ВВЕДЕНИЕ

Благодаря развитию передовых технологий повседневная жизнь человека становится удобнее и эффективнее. Развитие систем «умный дом» позволил автоматизировать большинство бытовых процессов, обеспечивая комфорт для пользователей. Современные решения позволяют управлять освещением, климатом, системами безопасности и другими элементами, создавая интеллектуальную и адаптивную среду.

Развитие универсальных стандартов связи значительно упростило взаимодействие устройств разных производителей, обеспечивая их совместимость и позволяя пользователям создавать гибкие, масштабируемые экосистемы для управления. В результате современные системы автоматизации становятся все более функциональными и доступными.

Несмотря на все преимущества, вопрос стоимости остается одним из ключевых факторов, ограничивающих массовое внедрение систем «умный дом». Высокие цены на оборудование, необходимость приобретения дополнительных шлюзов и контроллеров, а также расходы на установку и настройку могут стать серьезным барьером для пользователей. Кроме того, некоторые устройства требуют регулярного обновления или подписки на облачные сервисы, что увеличивает общую стоимость использования внедряемой системы.

Однако, помимо готовых коммерческих решений, все большую популярность приобретают DIY-устройства (Do It Yourself – «сделай сам»), позволяющие пользователям самостоятельно создавать и настраивать системы «умного дома» под свои нужды. Такой подход не только снижает затраты, но и дает возможность полной кастомизации, позволяя интегрировать исполнительные устройства в единую интеллектуальную сеть.

Целью данного дипломного проекта является разработка Android-приложения, которое обеспечит мониторинг и управление устройствами «умного дома», предоставляя пользователю интуитивно понятный интерфейс для эффективного контроля, настройки и автоматизации различных устройств.

Для достижения поставленной цели дипломного проекта важно разбить ее на конкретные задачи:

– прошивка и настройка шлюза «умного дома»;
– проектирование архитектуры системы «умный дом»;
– разработка Android-приложения для управление системой «умный дом»;
– тестирование и отладка разработанного приложения.

Данный дипломный проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет XX% (отчет о проверке на заимствования прилагается).

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор существующих аналогов

Одним из первых этапов разработки приложения для управления и мониторинга «умным домом» является определение перечня необходимых функций и выбор подходящих технологий. Для принятия обоснованных решений важно проанализировать преимущества и недостатки существующих решений на рынке.

Многие пользователи не имеют полного представления о возможностях современных систем «умного дома», их функционале и технических особенностях. Такие приложения могут включать широкий спектр функций – от управления освещением и климатом до мониторинга безопасности и автоматизации сценариев. В дальнейшем будут рассмотрены существующие решения, схожие по функционалу с разрабатываемым приложением.

1.1.1 Системы от «Xiaomi» предоставляет широкий ассортимент разнообразной техники, в том числе технику, которая отвечает требованиям «умного дома» [x]. В их ассортименте можно найти большое количество разнообразных устройств, что делает их продукцию универсальным решением для создания современного «умного дома».

Для удобного управления и взаимодействия с этими устройствами предоставляется специальное приложение, обеспечивающее полную совместимость устройств одной экосистемы и их простую настройку. Благодаря фокусу данного производителя на широкий потребительский рынок, все устройства легко подключаются и не требуют значительных временных затрат на настройку.

Для построения системы «умного дома» с устройствами Xiaomi и использованием протокола MQTT необходим централизованный хаб Xiaomi Smart Home Hub 2. Этот хаб выступает в роли шлюза между беспроводными устройствами, поддерживающими Zigbee 3.0, Bluetooth Mesh, Wi-Fi и Matter, обеспечивая их взаимодействие в единой экосистеме.

Xiaomi Smart Home Hub 2 подключается к сети через Ethernet или Wi-Fi и работает от розетки. В отличие от некоторых других хабов, он не оснащён встроенным аккумулятором, поэтому при отключении питания теряет связь с устройствами. Он поддерживает локальное управление через MQTT, что позволяет интегрировать его в собственные серверные решения, такие как HomeAssistant. Благодаря современному оборудованию хаб обеспечивает стабильную работу, минимальные задержки и возможность автоматизации без облачных сервисов.

Приложение Xiaomi Home является основным инструментом для управления устройствами «умного дома» от Xiaomi, а также совместимыми гаджетами других брендов, поддерживающих экосистему Mi Home. Оно доступно для Android и iOS, обеспечивая удобное удалённое взаимодействие с устройствами.

С помощью приложения можно добавлять устройства, управлять их настройками, создавать сценарии автоматизации и получать уведомления. Приложение поддерживает групповое управление, позволяя объединять устройства по комнатам и сценариям. Также реализована интеграция с голосовыми помощниками (для совместимых устройств).

Xiaomi Home позволяет настроить локальное управление для некоторых устройств, но большая часть автоматизаций выполняется через облачные серверы Xiaomi, что требует постоянного подключения к интернету. Интерфейс приложения интуитивно понятен, но скорость работы может зависеть от количества подключённых устройств и качества интернет-соединения.

Графический интерфейс приложения представлен на рисунке 1.1.

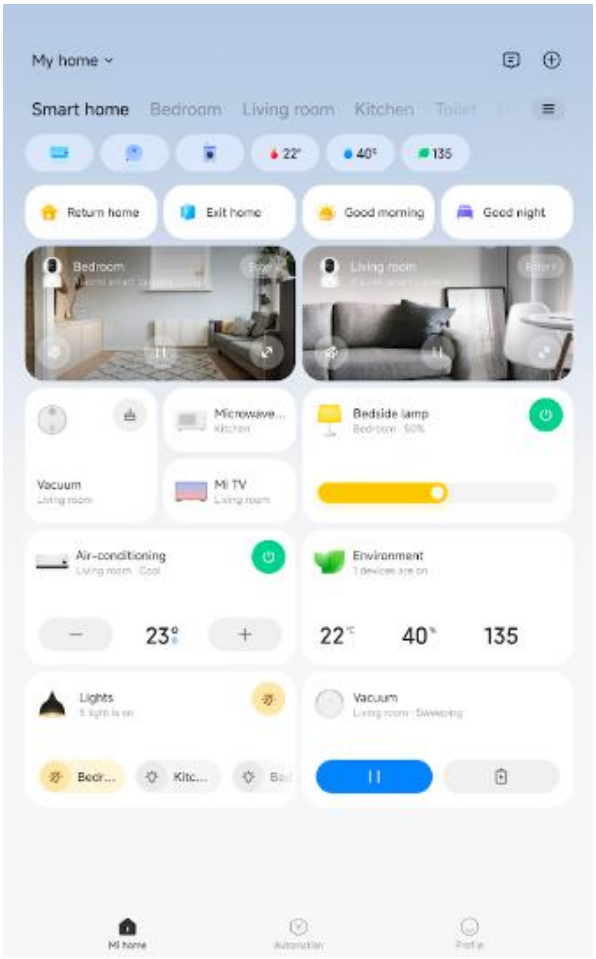

Рисунок 1.1 – Интерфейс приложения Xiaomi Home

Приложение Xiaomi Home предоставляет пользователям удобный способ управления устройствами умного дома от Xiaomi. Однако, на основе отзывов пользователей, можно выделить несколько недостатков:

1 Для корректной работы некоторых устройств и функций в приложении Xiaomi Home пользователям часто приходится выбирать определённый регион, например, Китай. Это связано с тем, что некоторые устройства могут быть доступны только в определённых регионах, а их функциональность

может отличаться в зависимости от выбранных настроек.

2 Для стабильной работы некоторых функций рекомендуется использовать английский язык. Это может быть неудобно для пользователей, предпочитающих русский интерфейс, так как требует дополнительной адаптации.

3 Приложение официально поддерживает русский язык, но некоторые разделы и плагины могут отображаться на китайском или английском. Это создаёт сложности при использовании и может привести к затруднениям в настройке устройств.

Таким образом приложение Xiaomi Home является удобным инструментом для управления устройствами умного дома, предлагая широкий функционал и интеграцию с экосистемой Xiaomi. Однако его использование может вызывать трудности из-за ограничений, связанных с региональными настройками. Несмотря на недостатки, приложение остаётся популярным благодаря своей универсальности, низкой стоимости и поддержке большого количества устройств.

1.1.2 Решения от компании «Яндекс» представляют собой экосистему, интегрированную с голосовым помощником Алисой. В ассортименте представлены устройства, такие как умные лампы, розетки, датчики движения и видеокамеры, а также шлюзы для взаимодействия с техникой сторонних производителей (например, Xiaomi, Philips Hue) через облачную интеграцию [x]. Основное преимущество системы – глубокая интеграция с сервисами Яндекса (Музыка, Погода, Расписания) и поддержка русского языка на всех уровнях взаимодействия.

В качестве шлюза чаще всего используется Яндекс Станция (умная колонка с голосовым управлением) или Яндекс Модуль, который подключается к сети через Wi-Fi и выступает мостом между устройствами умного дома и облачными сервисами. Шлюз поддерживает протоколы Wi-Fi и Bluetooth, но не Zigbee, что ограничивает совместимость с некоторыми устройствами. Управление через MQTT или локальные решения не предусмотрено – все команды обрабатываются через облако Яндекса. Это обеспечивает простоту настройки, но создаёт зависимость от интернет-соединения и серверов компании.

Приложение доступно для Android и iOS и служит центральным инструментом для управления устройствами. Ключевые функции данных приложений:

– добавление и группировка устройств по комнатам;

– создание сценариев автоматизации (например, «если датчик движения сработал, включить свет»);

– интеграция с голосовым помощником Алисой для управления через речь;

– уведомления о событиях (срабатывание датчиков, отключение устройств);

– совместимость с устройствами других брендов через облачные

плагины (например, TP-Link, Redmond).

Интерфейс приложения интуитивно понятен и полностью русифицирован (см. рисунок 1.2). Для настройки сценариев используется визуальный редактор, где пользователь может задавать условия и действия с помощью «перетаскивания» элементов.

