2 ПРОЕКТИРОВАНИЕ, РАЗРАБОТКА И ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

2.1 Проектирование архитектуры и описание состояний программного средства

В проектировании приложения под операционную систему *iOS* существует несколько архитектурных паттернов проектирования, самыми популярными являются *MV(X)* архитектуры: *MVC, MVP, MVVM,* а также *VIPER* и *Clean Swift* [9]. Общая направленность этих паттернов такова, что в любых программных средствах, построенных по данным моделям, должно обеспечиваться разделение данных от внешнего интерфейса, а также соблюдение принципов ООП и *SOLID* [10].

Clean Swift очень похож на *VIPER* своим разделением обязанностей, но взаимодействия сущностей происходят строже и по-другому принципу. Для начала разберем основную терминологию архитектуры. В *Clean Swift* приложение состоит из сцен, т.е. каждый экран приложения – это одна сцена. Основное взаимодействие в сцене идет через последовательный цикл между компонентами *ViewController, Interactor, Presenter* [11]. Это называется *VIP* цикл, который изображён ниже, на рисунке 2.1.

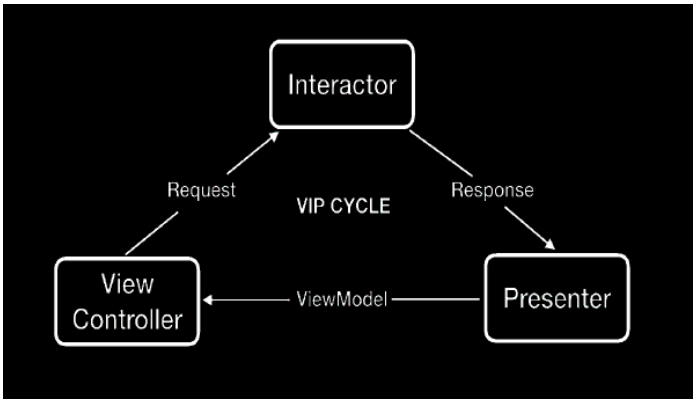

Рисунок 2.1 – Схема *VIP* цикла

Мостом между компонентами выступает файл *Models*, где находятся структуры, которые хранят в себе передаваемые данные. Каждый круг цикла имеет в себе 3 вида структур:

– *Request* – структура с данными для передачи из *ViewController'a* в *Interactor*;

– *Response* – структура с данными, например загруженными из сети, для передачи из *Interactor* в *Presenter*;

– *ViewModel* – структура с обработанными данными в *Presenter'e* для передачи обратно во *ViewController* [12].

ViewController отвечает только за конфигурацию и взаимодействие с View. В контроллере не должно находиться никакой бизнес логики, взаимодействия с сетью, вычислений и так далее. Его задача обрабатывать события с View, отображать или отправлять данные (без обработки и проверок) в Interactor.

Interactor получает запрос из *ViewController'a* (с данными или пустой), обрабатывает его и, если это требуется, передает новые данные в *Presenter*. *Interactor* содержит в себе бизнес логику сцены. Он работает с сетью, базой данных и модулями устройства.

Presenter занимается подготовкой данных для отображения. Как пример, добавить маску на номер телефона или сделать первую букву в названии заглавной. Обрабатывает данные, полученные из *Interactor'a*, после чего отправляет их обратно во *ViewController*. Разгружает *Interactor*, забирая на себя часть бизнес логики приложения, если *Interactor* стремительно разрастается. Ниже, на рисунке 2.2, представлена схема архитектуры *Clean Swift* [13].

