Министерство науки и высшего образования РФ
Федеральное государственное автономное образовательное учреждение высшего образования
«Пермский государственный национальный исследовательский университет»

Институт компьютерных наук и технологий

Разработка нативных мобильных приложений для iOS: современное состояние, проблемы и решения

Работу выполнил
студент группы ФИТ-2-2024 НМ 1 курса
Гурьянов Д.П.
«26» августа 2025 г.

Работу проверил
Юрков К.А.,
Старший преподаватель ИКНТ
«___» _____ 2025 г.

Пермь 2025

1. Обзор платформы

1.1. Доля рынка и распространение iOS

По состоянию на 2025 год рынок мобильных платформ фактически поделен между Android и iOS. iOS стабильно занимает второе место по глобальной доле рынка – порядка 25-30% устройств работают на iOS [1]. При этом в некоторых регионах (например, в Северной Америке) iOS опережает Android по популярности.

В экосистеме Apple наблюдается низкая фрагментация версий операционных систем: пользователи быстро обновляются до последних версий iOS. Так, по официальным данным Apple, 82% всех iPhone в июне 2025 года работали под управлением iOS 18 – самой свежей на текущий момент версии системы [2]. Быстрая миграция на новые версии означает, что разработчикам намного проще поддерживать актуальность приложений и не приходится долго учитывать устаревшие версии iOS.

1.2. Дизайн и принципы интерфейса

Важной чертой iOS является тщательно проработанный язык дизайна и единые рекомендации Apple по интерфейсам – Human Interface Guidelines. Apple уделяет особое внимание консистентности и удобству приложений, поэтому Human Interface Guidelines задает основные принципы: ясность, дифференциация, глубина и единообразие интерфейса [3].

Ясность подразумевает четкость текста и элементов, понятные иконки и прозрачную навигацию. Дифференциация означает, что интерфейс «уступает» контенту – оформление должно быть минималистичным, с акцентом на содержание. Глубина достигается использованием визуальных слоев, теней и анимаций – это создает ощущение многослойности и помогает пользователю ориентироваться [3]. Последовательность же требует единых шаблонов во всем приложении – одинакового поведения элементов, единого стиля оформления экранов, одинаковой реакции на пользовательский ввод.

2. Среда разработки

2.1. Технологический стек iOS

Современная разработка на iOS сосредоточена вокруг языка программирования Swift и IDE Xcode. Язык Swift, представленный в 2014 году, за последнее десятилетие практически вытеснил Objective-C для новых проектов. По оценкам, более 90% новых проектов под iOS пишутся на Swift [4], делая его основным языком экосистемы. Swift привлекает разработчиков современным синтаксисом, безопасностью (строгая типизация,

работа с опциональными значениями и пр.) и высокой производительностью, в то время как Objective-C сохраняет значение лишь для поддержки проектов с legacy-кодом.

Помимо языка программирования, Apple предоставляет IDE Xcode, в которой выполняется все – от написания кода до отладки и сборки приложений. Xcode содержит симуляторы устройств, инструменты профилирования и тестирования, а также интеграцию с системой контроля версий.

2.2. Инструменты UI и фреймворки

С 2019 года в распоряжении iOS-девелоперов появился декларативный фреймворк SwiftUI, который значительно упрощает создание адаптивного интерфейса. SwiftUI позволяет описывать UI декларативно посредством структур View и привязки данных через property wrappers. Система сама отслеживает изменения состояния, обновляя интерфейс. Данный подход сильно контрастирует с традиционным UIKit, где интерфейс строился императивно или в Interface Builder.

На практике современные проекты все чаще используют SwiftUI – благодаря более компактному коду и возможности разрабатывать максимально адаптивные интерфейсы. Тем не менее, UIKit по-прежнему актуален, особенно если требуется полная, тонкая кастомизация или поддержка старых версий iOS. Чаще всего в одном приложении существуют и SwiftUI, и UIKit.

Кроме того, Apple предлагает широкий набор фреймворков для реализации функциональности: SwiftData для локальной базы данных, AVFoundation lля работы с мультимедиа, Core Animation для анимаций, Core Location для геолокации и многие другие. Этот богатый стек нативных фреймворков – одно из преимуществ платформы iOS, позволяющее реализовать сложные функции с минимальным написанием низкоуровневого кода.