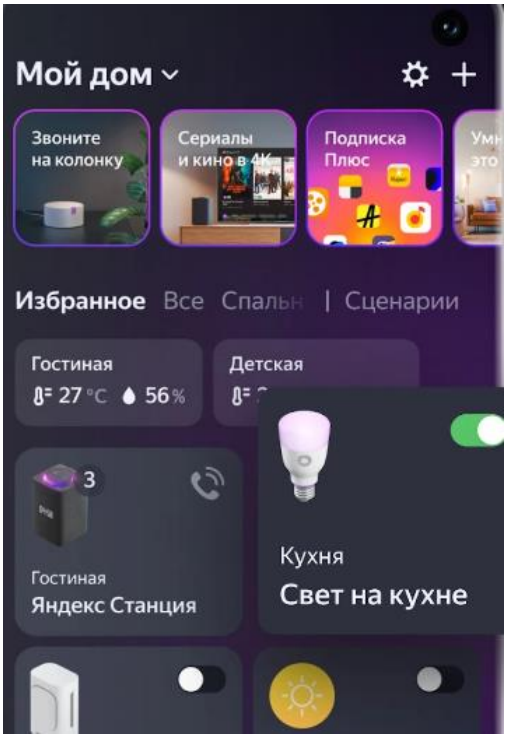

Рисунок 1.2 – Интерфейс приложения «Дом с Алисой»

Приложение «Яндекс.Умный дом» обеспечивает глубокую интеграцию с экосистемой сервисов Яндекса, что позволяет пользователям управлять устройствами через голосового помощника Алису и автоматически добавлять технику по QR-коду без ручного ввода данных. Локализация интерфейса, включая полную поддержку русского языка и голосовых команд, делает решение доступным для русскоязычной аудитории. Настройка устройств максимально упрощена: приложение не требует технических знаний, что идеально подходит для новичков. Кроме того, облачные сценарии автоматизации позволяют сохранять работоспособность функций даже при выключенном телефоне, так как логика обработки событий выполняется на серверах Яндекса.

Основным ограничением системы является сильная зависимость от облачных сервисов: большинство операций, включая управление устройствами и выполнение сценариев, требуют стабильного интернет-соединения, а локальное управление (например, через LAN) недоступно. Это создаёт риск потери контроля над умным домом при сбоях в сети. Кроме того, приложение поддерживает ограниченный набор протоколов – отсутствие Zigbee и MQTT сужает выбор совместимых устройств, делая экосистему закрытой.

Приложение «Яндекс.Умный дом» предлагает удобное решение для пользователей, которые ценят простоту и интеграцию с русскоязычными сервисами. Однако его зависимость от облака и закрытая экосистема ограничивают гибкость, что критично для продвинутых сценариев автоматизации. В отличие от Xiaomi, система Яндекса не поддерживает локальные протоколы (например, Zigbee), что делает её менее универсальной.

1.2 Zigbee

Беспроводной протокол Zigbee, предназначенный для передачи данных между устройствами в сетях с низким энергопотреблением. Он работает на частоте 2,4 ГГц и поддерживает три типа устройств:

– координатор (главное устройство, управляющее сетью);

– роутер (передаёт сигналы другим устройствам);

– оконечное устройство (датчики, выключатели и другие узлы, потребляющие минимум энергии).

Благодаря сетевой топологии Mesh устройства могут передавать данные друг через друга, увеличивая дальность связи.

Zigbee активно применяется в системах умного дома, промышленной автоматизации, медицине и IoT-решениях. Его ключевые преимущества – низкое энергопотребление, высокая надёжность за счёт самовосстанавливающейся сети и совместимость с разными производителями. Однако у него есть и недостатки: ограниченная пропускная способность делает его неподходящим для передачи мультимедиа, а необходимость координатора может усложнять настройку сети. Кроме того, возможны помехи от Wi-Fi, так как оба протокола работают на одной частоте.

1.3 MQTT

Большинство устройств «умного дома» находятся в непосредственной близости друг от друга и работают в локальной сети, но для их работы в качестве IoT-устройств необходимо подключение к сети Интернет. Это позволит организовать мониторинг и управление устройствами из любой точки планеты при наличии интернет-доступа. Этот подход значительно облегчает взаимодействие человека с системой, поскольку сегодня мобильный интернет доступен практически повсюду.

В большинстве современных систем «умный дом» управление системой производится с использованием мобильного телефона с выходом в интернет, в редких случая используются веб-приложения доступные как для телефонов, так и для персональных компьютеров с помощью веб-браузера. Такой подход делает мобильное устройство ещё одним IoT-устройством.

В непосредственной близости для обмена данными между устройствами и координатором умного дома применяются такие протоколы, как Zigbee, Z-Wave и другие. Однако для связи координатора со смартфоном эти протоколы не подходят, так как они имеют ограниченный радиус действия, не

поддерживаются в большинстве мобильных устройств и требуют специального оборудования для приема и передачи данных. Для этих целей используется подключение к интернету с использованием протокола MQTT.

Протокол MQTT, разработанный на основе TCP/IP, оптимизирован для потоковой передачи данных в сетях с низкой пропускной способностью, что делает его особенно востребованным при создании Android-приложений для удалённого управления системой «умный дом». Его преимущества перед другими протоколами и ключевые особенности включают:

– поддержка сохранённых сообщений (retained message), что позволяет новым клиентам получать актуальную информацию сразу после подключения;

– для работы в условиях нестабильного подключения реализован механизм QoS (Quality of Service) который обеспечивает надёжность доставки сообщений;

– имеет поддержку функционала LWT (Last Will and Testament, «последняя воля и завещание»), что используется для оповещения брокера о нештатном отключении клиента;

– минимальная задержка передачи данных, благодаря легковесности передаваемых сообщений.

Система связи, использующая протокол MQTT (см. рисунок 1.3), включает в себя издателей (publisher), сервер-брокер (MQTT broker) и одного или нескольких клиентов (subscribers). Издателю не требуется дополнительная настройка для определения числа или местоположения подписчиков, которые будут получать сообщения. Также подписчики не нуждаются в настройке для определения конкретного издателя. Издатель отправляет сообщения в темы (topic), откуда их могут читать клиенты, подписанные на данную тему. Издатель и подписчик не могу коммуницировать напрямую, поэтому они не знают о существовании друг друга.

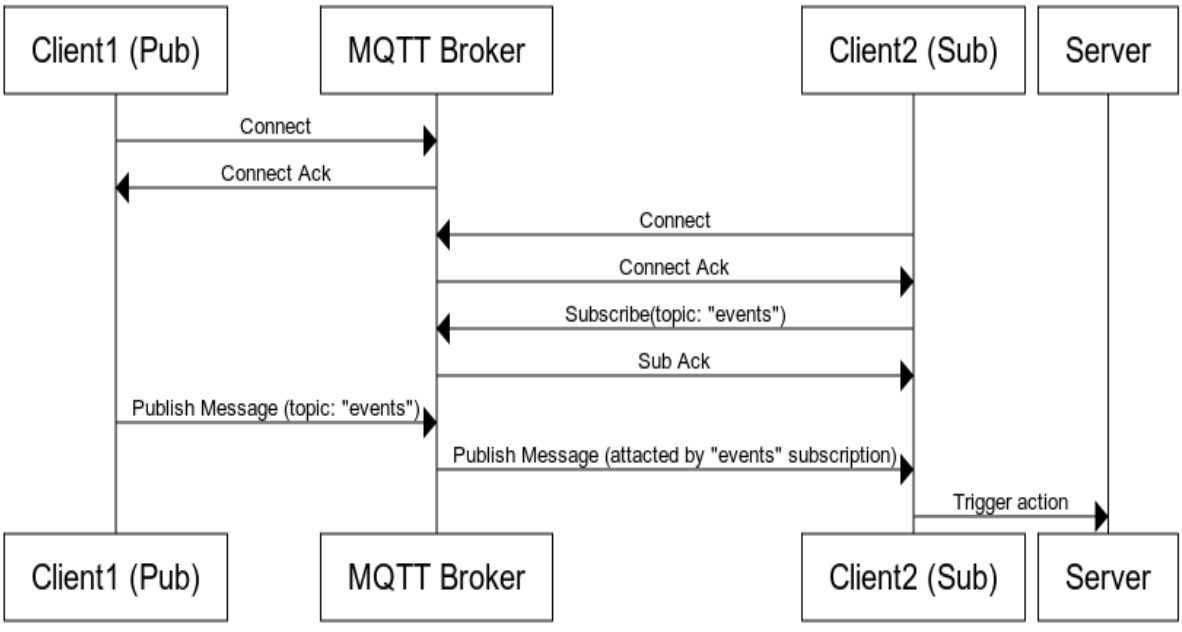

Рисунок 1.3 – Схема взаимодействия по протоколу MQTT

В MQTT данные передаются в виде payload, которые могут быть любыми байтовыми данными. Это означает, что информация может быть представлена в виде строки, JSON, двоичных данных или даже изображений – все зависит от того, что отправитель и получатель согласуют. Однако, в большинстве случаев, данные передаются в формате JSON, так как этот формат легко обрабатывается, обладает хорошей читаемостью и широко используется для обмена структурированными данными в приложениях, связанных с IoT и умными устройствами.

После отправки, данные хранятся в топиках, которые составлены из символов в кодировке UTF8 и имеют структуру, схожую с деревом файловых UNIX подобных операционных систем. Данный подход обеспечивает дополнительную читаемость для пользователей. В некоторых случаях устройства отправляют данные в топики, которые соответствует их адресу, например:

- home/kitchen/plug;
- home/restroom/light/ceiling;
- zigbee/0xA4C138DEB58ECDEC.

Данная структура топиков изначально задана в устройстве, но может быть изменена пользователем при необходимости.

Организация топиков с использованием дружественных имён для устройств позволяет наглядно отслеживать передаваемые данные и упрощает разработку и отладку кода, не требуя запоминания числовых адресов расположения данных.

Для взаимодействия устройств с брокером предусмотрены различные типы сообщений, ниже перечислены основные типы для функционирования системы:

- connect – установка соединения клиента с брокером;
- disconnect – разрыв соединения клиента с брокером;
- publish – публикация данных клиента в топик;
- subscribe – подписка клиента на получение сообщений из топика.

Таким образом, протокол MQTT является отличным средством для обмена данными в системах «умный дом». Его простота и гибкость позволяют легко интегрировать устройства с различными уровнями взаимодействия. Возможность использования структуры топиков, которая напоминает файловую систему, делает обмен данными более удобным и читаемым для пользователя. MQTT также поддерживает различные уровни качества обслуживания (QoS), что обеспечивает надежность передачи сообщений даже в условиях нестабильной сети. Эти особенности делают MQTT идеальным выбором для создания масштабируемых и устойчивых решений в умных домах, промышленности и других областях.

1.4 Интегрированная среда разработки Android Studio

Android Studio включает в себя редакторы кода, визуальные инструменты для разработки пользовательских интерфейсов, отладчики и

профилировщики производительности, а также средства для автоматизации и управления проектами.

Можно устанавливать и отлаживать приложения как в эмуляторе, так и на физических устройствах, подключённых по USB или беспроводным способом. В режиме отладки Android Studio предоставляет подробную информацию о работе приложения, позволяя приостанавливать его выполнение, анализировать состояние переменных и инспектировать пользовательский интерфейс.

Среда также включает библиотеку инструментов для работы с базами данных, мультимедийными файлами, сетевыми протоколами и графикой. Функция Live Edit позволяет мгновенно просматривать изменения в пользовательском интерфейсе Jetpack Compose без необходимости полной перекомпиляции приложения. Это помогает разработчикам оперативно тестировать идеи и настраивать внешний вид, а также проверять адаптацию интерфейса для разных устройств, ориентаций экрана и цветовых схем.