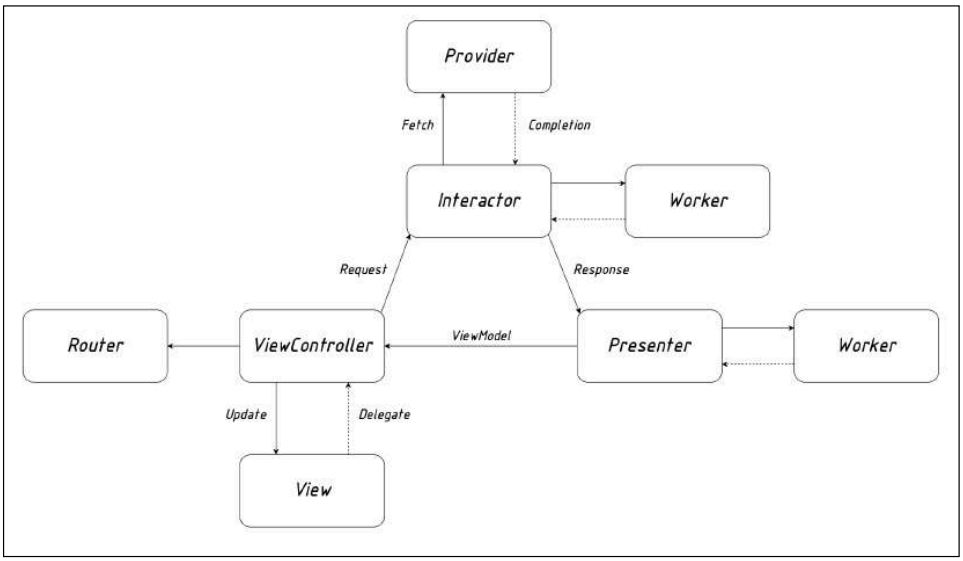

Рисунок 2.2 – Схема архитектуры *Clean Swift*

Так же есть *Router*, отвечающий за переход и передачу данных между сценами, и *Worker*, который берет часть логики *Interactor'a* на себя.

Исходя из всего вышесказанного было принято решение использовать *Clean Swift* в качестве архитектуры приложения. С данной архитектурой получена возможность распределить обязанности, улучшить удобство тестирования приложения, ввести заменяемость отдельных участков реализации.

Структура проекта включает в себя 7 папок.

1 *Application* – включает в себя все, что связано с базой самого приложения. В ней лежат файлы, которые описывают жизненный цикл приложения, формирование базовой сцены.

2 *AppRouting* – включает в себя файлы отвечающие за смену последовательности глобальной навигации.

3 *Elements* – включает в себя кастомные представления, которые используются во всем приложении.

4 *Screens* – включает в себя экраны приложения.

5 *Services* – включает в себя различные сервисы с отдельной функциональностью, такие как сеть, алерты, локализация и так далее.

6 *Extensions* – включают в себя расширения базовых классов.

7 *Resources* – включает в себя ресурсы приложения.

Структура проекта изображена на рисунке 2.3

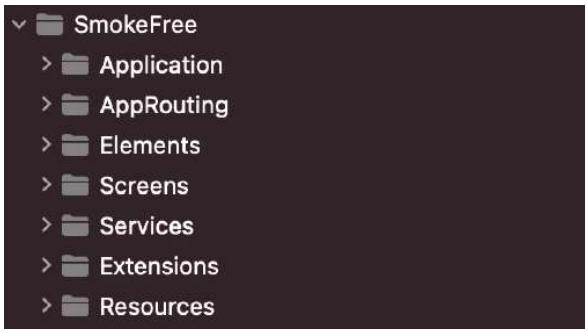

Рисунок 2.3 – Структура проекта

При этом каждая сцена имеет структуру, изображенную на рисунке 2.4.

Рисунок 2.4 – Структура сцены

В данном случае для построения сцены используются как стандартные для данной архитектуры сущности: *SignInViewController*, *SignInInteractor*, *SignInPresenter*, *SignInRouter, SignInModels,* так и дополнительные:

– *SignInAssembly* нужна для внедрения зависимостей между основными сущностями, такая сущность необходима для соответствия последнему принципу *SOLID,* а самое главное упрощение расширяемости сцены [14];

– *SignInProtocols* хранит в себе протоколы, под которые подписаны основные сущности, это необходимо для абстрактного взаимодействия с ними, также для подмены сущности на тестовую сущность при тестировании, чтобы не было жёстких зависимостей;

– *SignInDataLocalizable* является файлом, который хранит в себе тесктовые данные, которыми заполняется интерфейс, это необходимо для локализации программного средства.