2.3. Управление зависимостями и сборка

В последние годы основным решением стал Swift Package manager (SPM), встроенный прямо в Xcode. SPM обеспечивает простое подключение пакетов и их обновление. Apple активно продвигает SPM, и сейчас многие известные библиотеки (Alamofire, SwiftLint и др.) доступны через Swift Packages. Процесс сборки приложений полностью управляется Xcode: он компилирует код, линкует фреймворки, осуществляет кодовую подпись и упаковку. Благодаря интеграции всех шагов в IDE, конфигурация сборки (таргеты, схемы, профили) задается графически или через проектный файл .xcodeproj/.xcworkspace.

2.4. Архитектурные шаблоны и лучшая практика

С течением времени iOS сообщество разработало ряд архитектурных паттернов, помогающих поддерживать кодовую базу большого проекта. Долгое время стандартом де-факто был шаблон MVC (Model-View-Controller), рекомендуемый в официальной документации Apple. Контроллеры представления UIKit (UIViewController) стали центром приложения, что нередко приводило к проблеме «Massive View Controller», когда контроллер перегружен бизнес-логикой.

В ответ получили популярность более разделенные архитектуры – в первую очередь MVVM (Model-View-ViewModel) в сочетании с координирующим слоем. MVVM выделяет презентационную логику в отдельный класс ViewModel, который управляет состоянием UI и взаимодействует с моделью данных, а View лишь отображает привязанные к нему свойства. Этот шаблон замечательно сочетается с реактивными подходами и особенно уместен при использовании фреймворка SwiftUI, где структура View играет роль «View», а ViewModel – объект ObservableObject с свойствами @Published.

Дополнительно в сложных приложениях применяется паттерн координаторов: Coordinator берет на себя навигацию между экранами, создавая и показывая контроллеры, тем самым разгружая сами UIViewController от ответственности переходов. В итоге контроллеры остаются «глупыми» отображателями, ViewModel – поставщиками данных, а координаторы – навигаторами. Такой подход доказал эффективность: по оценкам, порядка 30% средних и крупных iOS-приложений сейчас используют архитектуру MVVM + Coordinator [5], так как она облегчает масштабирование приложения и поддержку кода. Кроме MVVM, в индустрии встречаются и другие паттерны – MVP, VIPER, Clean Architecture – но они менее распространены из-за сложности внедрения и избыточности для большинства задач.

Лучшими практиками в iOS-девелопменте сегодня считаются: разделение ответственности (Single Responsibility Principle для компонентов), отказ от больших монолитных классов, использование протоколов для абстракций (Dependency Injection по возможности), реактивное программирование при сложных асинхронных взаимодействиях.

3. Развертывание и дистрибуция

3.1. Публикация в App Store

Финальной стадией разработки является подготовка приложения к публикации в App Store – официальном и, на текущий момент, единственным магазином приложений для iOS. Apple предъявляет строгие требования к безопасности и качеству приложений.

Во-первых, каждое приложение должно быть подписано действительным сертификатом Apple. Подпись кода гарантирует, что приложение выпущено идентифицированным разработчиком и не было изменено посторонними – iOS просто не запустит бинарник без корректной подписи.

Во-вторых, перед публикацией приложение проходит модерацию (App Review): специалисты Apple проверяют его на соответствие правилам (контент, работоспособность, отсутствие вредоносного кода). Причем для получения девелоперского сертификата необходимо вступить в программу Apple Developer Program, членство в которой стоит 99 долларов в год, где личность или организация разработчика официально верифицируется Apple. Таким образом, экосистема iOS построена по принципу «walled garden»: только санкционированные Apple разработчики могут распространять приложения, и все попадающие в App Store программы проверены и привязаны к конкретным аккаунтам.

Процесс публикации включает подготовку маркетинговой страницы приложения в App Store Connect (иконки, скриншоты, описание и т.д.), загрузку сборки (.ipa файла) и ожидание проверки. После одобрения приложение становится доступным пользователям в App Store.

