МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет имени К.И.Сатпаева

Институт автоматики и информационных технологий

Кафедра "Программная инженерия"

Бейсембин Темирлан Даулетович

Разработка системы бюджетных расходов, финансовый помощник

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту

По специальность 5В070400 – Вычислительная техника и программное обеспечение

Алматы 2022

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет имени К.И.Сатпаева

Институт автоматики и информационных технологий

Кафедра "Программная инженерия"

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
канд. техн. наук, доцент, профессор
_____Молдагулова А.Н.
"_26_" _____05_____ 2022 г.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту

На тему: Разработка системы бюджетных расходов, финансовый помощник

по специальность 5В070400 – Вычислительная техника и программное обеспечение

Выполнил Бейсембин Т.Д.

Рецензент
доктор PhD, доцент
_____Утегенова А.У.
"_20_" _____05_____ 2022 г.

Научный руководитель
Ассоциированый профессор
_____Жекамбаева М.Н.
"_20_" _____05_____2022 г.

Алматы 2022

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет имени К.И.Сатпаева

Институт автоматики и информационных технологий

Кафедра "Программная инженерия"

5В070400 – Вычислительная техника и программное обеспечение

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
канд. техн. наук, доцент, профессор
_____Молдагулова А.Н.
"26" 05 2022 г.

ЗАДАНИЕ
на выполнение дипломного проекта

Обучающемуся Бейсембин Темирлан Даулетович

Тема: Разработка системы бюджетных расходов, финансовый помощник

Утверждена приказом проректора по академической работе: № 489-17/0 от "24" 12 2021 г.

Срок сдачи законченного проекта: "20" 05 2022 г.

Исходные данные к дипломному проекту: сбор теоретического материала, данные анализа по данной теме

Перечень подлежащих разработке в дипломном проекте вопросов:

а) Анализ предметной области, исследования рынка, актуальность проекта, составление и постановка задач.

б) Выбор инструментов для разработки системы.

в) Проектирование системы.

г) Разработка пользовательского интерфеса, верстка и разработка приложения.

д) Тестирование приложения.

Перечень графического материала (с точным указанием обязательных чертежей): представлены 16 слайдов презентации.

Рекомендуемая основная литература: из 24 наименований.

ГРАФИК

подготовки дипломного проекта

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю и консультантам	Примечание
1.Анализ предметной области и анализ рынка	25.01.2022	выполнено
2.Выбор технологий для ведения разработки системы.	05.02.2022	выполнено
3.Проектирование архитектуры системы и написание диаграмм.	20.02.2022	выполнено
4.Создание прототита и макета дизайна.	01.03.2022	выполнено
5.Верстка UI приложения на языке разметки XML.	15.03.2022	выполнено
6.Разработка внутреннего функционала приложения.	25.03.2022	выполнено
7.Тестирование проекта	28.03.2022	выполнено

Подписи

консультантов и нормоконтролера на законченный дипломный проект с указанием относящихся к ним разделов проекта

Наименования разделов	Консультанты, Ф.И.О (уч. степень, звание)	Дата подписания	Подпись
Нормоконтроллер	Жекамбаева М.Н. Ассоциированный профессор,	20.05.22	
Программное обеспечение	Марғұлан Қ. Магистр техн.наук., лектор.	20.05.22	

Научный руководитель _____ Жекамбаева М.Н.

Задание принял к исполнению обучающийся _____ Бейсембин Т.Д.

Дата «17» 11 2021 г.

АННОТАЦИЯ

Данный дипломный проект посвящен созданию приложения для контроля бюджета. Приложение было разработано для предоставления конечному пользователю легко контролировать свои траты. Дипломный проект рассматривает техническую и бизнес части разработки android-приложений.

Приложение ставит перед собой цель помочь пользователям следить за своими тратами, чтобы в дальнейшем анализировать какие траты являются лишними и на основе этого правильно планировать свой бюджет.

Дипломный проект состоит из четырех разделов, введения и заключения.

Первый раздел включает в себя цели, определения и общее описания.

Второй раздел описывает технологии, которыми мы будем пользоваться в разработке продукта.

Третий раздел посвящен архитектуре, базе данных и бизнес части приложения.

Четвертый раздел показывает нам саму реализацию основных частей приложения.

В заключении подводится итог реализации проекта.

Дальнейшее развитие проекта предполагает расширение функционала, настройка контроля версий и распространения приложения.

АҢДАТПА

Бұл дипломдық жоба бюджеттік бақылауға өтінімді құруға арналған. Қолданба соңғы пайдаланушыға шығындарын оңай басқаруға мүмкіндік беру үшін жасалған. Бітіру жобасы android қосымшаларын әзірлеудің техникалық және іскерлік бөліктерін қарастырады.

Қолданба пайдаланушыларға қандай шығындардың артық екенін әрі қарай талдау және осының негізінде өз бюджетін дұрыс жоспарлау үшін шығындарын қадағалауға көмектесуге бағытталған.

Дипломдық жоба төрт бөлімнен, кіріспеден және қорытындыдан тұрады.

Бірінші бөлім мақсаттарды, анықтамаларды және жалпы сипаттаманы қамтиды.

Екінші бөлімде біз өнімді әзірлеуде қолданатын технологиялар сипатталған.

Үшінші бөлім сәулетке, дерекқорға және қолданбаның бизнес бөлігіне арналған.

Төртінші бөлім бізге қосымшаның негізгі бөліктерінің нақты орындалуын көрсетеді.

Қорытындылай келе, жобаны іске асыру нәтижелері қорытындыланады.

Жобаны одан әрі дамыту функционалдылықты кеңейтуді, нұсқаны басқаруды орнатуды және қолданбаны таратуды қамтиды.

ANNOTATION

This graduation project is devoted to the creation of an application for budget control. The app has been designed to provide the end user with easy control over their spending. The graduation project considers the technical and business parts of the development of android applications.

The application aims to help users keep track of their spending to further analyze what expenses are superfluous and, based on this, plan their budget correctly.

The diploma project consists of four sections, introduction, and conclusion.

The first section includes objectives, definitions, and a general description.

The second section describes the technologies that we will use in product development.

The third section is devoted to the architecture, database, and business part of the application.

The fourth section shows us the actual implementation of the main parts of the application.

In conclusion, the results of the project implementation are summarized.

Further development of the project involves expanding the functionality, setting up version control and distribution of the application.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В условиях современных реалий вопрос экономии денег проявляется очень сильно. Используя банковские карты и онлайн-банкинг, многие попросту не следят за тратами. В связи с этим актуальным является использование приложения для контроля бюджета.

Проверив множество приложений для контроля бюджета, я пришел к мнению, что они довольно громоздкие и сложные. Различный функционал для работы с картами и депозитами, отсутствие ежемесячных трат (различные сервисы, кредиты), часто платные полные версии приложений, все это отталкивает от желания его использовать.

Данная работа направлена на создание приложения, с которым конечному пользователю будет проще управлять своими тратами, ограничить спонтанные покупки, анализировать свои траты по категориям ежемесячно. Возможно, пользователь хочет не экономить, а просто анализировать свои траты, это приложение ему также поможет.

1 Исследовательский раздел

1.1 Цель разработки

Цель данного проекта улучшить опыт конечного пользователя в управлении бюджетом, анализа покупок и трат. Управление бюджетом реализовано посредством добавления категорий и трат в локальную базу данных. В дальнейшей перспективе будет добавлена синхронизация с удаленной базой данных посредством регистрации.

Каждый элемент экрана и данные должны отслеживать изменения в базе данных и автоматически обновляться.

В техническом плане достоинстве данного приложения состоит в быстроте работы за счет хранения данных в локальной базе и упрощенном интерфейсе, но при этом упрощенный интерфейс никак не ограничит конечного пользователя, за счет проведенной UX/UI аналитики.

В качестве платформы выбраны устройства под управлением ОС Android, так как это самые распространенные устройства, в дальнейшем также будет реализовано приложение для iOS и Web.

1.2 Определения, термины и сокращения

Таблица 1.1 - Сокращения, термины и их определения

Сокращение или термин	Определение
ОС	Операционная система
БД	База данных
СУБД	Система управления базой данных
ПО	Программное обеспечение
UI	Пользовательский интерфейс
Android	Мобильная операционная система
Android SDK	Средство для разработки мобильных приложений под операционную систему Android
Java Runtime Environment (JRE)	Виртуальная машина, необходимая для исполнения Java-приложений
Java Virtual Machine (JVM)	Виртуальная машина Java, основная часть JRE исполняет байт-код Java.

Продолжение таблицы 1

Kotlin	Объектно-ориентированный язык программирования, работающий поверх Java Virtual Machine
Dependency Injection (DI)	Процесс предоставления внешней зависимости программному компоненту
Clean Architecture	Разработка приложений с использованием данной архитектуры предполагает разделения всех основных компонентов на слои
SQLite	СУБД с отрытым исходным кодом
Git	Система для контроля версий проекта

1.3 Анализ конкурентов

На данный момент существует множество прямых конкурентов данного приложения, но большинство сервисов по подсчету бюджета перегружены различным функционалом. Сюда входят наличие множества счетов и карт в приложении, в которых можно запутаться, перегруженный интерфейс в котором сложно разобраться. Один из таких примеров – приложение 1Money.

Как мы видим на рисунках 1 и 2 интерфейс перегружен и многое не понятно для конечного пользователя, так же заметим, что множество функций заблокировано, пока пользователь не приобретет премиум аккаунт.