1.5 Архитектура приложений

Разработка Android-приложений опирается на несколько архитектурных подходов, которые помогают организовать код, упростить поддержку и улучшить масштабируемость проекта. Архитектура определяет принципы разделения ответственности между компонентами приложения, что особенно важно в условиях сложных пользовательских интерфейсов и многопоточных операций.

Среди наиболее распространенных архитектур для Android можно выделить MVC (Model-View-Controller) и MVVM (Model-View-ViewModel). Эти модели помогают структурировать код так, чтобы интерфейс, логика обработки данных и бизнес-логика были четко разграничены, что снижает зависимость компонентов и делает приложение более гибким.

1.5.1 Model-View-Controller (MVC) – это архитектурный паттерн, который является основой для разработки приложений. Он обеспечивает разделение приложения на три основных взаимодействующих компонента: модель (Model), представление (View) и контроллер (Controller) представленных на рисунке 1.4.

Использование паттерна MVC позволяет создавать гибкие и масштабируемые приложения, где каждый компонент отвечает за свою часть функциональности.

Модель отвечает за хранение данных и логику обработки данных: в ней описывается бизнес-логика, правила валидации и другие операции, связанные с данными. Модель не имеет информации о представлении и контроллером, то есть не имеет доступа к их полям, функциям и иному. Она предоставляет данные через заранее определённые интерфейсы, что делает её независимой от пользовательского интерфейса.

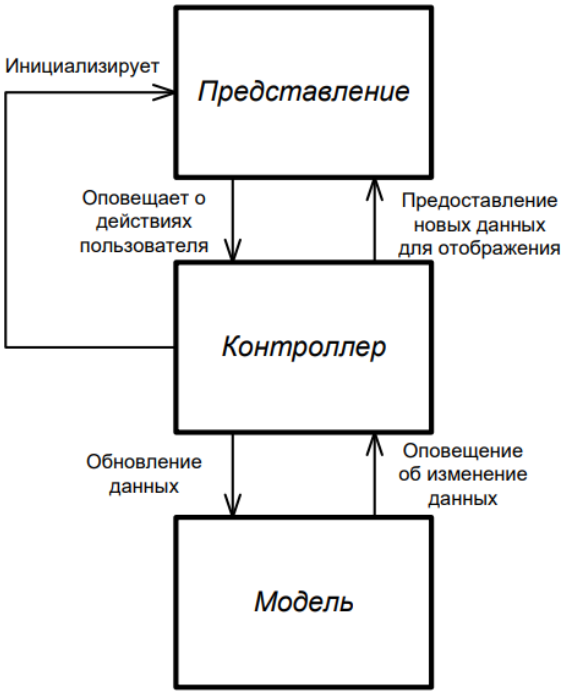

Рисунок 1.4 – Архитектурный паттерн MVC

Представление отображает данные пользователю и предоставляет возможность взаимодействия с приложением через графический интерфейс. В ней определяется внешний вид и макет интерфейса, это происходит в графическом редакторе. Представление не имеет представления о существовании модели и контроллера.

Контроллер является посредником между моделью и представлением. Он получает информацию от представления, обрабатывает её и вносит соответствующие изменения в модель. Таким образом, контроллер обеспечивает взаимодействие и согласованность между моделью и представлением, делая возможным реализацию функциональности приложения. Чаще всего контроллер является наиболее наполненным исходным кодом элементом приложения.

В контексте разработки Android-приложений паттерн MVC может использоваться для структурирования кода и разделения ответственности между компонентами. Однако в Android этот паттерн встречается реже, так как система активностей и фрагментов уже включает элементы контроллера и представления.

Хотя MVC может быть использован в Android, чаще применяются другие архитектурные паттерны, такие как MVVM (Model-View-ViewModel), которые лучше соответствуют особенностям Android-разработки и обеспечивают лучшую разделяемость кода.

1.5.2 Model-View-ViewModel (MVVM) является схожим с MVC паттерном, используемым для разделения компонентов приложения на три основных уровня: Модель, Представление и Модель Представления (ViewModel).

Одним из недостатков MVVM является некоторая его избыточность для простых приложений, которые имеют небольшое количеством логики и данных. MVVM требует больше кода, чем, например, простая парадигма Model-View-Controller. При этом эти недостатки нивелируются при разработке относительно больших приложений, так как MVVM предоставляет огромные возможности для повторного использования различных элементов Представления. На начальных этапах разработка приложения, использующего платформу MVVM, может быть затратной с точки зрения количества кода и времени, однако повторное использование и универсальность стандартизированных представлений упрощают поддержку и расширение проекта в будущем.

Структурная схема взаимодействия компонентов архитектурного паттерна MVVM представлена на рисунке ниже:

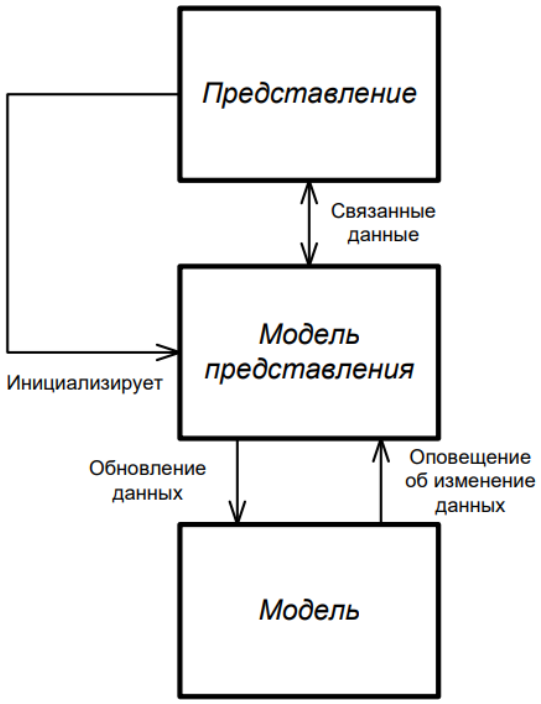

Рисунок 1.5 – Архитектурный паттерн MVVM

Модель, аналогична модели MVC, не имеет информации о представлении и модели представления.

Представление, частично похоже на MVC, отображает данные пользователю и обрабатывает пользовательский ввод. При это в отличии от MVC, представление напрямую обращается к модели представления, последняя чаще всего является полем представления.

Модель представления является посредником между моделью и представлением и чем-то схожа с контроллером MVC. Модель имеет общее с Представляем данные, которые автоматически обновляются и модели представления и в самом представлении при изменении их как стороны пользователя, так и со стороны модели. В MVC же контроллер при обновлении данных запускает метод обновления Представления. В каком-то смысле

представление совмещает контроллер и представление MVC. Модель представления также может обслуживать несколько представлений одновременно.

1.6 Язык программирования Kotlin

Kotlin – это современный статически типизированный язык программирования, разработанный компанией JetBrains. Он ориентирован на JVM (Java Virtual Machine) и является полностью совместимым с Java, что позволяет использовать существующие библиотеки и фреймворки без значительных изменений. Kotlin отличается лаконичностью, повышенной безопасностью кода и удобными конструкциями, которые помогают снизить количество ошибок в разработке.

Одним из ключевых преимуществ Kotlin является его безопасность при работе с null-значениями. В отличие от Java, в Kotlin используется строгая система типов, предотвращающая ошибки, связанные с NullPointerException. Также язык поддерживает функциональное программирование, позволяя использовать лямбда-выражения, коллекции с расширенными возможностями и другие удобные инструменты для обработки данных.

Kotlin активно используется в разработке Android-приложений и официально поддерживается Google как основной язык для платформы Android.

1.7 Jetpack Compose

Jetpack Compose – это современный инструмент для разработки интерфейсов в Android, созданный Google. Он основан на декларативном подходе, что позволяет описывать пользовательский интерфейс (UI) с помощью функций, а не традиционных XML-ресурсов. Это упрощает разработку, делает код более читаемым и снижает вероятность ошибок.

Одним из главных преимуществ Jetpack Compose является его гибкость и модульность. Компоненты интерфейса (Composable-функции) можно легко переиспользовать, комбинировать и изменять без необходимости пересоздавать всю иерархию элементов. Кроме того, Jetpack Compose тесно интегрирован с архитектурными паттернами, такими как MVVM, что делает его удобным инструментом для построения современных Android-приложений.

Compose полностью управляет состоянием UI, что позволяет разрабатывать динамические и отзывчивые интерфейсы без сложных манипуляций с View. Разработчики могут использовать State и MutableState для отслеживания изменений и автоматического обновления экрана. Это значительно упрощает работу с анимациями, пользовательскими действиями и изменениями данных.

Еще одно важное преимущество – интеграция с существующими Android-компонентами. Jetpack Compose совместим с View-based UI, поэтому

его можно постепенно внедрять в проекты, работающие на старых версиях Android. Кроме того, Google активно развивает Compose, добавляя новые возможности и улучшая производительность.

Jetpack Compose также поддерживает Material Design 3, что позволяет легко использовать современные UI-компоненты и стили. Это ускоряет процесс разработки и делает приложения более визуально привлекательными. В сочетании с другими библиотеками Jetpack (например, Navigation, Paging, Room) Compose становится мощным инструментом для создания надежных и масштабируемых Android-приложений.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После анализа предметной области разрабатываемого приложения были сформулированы основные требования, которые необходимо учитывать при его реализации. Поскольку приложение включает обширный функционал, включая работу с базой данных, обработку и отправку данных MQTT-брокеру, работа в локальной сети с использованием HTTP, возникает необходимость в проектировании структурной схемы программного средства.

Структура проекта состоит из следующих блоков:

- блок MQTT клиента;
- блок соединения с брокером;
- блок коммуникации MQTT;
- блок обработки и генерации сообщений;
- блок базы данных;
- блок моделей;
- блок бизнес-логики;
- блок моделей представления;
- блок представления.

Структура системы была организована таким образом, чтобы каждый блок выполнял строго определенную задачу и обеспечивал стабильную работу приложения в целом.

Далее будет рассмотрен принцип работы каждого из выделенных блоков, их основные функции, а также взаимодействие между ними. Взаимосвязь между основными блоками проекта отражена на структурной схеме ГУИР.400201.060 С1.

2.1 Блок MQTT клиента

Блок MQTT клиента отвечает за первичную настройку соединения, обеспечивая корректное взаимодействие с брокером. В рамках этой настройки задаются основные параметры подключения, включая адрес брокера, номер порта, уникальный идентификатор клиента, а также параметры безопасности, такие как логин и пароль.