На других сценах могут присутствовать папки *Network* и *Views:*

– *Network* хранит в себе файлы *Endpoint,* в котором собираются данные необходимые для запроса и *DTO,* в котором будут хранится данные, полученные из сетевого запроса;

– *Views* хранит в себе элементы представления, которые были вынесены из *ViewController*, чтобы уменьшить объём сущности, но не подходящие для переиспользования на других сценах.

На рисунке 2.5 приведен фрагмент диаграммы классов, которая позволяет продемонстрировать соответствие программного средства архитектуре.

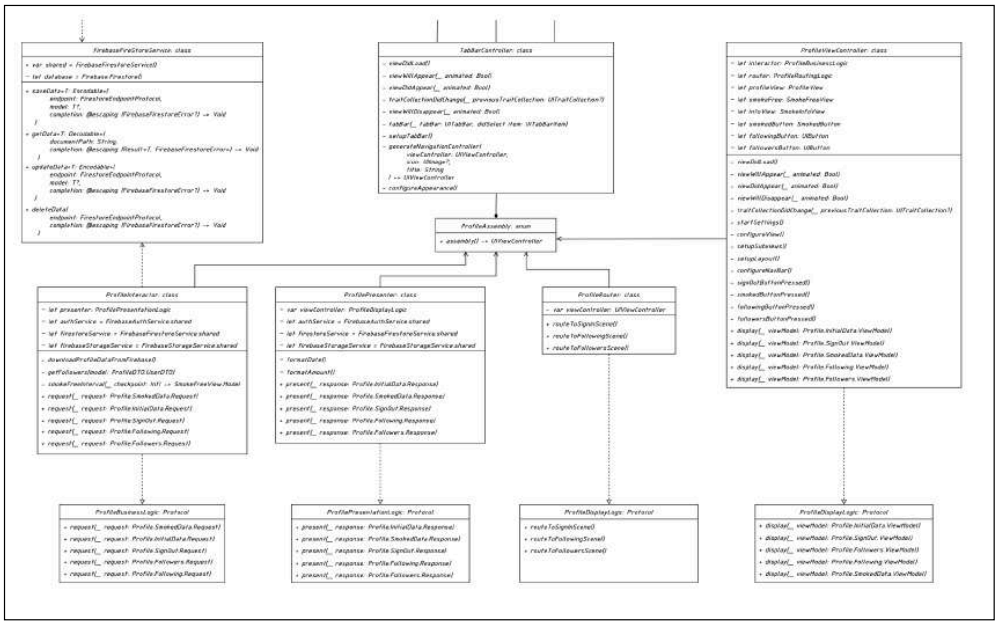

Рисунок 2.5 – Фрагмент диаграммы классов

После проектирования архитектуры программного средства предстоит описать состояния программного средства. В жизненном цикле разрабатываемого приложения существуют активный и фоновой режимы. Все основные задачи и процессы происходят в активном режиме, который содержит внутри основные состояние такие как: отображение пользовательского интерфейса, обработка бизнес процессов, обработка пользовательских сообщений. Важным аспектом является, что, за исключением определённых случаев, для пе-

рехода из одного состояния в другое программному средству необходимо пользовательское вмешательство.

Для понимания структуры программного средства, а также моделирования динамических аспектов системы нужно составить диаграмму состояний, которая покажет состояния, в которых может быть программное средство. Согласно ей, в самом начале пользователь попадает на экран авторизации. На нем он может авторизоваться или зарегистрироваться. После успешной авторизации, пользователь попадает на главный экран приложения, что дает ему возможность увидеть ленту со статьями о вреде курения. При нажатии на статью, пользователь переходит на экран детализации статьи.