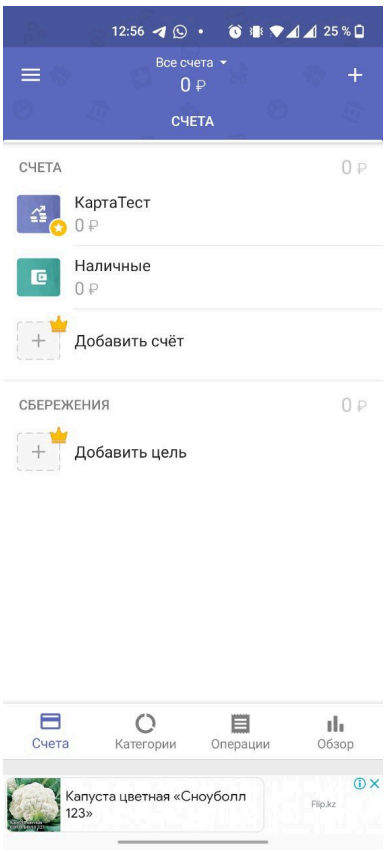

Рисунок 1.1 - Экран «Счетов» в приложении 1Money

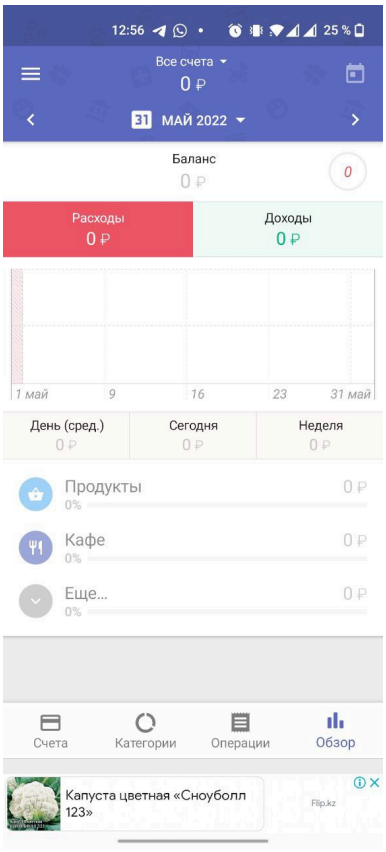

Рисунок 1.2 - Экран обзора графиков в приложении 1Money

По этим причинам большинство пользователей не используют данные приложения полностью, лишь добавляя траты и все на этом. Лично я скачав и использовав несколько приложений для контроля бюджета часто путался и постоянно прибегал к секции FAQ, чтобы найти ответы на вопросы по использованию приложений.

В связи с этим было решено написать новое приложение, которое будет проще и легче, но не терять основного функционала. Произведен анализ рынка и выделены основные функции, которые необходимы конечному пользователю, проведена UX/UI аналитика, чтобы пользователю было комфортно пользоваться приложением, и оно не вызывало желания закрыть его.

Часто в приложениях нет функций по периодическим постоянным тратам (подписки на сервисы, кредиты, рассрочки, квартира и коммунальные услуги), а также напоминания об их оплате. Отсутствие функционала по отслеживанию долгов.

Проанализировав рынок, я выделил основные задачи, которые необходимо реализовать:

– функционал добавления трат;
– функционал добавления категорий;
– функционал для отслеживания долгов (мне и кому я должен) ;
– функционал для напоминания о периодических тратах.

В дальнейшей перспективе в приложение будут добавлен новый функционал:

– функционал для семейного бюджета (общий бюджет) ;
– общие и личные заметки и напоминания о покупках;
– графики трат.

1.4 Предметная область

При выполнении проекта мне помогли предметные области, которые связаны с созданием приложением под ОС Android, клиент-серверной архитектурой, объектно-ориентированным программированием и управлением проектов.

1.4.1 Android

Предмет по изучению разработки Android приложений предполагает создание минимальных проектов для изучения основных частей приложения:

– верстка с помощью встроенного функционала Android Studio и языка разметки XML;
– многопоточность с помощью корутин;

– работа с API с использованием библиотек Retrofit+OkHTTP;
– архитектура приложения;
– хранение данных в локальной базе данных Room.

1.4.2 Software Engineering

Данная дисциплина предоставляет нам основные знания для проектирования архитектуры, которая будет правильно связана с бизнес-логикой проекта. Чаще всего проекты пишутся на заказ для бизнеса и для правильной работы необходимо правильно построить архитектуру. Архитектура должна быть масштабируемой, простой и не перегруженной, а также удобной для понимания любого разработчика, который придет на проект. Основная часть данной дисциплины – это различные диаграммы, которые отображают структуру проекта, действия пользователей, варианты использований и т. д.

1.4.3 Объектно-ориентированное программирование

Современные языки программирования используют различные методологии программирования, например, объектно-ориентированная, функциональная, предметно-ориентированная. Многие языки не ограничиваются одной методологией, а включают в себя несколько, так как во время разработки проекта разные способы решения задачи могут быть лучше, чем другие.

Язык Kotlin, на котором будет проходить разработка приложения, включает в себя объектно-ориентированную, функциональную, предметно-ориентированную методологии, которые работают в симбиозе.

Объектно-ориентированные языки программирования представляют программы в виде совокупности объектов, каждый из которых является экземпляром класса. Данная методология включает в себя 4 основных принципа:

– абстракция – отделение концепции от экземпляра, то есть использование только тех описаний объекта, которые с точностью представляют его в проекте;

– наследование – способность объекта основываться на другом объекте или классе, перенимая его открытые характеристики. Данный принцип предоставляет возможность переиспользовать код. Так как по принципу DRY (Don’t repeat yourself) необходимо избегать повтора кода, чтобы не «загрязнять» проект;

– инкапсуляция – ограничение доступа к объектам внутри других объектов;

– полиморфизм – способность передавать различные типы в аргументы функции.

На основе выше указанных принципов был создан свод основных правил SOLID, по которому надо следовать, чтобы проект было проще масштабировать и поддерживать.

– S – Single Responsibility (Принцип единственной ответственности). Каждый класс должен решать только одну задачу. Если класс отвечает за несколько задач сразу, то огромна вероятность появления ошибок.

– O – Open-Closed (Принцип открытости-закрытости). Классы должны быть открыты для расширения, но закрыты для модификации. По данному правилу лучше наследоваться от класса, который мы хотим расширить, чем изменять существующий, так как это может задеть другие части системы.

– L – Liskov Substitution (Принцип подстановки Барбары Лисков). Если класс A наследуется от B, то любые объекты типа B, присутствующие в системе, могут заменяться объектами типа A без негативных последствий для функциональности программы. В случаях, когда класс-потомок не предоставляет тот же функционал, что и родительский, огромна вероятность появления ошибок. Если у вас имеется класс и вы создаете на его базе другой класс, исходный класс становится родителем, а новый – его потомком. Класс-потомок должен производить такие же операции, как и класс-родитель.

– I – Interface Segregation (Принцип разделения интерфейсов). Не следует предоставлять клиенту функционал, который ему не нужен. Каждый интерфейс (доступ к методам) должен предоставлять клиенту только тот функционал, что ему необходим, иначе высока вероятность появления ошибок, а также это тратит ресурсы.

– D – Dependency Inversion (Принцип инверсии зависимостей). Модули верхнего уровня (классы, которые используют инструменты) не должны зависеть от модулей нижнего уровня (инструменты). И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Все должно соединяться через интерфейсы, класс не должен использовать инструмент напрямую. Принцип гласит, что ни интерфейс, ни класс, не обязаны вникать в специфику работы инструмента.

1.4.4 Управление проектами

Управление проектами неотъемлемая часть разработки ПО. Оно заключается в контроле, оценке и поддержке проекта ради достижения конкретно поставленных целей. Данная часть проекта включает в себя составления плана и сроков разработки, работы с бюджетом.

2 Технологический раздел

2.1 Android и Android SDK

Android это ОС от компании Google для смартфонов, планшетов и т.д. ОС основана на ядре Linux с собственной реализацией виртуальной машины Java компании Google.

Для создания приложений для ОС Android используется Android SDK и языка Kotlin. Android SDK написан в основном на языке Java и предоставляет все возможности для создания приложения:

– тестирование;

– отладка исходных кодов;

– оценивание работы приложения в режиме совместимости с разными версиями ОС Android;

– набор библиотек;

– телефонный эмулятор;

– наблюдать результат в реальном времени;

Android SDK предоставляет набор компонентов для реализации приложения. Основными из них являются:

– Activity – это отдельный экран со своим UI, он определяет и реагирует на действия на экран. Имеет свой жизненный цикл;

– Services – некая задача, которую надо выполнить. Сервис делится на background (работают на фоне и не говорят о своем присутствии, например синхронизация) и foreground (уведомляют конечного пользователя о своей работе, например загрузка файлов);

– Broadcast Receivers – это компонент, который доставляет данные из вне потока действий пользователя. Например, будильник, вдеь его приложение не всегда включено, оно работает в фоне, в ожидании своего времени, о котором его уведомит broadcast receiver. Также сообщает о системных уведомлениях, таких как уровень заряда батареи, был ли сделан снимок и т.д.;

– Content Providers – предоставляет данные из одного приложения другим по их запросу. Например, список контактов на устройстве, мы можем их получить из другого приложения, путем запроса к базе данных приложения «Контакты»;

– Дополнительные компоненты: Fragment, View, Layout, Intent, Resources, Manifest.

