Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования Московский государственный технический университет имени Н.Э. Баумана

Курсовой проект

по учебной дисциплине

«Базы данных»

На тему:

«Разработка Android-приложения «Электронная медицинская карта»

Выполнил:
Студент группы ИУ9-62
_____ Белогуров А.А.
«____» _____ 2017 г.
Руководитель курсового проекта:
_____ Домрачева А.Б.
«____» _____ 2017 г.

Москва, 2017 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современный мир информационных технологий трудно представить без использования баз данных. Уже при появлении самых простых информационных устройств, таких как перфокарты, возникла потребность хранить структурированный объем данных.

История появления БД начинается в 50-60 годах предыдущего столетия, а пик их роста начинается в 90-ые годы. Это время характерно появлением персональных компьютеров, которые стали ближе и доступнее каждому пользователю. Появилось множество программ, предназначенных для работы неподготовленных пользователей. Эти программы были просты в использовании и интуитивно понятны: это прежде всего различные редакторы текстов, электронные таблицы и другие. Простыми и понятными стали операции копирования файлов и перенос информации с одного компьютера на другой, распечатка текстов, таблиц и других документов.

Системы баз данных сегодня являются основой построения большинства информационных систем, которыми так же являются и мобильные устройства. Сейчас невозможно представить человека без телефона в руках, который без проблем может решить свою проблему при помощи пару нажатий на экран. В первую очередь – это всевозможные приложения, от тех, которые помогают составлять списки покупок и планировать бюджет, до почты и календаря. Однако, существуют приложения, которые помогают автоматизировать производство предприятий, магазинов или больниц. В основе этого все так же лежат базы данных, которые могут быстро предоставить нужную информацию пользователю.

Таким образом, можно сразу определить актуальность моей работы. Она заключается в том, что современный мир невозможно представить без информационных систем с интеграцией баз данных. В этом контексте мобильные

приложения позволяют решать множество задач, упрощают рутинную работу и увеличивают эффективность труда.

Объектом исследования являются ресурсоемкие базы данных для портативных устройств. Предметом исследования является отображение результатов базы данных на Android устройствах.

В ходе работы должны быть выполнены следующие задачи:

- обзор литературы, выбор языка программирования и базы данных на операционной системе Android;
- проектирование базы данных и её нормализация;
- реализация базы данных и приложения, в том числе заполнение данными;
- тестирование работоспособности приложения.

Целью данной курсовой работы является разработка приложения для Android устройств, которое содержит информацию о базе данных больницы и позволяет ею управлять.

1. Анализ возможностей организации СУБД на базе Android.

1.1. Тенденция развития мобильных устройств.

Первые портативные мобильные телефоны начали появляться в 80-90 годах предыдущего столетия, но они были очень дорогими, что отпугивало большинство покупателей, работали меньше часа без подзарядки и могли хранить в своей памяти только 8 номеров телефонов, например, телефон 1987 года «Mobira Cityman 900». Многие компании тогда скептически относились к рынку мобильных устройств, не все были готовы вкладывать огромные суммы денег на производство и исследование данной области. Однако, нашлись компании, которые были готовы рискнуть, они понимали, что за этим наше будущее.

Первопроходцами стали компании «Nokia», «Motorola» и «Siemens». Именно они задали ту тенденцию кнопочных телефонов. Рынок мобильных устройств после 90-го года начал развиваться очень стремительно, этому поспособствовала возникшая конкуренция между этими тремя компаниями. Каждый хотел показать, что именно его продукция самая лучшая. Например, один из самых популярных телефонов, выпущенный в 1989 году, «Motorola MicroTAC 9800x» выпускался вплоть до 1998 года в различных модификациях. Он весил всего 350 грамм и помещался в кармане рубашки.

Позже производители телефонов поняли, что кроме возможности звонков, телефон должен уметь хранить текстовые сообщения, историю вызовов, календарь, а также показывать время. В 1998 году компания «Nokia» выпустила «Nokia 9000 Communicator», который обладал всеми выше перечисленными функциями, но кроме этого мог обладал функционалом навигатора, для всего этого ему была доступна память в 8 МБ.