3.2. Тестирование и TestFlight

Перед широкой публикой разработчики часто распространяют сборки для ограниченного тестирования. Apple предоставляет удобный инструмент для этого – TestFlight. TestFlight интегрирован в App Store Connect и позволяет раздавать бета-версии без публикации в общий доступ. Разработчик может пометить сборку как бета и пригласить тестеров по электронной почте или через публичную ссылку. На одну программу допускается до 10 тысяч внешних тестеров [6]. TestFlight облегчает сбор отзывов: тестеры могут прямо на устройстве отправлять фидбек и скриншоты, а в случае краша система пришлет отчет разработчику.

3.3. Корпоративное развертывание (Enterprise)

Помимо App Store, Apple предоставляет канал дистрибуции приложений для внутренних нужд компаний. Если организация хочет создавать приватные приложения для своих сотрудников (например, корпоративные мессенджеры), она может вступить в Apple Developer Enterprise Program (ADEP) [7]. В рамках ADEP компания получает специальный enterprise-сертификат и профили, с которыми можно подписывать приложения для неограниченного числа устройств внутри организации [8]. Подобные приложения не публикуются в App Store, а устанавливаются напрямую.

Механизм подписи при этом различается, устройства, на которые ставится enterprise-приложение, должны доверять профилю компании – пользователь вручную должен подтвердить доверие к корпоративному сертификату. Любое подобное приложение все так же подписано выданным Apple сертификатом, просто проверяется принадлежность не к App Store, а к соответствующей организации.

3.4. Непрерывная интеграция и доставка (CI/CD)

Современные практики разработки мобильных приложений включают автоматизацию сборки, тестирования и развертывания. Continuous Integration/Continuous Delivery широко используется и в iOS-проектах, позволяя регулярно собирать приложение и выпускать обновления с минимальными затратами времени [9]. Благодаря CI серверу каждая новая версия кода автоматически проходит сборку и запуск тестов, гарантируя, что ошибки компиляции или регрессии обнаруживаются сразу, а не в конце цикла разработки. Настроенный пайплайн может автоматически разворачивать сборку: например, после прохождения всех тестов загружать .ipa на TestFlight или даже в App Store. Лучшими практиками CI/CD для iOS являются: быстрые сборочные пайплайны (параллельное выполнение задач, кэширование зависимостей, чтобы каждый коммит собирался быстрее), ранний запуск статического анализа и тестов (например, линтеры SwiftLint, юнит-тесты на каждом коммите), управление конфигурациями сборок (debug/release и различные таргет окружения) и обеспечение безопасности (хранение сертификатов и профилей подписи в защищенном хранилище CI)

4. Проблемы платформы и решения

4.1. Фрагментация устройств

В экосистеме iOS Apple контролирует как аппаратную часть, так и программную, поэтому фрагментация в ней существенно ниже, чем у Android. Количество моделей iPhone ограничено, и Apple стремится поддерживать актуальные устройства обновлениями ОС как можно дольше (порядка 5-6 лет обновлений в среднем). Тем не менее разработчики iOS сталкиваются с задачей адаптации приложения под различные размеры экранов. Линейка iPhone включает компактные модели (например, iPhone SE 3 с экраном 4,7 дюймов) и большие (iPhone 17 Pro Max с экраном 6,9 дюймов), с разными соотношением сторон (от классических 16:9 до более вытянутых) и разной плотностью пикселей.

Задача разработчика – сделать интерфейс гибко адаптируемым. Предлагает для этого довольно мощные инструменты: Auto Layout (набор констрейнтов, позволяющих задавать

относительное положение элементов, растяжение и отступы от краев экрана) и Size Classes (условные классы размеров – Compact или Regular по ширине и высоте).

Фактически, верстка UI в iOS производится не в абсолютных координатах, а через правила, что позволяет одному и тому же экрану корректно отображаться на экранах разных размеров и разрешений.

Помимо этого, следует добавить, что SwiftUI изначально построен с упором на адаптивность: интерфейс описывается декларативно, и система сама адаптирует его к различным размерам (разработчик может лишь указать адаптивные стейки, геометрии и условия для разных классов размеров).

Практические приемы включают использование безопасных зон (Safe Area) – область экрана вне системных элементов вроде «челки» и Dynamic Island. Например, стандартные контроллеры UITableView, UICollectionView автоматически подстраивают прокрутку и расположение под размер; текстовые элементы могут менять шрифт согласно настройках Dynamic Type пользователя.