Android SDK сильно упрощает разработку приложения так как, будучи фреймворком, предоставляет готовые функции и callback-функции. Упрощает работу с жизненным циклом приложения, ведь у каждого элемента приложения есть свой жизненный цикл, связанный с жизненным циклом его родительских объектов и за каждым надо следить, чем фреймворк и занимается. Давайте разберем пример жизненного цикла на основе Activity:

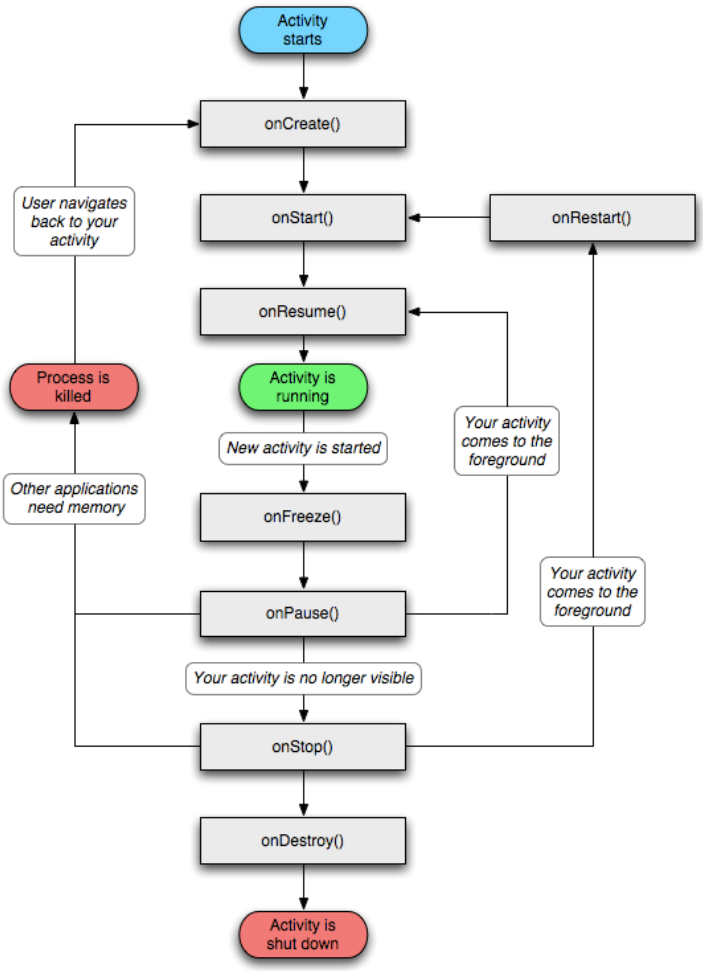

Рисунок 2.1 - Жизненный цикл Activity

Как мы видим на рисунке 1, жизненный цикл Activity начинает с onCreate, который проводит рендер экрана. Далее запуск Activity. При сворачивании приложения Activity переходит в состояние onPause и onStop, после возвращения к экрану, вызывается onRestart и onResume. Состояние onDestroy вызывается, когда экран был полностью закрыт, данное состояние чистит память.

2.2 Интегрированная среда разработки

Интегрированная среда разработки – это набор программных средств для разработки программного обеспечения. В качестве интегрированной среды разработки для создания приложений под ОС Android используется Android Studio.

Android Studio, основанная на ПО IntelliJ IDEA от компании JetBrains – официальное средство разработки Android приложений.

Возможности:

– сборка приложений, основанная на Gradle Build Tools – система автоматической сборки на языке Groovy;

– верстка дизайна через встроенный редактор дизайна;

– защита приложений с помощью ProGuard;

– статический анализатор кода, анализ производительности приложения и утечек памяти.

2.3 Room database

2.3.1 Преимущества

Room – библиотека которая предоставляет абстракцию над SQLite, то есть, удобное API для доступа к SQLite

SQLite – встраиваемая СУБД. Слово «встраиваемая» означает что SQLite работает как библиотека, которая становится частью программы, а не отдельный процесс клиент-серверной парадигмы. Таким образом работа с базой происходит путем вызова функций библиотеки. Такой подход сильно уменьшает затрачиваемые ресурсы, время отклика и в принципе упрощает программу. SQLite хранит базу данных в виде файла на устройстве, на котором выполняется сама программа.

Существует множество библиотек для реализации базы данных для хранения информации. Самой популярной из таковых является Room Database. Эту библиотеку Google создала и оптимизировала специально для android. Из преимуществ мы можем выделить скорость доступа к данным и их обработке, а также Room предполагает ручное написание SQL-запросов, что дает нам возможность лучше работать с SQLite.

2.3.2 Функции Room Database

Как уже говорилось выше, данная библиотека предполагает написание SQL-запросов. Все таблицы хранятся в виде объектов, что сильно облегчает работу с данными. Функции для доступа к SQLite реализованы с помощью рефлексии, то есть программа во время работы может отслеживать и модифицировать собственную структуру (создавать и видоизменять временные классы, с помощью которых мы и получаем доступ к данным), поэтому чтобы обращаться к функциям мы можем воспользоваться аннотации – форма синтаксических метаданных, которая дает необходимую информацию компилятору, генерирует код во временных файлах и может использоваться для получения данных с помощью рефлексии. Ниже мы рассмотрим основные аннотации библиотеки Room.

@Entity – аннотация, которая ставится над моделью данных и говорит компилятору, что из данной модели будет сгенерирован класс, который опишет таблицу для базы данных, далее в модели помеченной @Entuty мы указываем @PrimaryKey и @ColumntInfo (описание столбца таблицы).

@Query(“SQL-query”) – аннотация, которая ставится над методом, которая говорит, что данный метод будет реализован с помощью SQL-запроса.

@Transaction – аннотация, которая ставится над методом и говорит ему, что данный метод будет выполнять несколько последующих действий с базой данных.

@Insert – аннотация, которая ставится над методом и говорит ему, что данный метод будет использован для вставки данных в таблицу.

@Delete – аннотация, которая ставится над методом и говорит ему, что данный метод будет использован для удаления данных из таблицы.

2.4 Koin

2.4.1 Dependency Injection

Inversion of Control (Инверсия управления) – принцип ООП, используемый для уменьшения зависимости в системе. Также является архитектурным решением, которое упрощает расширение и масштабирование системы, при котором система управляется фреймворком.

Фреймворк – каркас для ПО, которое упрощает реализацию и хранит в себе множество готовых компонентов для разработки ПО. Например, Android SDK – каркас для разработки мобильного приложения, который является доступом к самой системе Android и предоставляет множество готовых компонентов для работы с ОС.

В обычной программе, без фреймворка, программист сам решает как должна работать программа, в какой последовательности вызывать функции и процедуры. Но, если использовать фреймворк, то программист расставляет необходимый код точках выполнения (например, callback-функции), затем запускает «main-функцию» фреймворка, которая обеспечит все вызовы, когда это будет необходимо. Сделаем вывод, что разработчик теряет контроль над выполнением кода – это и называется инверсией управления. Получается фреймворк управляет кодом разработчика, а не наоборот.

Одной из способов применения инверсии управления является внедрение зависимостей.

Dependency Injection (Внедрение зависимостей) – процесс предоставления зависимости другим компонентам проекта, предоставить объект другому объекту, без ручного его создания и слежки за его жизнедеятельностью. В соответствии с принципом единственной ответственности объект отдает заботу о построении требуемых ему зависимостей внешнему общему механизму.

При использовании паттерна DI объект не пытается выяснять свои зависимости, а предоставляет для этого сеттеры или принимает в свой конструктор, посредством чего внедряются зависимости.

2.4.2 Преимущества

Koin – фреймворк для внедрения зависимостей в Android проект.

Koin позволяет нам ограничивать область живучести зависимостей относящихся к конкретному компоненту, то есть фреймворк контролирует создание и жизнедеятельность объекта (при необходимости сам создает и уничтожает объект).

Популярным фреймворком для реализации DI в Android-приложениях является Dagger, но он требует глубокого изучения. Одной из лучших альтернатив ему – Koin, библиотека написанная на чистом Kotlin.

Из преимуществ выделим облегченный способ настройки DI в проекты (помогает этому Kotlin DSL), а также повышенная скорость сборки приложения (за счет понижения производительности работы самого приложения). Dagger в этом случае будет лучше, так как он не ограничивает производительность приложения, а лишь увеличивает время сборки. Также проблему сложности изучения Dagger фреймворка решила наработка компании Google в виде библиотеки Hilt, что является абстракцией над Dagger.

2.4.3 Функции Koin

Koin ограничивает область живучести зависимостей, то есть предоставляет временную область для существования объекта. Существует 3 вида областей:

– Single (одиночный объект) – создается объект, который сохраняется в течение периода существования контейнера (аналогично синглтону);

– Factory (фабрика объектов) – каждый раз создается новый объект, без сохранения в контейнере (совместное использование объектом невозможно);

– viewModel (фабрика объектов) – как и при области factory, постоянно создается новый объект, но данная область работает только с классами, родительским классом которых является ViewModel.

После определения всех областей и привязанных к ним объектам, мы можем передавать их друг другу в качестве параметров конструктора и не переживать о создании объекта или удалении его для очистки памяти.

2.5 Retrofit + OkHttp

OkHttp – библиотека, которая позволяет отправлять и получать веб-запросы. С помощью данной библиотеки мы можем создать в нашем приложении веб-клиент для работы с сетью.