Дополнительно можно задать интервал Keep Alive, который определяет, как часто клиент будет отправлять сигналы активности брокеру, предотвращая разрыв соединения. Также можно настроить механизм Last Will and Testament (LWT) – заранее заданное сообщение, которое брокер опубликует в случае неожиданного отключения клиента. Это позволяет другим участникам системы получать актуальную информацию о его состоянии.

Кроме того, при настройке соединения можно указать, должна ли сессия быть очищаемой (Clean Session). Если параметр включен, при каждом новом подключении клиент начинает работу с нуля, без сохраненных подписок. Если отключен, брокер запоминает подписки и недоставленные сообщения, обеспечивая более надежную связь при временных разрывах соединения.

При первом подключении системы «умного дома» данные для подключения вводятся пользователем через блок представления. Вводимые параметры, такие как адрес брокера, логин, пароль и другие настройки, проверяются на корректность и сохраняются в базе данных только при успешном вводе и установлении соединения. Это обеспечивает их последующее использование и предотвращает сохранение некорректных данных. При следующем запуске приложения блок MQTT клиента загружает сохраненные параметры из базы данных и автоматически устанавливает соединение с брокером, исключая необходимость повторного ввода данных. Такой подход повышает удобство работы пользователя и снижает вероятность ошибок, связанных с ручной настройкой.

Благодаря этому механизму обеспечивается не только удобное управление подключением, но и стабильность работы системы, так как параметры сохраняются и могут быть использованы для восстановления соединения в случае сбоя.

2.2 Блок соединения с брокером MQTT

Блок соединения с MQTT-брокером отвечает за установление и поддержание стабильного соединения между клиентом и брокером. Его главная задача – инициировать подключение, следить за его состоянием и восстанавливать связь в случае разрыва.

При запуске системы блок загружает сохраненные параметры подключения, такие как адрес брокера, номер порта, логин, пароль, уникальный идентификатор клиента, а также дополнительные настройки, включая Keep Alive, Clean Session и Last Will and Testament (LWT).

Далее создается MQTT-клиент с заданными параметрами, после чего он отправляет запрос на подключение к брокеру, используя аутентификацию по логину и паролю. Если соединение успешно установлено, блок уведомляет другие компоненты системы, в частности блок коммуникации MQTT, который затем подписывается на нужные топики и начинает обмен сообщениями.

Для поддержания соединения используется механизм Keep Alive, который позволяет клиенту периодически отправлять сигналы активности, предотвращая разрыв связи. Если же соединение все же прерывается, блок оперативно обнаруживает это и инициирует процедуру переподключения. При повторных неудачах используется механизм увеличения задержки между попытками восстановления связи, чтобы снизить нагрузку на используемую сеть и брокер.

При каждом успешном подключении блок соединения может сохранять актуальные параметры в базе данных, чтобы использовать их при следующих запусках системы. Это позволяет исключить необходимость повторного ввода данных вручную.

Таким образом, блок соединения с MQTT-брокером выполняет ключевую функцию в системе, обеспечивая надежное подключение и автоматическое восстановление связи при сбоях.

2.3 Блок коммуникации MQTT

Блок коммуникации MQTT играет ключевую роль в системе «умного дома», обеспечивая обмен сообщениями между приложением и удаленным брокером. Его основная задача – установление защищенного соединения, подписка на топики и публикация сообщений, что позволяет передавать данные о состоянии устройств и отправлять им управляющие команды.

После подключения к брокеру с использованием аутентификации блок подписывается на определенные топики, что дает возможность получать актуальную информацию о состоянии устройств. При поступлении сообщения от брокера блок передает его в блок обработки и генерации сообщений, который анализирует содержимое и определяет дальнейшие действия. Если данные содержат информацию о новом состоянии устройства, они передаются в пользовательский интерфейс для отображения актуальной информации.

Кроме приема сообщений, блок коммуникации выполняет отправку данных, обеспечивая управление устройствами. Когда пользователь изменяет параметры в приложении, например, включает освещение или задает температуру, блок обработки формирует соответствующую команду и передает ее в блок коммуникации, который публикует сообщение в нужный топик MQTT. Это позволяет системе обеспечивать двусторонний обмен данными, необходимый для стабильной работы «умного дома».

Для поддержания соединения блок включает механизмы обработки ошибок и восстановления связи. В случае разрыва соединения он уведомляет блок соединения с брокером, который предпринимает попытки восстановления. Также он контролирует целостность передаваемых данных, предотвращая некорректные сообщения и дублирование команд.

2.4 Блок обработки и генерации сообщений

Блок обработки и генерации сообщений играет самую важную роль в системе, обеспечивая корректное взаимодействие между приложением и устройствами «умного дома» через MQTT-протокол. Он выполняет обработку входящих сообщений от брокера, анализируя их содержимое, и формирует исходящие команды для управления устройствами. Основная цель этого блока – преобразование сырых данных в понятный формат, фильтрация лишней информации и передача обработанных данных в соответствующие компоненты системы.

При получении сообщения от MQTT-брокера блок обработки и генерации анализирует его, проверяет на наличие ошибок и определяет, к какому устройству оно относится. Если данные содержат информацию о состоянии устройства, например, текущую температуру, уровень освещенности или статус реле, они передаются в пользовательский интерфейс для отображения актуальной информации.

Кроме обработки входящих сообщений, этот блок также отвечает за генерацию исходящих данных. Когда пользователь в приложении изменяет

параметры работы устройства, блок обработки получает соответствующий запрос, преобразует его в формат MQTT-сообщения и передает в блок коммуникации для отправки брокеру. Это позволяет осуществлять двусторонний обмен данными между клиентом и устройствами «умного дома».

Дополнительно блок включает механизмы фильтрации и валидации данных, предотвращая отправку некорректных команд и обеспечивая корректную работу системы. В случае обнаружения ошибки он может отправлять уведомления в интерфейс пользователя или запускать механизмы повторной передачи сообщения. Все это делает блок обработки и генерации сообщений важным элементом системы, который обеспечивает стабильную и надежную работу обмена данными через MQTT.

2.5 Блок базы данных

Информация, получаемая из блока бизнес-логики после обработки сообщений от брокера, записывается в базу данных, где она структурируется в таблицах, предназначенных для хранения данных о системе и подключенных устройствах. В качестве хранилища используется SQLite с библиотекой Room, обеспечивающей удобную работу с базой данных через объектно-реляционное отображение (ORM).

Блок базы данных отвечает за сохранение параметров подключения к удаленному брокеру сообщений, включая адрес брокера, номер порта, учетные данные, уникальный идентификатор клиента, а также дополнительные настройки, такие как Keep Alive, Clean Session и Last Will and Testament. Это позволяет автоматически восстанавливать соединение после перезапуска приложения.

В таблицах хранятся сведения о подключенных устройствах, включая уникальный сетевой адрес Zigbee, последние полученные значения от MQTT-брокера, пользовательские настройки (например, дружественное имя для устройства), параметры управления (каналы переключения реле, регулировки освещения) и тип устройства. Для ускорения работы приложения и снижения нагрузки на сеть используется кеширование данных, позволяющее загружать актуальные параметры устройств без необходимости частых запросов к брокеру. Механизмы обновления и удаления данных реализованы через Room API с учетом каскадных изменений и целостности связей между таблицами.

2.6 Блок моделей

Блок моделей отвечает за структуру данных, используемых в приложении. Он содержит классы, описывающие подключенные устройства умного дома, а также параметры подключения к MQTT-брокеру. Модели обеспечивают единообразное представление данных, используемых в бизнес-логике, базе данных и пользовательском интерфейсе.

Данные об устройствах включают уникальный идентификатор, тип

устройства, текущее состояние, доступные команды и другие параметры, необходимые для взаимодействия. Также в моделях хранятся сведения о подключении к MQTT-брокеру, такие как адрес сервера, логин, пароль и настройки соединения.

Модели упрощают обработку и передачу данных между различными компонентами системы, обеспечивая их согласованность и целостность. Кроме того, они позволяют эффективно управлять состоянием устройств и взаимодействовать с MQTT-брокером для получения и отправки данных в реальном времени.

2.7 Блок бизнес-логики

Блок бизнес-логики играет ключевую роль в обеспечении стабильной работы приложения, связывая между собой все основные компоненты системы. Он отвечает за обработку данных, управление MQTT-сообщениями, взаимодействие с базой данных и передачу данных в модель представления. Вся логика работы с устройствами умного дома, включая обработку пользовательских команд, обновление состояний и выполнение автоматических сценариев, сосредоточена именно здесь.

Для повышения надежности блок бизнес-логики реализует механизмы обработки ошибок и повторной отправки сообщений в случае сбоев соединения. Он также контролирует целостность данных, предотвращая их дублирование или потерю. Благодаря такому подходу обеспечивается бесперебойная работа системы и точное выполнение всех команд пользователя.

2.8 Блок представлений

Блок представлений отвечает за отображение данных и взаимодействие пользователя с системой «умного дома». В проекте используется Jetpack Compose, который позволяет создавать гибкий и переиспользуемый UI, адаптируемый под разные устройства. Представления работают в тесной связке с ViewModel, которая управляет состоянием, обрабатывает события и передает данные в интерфейс. Это обеспечивает четкое разделение ответственности, где UI остается декларативным, а бизнес-логика сосредоточена во ViewModel.

Одним из важных аспектов является тестируемость представлений. ViewModel можно проверять отдельно, используя инструменты для работы с потоками данных, а UI тестируется с помощью Compose UI Testing. Это позволяет выявлять ошибки на ранних этапах разработки. Кроме того, в Jetpack Compose предусмотрены механизмы обработки ошибок, такие как отображение уведомлений и диалогов, что помогает улучшить пользовательский опыт. Благодаря такому подходу представления остаются простыми, удобными в сопровождении и эффективно взаимодействуют с бизнес-логикой приложения.

2.9 Блок моделей представлений

В Jetpack Compose корневым представлением является композиционная функция (Composable), с которой начинается работа приложения. Это представление содержит ссылки на другие представления, формируя иерархическую структуру интерфейса. При этом всегда возможна навигация назад, а также циклические переходы, когда одно представление может ссылаться на себя.

В Jetpack Compose любое отображаемое на экране содержимое является представлением (Composable). Например, кнопка – это отдельное представление, описанное декларативным способом, а содержащий ее список также является представлением. Таким образом, одно представление может включать в себя множество других, а благодаря механизму параметров при инициализации обеспечивается гибкость и переиспользуемость компонентов.

Представления должны работать одинаково на всех устройствах Android и адаптироваться под разные размеры экранов. Для этого используется механизм адаптивного пользовательского интерфейса, обеспечивающий корректное отображение элементов на смартфонах, планшетах и других устройствах.

