medication.time)

let minutes = calendar.component(.minute, from: medication.time)

self.setUpLocalNotification(hour: hour, minute: minutes, date: medication.start\_date)

do {

try viewContext.save()

print("Order saved.")

} catch {

print(error.localizedDescription)

}

name = ""

dosage = ""

medication.finish\_date = Date()

medication.start\_date = Date()

medication.doze = ""

}

self.showAlert = true

})

{

Text("Add")

.foregroundColor(.white)

.fontWeight(.bold)

.padding(.vertical)

.frame(width: UIScreen.main.bounds.width - 200)

}.alert(isPresented:$showAlert) {

switch activeAlert{

case .first:

return Alert(

title: Text("Invalid input!"),

message: Text("Fill content properly"),

dismissButton: .default(Text("Got it!"))

)

case .second:

return Alert(

title: Text("A new medication has been added successfully"),

message: Text("Scheduled medications are shown on home screen"),

dismissButton: .default(Text("Got it!"))

)

}

}

.background(LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(0.9), Color(ColorsSaved.gradientTorquise).opacity(0.9)]), startPoint: .topLeading, endPoint: .bottomTrailing))

.cornerRadius(14)

.offset(y: 15)

.padding(.leading, 100)

.shadow(radius: 15)

}.padding(.bottom, 50)

}.padding(.horizontal, 20)

.cornerRadius(25)

}

}

func setUpLocalNotification(hour: Int, minute: Int, date: Date) {

if(date <= Date() || notification){

let calendar = NSCalendar(identifier: .gregorian)!;

var dateFire = Date()

var fireComponents = calendar.components( [NSCalendar.Unit.day, NSCalendar.Unit.month, NSCalendar.Unit.year, NSCalendar.Unit.hour, NSCalendar.Unit.minute], from:dateFire)

if (fireComponents.hour! > hour

|| (fireComponents.hour == hour && fireComponents.minute! >= minute) ) {

dateFire = dateFire.addingTimeInterval(86400)

fireComponents = calendar.components( [NSCalendar.Unit.day, NSCalendar.Unit.month, NSCalendar.Unit.year, NSCalendar.Unit.hour, NSCalendar.Unit.minute], from:dateFire);

}

fireComponents.hour = hour

fireComponents.minute = minute

dateFire = calendar.date(from: fireComponents)!

let localNotification = UNUserNotificationCenter.current()

let content = UNMutableNotificationContent()

content.title = "PillQuipo"

content.subtitle = "Hey,\(username) it's time to take care of your health"

content.sound = UNNotificationSound.default

var dateComponents = DateComponents()

dateComponents.hour = hour

dateComponents.minute = minute

let trigger = UNCalendarNotificationTrigger(dateMatching: dateComponents, repeats: false)

let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: trigger)

localNotification.add(request)

}

}

}

struct addByScanner: View{

@State var isPresentingScanner = false

@State var scannedCode: String?

@State private var isShowingScanner = false

@State var medication = MedicationModel()

@State var result = [Resul]()

@State var showingAlert = false

@State var name = ""

@State var form = ""

@FetchRequest(

entity: DataForScan.entity(),

sortDescriptors: [

NSSortDescriptor(keyPath: \DataForScan.generic\_name, ascending: true)

]

) var medications: FetchedResults<DataForScan>

var body: some View {

ZStack{

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(1), Color(ColorsSaved.gradientTorquise).opacity(1)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

VStack{

ZStack() {

CodeScannerView(

codeTypes: [.qr, .ean13, .code128],

completion: { result in

if case let .success(code) = result {

self.scannedCode = code

print(self.medications)

for item in medications{

if item.product\_ndc == scannedCode{

print(scannedCode!)

form = item.dosage\_form!

name = item.generic\_name!