Адаптивный дизайн iOS во многом облегчен тем, что Apple контролирует дизайны устройств – т.е. нет сотен вариаций, как в Android, поэтому достаточно проверить приложение на нескольких устройствах (например, маленький iPhone SE3, средний iPhone 17 и большой iPhone 17 Pro Max). Кроме размеров, устройства отличаются и железом: процессорной мощностью и объемом памяти. Это требует оптимизации производительности – например, графические эффекты и анимации (особенно это актуально сейчас с новой дизайн-философией Liquid Glass) должны автоматически деградировать на старых моделях, чтобы сохранять плавность работы. Apple часто внедряет API для оптимизации: например, Metal Perfomance Shaders могут использоваться для тяжелых графических расчетов, а где недоступно – библиотека сама переключается на CPU.

4.2. UI-компоненты и адаптивный дизайн

Apple снабжает разработчиков обширной библиотекой стандартных элементов: кнопки, переключатели, списки, навигационные бары, таб-бары и др. Компоненты изначально спроектированы таким образом, чтобы выглядеть и работать оптимально на разных устройствах. Например, UINavigationController на iPhone показывает контроллеры во весь экран, а на iPad по умолчанию использует форму popover или split-view для деталей. Используя стандартные контроллеры, разработчик делегирует Apple заботу о корректном отображении. Также Human Interface Guidelines устанавливает базовые паттерны адаптивного дизайна: например, рекомендует на iPad вместо меню во весь экран использовать сайдбар, показывать больше контента на экране. С выходом iPadOS Apple

даже отделила некоторые аспекты интерфейса – например, виджет DatePicker на iPhone показывается модально на весь экран, а на iPad как выпадающее окно.

Подводя итог, хотя фрагментация iOS устройств минимальна, как у Android, ей следует уделять внимание. Решения заключаются в адаптивном дизайне с помощью Auto Layout/SwiftUI, использовании стандартных компонентов, тщательном тестировании на разных моделях и оптимизации для менее мощных устройств. Следуя рекомендациям Apple и применяя принцип «responsive design», можно разработать нативное iOS-приложение, которое будет выглядеть и работать корректно на всем спектре поддерживаемых устройств.

5. Интеграция с оборудованием

5.1. Датчики и возможности устройств

Одним из преимуществ нативной разработки под iOS является прямой доступ к разнообразным аппаратным сенсорам и функциям устройств Apple. iPhone оснащены огромным множеством датчиков: GPS-модуль, акселерометр, гироскоп, датчики приближения и освещенности, а также камеры, микрофоны, Face ID [11]. iOS предоставляет унифицированные API для работы с ними: Core Location для получения геопозиции, Core Motion для движения и ориентации, AVFoundation и UIKit AV для камеры и микрофона, Core Bluetooth для работы с BLE-устройствами, HealthKit и Core ML для доступа к специфичным сенсорам здоровья и нейросетевым функциям. Интеграция с аппаратными возможностями обычно сводится к подключению соответствующего фреймворка и вызову высокоуровневых методов. Например, чтобы отслеживать GPS-координаты, достаточно создать CLLocationManager и запросить обновления локации – система сама воспользуется GPS, Wi-Fi или сотовыми данными для определения местоположения.

5.2. Работа с разрешениями

Доступ к чувствительным датчикам требует явного разрешения пользователя. Приложение должно описать в Info.plist цель использования, а при первом обращении к API система автоматически покажет диалог запроса доступа (локации, камеры, микрофона, контактов и др.). Разработчик обязан быть готов к отказу пользователя и к тому, что API вернет ошибку. Лучше всего запрашивать разрешение у пользователя только тогда, когда это действительно необходимо для функции.

5.3. Батарея и эффективное использование датчиков

Взаимодействие с аппаратными компонентами часто связано с расходом энергии. Например, непрерывное получение GPS-координат быстро сажает батарею, постоянное считывание акселерометра нагружает CPU, а фоновая работа Wi-Fi/Bluetooth препятствует

уходу устройства в глубокий сон. Поэтому эффективное использование сенсоров – критически важная задача. Разработчики должны обрабатывать данные датчиков экономно, опрашивая их не чаще необходимого и выключая обновления, когда они не нужны [11]. Apple предоставляет альтернативные режимы, снижающие энергопотребление: вместо постоянного GPS можно использовать Significant Location Change (обновления только при смещении на >500 м) или режим Visits (фиксирует прибытие/отбытие в определенное место). Вместо частого опроса акселерометра лучше полагаться на push-методы Core Motion с определенным интервалом обновлений.