Retrofit – библиотека, которая упрощает работу с REST API сайта, является надстройкой (абстракцией) над OkHttp. Библиотека позволяет нам отправлять GET, POST, PUT, DELETE запросы и может работать в асинхронном режиме.

Для начала работы с сетью мы создадим веб-клиент, который строится на основе OkHttp. Далее нам надо будет описать методы для работы с бэкендом и в дальнейшем работать с предоставленными нам эндпоинтами.

Также данная библиотека предоставляет нам работу с перехватчиками (intereptors), которые могут изменять заголовки запросов. Например, мы можем использовать их для работы с токенами авторизации в заголовке Authorization.

3 Проектная часть

3.1 Архитектура проекта

Приложение было разработано основе такого стека технологий, как Android, Room, Retrofit+OkHttp, Koin, MVVM с применением Clean Architecure

Данные будут храниться в реляционной базе данных SQLite. Через библиотеку Room мы получим возможность получать, редактировать и всячески изменять данные из базы данных. А с помощью Retrofit+OkHttp мы сможем получать данные с удаленного бэкенда.

Проект реализован с помощью паттерна архитектуры MVVM (Model-View-ViewModel) с применением Clean Architecture. Данный шаблон проектирования разделяет проект на три абстракции, каждая из которых занята своим делом.

Model – представляет логику работы с данными.

View – представление, представляет отображение данных на экраны через Activity/Fragment (виды экранов в android приложении).

ViewModel – связь между абстракциями model и view, предоставляет возможность видоизменять полученные из model данные для отображения во view. Данный класс наследуется от ViewModel, что предоставляет следить за жизненной деятельностью Activity/Fragment и не терять данные во время работы приложения, что предоставляет нам надёжность.

Архитектура – это как при строительстве, ведь программы строятся из блоков, описание того, как эти блоки должны быть соединены.

Целью программной архитектуры является минимизация количества человеческих ресурсов, необходимых для создания и поддержки системы.

Clean Architecture – программная архитектура, которая предполагает разделение проекта на слои. Создана Робертом Мартиным (Дядя Боб), человеком, который пишет код уже более 50 лет и за это время создал множество программ. Он сделал вывод, что везде правила архитектуры одни и те же и за 50 лет никак не изменились. На основе данной информации он предложил свое видение архитектуры с разделением всего на независимые слои.

Рисунок 3.1 - Схема построения чистой архитектуры

Данная архитектура предоставляет нам множество преимуществ против монолитной архитектуры, где все компоненты связаны друг с другом. Преимущества:

– Независимость от фреймворка (в нашем случае Android SDK): архитектура системы не зависит от какой-либо библиотеки, что позволяет нам использовать фреймворк в качестве инструмента, что не ограничивает наш проект;

– Тестируемость: бизнес-часть проекта может быть протестирована без UI, БД и веб-сервера;

– Независимость от UI: интерфейс можно легко изменить, не трогая остальную систему;

– Независимость от базы данных: предоставляет нам возможность изменить базу данных на любую другую без больших изменений архитектуры проекта, ведь наша бизнес-часть не зависима от данных;

– Независимость от внешних сервисов: наша бизнес-часть не зависима ни от какой части.

Для работы чистой архитектуры нам в основном необходимо придерживаться одного из правил принципов SOLID, а именно Dependency Inversion Principle – модули внешних уровней не должны импортировать сущности из модулей внутренних уровней и наоборот. Все зависит от

абстракций, который не должны зависеть от деталей и наоборот. А также Inversion of Control – фреймворк управляет кодом программиста, а не программист фреймворком. То есть обычно программист сам определяет как код будет вызывать методы и функции. Но если используется фреймворк, программист разместит свой код в определенных точках выполнения (используя так называемые callback-функции, функции обратного вызова, то есть вызывается самим фреймворком), затем запустить main-функцию, которая сама все выполнит за программиста.

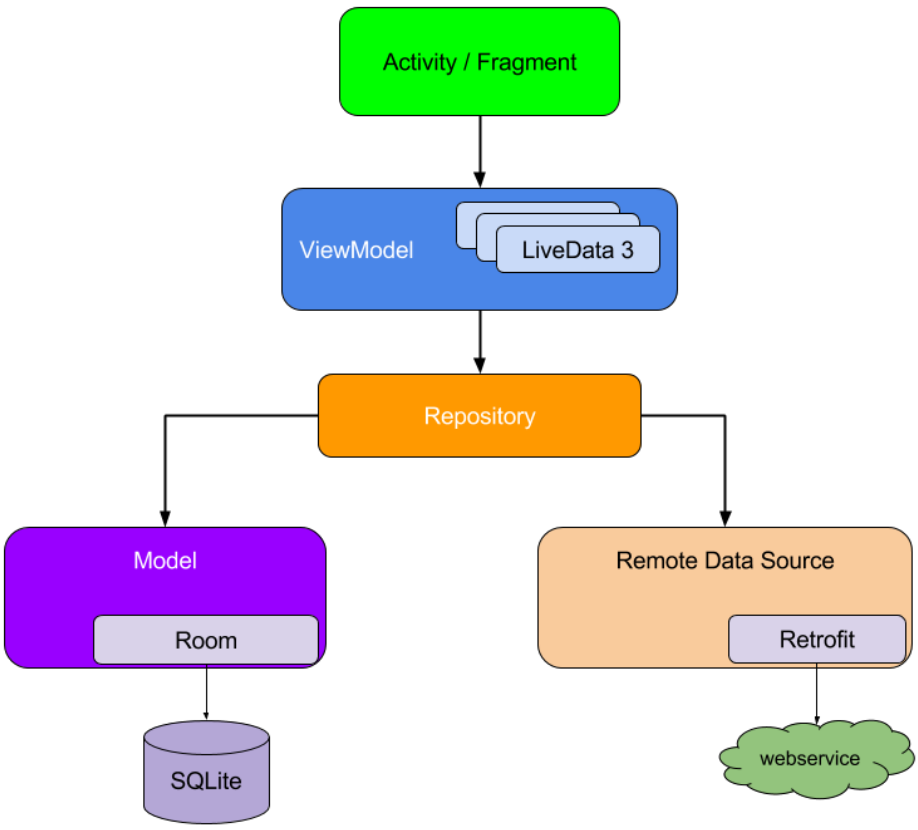

Рисунок 3.2 - Стек технологий для разработки приложения

Рассмотрим архитектуру нашего проекта на шаблоне на рисунке 2.

В android приложениях при использовании чистой архитектуры проект делят на 3 слоя:

– data – сюда относятся Model и Remote Data Source, которые отвечают за получение данных из локального и удаленного хранилищ;

– domain – сюда относится repository, то есть наша бизнес-часть, она обрабатывает полученные данные из data слоя;

– ui – сюда относятся viewModel и Activity/Fragment, где viewModel подготавливает данные для отображения, а Activity/Fragment просто отображают эти данные.

Данные поступают в репозиторий, где обрабатываются и оттуда идут во viewModel через LiveData (встроенный в Android SDK паттерн наблюдателя, который постоянно прослушивает данные). Далее Activity/Fragment прослушивая LiveData из viewModel обновляют Ui, отображая новые данные.

3.2 Описание диаграммы базы данных

Схема «сущность-связь» (ER-диаграмма) – показывает, как «сущности» связаны между собой внутри системы. Чаще всего используются для проектирования баз данных.

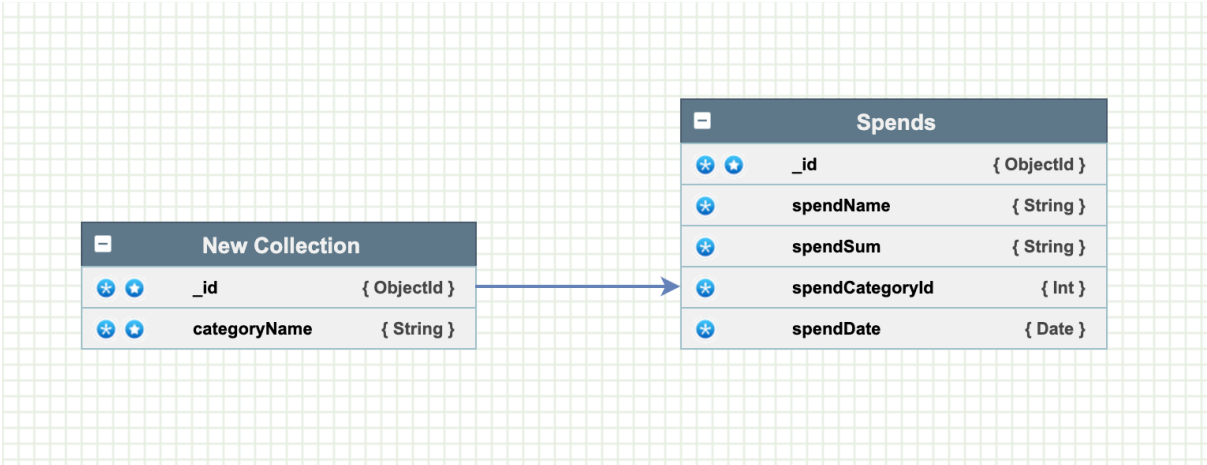

Рисунок 3.3 - ER-диаграмма базы данных

База данных в нашем проекте получается небольшая в связи с небольшим количеством данных для хранения. Связь между таблицами один-ко-многим, то есть на одну категорию может быть множество трат.