Представление взаимодействует с моделью представления (ViewModel), получая из нее данные и отправляя события на обработку. Однако допускается выполнение простых вычислений непосредственно в представлениях. В некоторых случаях, когда представление содержит только статические данные или не требует сложной логики, оно может работать напрямую с моделью данных без использования ViewModel. Это особенно актуально для простых компонентов, таких как кнопки или отдельные элементы списка.

Представления также могут взаимодействовать с Composition Local, получая данные из общего контекста, что упрощает управление состоянием и передачу зависимостей. Функциональность представлений в системе «умного дома»:

1 Пользовательский опыт. Важно учитывать удобство использования интерфейса: логичную навигацию, интуитивные элементы управления и четкую обратную связь при взаимодействии с системой.

2 Визуализация данных. Jetpack Compose позволяет создавать динамические и адаптивные интерфейсы для отображения данных. Это могут быть графики, диаграммы, карточки или таблицы, отображающие текущее состояние устройств системы «умного дома».

3 Элементы управления, такие как Button, Switch, Slider, TextField, позволяют пользователю управлять устройствами «умного дома». Использование состояний (State) и событий (Event) обеспечивает интерактивность и отзывчивость интерфейса.

Таким образом, использование Jetpack Compose в разработке интерфейса приложения «умного дома» позволяет создавать гибкие, адаптивные и удобные представления, обеспечивая качественный пользовательский опыт.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Функциональное проектирование фокусируется на создании корректно работающего приложения, определяя его ключевые возможности и структуру. В этом разделе рассматриваются логические блоки системы, их классы, методы и выполняемые функции. Также представлены диаграмма классов на чертеже ГУИР.400201.060 РР.1 и диаграмма последовательности на чертеже ГУИР.400201.060 РР.2.

3.1 Представления

Разработка приложения производится с использованием фреймворка Jetpack Compose, который основан на декларативном и функциональном подходе к созданию пользовательского интерфейса. В рамках данного подхода представления строятся с помощью функций-компонентов (Composable), которые возвращают описание UI-элементов. Каждый экран приложения представлен в виде набора таких функций, которые принимают параметры, управляют состоянием и отрисовывают элементы интерфейса.

3.1.1 Функция `AuthorizationScreen` отвечает за графический интерфейс экрана авторизации в приложении, предназначенном для управления системой «умный дом». Она предоставляет пользователю возможность ввести учетные данные MQTT-брокера, необходимые для установления соединения с сервером.

Основные компоненты интерфейса:

1 Элемент `Image` используется для отображения логотипа приложения посредине в верхней части экрана.

2 Форма `BrokerInputForm` включает поля для ввода адреса (URI), порта, имени пользователя и пароля, а также кнопку (`Button`), вызывающую функцию `onAddBroker` для сохранения введённых данных и добавления нового брокера в приложении.

3 Список `BrokerList` отображает последний MQTT-брокер, к которому производилось подключение, с помощью `Text` и `BrokerItem`, позволяя повторно подключиться (`onLogin`) или удалить запись (`onDelete`).

4 Компонент `BrokerItem` оформляет информацию о брокере в `Card`, включая `Text` для отображения URI, порта, а также (если указаны) логина и замаскированного пароля. Внизу расположены две кнопки: `Button` для подключения к брокеру и `OutlinedButton` для удаления записи.

Данный интерфейс предоставляет возможность добавить информацию для подключения к новому MQTT-брокеру, используя соответствующие учётные данные, а при повторном входе – без ввода данных подключиться к последнему использованному брокеру или выбрать другой из списка сохранённых.

3.1.2 `HomeScreen` является основным экраном приложения, предоставляющий удобный интерфейс для навигации между ключевыми разделами: устройства, комнаты и настройки. Пользователь может легко переключаться между вкладками, что обеспечивает быстрый доступ к необходимым данным без задержек и перезагрузки.

Функциональные особенности:

– горизонтальная навигация реализована через `LazyRow`, где каждая вкладка представлена компонентом `TabButton`, визуально выделяющим активный раздел;

– состояние активной вкладки отслеживается через переменную `selectedTab`, а `previousTab` используется для анимации плавных переходов;

– динамическое обновление контента осуществляется с помощью `AnimatedContent`, который меняет отображаемый экран в зависимости от текущей вкладки;

– анимации переходов включают эффекты `slideInHorizontally`, `slideOutHorizontally`, `fadeIn` и `fadeOut`, создавая плавный и интуитивный пользовательский опыт;

– адаптация интерфейса под различные размеры экранов: если ширины экрана недостаточно, `LazyRow` позволяет прокручивать вкладки;

– оптимизация производительности, позволяющая избежать лишних перерисовок и снизить нагрузку на устройство, что особенно важно для слабых смартфонов.

Благодаря этим особенностям `HomeScreen` обеспечивает удобную и быструю работу с основным функционалом приложения.

3.1.3 Интерфейс `DevicesScreen` отвечает за отображение списка устройств, доступных в системе, и предоставляет пользователю возможность управления их состоянием.

Основные компоненты экрана:

– `Scaffold` – используется в качестве контейнера, обеспечивающего корректное размещение элементов и управление отступами;

– `Column` – главный контейнер, содержащий все элементы экрана и обеспечивающий вертикальное расположение компонентов;

– вертикальная прокрутка `verticalScroll()` позволяет комфортно просматривать список устройств при их большом количестве;

– `DeviceCard` – карточка устройства, содержащая изображение, название, тип устройства и его текущее состояние с возможностью управления.

Для загрузки данных используется `devicesViewModel.devices`, который с помощью `collectAsState()` получает актуальный список всех устройств. Фильтрация по типу выполняется с помощью функции `getDevicesByTypeFlow(type)`, где в качестве аргумента указывается тип устройства, а актуальные состояния устройств отслеживаются в `DeviceState.devicesData`, что позволяет оперативно обновлять

информацию.

Динамическое обновление интерфейса обеспечивается автоматическим реагированием UI на изменения в данных `devicesViewModel.devices`, благодаря чему состояние устройств остается актуальным без необходимости ручного обновления. Текущее состояние каждого устройства извлекается из `deviceState`, что гарантирует мгновенное отображение изменений.

3.1.4 `SettingsScreen` – это экран настроек, отвечающий за управление дополнительными параметрами и средствами в системе умного дома. В частности, здесь реализовано управление некоторыми настройками шлюза, включая режим поиска устройств.

В данном представлении реализован функционал:

– включение и отключение режима обнаружения новых устройств;

– отображение времени работы режима обнаружения;

– управление настройками подсветки координатора.

Экран настроек состоит из следующих компонентов:

– основной контейнер (`Column`) – обеспечивает расположение элементов в столбец с отступами между ними;

– карточки (`Card`) – используется для отображения различных функциональных компонентов системы;

– переключатель (`Switch`) – позволяет производить управление переключаемых параметров координатора;

– текстовые элементы (`Text`) – отображают информацию о работе координатора.

Метод `onDiscoverySwitchChanged(enabled: Boolean)` отвечает за управление режимом обнаружения новых устройств. При передаче `true` вызывается функция `startDiscovery()`, иначе – `stopDiscovery()`, прекращающая поиск.

Функция `LaunchedEffect(discoveryState)` асинхронно отслеживает изменение состояния поиска (`discoveryState`). Если поиск включён, переменная `remainingTime` инициализируется значением 255, соответствующее 255 секундам, затем в цикле каждую секунду значение уменьшается на единицу функцией управления задержками `delay(1000)`, пока не достигнет нуля. Если поиск отключается, значение `remainingTime` устанавливает значение 0.

3.1.5 Функция `RoomsScreen` реализует функционал управления системой группирования устройств по их территориальному расположению в помещении. Данный пользовательский интерфейс позволяет просматривать список существующих комнат, отображать количество устройств в каждой из них и добавлять новые комнаты в систему.

В данной функции описаны следующие атрибуты:

– `showDialog` – переменная состояния, которая отвечает за отображение диалогового окна добавления новой комнаты. Если `true`, отображается

диалоговое окно AddRoomDialog, иначе компонент скрыт.

– rooms – коллекция комнат, полученная из roomsViewModel, используемая для построения списка комнат в LazyColumn.

Для добавления новой комнаты используется FloatingActionButton, который при нажатии изменяет состояние showDialog на true, вызывая отображение AddRoomDialog.

После ввода названия комнаты и подтверждения действий, вызывается функция roomsViewModel.addRoom(roomName), которая добавляет новую комнату в список. После этого состояние отображения диалогового окна устанавливается в false.

3.1.6 Интерфейс DeviceDetailScreen

3.2 Модели представлений

Взаимодействие пользовательского интерфейса с данными и бизнес-логикой осуществляется с использованием архитектурного компонента ViewModel. ViewModel отвечает за хранение и управление состоянием экрана, обеспечивая его устойчивость к изменениям конфигурации, таким как поворот экрана. Он предоставляет данные в Composable-функции представлений через механизмы хранения состояний, а также содержит методы для обработки пользовательских действий.

Для каждого представления реализован соответствующий фабричный класс ViewModelProvider.Factory, который отвечает за создание экземпляров классов с передачей необходимых зависимостей, обеспечивая инъекцию зависимостей и соблюдение принципов инверсии управления.

3.2.1 Класс AuthorizationViewModel реализует логику управления данными и состоянием экрана авторизации.

Основные атрибуты класса:

1 _brokers – приватное свойство типа MutableState<List<Broker>>, хранящее текущий список брокеров. Оно используется для внутреннего управления состоянием и позволяет изменять данные внутри ViewModel.

2 brokers – публичное свойство типа State<List<Broker>>, предоставляющее доступ к списку брокеров только для чтения. Оно обеспечивает реактивное обновление UI при изменении данных (например, при добавлении или удалении брокеров).

Методы класса:

1 addBroker() асинхронно добавляет нового MQTT-брокера в локальную базу данных и обновляет UI. Метод выполняется в корутине через viewModelScope.launch, создаёт объект Broker с переданными serverUri, serverPort, user и password (поддерживая анонимное подключение), затем сохраняет его в базе данных с помощью brokerDao.insert(), после этого вызывается метод loadBrokers(), чтобы загрузить обновлённый список

брокеров, что автоматически обновляет UI благодаря реактивному состоянию `_brokers`.

2 `deleteBroker()` асинхронно удаляет информацию о брокере из базы данных и обновляет UI. Он выполняется в корутине, сначала отключает MQTT-клиент с помощью `MQTTClient.getInstance().disconnect()`, затем удаляет брокера через `brokerDao.deleteBroker(broker)`, после чего вызывает `loadBrokers()`.

3 `handleLogin()` устанавливает текущий брокер, создаёт MQTT-клиент и выполняет подключение. Он обновляет `BrokerState` с `broker.id`, затем переинициализирует клиент `MQTTClient` с указанным брокером и обработчиком сообщений. Если подключение успешно, вызывается функция `onSuccess()`. Далее асинхронно с использованием `viewModelScope.launch(Dispatchers.IO)` клиент подписывается на необходимые для работы топики.