5.4. Аппаратные возможности и управление ими

Некоторые функции требуют не только разрешения, но и особых настроек: приложение может работать в фоновом режиме с местоположением или воспроизведением аудио, но для этого в возможностях надо включить соответствующие Background Modes. Злоупотребление ими недопустимо – Apple проверяет, чтобы фоновая работа была оправданна (например, плеер или навигатор). Работа с фоновыми задачами на iOS организована через механизмы UIApplication (для коротких задач при сворачивании) и BGTaskScheduler (для долгих фоновых обновлений по расписку), но система строго лимитирует фоновые процессы – чтобы беречь аккумулятор, iOS может приостанавливать приложение через несколько минут фоновой активности, если оно не относится к разрешенным категориям (навигация, аудио, VoIP). Для длительных операций (загрузки файлов, синхронизация) Apple рекомендует использовать URLSession с background configuration, тогда передача продолжится даже после ухода приложения в фон, под контролем системы.

6. Безопасность

6.1. Аспекты безопасности на iOS

Закрытая природа iOS обеспечивает высокую базовую безопасность – приложения запускаются в песочнице, изолированы друг от друга, система шифрует пользовательские данные на диске. Темн е менее, мобильные приложения обрабатывают массу чувствительных данных, поэтому безопасность – ключевой фактор успеха. В 2025 году количество атак на мобильные приложения выросло более чем на 80% по сравнению с предыдущим годом [12]. Даже при жестком контроле App Store, исследования показывают, что многие iOS приложения имеют критические уязвимости – захардкоденные пароли, слабое шифрование, неправильные настройки разрешений, небезопасное хранение данных и т.д. [12].

6.2. Безопасное хранение данных

Мобильные приложения часто нуждаюбтся в хранении конфиденциальной информации: токены сессий, пароли, личные данные пользователя. Категорически запрещено сохранять такие сведения в plain тексте, как, например, в UserDefaults или файлах. Вместо этого следует использовать Keychain – систему безопасного хранилища Apple. Keychain – стандартный механизм безопасного хранения чувствительных данных в iOS [12]. По сути, Keychain является зашифрованной базой данных, управляемой системой: каждое приложение имеет в ней свой изолированный раздел, защищенный аппаратно, Все записи Keychain шифруются алгоритмом AES-256-GCM с аппаратным ключом, хранящимся в Secure Enclave (специальном доверенном чипе) [12]. Разработчик может сохранять пароли, ключи шифрования, токены в Keychain, задавая политики доступа. При правильном использовании Keychain практически исключает возможность компрометации данных – даже при полном дампе памяти или взломе приложения расшифровать данные невозможно без ключа Secure Enclave. В дополнение к Keychain, Apple предлагает API для шифрования данных – CryptoKit, позволяющий легко выполнять крипто-операции с аппаратным ускорением. Однако управление ключами все равно желательно доверить Keychain, а не хранить их в коде или файлах [12]. Помимо паролей, стоит думать о безопасности любой личной информации: если приложение кеширует какую-то часть пользовательских данных на диск, стоит пометить файл атрибутом .completeFileProtection (шифрование на диске до разблокировки устройства). Также надо очищать чувствительные данные из памяти и избегать попадания их в логи или бэкапы.

6.3. Безопасность сетевого взаимодействия

Практически любое мобильное приложение общается с сервером, поэтому важна защита данных в транзите. Минимальное требование – использование шифрования трафика TLS (HTTPS) для всех подключений. С 2016 года Apple ввела политику App Transport Security (ATS), требующую, чтобы все запросы шли по HTTPS, отключение ATS требует обоснования и Apple может отклонить приложение за небезопасный трафик.

Кроме шифрования канала, стоит продумать аутентификацию и авторизацию: использовать токены вместо постоянного хранения паролей, внедрять механизмы истечения сессий, MFA для критичных действий. Любые ключи API, токены доступа не должны быть закодированы напрямую в приложении – при необходимости их можно загружать с сервера после аутентификации пользователя или хранить в Secure Enclave.