}

} }

}

)

if name != ""{

ZStack{

LinearGradient(gradient: Gradient(colors: [Color(ColorsSaved.gradientBlue).opacity(1), Color(ColorsSaved.gradientTorquise).opacity(1.0)]), startPoint: .topLeading, endPoint: .bottomTrailing).edgesIgnoringSafeArea(.all)

addCustomPill(name: name, dosage: "", dosage\_form: form, start\_date: Date(), finish\_date: Date())

}

}

func data(){

for item in medications{

if item.product\_ndc == scannedCode{

print(scannedCode!)

medication.dosage\_form = item.dosage\_form!

medication.generic\_name = item.generic\_name!

}

}

}

}

struct addBySearch: View{

@State var result = [Resul]()

@State var text: String

@State var results = MedicationModel()

@State var searchText = ""

@State var isSearching = false

@State var hideBar = false

@Environment(\.managedObjectContext) var viewContext

@FetchRequest(

sortDescriptors: [NSSortDescriptor(keyPath: \DataForScan.generic\_name, ascending: true)],

animation: .default)

private var items: FetchedResults<DataForScan>

var body: some View {

ZStack(alignment: .top){

RoundedRectangle(cornerRadius: 25.0)

.foregroundColor(Color(ColorsSaved.almostWhite))

.frame(width: 360, height: 480, alignment: .center)

NavigationView {

List{

SearchBar(searchText: $searchText, isSearching: $isSearching)

.frame(alignment: .center)

VStack{

if result.isEmpty{

LoadingView()

.padding(.top, 100)

}

List((result).filter({ "\($0)".contains(searchText) || searchText.isEmpty }), id: \.self) { item in

VStack(alignment: .leading){

NavigationLink(destination: addCustomPill(name: item.generic\_name, dosage: item.dosage\_form, dosage\_form: item.dosage\_form, start\_date: Date(), finish\_date: Date())) {

Text("\(item.generic\_name), \(item.labeler\_name)")

.font(Font.custom("Karla-Bold", size: 15))

}.navigationBarHidden(true)

.foregroundColor(Color(ColorsSaved.textBlue))

}.navigationBarHidden(true)

}.frame(width: 330, height: 470)

.navigationBarHidden(true)

} .navigationBarHidden(true)

} .navigationBarTitle("")

.navigationBarHidden(true)

}.onAppear(perform: loadData)

.navigationBarHidden(true)

.padding(.top, 5)

.cornerRadius(25)

.frame(width: 350, height: 475, alignment: .center)

.foregroundColor(Color(ColorsSaved.almostWhite))

}

}

func loadData(){

guard let url = URL(string: "https://api.fda.gov/drug/ndc.json?limit=1000")

else{

print("Invalid URL")

return

}

let request = URLRequest(url: url)

URLSession.shared.dataTask(with: request){ data, response, error in

if let data = data{

if let decodedResponse = try? JSONDecoder().decode(Responses.self, from: data){

DispatchQueue.main.async{

self.result = decodedResponse.results

}

return

}

}

print("Fetch failed")

}.resume()

}

}

/\*UserController.swift\*/

final class TodoController: RouteCollection {

func boot(router: Router) throws {

let usersRoute = router.grouped("api", "users")

usersRoute.get(User.parameter, use: getOneHandler)

usersRoute.post(use: createHandler)

usersRoute.put(User.parameter, use: updateHandler)

usersRoute.delete(User.parameter, use: deleteHandler)

}

func getOneHandler(\_ req: Request) throws -> Future<User> {

return try req.parameters.next(User.self)

}

func createHandler(\_ req: Request) throws -> Future<User> {

return try req.content.decode(User.self).flatMap { (user) in

return user.save(on: req)

}

}

func updateHandler(\_ req: Request) throws -> Future<User> {

return try flatMap(to: User.self, req.parameters.next(User.self), req.content.decode(User.self)) { (user, updatedUser) in

user.key = updatedUser.key

user.id = updatedUser.id

user.name = updatedUser.name

return user.save(on: req)

}

}

func deleteHandler(\_ req: Request) throws -> Future<HTTPStatus> {

return try req.parameters.next(User.self).flatMap { (user) in

return user.delete(on: req).transform(to: HTTPStatus.noContent)