3.2.2 `DevicesViewModel` это модель представления, которая служит для управления устройствами умного дома, обеспечивая их загрузку, обновление и взаимодействие с MQTT брокером.

Атрибут `_devices` – это `MutableStateFlow`, хранящий список устройств и используемый для управления их состоянием. Он инициализируется пустым списком и изменяется внутри `viewModelScope.launch`, что позволяет выполнять загрузку устройств из базы данных в фоновом режиме и не блокировать основной поток выполнения программы. Для получения данных используется атрибут `devices` типа `StateFlow`.

Методы, используемые для работы с данными:

1 `loadDevices(brokerId: Int)` – асинхронно загружает устройства, связанные с брокером, подписываясь на `Flow` из базы данных. При обновлении данных обновляет `_devices`, что автоматически отражается в пользовательском интерфейсе.

2 `getDeviceIdByIeeeAddr(addr: String, callback: (Int?) -> Unit)` – получает `ID` устройства по его адресу и передает результат в `callback`.

3 `getDevicesByTypeFlow(type: String)` – возвращает `StateFlow` с устройствами указанного типа. Фильтрует `_devices` на основе команд, полученных через `Flow`.

4 `getDevicesByRoomIdFlow(roomId: Int)` – аналогично предыдущему методу фильтрует устройства по их типу, но дополнительно учитывает идентификатор комнаты `roomId`, обновляя `resultFlow`.

5 `addDeviceIfNotExists(device: Device)` – добавляет устройство в локальное хранилище, если его там нет. Проверяет наличие устройства по его физическому адресу `ieeeAddr`, при отсутствии добавляет новую запись с информацией об устройстве.

6 `updateDeviceName(deviceId: Int, newName: String)` – изменяет имя устройства, обновляя его в базе и списке `_devices`.

7 `addCommandIfNotExists(command: Command)` – добавляет команду в базе данных, если ее еще нет, проверяя по `commandTopic`.

8 `sendCommandToMqtt(topic: String, command: String)` – публикует команду управления устройством в заданный топик.

9 `assignRoomToDevice(deviceId: Int, roomId: Long?)` – обновляет `roomId` устройства в базе.

3.2.3 Класс `SensorsViewModel`

3.3 Модели

В данном подразделе представлены таблицы моделей данных и связи между ними. Типы данных указаны в нотации Kotlin, как они описаны в исходном коде. Опциональность значений обозначается вопросительным знаком после типа данных.

Для создания сущности и соответствующей ей таблицы в базе данных используется аннотация `@Entity`. В её параметрах указывается служебная информация для библиотеки Room, такая как:

– `tableName` – название таблицы;

– `foreignKeys` – внешний ключ, определяющий связи между таблицами;

– `indicies` – список индексов, ускоряющих поиск данных при большом объёме информации.

Описание модели состоит из двух частей:

– описание атрибутов таблицы – перечень столбцов и их характеристики;

– описание связей таблицы – указание внешних ключей, индексов и каскадных правил.

3.3.1 Класс `Device` представляет собой сущность базы данных, представленной в таблице `devices`, предназначенную для хранения информации об устройствах в системе «умного дома».

Описание атрибутов таблицы:

– `id: Int` – первичный ключ таблицы, автоматический генерируемый с помощью аннотации `@PrimaryKey(autoGenerate = true)`;

– `ieeeAddr: String` – уникальный физический идентификатор устройства;

– `friendlyName: String` – удобочитаемое название устройства, значение которого задается пользователем для его идентификации;

– `modelId: String` – уникальный идентификатор модели устройства, соответствующий заводскому обозначению производителя и определяющий тип и функциональность устройства;

– `topic: String` – MQTT топик, в котором публикуется информация о состоянии устройства;

– roomId: Long? – идентификатор комнаты, в которой установлено устройство;

– brokerId: Int – идентификатор брокера, через которого устройство обменивается информацией.

Связи между таблицами определены с помощью внешних ключей. Поле brokerId связано с id таблицы Broker, и при удалении брокера все связанные устройства также удаляются (с параметром onDelete = ForeignKey.CASCADE). Параметр roomId связан с id таблицы RoomEntity, и, если комната удаляется, у всех устройств, связанных с этой комнатой, значение roomId становится null (с параметром onDelete = ForeignKey.SET_NULL).

Объект компаньон в классе Device используется для создания нового устройства с автоматическим формированием топика. Топик генерируется на основе физического адреса устройства в формате zigbee/0x<ieeeAddr>.

Кроме того, в таблице используется индексация по полю roomId, что значительно ускоряет поиск устройств, принадлежащих определённой комнате.

3.3.2 Класс Broker представляет собой сущность базы данных, отображаемую в таблице brokers, предназначенную для хранения информации о MQTT-брокерах, используемых в системе «умного дома».

Поля класса Broker:

– id: Int – первичный ключ таблицы с автоматической генерацией;

– serverUri: String – URI MQTT-брокера, который используется для подключения к нему;

– serverPort: Int – порт, через который осуществляется подключение к брокеру;

– user: String? – имя пользователя для аутентификации при подключении к брокеру (может быть null, если аутентификация не требуется);

– password: String? – пароль пользователя для подключения к брокеру (может быть null, если аутентификация не требуется).

Поле id используется в таблице devices как внешний ключ (brokerId). Если брокер удаляется, все связанные с ним устройства также удаляются, что обеспечивается параметром onDelete = ForeignKey.CASCADE в определении внешнего ключа.

В таблице brokers можно хранить несколько брокеров, что позволяет системе «умного дома» работать с разными MQTT-серверами.

3.3.3 Класс Command представляет собой сущность базы данных, отображаемую в таблице commands, предназначенную для хранения информации о командах управления устройствами в системе «умного дома».

Атрибуты таблицы:

– id: Int – первичный ключ таблицы;

– deviceId: Int – внешний ключ, связывающий команду с устройством из таблицы devices. При удалении устройства все связанные с ним команды также удаляются;

– commandTopic: String – MQTT-топик, в который отправляются команды для управления устройством;

– payloadOn: String? – полезная нагрузка для включения устройства (может быть null, если команда не предназначена для переключателя);

– payloadOff: String? – полезная нагрузка для выключения устройства (может быть null, если команда не предназначена для переключателя);

– options: Map<String, String>? – список доступных опций для команд выбора (например, вариантов переключения режимов устройства), для хранения в базе данных с использованием MapTypeConvertor;

– commandTemplate: String? – шаблон команды, который может быть использован для формирования сложных сообщений в MQTT (может быть null, если команда не требует шаблонизации);

– commandType: String – тип команды (например, switch, select, custom), определяющий логику её обработки.

Поле deviceId связано с идентификатором таблицы devices. При удалении устройства все связанные с ним команды автоматически удаляются (onDelete = ForeignKey.CASCADE).

Таблица commands позволяет хранить команды управления устройствами умного дома, поддерживая как простые команды (включить/выключить), так и более сложные (выбор режима работы).

3.3.4 Класс RoomEntity представляет собой сущность, информация о которой хранится в таблице rooms.

Поля сущности:

– id: Long – идентификатор, выступающий в роли первичного ключа таблицы;

– name: String – название комнаты, которое задается пользователем для удобного восприятия в системе.

Поле id используется в таблице devices как внешний ключ (roomId). Если комната удаляется, у всех связанных с ней устройств значение roomId принимает значение null (onDelete = ForeignKey.SET_NULL), что позволяет избежать удаления устройств из системы при удалении комнаты.

Таблица rooms позволяет группировать устройства по местоположению в помещении.

3.4 Хранение данных

В приложении используется локальная база данных на основе SQLite, управляемая через библиотеку Room. Это позволяет эффективно хранить и обрабатывать информацию обеспечивая удобный доступ к данным и их

целостность.

Взаимодействие с хранимой информацией осуществляется через DAO-интерфейсы. Помимо постоянного хранения данных в базе, в приложении используются singleton-объекты, отвечающие за временное хранение состояний системы. Их основная задача – обеспечение быстрого доступа к актуальной информации без необходимости выполнения запросов к базе данных. Эти состояния существуют только во время работы приложения и не сохраняются в базе, поскольку их значения могут динамически изменяться в зависимости от текущего сеанса работы системы.

3.4.1 Класс `RoomLocalDatabase` является основной точкой доступа к базе данных SQLite, реализованной с использованием библиотеки `Room`.

Атрибуты класса базы данных задаются в качестве параметров аннотации `@Database`, в которой указываются:

– `entities` – список моделей используемых в базе данных (`Broker, Device, Command`);

– `version` – версия базы данных;

– `exportSchema` – флаг экспорта схемы, установленный в значение false, что означает создание таблиц производится на основе моделей, описанных в приложении.

Методы класса:

– `brokerDAO(), deviceDAO(), commandDAO(), roomEntityDAO()` – абстрактные методы, предоставляющие доступ к DAO-объектам для работы с таблицами brokers, devices, commands и rooms соответственно;

– `getInstance()` – статический метод, реализующий порождающий паттерн Singleton для создания и получения единственного экземпляра базы данных, используя `Room.databaseBuilder`.

3.4.2 Класс `BrokerState` определён с помощью ключевого слова `object`, что делает его singleton-объектом – он создаётся один раз и существует в единственном экземпляре в рамках всего приложения.

Этот класс предназначен для хранения идентификатора текущего MQTT-брокера, который используется в данный момент.

Основные компоненты данного класса:

– `_brokerId` – приватное поле типа `MutableStateFlow<Int?>`, хранящее идентификатор брокера и позволяющее его изменять внутри методов класса.

– `brokerId` – публичное свойство типа `StateFlow<Int?>`, предоставляющее неизменяемый поток данных, содержащий идентификатор текущего брокера;

– `setBrokerId(id: Int)` – метод, обновляющий идентификатор брокера, устанавливая новое значение в `_brokerId`.

3.4.3 Класс `DeviceState` так же определён с помощью ключевого слова `object`, что делает его singleton-объектом.

Данный класс используется для хранения состояний устройств «умного дома», содержащего актуальные данные, передаваемые устройствами.

Основные компоненты класса:

– `_devicesData` – приватное поле типа `MutableStateFlow<Map<Int, Map<String, Any>>>`, хранящее данные всех устройств, где ключом верхнего уровня является `deviceId`, а значением – параметры устройств, представленных в виде структуры `Map`;

– `devicesData` – публичное свойство типа `StateFlow<Map<Int, Map<String, Any>>>`, предоставляющее неизменяемый поток данных с текущими состояниями устройств;

– `updateDeviceData(deviceId: Int, payload: String)` – метод, обновляющий данные конкретного устройства, принимая идентификатор устройства `deviceId` и строку в формате JSON `payload`;

– `parseJson(json: String)` – метод, преобразующий JSON-строку в структуру `Map<String, Any>` используемую для хранения состояния устройства;

– `getDeviceValue(deviceId: Int, key: String)` – метод, возвращающий конкретное значение из состояния устройства по его идентификатору и заданному ключу `key`.