От других диаграмм диаграмма состояний отличается тем, что описывает процесс изменения состояний только одного экземпляра определенного класса – одного объекта, причем объекта реактивного, то есть объекта, поведение которого характеризуется его реакцией на внешние события [15].

На рисунке 2.5 изображена *UML* диаграмма состояний.

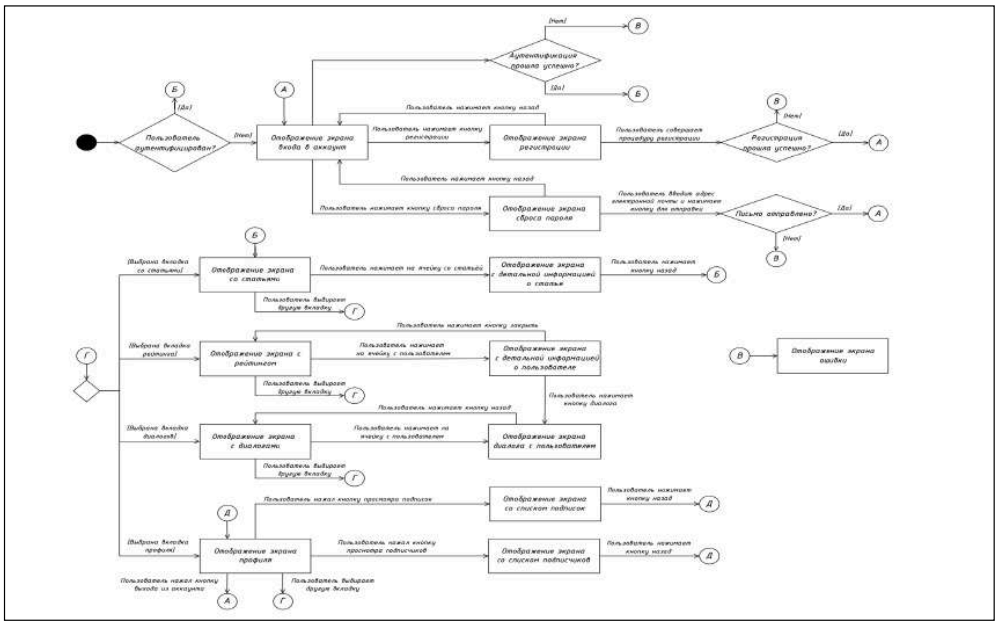

Рисунок 2.5 – *UML* диаграмма состояний

Также важной частью описания архитектуры приложения является описание пакетов, подключенных к нему. В данном случае использовался нативный инструмент компании *Apple – Swift Package Manager*.

Разделение приложений на пакеты произведено в связи с двумя последними принципами *SOLID* – принцип разделения интерфейсов и принцип инверсии зависимостей.

Принцип разделения интерфейсов подразумевает то, что не нужно добавлять в сущность функциональность, которая ему не нужна. Сущность

должна производить только те операции, которые обеспечивают правильность выполнение его методов.

Принцип инверсии зависимостей подразумевает то, что модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Модули или классы верхнего уровня равно классы, которые выполняют операцию при помощи инструмента. Модули или классы нижнего уровня это – инструменты, которые нужны для выполнения операций. Абстракции – представляют интерфейс, соединяющий два класса. Детали – это специфические характеристики работы инструмента. Согласно данному принципу, класс не должен соединяться с инструментом, который применяет для выполнения операции. Вместо этого он должен быть соединён с интерфейсом, который поможет установить связь между инструментом и классом. Кроме того, принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента. Напротив, это инструмент должен подходить под требования интерфейса.

Структура пакетов изображена на рисунке 2.6.