3.3 Описание диаграммы вариантов использования

Диаграмма вариантов использования – это диаграмма с множеством логически связанных ролей, исполняющих различные последовательности действий. На диаграмме отображаются Участники (роли) и Прецеденты (use cases). Прецедент не показывает «как» достичь результата, а только «что» можно выполнить.

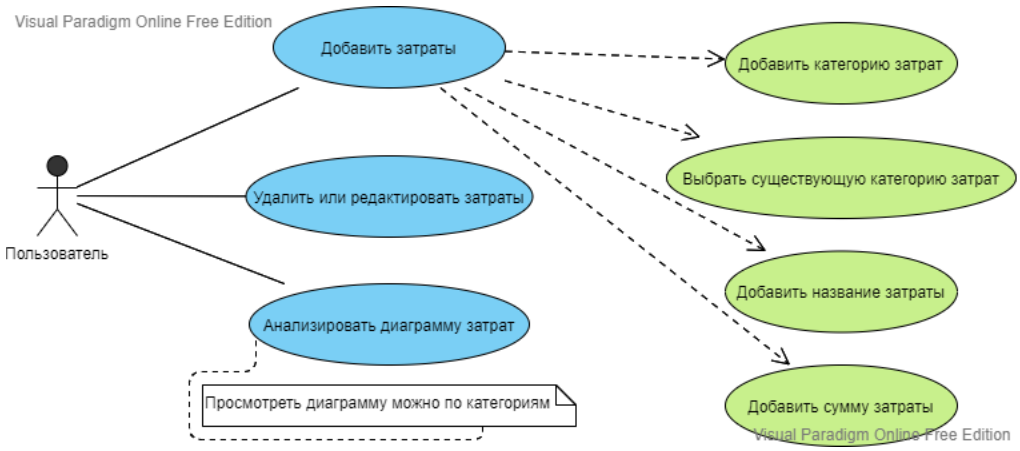

Рисунок 3.4 - Диаграмма вариантов использования

Как мы видим на рисунке 4, у нас только одна роль – пользователь – которая может выполнить несколько действий:

- добавлять данные: создать новую категорию или новую затрату;
- удалять и редактировать существующие затраты и категории;
- анализировать диаграмму затрат по категориям.

3.4 Описание диаграммы деятельности

Диаграмма деятельности – представляет последовательности действий, которые пользователь может произвести.

Данное приложение стартует с главной страницы, на которой отображается список трат с возможностью перейти к списку категорий. Пользователю необходимо добавлять категории и траты, путем нажатия на кнопку добавления. Далее открывается страницы заполнения данных, после чего он возвращается на главную страницу, видя измененный список.

После добавления категорий и затрат у пользователя они отобразятся на главной странице, и он сможет на них нажимать, при нажатии открывается часть экрана, где он сможет удалить или изменить данные, путем ввода новых. Далее пользователь возвращается на главную страницу.

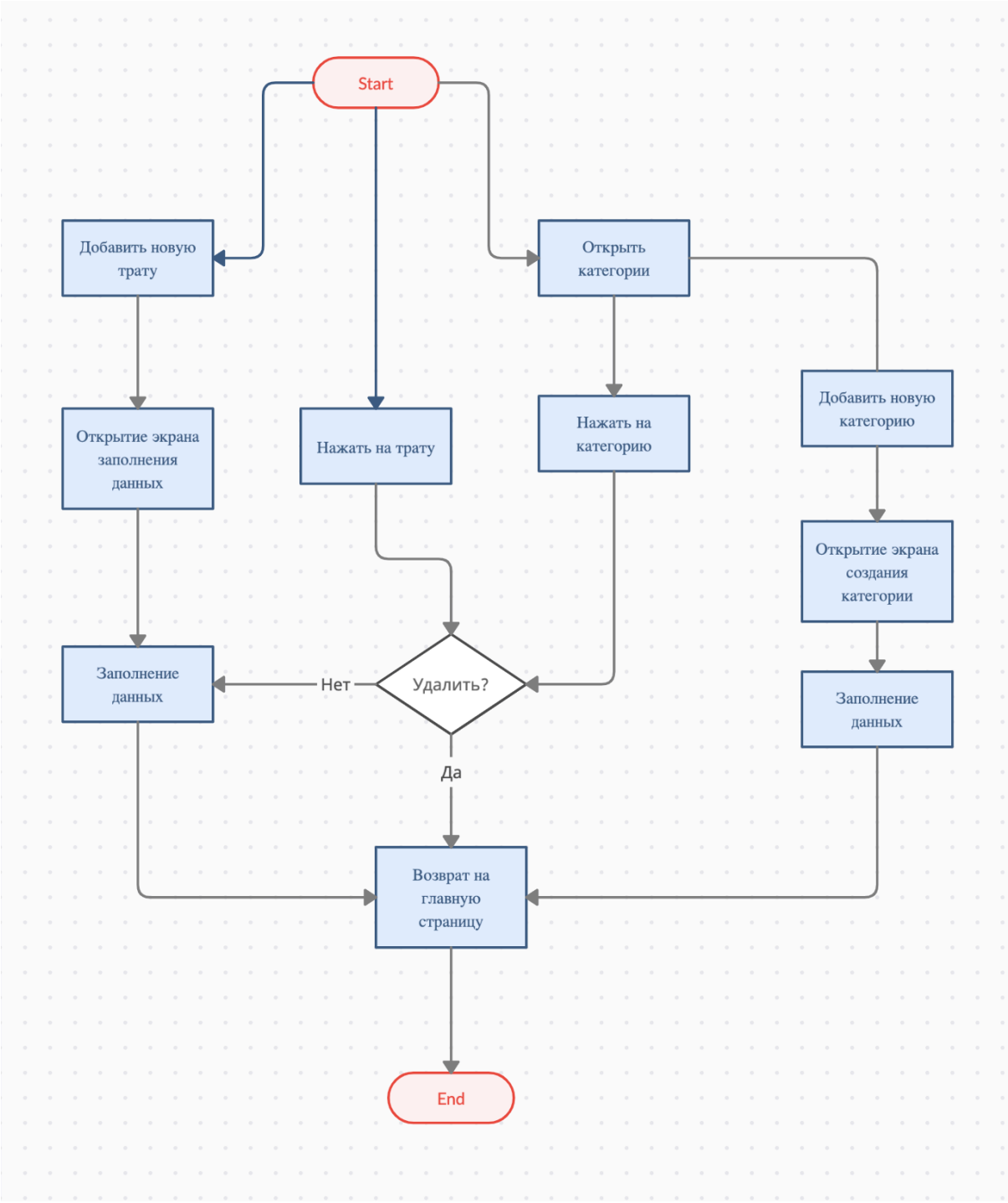

Рисунок 3.5 - Диаграмма деятельности

3.5 Разработка UX/UI дизайна приложения

Работа с приложением для телефона должна быть привычной и интуитивной, понятной любому пользователю. Поэтому в разработке android приложений неотъемлемой частью является верстка дизайна.

Верстка дизайна встроена в интегрированную среду разработки, нам предоставляется инструмент верстки с возможностью создания экрана путем перетаскивания элементов прямо на экран, только это не так удобно, как писать код. Верстка кодом в android происходит на языке разметки XML. XML – расширяемый язык разметки, в котором синтаксис расширяется путем создания новой грамматики со словарем.

Фреймворк Android SDK предоставляет множество способов верстки. Верстка вложенная, получается каждая часть экрана является ребенком родительской части. В качестве родительского контейнера мы можем использовать несколько предложенных макетов:

– ConstraintLayout – макет, где каждое дочернее view связано с соседними через «цепи»;

– NestedScrollLayout – макет, в котором нам предоставляется возможность листать экран вверх и вниз;

– CoordinatorLayout – макет, который предоставляет нам возможность анимировать взаимодействие пользователя с экраном;

– LinearLayout – элементарный макет, в котором все определяется через gravity view, который говорит место расположение view (center, end, start, bottom, top).

В качестве дочерних view мы можем использовать либо сами макеты, а в них вкладывать другие view, либо создавать свои собственные, так называемые, custom view, которые по сути, мы сами и рисуем, либо выбрать и огромного набора готовых:

– TextView – view для вывода текста на экран;

– RecyclerView – view для вывода списка на экран;

– EditText – view для ввода текста;

– Button – кнопка;

– Toolbar – верхняя часть экрана;

– BottomNavBar – нижняя навигационная часть, для смены страниц;

– Множество других.

Для прототипа дизайна использовался онлайн-сервис Figma. Figma – сервис для разработки интерфейсов и прототипирования. Он используется как для создания простых прототипов и интерфейсов, так и для детальной проработки дизайна интерфейсов мобильных приложений, веб-сайтов и иного ПО, которое имеет UI.

Во время продумывания дизайна приложения предполагалось сделать все как можно проще, чтобы конечному пользователю не пришлось ничего искать. Использовалось руководство по дизайну Material Design 3 от компании Google.

В котором расписаны самые удобные для пользователя расположения элементов экрана, а также их размеры и цвета.

Цвета использовались нежные и приятные, чтобы не отпугивать пользователя, и он хотел возвращаться в приложение.

Material Design – стиль дизайна UI для ПО и приложений, разработанный компанией Google. Стиль предполагает широкое применение строгих макетов, анимации и переходов, отступов и эффектов глубины (света и тени). По идее Google, у приложений не должно быть острых углов, все должно быть плавно и неприметно.