3.4.4 Класс `DiscoveryState` определен ключевым словом `object`. Данный класс используется для управления процессом обнаружения и подключения устройств в систему «умный дом». Он содержит методы для запуска и остановки процесса обнаружения, а также отслеживает его текущее состояние.

Атрибуты класса:

1 `_isDiscoveryActive` – поле типа `MutableStateFlow<Boolean>`, которое хранит текущее состояние процесса обнаружения устройств. Если значение равно `true`, это означает, что процесс обнаружения активен, если `false` — процесс неактивен;

2 `isDiscoveryActive` – атрибут типа `StateFlow<Boolean>`, которое предоставляет доступ к текущему состоянию процесса обнаружения устройств. Это свойство является неизменяемым, что позволяет внешним компонентам подписываться на изменения состояния, но не изменять его напрямую;

3 `resetJob` – приватное поле типа `Job?` которое хранит ссылку на корутину, отвечающую за автоматическое завершение процесса обнаружения через определённый промежуток времени. Это поле может быть `null`, если процесс обнаружения неактивен.

Методы, описанные в классе:

1 метод `startDiscovery()` устанавливает значение `true` для атрибута `_isDiscoveryActive`, что указывает на начало процесса обнаружения

устройств. Он отправляет MQTT-сообщение в топик DISCOVERY_TOPIC со значением true, чтобы разрешить устройствам Zigbee подключаться к сети. Если существует предыдущая корутина, она отменяется. Затем запускается новая корутина, которая через DISCOVERY_TIME автоматически завершает процесс обнаружения, устанавливая _isDiscoveryActive в false.

2 stopDiscovery() устанавливает значение _isDiscoveryActive в false, что указывает на завершение процесса обнаружения. Он отправляет MQTT-сообщение в аналогичный топик со значением false, чтобы запретить устройствам Zigbee подключаться к сети. Если текущая корутина существует, она отменяется.

3.4.5 Класс DeviceDAO представляет собой DAO-интерфейс (Data Access Object) для работы с таблицей устройств в базе данных. Он определён с помощью аннотации @Dao и содержит методы для взаимодействия с данными.

Основные методы:

– getDevicesByBroker(brokerId: Int) – получает список информации об устройствах, связанных с указанным брокером используя его ID;

– getDevicesByBrokerFlow(brokerId: Int) – возвращает поток (Flow), содержащий информацию об устройствах для заданного брокера по его идентификационному номеру;

– getAllDevices() – загружает информацию о всех устройствах, хранящиеся в базе данных;

– getDeviceByIeeeAddr(ieeeAddr: String) – ищет устройство с использованием его физического адреса (IEEE адрес), возвращает null, если устройство не найдено;

– getDeviceById(deviceId: Int) – получает устройство по его deviceId;

– insertDevice(device: Device) – добавление устройства в базу данных, игнорируя конфликты. Возвращает deviceId добавленной записи или -1, если устройство уже существует;

– getDevicesByRoomIdFlow(roomId: Int) – возвращает поток Flow, содержащий список устройств, принадлежащих указанной комнате (roomId);

– updateDevice(device: Device) – обновляет информацию об устройстве в базе данных;

– getDeviceCountForRoom(roomId: Long) – получает количество устройств, связанных с заданной комнатой (roomId).

3.4.6 Класс BrokerDAO является DAO-интерфейсом для взаимодействия приложения с таблицей, содержащей информацию о брокерах.

Методы класса:

– insert(broker: Broker) – добавляет брокера в базу или обновляет его при конфликте;

– `getAllBrokers()` – получает список всех брокеров;

– `getLastBroker()` – получает последнего добавленного брокера (по убыванию ID), возвращает `null`, если брокеров нет;

– `deleteBroker(broker: Broker)` – удаляет запись содержащую информацию об указанном брокере.

3.4.7 Класс `CommandDAO` представляет собой интерфейс, обеспечивающий доступ к данным команд, хранящимся в базе.

Методы, реализуемые в данном интерфейсе:

– `getCommandByCommandTopic(commandTopic: String)` – получает команду по заданной теме (`commandTopic`), возвращает `null`, если такой команды нет;

– `getSwitchCommandByDeviceId(deviceId: Int)` – ищет команду типа «`switch`» по `deviceId`, возвращает `null`, если команда не найдена;

– `insertCommand(command: Command)` – добавляет новую команду в базу или обновляет существующую при конфликте;

– `getCommandsByTypeFlow(type: String)` – возвращает `Flow` со списком команд заданного типа.

3.4.8 `RoomEntityDAO` – класс, предназначенный для управления данными о комнатах в базе данных.

Доступные функции для работы с данными:

– `getAllRooms()` – возвращает поток `(Flow)` со списком всех комнат;

– `insertRoom(room: RoomEntity)` – добавляет новую комнату или обновляет существующую при конфликте.

3.5 MQTT и взаимодействие с брокером

В данном разделе рассматривается организация взаимодействия с MQTT-брокером в разрабатываемом приложении. Основной задачей является обмен сообщений через MQTT-протокол для управления устройствами умного дома.

3.5.1 Интерфейс `MQTTMessaging`

3.5.2 Интерфейс `MQTTConnection`

3.5.3 Класс `MQTTClient` представляет собой реализацию MQTT-клиента для взаимодействия с брокером. Данный класс реализует интерфейсы `MQTTMessaging` и `MQTTConnection`. Он позволяет настроить соединение с учетом параметров аутентификации, обрабатывать входящие сообщения через функции-колбэки и управлять подписками.

Атрибуты класса:

– `mqttClient: MqttClient?` – объект клиента MQTT;

– `broker: Broker?` – содержит информацию о брокере;

– `messageHandler: MQTTMessageHandler?` – обработчик входящих сообщений.

Методы, реализованные в классе:

– `initialize(Broker, MQTTMessageHandler)` – выполняет инициализацию клиента, устанавливая параметры подключения к брокеру и назначая обработчик сообщений;

– `reinitialize(Broker, MQTTMessageHandler)` – завершает текущее соединение, перенастраивает параметры подключения и повторно инициализирует клиента с новым брокером;

– `connect(): Boolean` – выполняет подключение к брокеру с учетом учетных данных. Возвращает true в случае успешного соединения и false при возникновении ошибки;

– `subscribe(topic: String)` – подписывает клиента на указанный MQTT-топик и передает полученные сообщения в `messageHandler`;

– `publish(topic: String, payload: String, qos: Int, retained: Boolean)` – отправляет сообщение в указанный топик с заданным уровнем качества обслуживания (QoS) и возможностью хранения последнего значения (retained);

– `disconnect()` – разрывает соединение с MQTT-брокером и освобождает ресурсы.

3.5.4 Класс `MQTTMessageHandler` отвечает за обработку входящих MQTT-сообщений и взаимодействие с моделями данных приложения.

Метод `handleMessage(topic: String, payload: String)` является основной функцией для обработки входящих MQTT-сообщений. Анализирует `topic` сообщения и вызывает соответствующий обработчик в зависимости от его типа. Это позволяет корректно маршрутизировать данные в системе.

Метод `handleDeviceStateMessage(topic: String, payload: String)` отвечает за обновление состояния устройств. Разбирает данные, полученные в сообщении, и обновляет соответствующую запись в модели состояния устройств. Может использоваться для синхронизации данных между физическими устройствами и интерфейсом пользователя.

Метод `handleDeviceCommandMessage(topic: String, payload: String)` предназначен для обработки команд, отправленных устройствам. Проверяет, существует ли уже такая команда в базе данных, и, если нет, сохраняет её. Это позволяет избежать дублирования команд и обеспечивает корректное управление устройствами.

Метод `handleDeviceListMessage(payload: String)` выполняет обработку JSON-данных, содержащих список доступных устройств. Если какое-либо устройство еще не добавлено в систему, оно записывается в базу данных. Этот механизм позволяет автоматически обновлять список доступных устройств в приложении.

Данный класс обеспечивает корректную маршрутизацию MQTT сообщений и взаимодействие с внутренними структурами данных

приложения.

3.5.5 Класс `Topics` содержит предопределенные константы, представляющие MQTT-топики, с которыми взаимодействует приложение. Эти топики используются для обмена данными с MQTT-брокером и управления устройствами умного дома.

Атрибуты:

– `DISCOVERY_TOPIC` – топик для включения режима сопряжения новых Zigbee-устройств;

– `DEVICE_STATE_TOPIC` – шаблон топиков, содержащих информацию о состоянии устройств;

– `DEVICE_COMMANDS_TOPIC` – базовый топик для получения информации о командах для управления устройствами;

– `DEVICE_LIST_TOPIC` – топик для получения списка доступных устройств.

3.6 Вспомогательные компоненты системы

3.6.1 Разметка AndroidManifext.xml
3.6.2 Класс Logger
3.6.3 Класс Constants
3.6.4 Объект Extensions

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

6 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И РЕАЛИЗАЦИИ НА РЫНКЕ ANDROID-ПРИЛОЖЕНИЯ ДЛЯ УПРАВЛЕНИЯ И МОНИТОРИНГА УСТРОЙСТВАМИ УМНОГО ДОМА

7.1 Характеристика программного средства, разрабатываемого для реализации на рынке

Созданный дипломный проект представляет собой нативное приложение для операционной системы Android, которое позволяет пользователям управлять устройствами «умного дома» через протокол MQTT для DIY систем.

Целью разработки проекта является упрощение управления устройствами «умного дома» и обработки данных о их состоянии. Приложение предназначено для пользователей, позволяя им эффективно контролировать устройства, настраивать автоматизации и оперативно получать информацию об их работе.

Целевой аудиторией данного приложения являются пользователи систем «умного дома», которым необходим удобный инструмент для управления устройствами через протокол MQTT. Также потенциальными пользователями могут быть энтузиасты DIY-решений, использующие платформы, такие как HomeAssistant, а также владельцы экосистем Aqara, Xiaomi, Tuya и Яндекс.

На момент разработки существует большое количество решений для управления устройствами «умного дома», однако большинство из них ориентированы на конкретные экосистемы, такие как Aqara, Xiaomi, Tuya и Яндекс. Существующие приложения часто имеют ограниченный функционал и не поддерживают интеграцию с DIY-системами.

Планируется распространение приложения через Google Play с возможностью монетизации, включая бесплатную и расширенную платную версии.