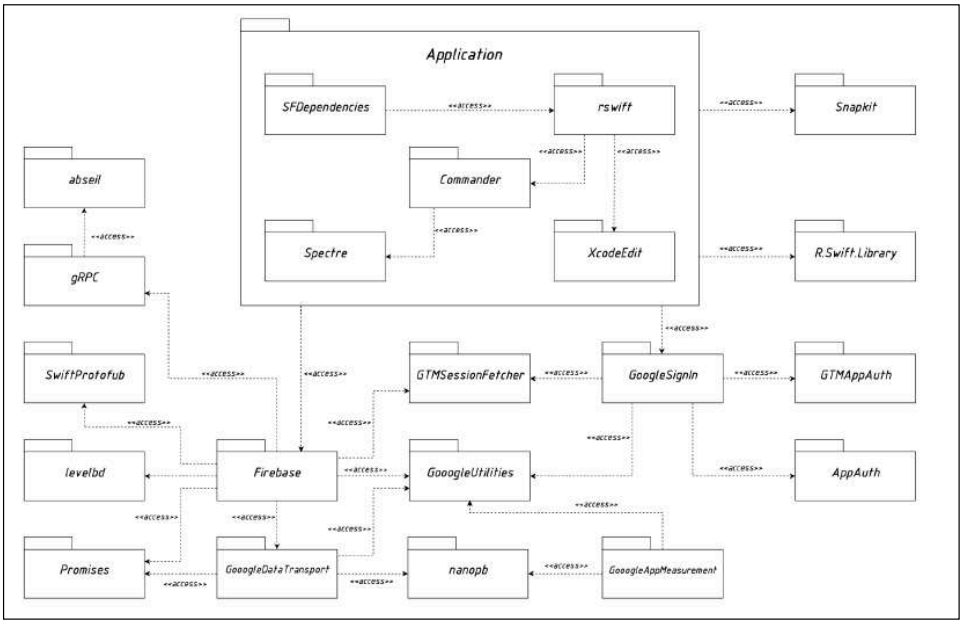

Рисунок 2.6 – Диаграмма пакетов приложения

Использование системы пакетов в архитектуре дает возможность грамотно разделить все согласно вышеперечисленным принципам [16].

Таким образом, была выбрана и спроектирована архитектура, которая отлично масштабируется и тестируется, а также отлично подходит для долгоиграющих проектов с большим количеством бизнес логики. А также были описаны основные состояния разрабатываемого программного средства.

2.2 Формализация предметной области программного средства

Предметная область – это целенаправленная первичная трансформация картины внешнего мира в некоторую умозрительную картину, определенная часть которой фиксируется в информационной системе в качестве алгоритмической модели фрагмента действительности. Иными словами – это часть реального мира, подлежащая изучению с целью организации управления и, в конечном счете, автоматизации.

В качестве предметной области данного дипломного проекта выбрана помощь страдающим от табачной зависимости, в связи с высокой актуальностью данной темы. Такие сервисы пользуются довольно высоким спросом, ведь в большей или меньшей степени каждый страдающий от табачной зависимости ищет способ бросить курить.

Одна из первых задач, с решением которых сталкивается разработчик программной системы – это изучение, осмысление и анализ предметной области. Дело в том, что предметная область сильно влияет на все аспекты проекта: требования к системе, взаимодействие с пользователем, модель хранения данных, реализацию и т.д. Анализ предметной области, позволяет выделить ее сущности, определить первоначальные требования к функциональности и определить границы проекта.

Результатом проведения анализа предметной области должен стать перечень системных требований, спецификаций, информационных потоков и их описание. Очень часто для этого применяются стандартные способы описания предметной области с использованием UML (унифицированный язык моделирования).

Язык моделирования – это набор графических нотаций, которые используются для описания моделей в процессе проектирования. Нотация представляет собой совокупность графических объектов, используемых в модели, и является синтаксисом языка моделирования. Язык моделирования, с одной стороны, должен делать решения проектировщиков понятными пользователю, с другой – предоставлять проектировщикам средства достаточно формализованного и однозначного определения проектных решений, подлежащих реализации в виде программных комплексов, образующих целостную систему. Комплексность подхода и использование единой нотации очень важно не только на этапе моделирования предметной области, но и на последующих этапах разработки программной системы [17].