С приходом Android 12 Material Design был заменен на Material You, в котором строгость макетов стала менее явной, а плавность и округленность стали чрезмерны.

4 Экспериментальная часть

4.1 Использование Clean Architecture

Для реализации проекта было решено использовать Clean Architecture. Clean Architecture подразумевает разделение проекта на слои. Чаще всего используется 3 слоя, в виде пакетов приложения:

– data – связь с базой данных, бэкенд API;

– domain – получив данные с data слоя, обрабатывает их в так называемых use cases и создает модельки, на основе бизнес-процессов приложения, в более удобные для отображения во view слое. По сути является связью между data и view слоями;

– view – слой, описывающий фронт часть проекта.

Пакет (package) – набор классов, которые описывают одну из частей проекта. Более крупные проекты, которые имеют более сложные бизнес-процессы, описывают с помощью модулей.

Модуль – набор пакетов, которые описывают одну из частей проекта. Каждый модуль можно рассматривать как отдельное приложение.

Data и domain слои связаны посредством интерфейса DatabaseDAO и FirebaseAPI callback-функций. Domain и presentation слои связаны с помощью ViewModel и массивов типа LiveData, реализованного посредством паттерна Observer.

При построении архитектуры, в связи с тем, что проект не является большим и отсутствуют сложные бизнес-процессы, было решено отказаться от domain слоя. В итоге полученную архитектуру мы видим на рисунке N. Также на рисунке мы можем заметить пакет di, который хранит в себе файл с описанием dependency injection.

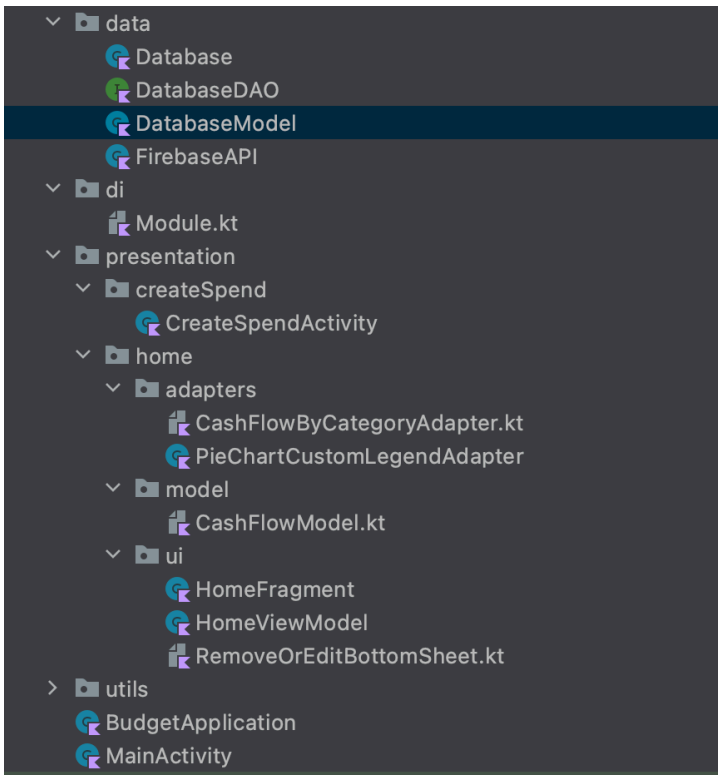

Рисунок 4.1 - Архитектура приложения

В пакете data описывается instance базы данных, интерфейс DAO для базы данных, сама база данных в виде моделей данных, а также FirebaseAPI в которой описан instance Firebase Realtime Database и callback-функции для доступа к данным с внешней базы данных.

В пакете presentation описывается view приложения. Данный пакет хранит в себе несколько пакетов, каждый из которых описывает отдельный экран. Более сложные экраны хранят в себе еще несколько пакетов по типу adapter, ui и model. Adapter класс описывает списки на экране, UI – сам экран, а model – viewModel для связи с data/domain слоями.

Пакет utils – дополнительные inline, extension, lambda функции необходимые для разных функций в разных местах.

Inline-функции – функции которые «встраиваются» в код во время компиляции приложения, что увеличивает скорость компиляции.

Extension-функции – функции расширения, используются для расширения уже существующих классов. Например, нам необходимо добавить некоторый метод для работы со строками, чтобы использовать его по всем проекту, но так как класс String является закрытым, то мы воспользуемся extension-функцией.

Lambda-функция – функция, которая принимает в себя другое выражение или функцию.

Класс MainActivity является отправной точкой фронт части приложения, чаще всего именно это активити запускается первым.

Класс BudgetApplication является отправной точкой до рендеринга фронт части приложения, здесь описываются настройки, которые должны сработать до и во время отображения первого экрана.

4.2 Реализация базы данных и моделей

Для хранения локальных данных мы воспользовались базой данной, которая хранится непосредственно на устройстве пользователя. Для этого мы подключили библиотеку Room Database, которая является самой популярной и удобной в разработке андроид приложений на данный момент.

Для реализации Room Database мы создадим абстрактный класс, который наследуется от Database класса и дает нам доступ к библиотеке. Над классом объявим аннотацию @Database, в которой укажем версию базы и массив из таблиц.

```
import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
    entities = [
        DatabaseModel::class,
        CategoryModel::class
    ],
    version = 1
)
abstract class Database : RoomDatabase() {
    abstract val dao: DatabaseDAO
}
```

Рисунок 4.2 - Реализация абстрактного класса базы данных

При создании объекта базы данных он обязательно должен быть в единственном instance, чтобы не создать несколько баз, которые никак не будут друг с другом связаны и будут только мешать друг другу, пытаясь одновременно получить доступ к одному и тому же файлу на устройстве пользователя.

Поэтому, чтобы создать объект базы данных мы объявим его в настройке dependency injection проекта:

```
val database = module { this: Module
    fun provideDataBase(application: Application) =
        Room.databaseBuilder(application, Database::class.java, name: "Database")
            .fallbackToDestructiveMigration()
            .build()

    fun provideDao(database: Database) =
        database.dao

    single { provideDataBase(androidApplication()) }
    single { provideDao(get()) }
}
```

Рисунок 4.3 - Создание instance базы данных

Как мы видим на рисунке выше, мы создаем два single объекта database и dao, посредством паттерна синглтон. Функция provideDatabase создает базу данных с помощью callback функции databaseBuilder, а функция provideDao создает интерфейс для доступа к БД.

Далее создадим таблицы для БД, в котором объявим таблицы в виде data классов, а над каждым классом объявим аннотацию @Entity, а в аннотации укажем название таблицы.

Над одной из переменных указываем аннотацию @PrimaryKey, которая укажет что эта переменная – первичный ключ.

```
import androidx.annotation.DrawableRes
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "spends")
data class DatabaseModel(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val categoryName: String,
    @DrawableRes
    val categoryImage: Int,
    val spendName: String,
    val totalSum: Int = 0
)

@Entity(tableName = "categories")
data class CategoryModel(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val categoryName: String
)
```

Рисунок 4.4 - Таблицы база данных

После нам необходимо создать интерфейс, через который мы будем получать доступ к библиотеке. Для этого создадим интерфейс и объявим над ним аннотацию @Dao.

В интерфейсы создадим методы, над которым объявим аннотации @Query, @Insert и @Delete.

```
@Dao
interface DatabaseDAO {

    @Query( value: "SELECT * FROM spends")
    fun getAllSpends(): LiveData<List<DatabaseModel>>

    @Query( value: "SELECT * FROM spends WHERE spendName=:spendName")
    fun getSpend(spendName: String): LiveData<List<DatabaseModel>>

    @Insert(onConflict = REPLACE)
    fun insertSpend(spend: DatabaseModel)

    @Delete
    fun deleteSpend(spend: DatabaseModel)
}
```

Рисунок 4.5 - DAO интерфейс базы данных

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта и разработки мобильного приложения были реализованы все поставленные цели. Проведя опросы, а на основе опыта стало понятно, что в современной ситуации следить за личными тратами сложно, ведь у многих людей множество карт, депозитов и не видно их сразу и все, также повсеместно вводятся сервисы онлайн оплаты, оплаты картой, QR и перевода средств на карты напрямую.

Для достижения цели были проделаны работы по следующим этапам разработки:

– изучение рынка и конкурентов;

– определение задач и требований;

– проектирование архитектуры и UML диаграмм;

– разработка дизайна;

– разработка самого приложения.

В настоящее время существует множество приложений по контролю бюджета, но, проанализировав рынок, был выделен перегруженный или же недостающий функционал. Приложение по контролю бюджета не должно быть перегруженным, все должно быть быстро и просто, чтобы пользователь не утомлялся и у него сохранялся интерес в отслеживании и построении своего бюджета.

Большая часть времени была потрачена на разработку удобного дизайна, чтобы пользоваться приложением было не сложно и удобно.

В данный момент приложение имеет функционал по добавлению трат и категорий, в дальнейшем проект будет дополняться новым функционалом и выложен в Play Market.

В итоге было создано адаптивное и динамически обновляемое приложение, с использованием чистой архитектуры для дальнейшего расширения.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1 Документация к разработке Android приложений // Электронная версия на сайте https://developer.android.com/guide

2 Общая информация об IDE Android Studio // Электронная версия на сайте https://developer.android.com/studio

3 Роберт С. Мартин Быстрая разработка программ. Принципы, примеры, практика. – Вильямс, 2004.