6.4. Аутентификация пользователя

iOS предлагает удобные и безопасные механизмы для реализации логина в виде интеграции Face ID через фреймворк LocalAuthentication, позволяющего реализовать биометрическую аутентификацию для разблокировки приложения или выполнения определенных действий. При этом данные Face ID никогда не покидают устройстве, поскольку проверка происходит в Secure Enclave, а приложение получает только факт успешной аутентификации.

Помимо этого, Apple продвигает Sign in with Apple – OAuth2 сервис единого входа, который скрывает электронную почту пользователя и предотвращает слежку. С точки зрения приложения, внедрение Sign in with Apple избавляет от хранения паролей вовсе – Apple берет на себя проверку личности.

6.5. Кодовая подпись и верификация приложений

Как упоминалось выше, iOS строго требует подписания приложений – обязательная кодовая подпись является фундаментом безопасности: система не позволит выполнить код, не прошедший проверку подписи. Эта схема позволяет защитить приложения от модификации, поскольку злоумышленник не может подменить часть вашего приложения, так как подпись станет недействительной. Третьи лица также не могут выпустить фальшивое приложение от вашего имени, так как сертификаты выдаются только проверенным аккаунтом, прошедшим верификацию через программу Apple Developer Program (см, п. 3.1.).

7. Оптимизация энергопотребления

Пользователи современных смартфонов чувствительны к тому, как приложение влияет на заряд аккумулятора. Приложение, которое заметно разряжает устройство, рискует быть удалено, независимо от полезности [13], поэтому разработчики обязаны проектировать приложения с учетом энергоэффективности.

В iOS присутствуют встроенные механизмы, которые ограничивают фоновую активность приложений: режим низкого энергопотребления, который система может включить при 20% батареи или пользователь по своему желанию, или режим адаптивного питания, представленный в iOS 26.

7.1. Основные источники расхода аккумулятора

- Частые или неэффективные сетевые запросы держат активным радио-модуль устройства (Wi-Fi или Cellular), что потребляет много энергии [13]. В особенности это заметно при мобильной передаче данных, когда каждое соединение требует пробуждения модема, сигнала к вышке и т.д.

- Интенсивные вычисления, часто работающий на полной мощности процессор быстро разряжают батарею [13]. Сюда относятся неоптимальные алгоритмы, бесконечные циклы ожидания и пр.
- Постоянное использование GPS – один из самых «прожорливых» сценариев [13]. Чип GPS и сопутствующие вычисления (определение координат по спутникам) потребляют значительную мощность, поэтому длительное слежение за местоположением разряжает телефон.
- Любые задачи, выполняемые в фоне (обновление контента, фоновое прослушивание музыки и т.д.) препятствуют устройству переходить в спящий ежим, не дают выключить радио-модуль, могут будить процессор – это все вместе сокращает время работы [13].

7.2. Стратегии оптимизации

- Чтобы снизить сетевую активность, существует несколько подходов. Во-первых, уменьшить число запросов – агрегировать данные, использовать эффективные протоколы. Например, собрать несколько API-вызовов в один, если это возможно, или использовать сжатие данных (сжимать JSON, отправлять бинарные форматы). Во-вторых, кэширование – не запрашивать с сервера то, что можно сохранить локально. Если приложение обновляет данные, вместо частого опроса сервера лучше внедрить push-уведомления или использовать Background Fetch, чтобы система сама по расписанию запуска обновление, когда это оптимально [13]. Также iOS позволяет планировать сетевые запросы через специальные классы (NSURLSession background tasks), которые выполняют передачу в фоновом режиме, а система агрегирует такие задачи от разных приложений, минимизируя время активного радио-модуля.
- Для оптимизации CPU нужно стремиться, чтобы код выполнялся как можно более эффективно. Это включает выбор правильных структур данных (избегать квадратичных алгоритмов на больших списках), кэширование результатов, использование векторных API Apple (vDSP для математики, например). Одно из правил – объединение мелких задач в одну крупную (batching), чтобы процессор реже просыпался [13]. Вместо частого вызова таймера раз в 0,1 секунду лучше выполнять нужные действия раз в 1 секунду, накопив работу. Помимо этого, важно не злоупотреблять фоновыми потоками: многопоточность помогает распределить нагрузку, но создание множества потоков само по себе тратит ресурсы.
- Если приложению нужны координаты, крайне важно настроить подходящую точность и дистанционный фильтр. Например, для погодного приложения не имеет