}

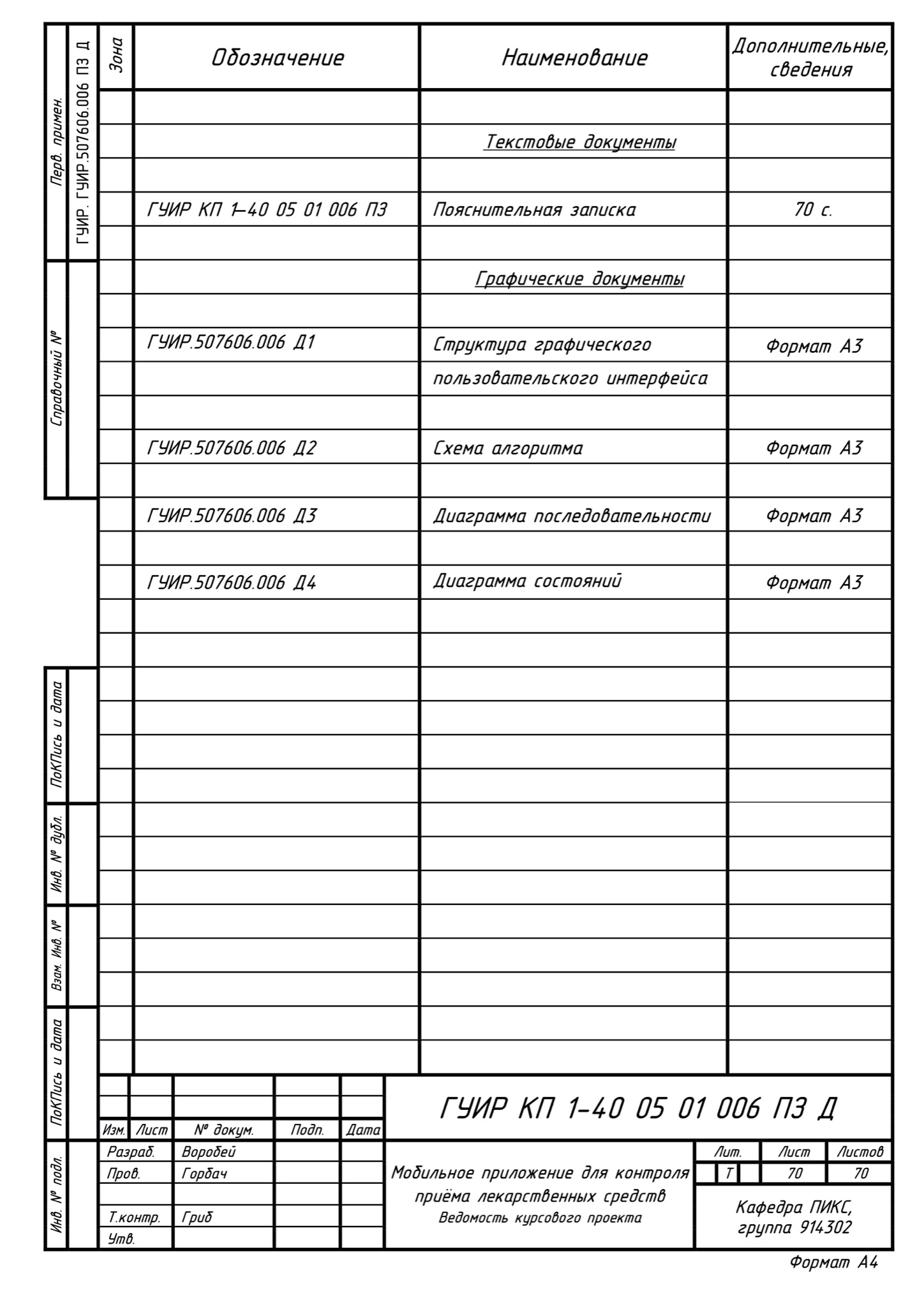
}

}

# ПРИЛОЖЕНИЕ В

# (обязательное)

# Ведомость документов



# 