4 Роберт С. Мартин Принципы, паттерны и методики гибкой разработки на языке C#. – Символ-Плюс, 2011

5 Документация к работе с Activity // Электронная версия на сайте https://developer.android.com/reference/android/app/Activity

6 Application Fundamentals // Электронная версия на сайте https://developer.android.com/guide/components/fundamentals

7 Save data in a local database using Room // Электронная версия на сайте https://developer.android.com/guide/components/fundamentals

8 Reflection API. Рефлексия. Темная сторона Java // Электронная версия на сайте https://javarush.ru/groups/posts/513-reflection-api-refleksija-temnaja-storona-java

9 М. Фоулер InversionOfControl // Электронная версия на сайте http://martinfowler.com/bliki/InversionOfControl.html

10 Роберт С. Мартин Чистый код. Создание, анализ, рефакторинг. – Питер, 2019

11 Dependency injection in Android // Электронная версия на сайте https://developer.android.com/training/dependency-injection

12 Dependency injection in Android // Электронная версия на сайте https://medium.com/@nikitaverma081996/dependency-injection-in-android-6eca8d41ae17

13 М. Симан Внедрение зависимостей в .NET. – MANNING, 2013

14 Роберт С. Мартин Clean Code: A Handbook of Agile Software Craftsmanship – Pearson Education, 2008

15 Документация к Koin // Электронная вересия на сайте https://insert-koin.io/docs/reference/introduction

16 Koin — библиотека для внедрения зависимостей, написанная на чистом Kotlin // Электронная версия на сайте https://habr.com/ru/company/otus/blog/530024/

17 Документация к OkHttp // Электронная версия на сайте https://square.github.io/okhttp/

18 Документация к Retrofit // Электронная версия на сайте https://square.github.io/retrofit/

19 Роберт С. Мартин Чистая архитектура. Искусство разработки программного обеспечения. – Питер, 2020

20 Clean Architecture in Android – A Beginner Approach // Электронная версия на сайте https://medium.com/swlh/clean-architecture-in-android-a-beginner-approach-be0ce00d806b

21 Detailed Guide on Android Clean Architecture // Электронная версия на сайте https://medium.com/android-dev-hacks/detailed-guide-on-android-clean-architecture-9eab262a9011

22 Guide to app architecture // Электронная версия на сайте https://developer.android.com/topic/architecture

23 MVVM на Android с компонентами архитектуры + библиотека Koin // Электронная версия на сайте https://medium.com/nuances-of-programming/mvvm-%D0%BD%D0%B0-android-%D1%81-%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%B0%D0%BC%D0%B8-%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D1%8B-%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0-koin-e2e77b77950e

24 Е. Мацюк, А. Бадретдинов, А. Блинов AndroidArchitectureBook // Электронная версия на сайте https://github.com/AndroidArchitecture/AndroidArchitectureBook/blob/master/Intro.md

Приложение А
(обязательное)

Техническое задание

А.1.5 Техническое задание на разработку мобильного приложения для контроля бюджета

Настоящее техническое задание распространяется на разработку приложения для контроля бюджета, предназначенной для хранения информации о тратах пользователя. Предполагается, что данное приложение будут использовать обычные пользователи. Данная система позволит конечному пользователю следить за своими тратами и контролировать бюджет.

А.1.5.1 Основание для разработки

Приложение разрабатывается на основе устного согласия научного руководителя выбранной дипломником темой.

А.1.5.2 Назначение

Система предназначена для хранения и обработки сведений о тратах конечного пользователя приложения.

А.1.5.3 Требования к функциональным характеристикам

Система должна обеспечить возможность выполнения следующих функций:

– отображение трат
– хранение трат в локальной базе данных
– возможность добавления трат
– возможность добавления категорий
– возможность удаления и изменения трат и категорий

А.1.5.4 Требования к надежности

Предусмотреть контроль вводимой информации. Предусмотреть блокировку некорректных действий пользователя при работе с ситемой. Обеспечить целостность хранимой информации.

А.1.5.5 Требованию к составу и параметрам технических средств

Система должна работать на мобильных устройствах с количество ОЗУ не ниже 1 Гб.

А.1.5.6 Требования к информационной и программной совместимости

Система должна работать на устройствах под управлением ОС Android 7.0 и выше. Обязательно наличие Google сервисов.

Приложение Б
(обязательное)

Текст программы

```
package kz.temirlan.controlbudget

import android.app.Application
import kz.temirlan.controlbudget.di.budgetViewModel
import kz.temirlan.controlbudget.di.database
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.core.context.startKoin

class BudgetApplication : Application() {

    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidLogger(level = org.koin.core.logger.Level.ERROR)
            androidContext(this@BudgetApplication)
            modules(
                database,
                budgetViewModel,
            )
        }
    }
}

package kz.temirlan.controlbudget

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.viewbinding.library.activity.viewBinding
import androidx.navigation.Navigation
import androidx.navigation.findNavController
import androidx.navigation.fragment.NavHostFragment
import androidx.navigation.ui.NavigationUI
import androidx.navigation.ui.setupWithNavController
import com.faskn.lib.PieChart
import com.faskn.lib.Slice
import kz.temirlan.controlbudget.databinding.ActivityMainBinding
import kotlin.random.Random
```

Продолжение приложения Б

```
class MainActivity : AppCompatActivity(R.layout.activity_main) {

    private val binding by viewBinding<ActivityMainBinding>()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
}

package kz.temirlan.controlbudget.presentation.home.ui

import android.content.Intent
import android.os.Bundle
import android.view.View
import android.viewbinding.library.fragment.viewBinding
import androidx.fragment.app.Fragment
import com.faskn.lib.Slice
import com.faskn.lib.buildChart
import kz.temirlan.controlbudget.R
import kz.temirlan.controlbudget.databinding.FragmentHomeBinding
import
kz.temirlan.controlbudget.presentation.createSpend.CreateSpendActivity
import
kz.temirlan.controlbudget.presentation.home.adapters.CashFlowByCategoryAd
apter
import
kz.temirlan.controlbudget.presentation.home.adapters.CashFlowInterface
import
kz.temirlan.controlbudget.presentation.home.adapters.PieChartCustomLegend
Adapter
import
kz.temirlan.controlbudget.presentation.home.model.CashFlowByCategory
import org.koin.androidx.viewmodel.ext.android.sharedViewModel
import kotlin.random.Random

class HomeFragment : Fragment(R.layout.fragment_home), CashFlowInterface
{

    private val binding by viewBinding<FragmentHomeBinding>()
    private val viewModel by sharedViewModel<HomeViewModel>()
    private val adapter = CashFlowByCategoryAdapter(this@HomeFragment)
```