смысла запрашивать точность до метра – достаточно до километра, что существенно экономнее. Использовать геофенсинг или режимы significant change: вместо непрерывного стрима GPS можно подписаться на события входа/выхода из определенных зон (геофенс) или крупные перемещения (significant change), чтобы получать редкие, но существенные обновления [13]. Это позволит, к примеру, проснуться приложению только тогда, когда пользователь переместился на значительное расстояние.

- Если приложению не жизненно необходимо работать в фоне – лучше не включать Background Modes. Соцсети могут полагаться на push-уведомления вместо фонового обновления ленты. Если же фоновые возможности нужны, то воспользоваться новыми API: библиотека BackgroundTasks (BGTaskScheduler) позволяет планировать короткие фоновые обновления контента, при этом система сама решает, когла их запустить. От разработчика требуется обеспечение того, чтобы фоновые задачи выполнялись быстро и эффективно, по возможности группируя их. Отправка silent push-уведомлений также должна использоваться умеренно, поскольку Apple может ограничить приложение, которое слишком часто будит устройство без ведома пользователя [13].

8. Тестирование

8.1. Инструменты и фреймворки для тестирования

В экосистеме iOS основным инструментом является встроенный XCTest – фреймворк для модульных тестов и UI-тестов, поставляемый вместе с Xcode. Разработчик может писать юнит-тесты на Swift или Objective-C, проверяя корректность работы функций, моделей, логики. XCTest предоставляет assert-методы, возможность группировать тесты в сьюты и измерять производительность отдельных участков (через метод measure для бенчмаркинга).

Для тестирования UI Apple предлагает XCTest UI. Это специальный целевой модуль, который запускает приложение и эмулирует действия пользователя – нажатие кнопок, ввод текста, жесты с последующей проверкой ожидаемого результата на экране. UI-тесты позволяют автоматизировать сценарии, которые вручную проверять долго и муторно: например, логин и переход по нескольким экранам, или добавление товара в корзину и оформление заказа в e-commerce приложении.

8.2. Тестирование на устройствах vs эмуляторы

Xcode Simulator – мощный инструмент, позволяющий быстро запускать приложение на виртуальных устройствах. Однако между симуляцией и реальными девайсами есть отличия: симулятор выполняется на архитектуре Mac (ARM64 на Apple Silicon или x86_64 на Intel), не эмулируя реальное железо, а лишь поведение iOS. Симулятор не способен воспроизвести некоторые аппаратные особенности: нет реального GSM/LTE модуля, датчиков, отсутствует камера, микрофон. Невозможно симулировать низкий объем памяти, нагрев устройства, прерывания, такие как входящий звонок [14].

Реальные устройства работают в совершенно иной среде: у пользователя может быть плохой сигнал, забита память другими приложениями, фоновые процессы, энергосбережение – все это симулятор не отражает, поэтому неотъемлемое правило мобильной разработки – тестировать на реальных устройствах перед выпуском приложения.

8.3. Подходы к тестированию UI

Тестирование UI сочетает в себе автоматические и ручные методики. Автоматизированные UI-тесты полезны для регрессии: можно прогнать скрипт по основным сценариям приложения и убедиться, что после изменений все еще можно залогиниться, создать запись, оформить заказ и т.д. Однако UI-тесты пишутся и поддерживаются довольно дорого: малейшие изменения в интерфейсе могут сломать тест. Поэтому обычно автоматизируют критичный минимальный набор сценариев.

Остальное покрывается ручным тестированием – QA инженеры или сами разработчики кликают по приложению, проверяя различные кейсы, пограничные условия.

8.4. Тестирование производительности

Помимо логической корректности, необходимо проверять, что приложение быстро запускается, плавно работает и не падает под нагрузкой. Apple рекомендует измерять время запуска приложения и следить, чтобы оно не превышало 1-2 секунды для холодного старта. Инструменты вроде XCTest measure помогают отловить методы, выполняющиеся слишком медленно – тестовый кейс может, например, вызывать парсинг JSON тысячу раз и проверить, что в среднем операция укладывается в заданный порог времени.