7.2 Расчет инвестиций в разработку программного средства

7.2.1 Расчет зарплат на основную заработную плату разработчиков производится исходя из количества людей, которые занимаются разработкой программного продукта, месячной зарплаты каждого участника процесса разработки и сложности выполняемой ими работы. Затраты на основную заработную плату рассчитаны по формуле:

$$З_о = К_{пр} \sum_{i=1}^{n} З_{чi} \cdot t_i \,, \qquad (7.1)$$

где n – количество исполнителей, занятых разработкой конкретного ПО;

$К_{пр}$ – коэффициент, учитывающий процент премий;

$З_{ч.i}$ – часовая заработная плата i-го исполнителя, р.;

t_i – трудоемкость работ, выполняемых i-м исполнителем, ч.

Разработкой всего приложения занимается инженер-программист, обязанности тестирования приложения лежат на инженере-тестировщике. Задачами инженера-программиста, который занимается являются создание модели данных, графического интерфейса, связи между моделью данных и графическим интерфейсом. Инженер-тестировщик занимается выявлением неработоспособных частей приложения, а также оценивает пользовательский опыт, получаемый от использования приложения.

Месячная заработная плата основана на медианных показателях для Junior инженера-программиста за 2024 год по Республике Беларусь, которая составляет примерно 807 долларов США в месяц, а для Junior инженера-тестировщика – 466 долларов США [+]. По состоянию на 21 февраля 2025 года, 1 доллар США по курсу Национального Банка Республики Беларусь составляет 3,2239 белорусских рубля.

В перерасчете на белорусские рубли месячные оклады для инженера-программиста и инженера-тестировщика составляют 2601,68 и 1502,33 белорусских рублей соответственно.

Часовой оклад исполнителей высчитывается путем деления месячного оклада на количество рабочих часов в месяце, то есть 160 часов.

За количество рабочих часов в месяце для инженера-программиста и инженера-тестировщика принято соответственно 196 и 32 часа. Коэффициент премии приравнивается к единице, так как она входит сумму заработной платы. Затраты на основную заработную плату приведены в таблице:

Таблица 7.1 – Затраты на основную заработную плату сотрудников

Категория исполнителя	Месячный оклад, р	Часовой оклад, р	Трудоемкость работ, ч	Итого, р
Инженер-программист	2601,68	16,26	196	3186,96
Инженер-тестировщик	1502,33	9,39	32	300,46
Итого				3487,42
Премия и стимулирующие зарплаты (0%)				0
Общие затраты на основную заработную плату разработчиков				3487,42

7.2.2 Расчет затрат на дополнительную заработную плату разработчиков, предусмотренных законодательством о труде, осуществлялся по формуле 7.2:

$$З_д = \frac{З_о \cdot Н_д}{100\%}, \quad (7.2)$$

где Зо – затраты на основную заработную плату, р.;

$Н_д$ – норматив дополнительной заработной платы.

Норматив дополнительной заработной платы был принят равным 10%.

Размер дополнительной заработной составил:

$$З_д = \frac{З_о \cdot Н_д}{100\%} = \frac{3\ 487{,}42 \cdot 10}{100\%} = 348{,}724 \text{ р.}$$

7.2.3 Расчет отчислений на социальные нужды производился в соответствии с действующими законодательными актами по формуле 7.3:

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100\%}, \quad (7.3)$$

где $Н_{соц}$ – норматив отчислений от фонда оплаты труда.

Норматив отчислений от фонда оплаты труда дополнительной заработной платы был принят равным 35%.

Размер дополнительной заработной платы составил:

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100\%} = \frac{(3\ 487{,}42 + 348{,}724) \cdot 35}{100\%} = 1\ 342{,}65 \text{ р.}$$

7.2.4 Расчет затрат на прочие расходов был произведен по формуле 7.4:

$$Р_{пр} = \frac{З_о \cdot Н_{пр}}{100\%}, \quad (7.4)$$

где $Н_{пр}$ – норматив прочих расходов.

Норматив прочих расходов был принят равным 30%.

Размер прочих расходов составил:

$$Р_{пз} = \frac{З_о \cdot Н_{пр}}{100\%} = \frac{3\ 487{,}42 \cdot 30}{100\%} = 1046{,}226 \text{ р.}$$

7.2.5 Расчет расходов на реализацию продукта рассчитан по формуле 7.5:

$$Р_{пр} = \frac{З_о \cdot Н_р}{100\%} \quad (7.5)$$

где $Н_р$ – норматив расходов на реализацию

Норматив расходов на реализацию был принят 3%.

Размер расходов на реализацию составил:

$$Р_{пз} = \frac{З_о \cdot Н_р}{100\%} = \frac{3\ 487{,}42 \cdot 3}{100\%} = 104{,}6226 \text{ р.}$$

Полная сумма затрат на разработку программного средства представлена в таблице 7.2.

Таблица 7.2 – Полная сумма затрат на разработку программного средства

Наименование статьи затрат	Сумма, р.
Основная заработная плата разработчиков	3 487,42
Дополнительная заработная плата разработчиков	348,72
Отчисления на социальные нужды	1 342,65
Прочие расходы	1 046,22
Расходы на реализацию	104,622
Общая сумма инвестиций (затрат) на разработку	6 329,64

7.3 Расчёт экономического эффекта от реализации программного средства на рынке

Для расчёта экономического эффекта организации-разработчика программного средства, а именно чистой прибыли, необходимо знать такие параметры как объем продаж, цену реализации и затраты на разработку.

Соответственно необходимо создать обоснование возможного объема продаж, количество проданных лицензий расширенной версии программного средства, купленного пользователями.

В Беларуси проживает около 9,2 миллиона человек, из которых примерно 8,48 миллионов являются активными интернет-пользователями [+]. По данным [+], доля пользователей Android среди мобильных ОС в Беларуси на 2024 год составляет 83%, что делает платформу наиболее популярной среди владельцев смартфонов.

Учитывая, что использование систем «умный дом» требует наличия совместимых устройств, примем, что 3% от общего числа активных интернет-пользователей уже обладают устройствами умного дома. Из них 80% (будут использовать приложения аналоги от производителей устройств, а оставшиеся 20% (42 200 человек) установят разработанное программное средство. Из них 5000 пользователей приобретут расширенную версию программного обеспечения.

С учетом цены на расширенную версию приложения, которая составляет 2,99 долларов США, и с учетом обменного курса доллара к белорусскому рублю, отпускная стоимость программного средства составит примерно 9,62 белорусских рубля.

Для расчёта прироста чистой прибыли необходимо учесть налог на добавленную стоимость, который высчитывается по следующей формуле:

$$\text{НДС} = \frac{\text{Ц}_{\text{отп}} \cdot N \cdot \text{Н}_{\text{д.с}}}{100\% + \text{Н}_{\text{д.с}}}, \tag{7.6}$$

где N – количество копий программного продукта, реализуемое за год, шт.;

$\text{Ц}_{\text{отп}}$ – отпускная цена копии программного средства, р.;

N – количество приобретённых лицензий;

$\text{Н}_{\text{д.с}}$ – ставка налога на добавленную стоимость, %.

Ставка налога на добавленную стоимость по состоянию на 15 апреля 2024 года, в соответствии с действующим законодательством Республики Беларусь, составляет 20%. Используя данное значение ставки налога, посчитаем НДС:

$$\text{НДС} = \frac{9{,}62 \cdot 5\,000 \cdot 20\%}{100\% + 20\%} = 8\,016 \text{ р.}$$

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли, которую получит разработчик от продажи программного продукта. Для этого используется формулу:

$$\Delta П_{ч}^{р} = (Ц_{отп} \cdot N - \text{НДС}) \cdot Р_{пр} \cdot \left(1 - \frac{Н_{п}}{100}\right), \quad (7.7)$$

где N – количество копий программного продукта, реализуемое за год, шт.;
$Ц_{отп}$ – отпускная цена копии программного средства, р.;
НДС – сумма налога на добавленную стоимость, р.;
$Н_{п}$ – ставка налога на прибыль, %;
$Р_{пр}$ – рентабельность продаж копий.

Ставка налога на прибыль, согласно действующему законодательству с 1 января 2025 равна 20%. Рентабельность продаж копий взята в размере 30%. Зная ставку налога и рентабельность продаж копий (лицензий), рассчитывается прирост чистой прибыли для разработчика:

$$\Delta П_{ч}^{р} = (9{,}62 \cdot 5\,000 - 8\,016) \cdot 30\% \cdot \left(1 - \frac{20}{100}\right) = 9\,620 \text{ р.}$$

7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке

Для того, чтобы оценить экономическую эффективность разработки и реализации программного средства на рынке, необходимо рассмотреть результат сравнения затрат на разработку данного программного продукта, а также полученный прирост чистой прибыли за год.

Сумма затрат на разработку меньше суммы годового экономического эффекта, поэтому можно сделать вывод, что такие инвестиции окупятся менее, чем за один год.

Таким образом, оценка экономической эффективности инвестиций производится при помощи расчёта рентабельности инвестиций (Return on Investment, ROI). Формула для расчёта ROI:

$$ROI = \frac{\Delta П_{ч}^{р} - З_{р}}{З_{р}} \cdot 100\%, \quad (7.8)$$

где $\Delta П_ч^р$ – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.;

$З_р$ – затраты на разработку и реализацию программного средства, р.

$$ROI = \frac{9\,620 - 8\,016}{8\,016} \cdot 100\% = 20\%$$

7.5 Вывод об экономической целесообразности реализации проектного решения

Проведенные расчеты технико-экономического обоснования позволяют сделать предварительный вывод о целесообразности разработки программного продукта для умного дома. Общая сумма затрат на его разработку и реализацию составила 6 329,64 белорусских рублей, а отпускная цена установлена на уровне 9,62 белорусских рублей.

Прогнозируемый прирост чистой прибыли за год, основанный на предполагаемом объеме продаж в размере 5 000 расширенных версий в год, составляет 9 620 белорусских рублей. Рентабельность инвестиций за год оценивается в 20%.

Такие результаты говорят о том, что разработка данного программного продукта является перспективной и имеет экономическое обоснование. Однако, необходимо учитывать возможные риски, связанные с конкуренцией на рынке и возможным недооцениванием продукта со стороны потребителей.

Такой показатель рентабельности может быть приемлем для стабильного бизнеса, но для увеличения доходности следует рассмотреть стратегии оптимизации затрат, повышения ценности продукта для пользователей и расширения рынка сбыта. Дополнительные инвестиции в маркетинг, улучшение функционала и внедрение подписочных моделей могут способствовать росту рентабельности в будущем.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

https://salaries.devby.io/ 17.03.2025

https://bvn.by/2024/04/15/kakimi-smartfonami-chashhe-vsego-polzujutsja-belorusy 17.03.2025

https://myfin.by/article/tekhnologii/tiktok-nabiraet-popularnost-u-belorusov-issledovanie 17.03.2025

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

ПРИЛОЖЕНИЕ Б
(обязательное)

Спецификация

ПРИЛОЖЕНИЕ В
(обязательное)

Ведомость документов