Продолжение приложения Б

```
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setupView()
    }

    private fun setupView() {
        binding.addButton.setOnClickListener {
            val intent = Intent(requireContext(), CreateSpendActivity::class.java)
            startActivity(intent)
        }
        initPieChart()
        initAdapter()
        observeViewModel()
    }

    private fun updateTitle() {
        binding.toolbar.title = "${viewModel.allSpends.sumOf { it.totalSum }} $"
    }

    private fun observeViewModel() {
        viewModel.updateRecycler.observe(viewLifecycleOwner) {
            adapter.setData(viewModel.allSpends)
            updateTitle()
            initPieChart()
        }
    }

    private fun initPieChart() {
        val pieChart = buildChart {
            slices { provideSlices() }
            sliceWidth { 80f }
            sliceStartPoint { 0f }
        }

        binding.pieChart.setPieChart(pieChart)
        binding.pieChart.showLegend(binding.legendLayout,
PieChartCustomLegendAdapter())
    }

    private fun initAdapter() {
        binding.cashFlowRecycler.apply {
```

```
        adapter = this@HomeFragment.adapter
    }
    adapter.setData(viewModel.allSpends)
}

private fun provideSlices(): ArrayList<Slice> {
    return if (viewModel.allSpends.isEmpty()) {
        arrayListOf(
            Slice(
                10.0F,
                R.color.dark_purple,
                "Food"
            ),
            Slice(
                10.0F,
                R.color.orange,
                "Apartment"
            ),
            Slice(
                10.0F,
                R.color.sky_blue,
                "Services"
            )
        )
    }
    else {
        arrayListOf(
            Slice(
                viewModel.allSpends.filter { it.categoryName == "Food" }
                    .sumOf { it.totalSum }.toFloat(),
                R.color.dark_purple,
                "Food"
            ),
            Slice(
                viewModel.allSpends.filter { it.categoryName == "Apartments" }
                    .sumOf { it.totalSum }.toFloat(),
                R.color.orange,
                "Apartments"
            ),
            Slice(
                viewModel.allSpends.filter { it.categoryName == "Services" }
```

```
                .sumOf { it.totalSum }.toFloat(),
                R.color.sky_blue,
                "Services"
            )
        )
    }
  }

  override fun onResume() {
    super.onResume()
    adapter.setData(viewModel.allSpends)
    updateTitle()
    initPieChart()
  }

  override fun onClick(id: Int, item: CashFlowByCategory) {
    val bottomSheet = RemoveOrEditBottomSheet()
    val bundle = Bundle()
    bundle.putParcelable("item", item)
    bundle.putInt("idItem", id)
    bottomSheet.arguments = bundle
    bottomSheet.show(parentFragmentManager, "tag")
  }
}

package kz.temirlan.controlbudget.presentation.home.adapters

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.RecyclerView
import
kz.temirlan.controlbudget.databinding.ItemCashFlowByCategoryBinding
import
kz.temirlan.controlbudget.presentation.home.model.CashFlowByCategory

interface CashFlowInterface {
  fun onClick(id: Int, item: CashFlowByCategory)
}

class CashFlowByCategoryAdapter(
  private val clickListener: CashFlowInterface
```

```
) :
RecyclerView.Adapter<CashFlowByCategoryAdapter.CashFlowByCategoryV
iewHolder>() {

    private val items = mutableListOf<CashFlowByCategory>()

    fun setData(newItems: MutableList<CashFlowByCategory>) {
        items.clear()
        items.addAll(newItems)
        notifyDataSetChanged()
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        CashFlowByCategoryViewHolder(
            ItemCashFlowByCategoryBinding.inflate(
                LayoutInflater.from(parent.context),
                parent,
                false
            )
        )

    override fun onBindViewHolder(holder: CashFlowByCategoryViewHolder,
position: Int) {
        holder.bind(items[position])
    }

    override fun getItemCount() = items.size

    inner class CashFlowByCategoryViewHolder(private val binding:
ItemCashFlowByCategoryBinding) : RecyclerView.ViewHolder(binding.root)
{

        fun bind(item: CashFlowByCategory) = with(binding) {
            categoryImage.setImageResource(item.categoryImage)
            categoryName.text = item.categoryName
            spendName.text = item.spendName
            totalSum.text = "${item.totalSum} $"
            root.setOnClickListener {
                clickListener.onClick(adapterPosition, item)
            }
        }
```

```
    }
}

package kz.temirlan.controlbudget.presentation.home.ui

import android.os.Bundle
import android.util.Log
import android.view.Gravity
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.view.animation.AnimationUtils
import android.widget.ArrayAdapter
import android.widget.Button
import android.widget.FrameLayout
import android.widget.Toast
import androidx.coordinatorlayout.widget.CoordinatorLayout
import com.google.android.material.bottomsheet.BottomSheetBehavior
import com.google.android.material.bottomsheet.BottomSheetDialog
import com.google.android.material.bottomsheet.BottomSheetDialogFragment
import kz.temirlan.controlbudget.R
import
kz.temirlan.controlbudget.databinding.BottomSheetDeleteFragmentBinding
import
kz.temirlan.controlbudget.presentation.home.adapters.CashFlowByCategoryAd
apter
import
kz.temirlan.controlbudget.presentation.home.model.CashFlowByCategory
import kz.temirlan.controlbudget.utils.gone
import kz.temirlan.controlbudget.utils.hide
import kz.temirlan.controlbudget.utils.show
import org.koin.androidx.viewmodel.ext.android.sharedViewModel

private const val COLLAPSED_HEIGHT = 228

class RemoveOrEditBottomSheet : BottomSheetDialogFragment() {

    lateinit var binding: BottomSheetDeleteFragmentBinding
    private val viewModel by sharedViewModel<HomeViewModel>()
    private var item: CashFlowByCategory? = null
    private var idItem: Int? = 0
```

Продолжение приложения Б

```
    override fun getTheme() = R.style.AppBottomSheetDialogTheme

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        binding = BottomSheetDeleteFragmentBinding.bind(
            inflater.inflate(R.layout.bottom_sheet_delete_fragment, container)
        )
        return binding.root
    }

    override fun onStart() {
        super.onStart()

        getBundleArguments()
        setupView()

        val animUpDown = AnimationUtils.loadAnimation(requireContext(),
R.anim.up_down)
        binding.arrow.startAnimation(animUpDown)

        val density = requireContext().resources.displayMetrics.density

        dialog?.let {
            val bottomSheet =
it.findViewById<View>(com.google.android.material.R.id.design_bottom_she
et) as FrameLayout
            val behavior = BottomSheetBehavior.from(bottomSheet)

            behavior.peekHeight = (COLLAPSED_HEIGHT * density).toInt()
            behavior.state = BottomSheetBehavior.STATE_COLLAPSED

            val coordinator = (it as
BottomSheetDialog).findViewById<CoordinatorLayout>(com.google.android.
material.R.id.coordinator)
            val containerLayout =
it.findViewById<FrameLayout>(com.google.android.material.R.id.container)
```

Продолжение приложения Б

```
        val buttons = it.layoutInflater.inflate(R.layout.bottom_sheet_button,
null)
        buttons.findViewById<Button>(R.id.button_a).setOnClickListener {
            item.let { item ->
                if (item != null) {
                    viewModel.deleteSpend(item)
                }
                viewModel.sendEvent()
            }
            dismiss()
        }
        buttons.layoutParams = FrameLayout.LayoutParams(
            FrameLayout.LayoutParams.MATCH_PARENT,
            FrameLayout.LayoutParams.WRAP_CONTENT
        ).apply {
            height = (60 * density).toInt()
            gravity = Gravity.BOTTOM
        }

        containerLayout?.addView(buttons)

        buttons.post {
            (coordinator?.layoutParams as
ViewGroup.MarginLayoutParams).apply {
                buttons.measure(
                    View.MeasureSpec.makeMeasureSpec(0,
View.MeasureSpec.UNSPECIFIED),
                    View.MeasureSpec.makeMeasureSpec(0,
View.MeasureSpec.UNSPECIFIED)
                )
                this.bottomMargin = (buttons.measuredHeight - 8 *
density).toInt()
                containerLayout?.requestLayout()
            }
        }

        behavior.addBottomSheetCallback(object :
BottomSheetBehavior.BottomSheetCallback() {

            override fun onStateChanged(bottomSheet: View, newState: Int) {
```

```
            }

            override fun onSlide(bottomSheet: View, slideOffset: Float) {
                with(binding) {
                    if (slideOffset > 0) {
                        layoutCollapsed.alpha = 1 - 2 * slideOffset
                        layoutExpanded.alpha = slideOffset * slideOffset

                        if (slideOffset > 0.5) {
                            layoutCollapsed.gone()
                            arrow.clearAnimation()
                            layoutExpanded.show()
                        }

                        if (slideOffset < 0.5 && binding.layoutExpanded.visibility
== View.VISIBLE) {
                            layoutCollapsed.show()
                            arrow.startAnimation(animUpDown)
                            layoutExpanded.hide()
                        }
                    }
                }
            }
        })
    }
}

private fun setupView() = with(binding) {
    initDropDownMenu()
    backButton.setOnClickListener {
        dismiss()
    }
    buttonEdit.setOnClickListener {
        item?.let { item ->
            idItem.let { id ->
                if (id != null) {
                    viewModel.editSpend(
                        id,
                        CashFlowByCategory(
                            categoryName = categoryDropdown.text.toString(),
```

```
                        categoryImage =
CashFlowByCategory.getCategoryImage(categoryDropdown.text.toString()),
                        spendName = nameEditText.text.toString(),
                        totalSum = sumEditText.text.toString().toInt()
                    )
                )
            }
        }
        viewModel.sendEvent()
        dismiss()
    }
}

private fun getBundleArguments() {
    item = arguments?.getParcelable("item")
    idItem = arguments?.getInt("idItem")
}

private fun initDropDownMenu() {
    val categories = listOf(
        "Food",
        "Services",
        "Apartments"
    )
    binding.categoryDropdown.setAdapter(
        ArrayAdapter(
            requireContext(),
            R.layout.item_dropdown,
            categories
        )
    )
}
}

package kz.temirlan.controlbudget.data

import androidx.room.Database
import androidx.room.RoomDatabase

@Database(
```

```
    entities = [
        DatabaseModel::class,
        CategoryModel::class
    ],
    version = 1
)
abstract class Database : RoomDatabase() {
    abstract val dao: DatabaseDAO
}

package kz.temirlan.controlbudget.data

import androidx.annotation.DrawableRes
import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName = "spends")
data class DatabaseModel(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val categoryName: String,
    @DrawableRes
    val categoryImage: Int,
    val spendName: String,
    val totalSum: Int = 0
)

@Entity(tableName = "categories")
data class CategoryModel(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    val categoryName: String
)

package kz.temirlan.controlbudget.data

import androidx.lifecycle.LiveData
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy.REPLACE
```

Продолжение приложения Б

```
import androidx.room.Query

@Dao
interface DatabaseDAO {

    @Query("SELECT * FROM spends")
    fun getAllSpends(): LiveData<List<DatabaseModel>>

    @Query("SELECT * FROM spends WHERE spendName=:spendName")
    fun getSpend(spendName: String): LiveData<List<DatabaseModel>>

    @Insert(onConflict = REPLACE)
    fun insertSpend(spend: DatabaseModel)

    @Delete
    fun deleteSpend(spend: DatabaseModel)
}

package kz.temirlan.controlbudget.di

import android.app.Application
import androidx.room.Room
import kz.temirlan.controlbudget.data.Database
import kz.temirlan.controlbudget.presentation.home.ui.HomeViewModel
import org.koin.android.ext.koin.androidApplication
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.dsl.module
import kotlin.math.sin

val database = module {
   fun provideDataBase(application: Application) =
      Room.databaseBuilder(application, Database::class.java, "Database")
         .fallbackToDestructiveMigration()
         .build()

   fun provideDao(database: Database) =
      database.dao

   single { provideDataBase(androidApplication()) }
   single { provideDao(get()) }
}
```

Продолжение приложения Б

```
val budgetViewModel = module {
    single {
        HomeViewModel()
    }
}
```