Тестирование потребления памяти – отдельный важный момент. Симулятор может не показывать утечки памяти, которые на устройстве приведут к выгрузке приложения при достижении лимита, поэтому используют инструмент Leaks для поиска утечек и Allocations для отслеживания роста памяти.

8.5. Специфические аспекты тестирования iOS

Важно проверять совместимость с разными версиями iOS: если минимальная поддерживаемая версия iOS 16, нужно протестировать на 16 и на последней на предмет отсутствия проблем. Иногда возникают нюансы: API меняются, или на старой версии что-то работает иначе.

Если приложение мультиязычное, проверить все языки (нет ли текстов, выходящих за границы кнопки и т.п.). Протестировать, как VoiceOver читает элементы интерфейса для большей доступности: все ли помечено семантически, удобны ли элементы для людей с ограничениями.

Список источников

1. StatCounter Global Stats. «Mobile Operating System Market Share Worldwide – July 2025». URL: https://gs.statcounter.com/os-market-share/mobile/worldwide (Дата обращения: 26.08.2025).
2. Apple. «iOS and iPadOS usage as measured by devices that transacted on the App Store (June 4, 2025).» Apple Developer Support, 2025. URL: https://developer.apple.com/support/app-store/ (Дата обращения: 26.08.2025).
3. BairesDev. «iOS App Design Guidelines for 2025.» BairesDev Blog, 2023. URL: https://www.bairesdev.com/blog/ios-design-guideline/ (Дата обращения: 26.08.2025).
4. Crooks, Melissa. «Uncover The Key Differences of Objective-C vs. Swift.» Medium, Aug 20, 2024. URL: https://medium.com/@melissacrooks/uncover-the-key-differences-of-objective-c-vs-swift-6bba10033ce0 (Дата обращения: 27.08.2025).
5. Max Kalik. «A Look at the MVVM Design Patterms for iOS» Hackernoon, Nov 8, 2022. URL: https://hackernoon.com/mvvm-for-ios-family (Дата обращения: 2.09.2025).
6. Apple. «Beta Testing made simple with TestFlight». URL: https://developer.apple.com/testflight/ (Дата обращения: 2.09.2025).
7. Apple Developer Enterprise Program. URL: https://developer.apple.com/programs/enterprise/ (Дата обращения: 4.09.2025)
8. Apple. «Процесс подписи кода приложений в iOS, iPadOS, tvOS, watchOS и visionOS». Apple Platform Security, Apple Support. URL: https://support.apple.com/ru-ru/guide/security/sec7c917bf14/web (Дата обращения: 4.09.2025)
9. Amarildo Lucas. «iOS CI/CD: Continuous Intergration and Continuous Delivery Explained» Semaphore, Apr 3, 2024. URL: https://semaphore.io/ios-continuous-integration (Дата обращения: 4.10.2025).
10. Apple Developer. Human Interface Guidelines. URL: https://developer.apple.com/design/human-interface-guidelines (Дата обращения: 4.10.2025).
11. 30DaysCoding. «Unlocking the Power of iOS Sensors and Hardware Integration». 30DaysCoding Blog. URL: https://30dayscoding.com/blog/working-with-ios-sensors-and-hardware-integration (Дата обращения: 4.10.2025).
12. Jadhav Tushar. «iOS App Security Checklist: Top Best Practices for Developing Safer Mobile Apps». Mobisoft Infotech Blog, July 4, 2025. URL:

https://mobisoftinfotech.com/resources/blog/app-security/ios-app-security-checklist-best-practices (Дата обращения: 8.10.2025).

13. Astro Lab Tech. «Optimizing Battery Usage in Your iOS App: A Comprehensive Guide» Medium, Oct 21, 2024. URL: https://medium.com/@edabdallamo/optimizing-battery-usage-in-your-ios-app-a-comprehensive-guide-8203cb074b8a (Дата обращения: 8.10.2025).
14. Stack Overflow. «Testing iOS on real devices vs. Simulator» StackOverflow, Nov 5, 2012. URL: https://stackoverflow.com/questions/13235702/testing-ios-testing-on-real-devices-vs-simulator (Дата обращения: 8.10.2025).