При разработке модели предметной области определяют некоторые границы, в пределах которых можно развивать логическую модель данных, т.е. моделировать объекты, не выходящие за пределы рассматриваемой предметной области. При этом все второстепенные детали опускаются, чтобы чрезмерно не усложнять процесс анализа и исследования полученной модели.

Под моделью предметной области понимается некоторая система, имитирующая структуру или функционирование исследуемой предметной области. Модель должна отражать все аспекты функционирования программного

обеспечения, и необходима на всех этапах жизненного цикла программного обеспечения.

Модель предметной области – это формализованные знания о предметной области, выраженные при помощи каких-либо средств. В качестве таких средств могут выступать текстовые описания предметной области, но более информативными и полезными при разработке программного обеспечения и баз данных являются описания предметной области, выполненные при помощи специализированных графических нотаций. От того, насколько правильно смоделирована предметная область, зависит успех дальнейшей разработки системы.

В качестве графического представления для выделения основных процессов системы будет построена диаграмма деятельности.

Диаграмма деятельности – это технология, позволяющая описывать логику процедур, бизнес-процессы и потоки работ. Это поведенческая диаграмма, которая может быть использована для изучения бизнес-процессов с целью определения их потока и требований. Описанная диаграмма представлена на рисунке 2.7.

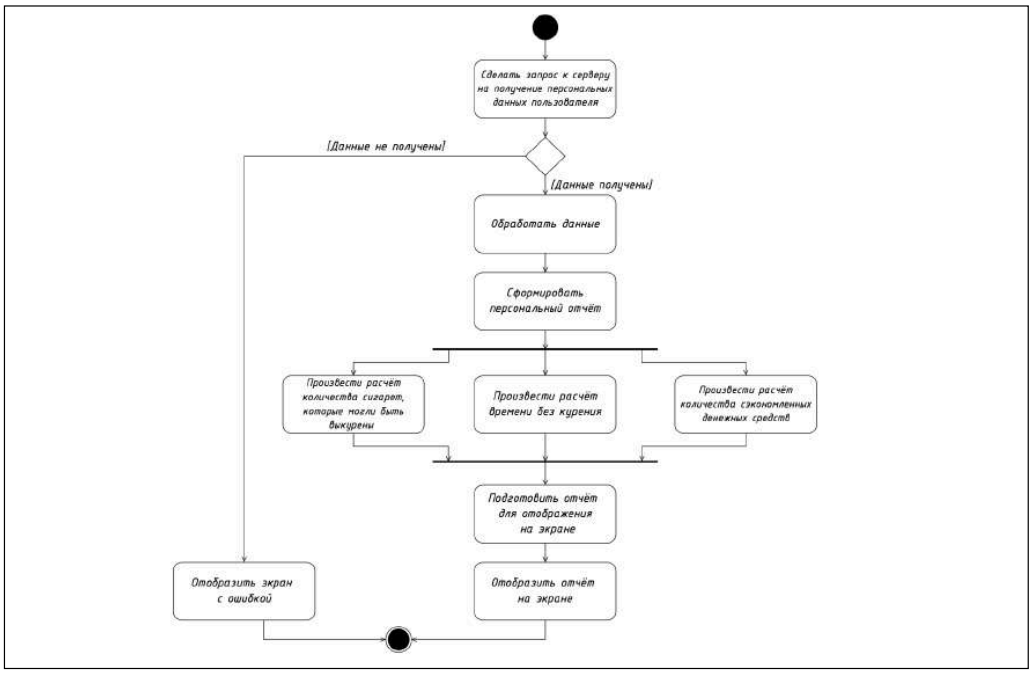

Рисунок 2.7 – *UML* Диаграмма деятельности

Таким образом была создана модель предметной области, описанная при помощи унифицированного языка моделирования UML, а также описана формализация предметной области программного средства.