Каждая компания выпускала телефоны на своей операционной системе. Переходя с телефона одного производителя на телефон другого, пользователь

мог долго разбираться в новых возможностях приобретенного телефона. Именно в 9 января 2007 года компанией Apple был представлен iPhone первого поколения, он работал на совершенно новой и непохожей на другие операционные системы - iPhone OS, которая в будущем будет переименована в iOS. Однако данная операционная система может быть установлена только на продукцию компании Apple.

В компании Google захотели сделать конкурента iPhone OS, но с тем условием, что новую ОС сможет использовать любой производитель в своих телефонах. 23 сентября 2008 года была показана Android 1.0 «Apple Pie» на телефоне «T-Mobile G1», который в Европе продавался под именем «HTC G1». В то время это было революцией для рынка мобильных устройств, именно так зародились две самые популярные операционные системы на сегодняшний день.

Первые версии iPhone OS и Android не предполагали установку сторонних приложений, однако в течении двух лет это было исправлено, и каждый разработчик мог написать свое приложение для этих операционных систем. Так как в смартфонах память ограничена, то сразу же вставала острая проблема «где и как хранить информацию в мобильном устройстве?». Далее в моей курсовой работе будут рассматриваться способы хранения данных от приложений в мобильных устройствах на операционной системе «Android».

Разработчики ОС от Google предусмотрели три основных способа хранения информации:

- внутренняя (внешняя) память;
- Shared Preferences;
- SQLite;
- сторонние библиотеки.

1.2. Анализ способов хранения информация в ОС Android.

1.2.1. Внутренняя и внешняя память.

При первом запуске приложения в его папке создается файл методом «*FileOutputStream openFileOutput(String name, int mode)*», в который можно записывать информацию, которая может потребоваться при последующих запусках приложения.

По умолчанию файл во внутреннем памяти недоступны приложениям и пользователю, данный файл удаляется вместе с приложением. Для внешней памяти используется SD-карта или встроенная память, файлы доступны всем и могут быть изменены (или удалены) пользователем.

Данный метод хранения информация мало эффективен и почти нигде не используется, так как информация не структурирована и представлена в виде последовательности строк. Для получения данных в конце файла необходимо будет пройтись по всему файлу.

1.2.2. Интерфейс Shared Preferences.

Shared Preferences Позволяет сохранять данные примитивных типов: *boolean, float, int, long, string*. Так же данные можно сериализовать, что дает возможность сохранять данные любых типов и обращаться к ним. Данные хранятся парами ключ-значение.

Чтобы их прочитать, необходимо знать ключ. Сохранение данных выполняется для всех сессий пользователя, даже если приложение будет убито. Они недоступны извне приложения.

Данный способ используется для хранения небольших данных, вроде состояний окна и информации о пользователе. Нет смысла использовать Shared Preferences для хранения больших структурированных данных, так как для каждого поля придется придумывать свой ключ и хранить его отдельно в памяти.

1.2.3. СУБД SQLite.

SQLite - компактная встраиваемая реляционная база данных с открытым исходным кодом. Встраиваемая означает что данная база данных не использует парадигму сервер-клиент. SQLite предоставляет библиотеку, с которой программа компонуется, и, таким образом, в качестве протокола обмена используются вызовы функций API библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу.

Сама библиотека написана на языке программирования C, простота и удобство встраивания привели к тому, что SQLite используется в браузерах, музыкальных плеерах и многих других программах, например, Skype, Mozilla Firefox.

Поэтому для больших одинаковых и структурированных объемов данных в Android началась использоваться база данных SQLite. Она поддерживает внутренний синтаксис, однако данными можно управлять и с помощью языка запросов, как в T-SQL.

SQLite доступен на любом Android-устройстве и его не нужно устанавливать отдельно.

Данная БД поддерживает следующие типы данных:

- NULL - пустое значение
- INTEGER - целочисленное значение
- REAL - значение с плавающей точкой
- TEXT - строки или символы с кодировкой UTF-8, UTF-16BE или UTF-16LE
- BLOB - бинарные данные

Остальные типы следует конвертировать, прежде чем сохранять в базе данных. SQLite не поддерживает тип boolean, для этого используется цифры 0 и 1. Для хранения картинок и других изображений не рекомендуется использовать тип BLOB, лучше хранить в базе путь к изображениям, а сами изображения хранить в файловой системе. Так же здесь нельзя использовать varchar(n) для

ограничения длины строки. Нельзя удалить или изменить столбец в таблице, есть триггеры, но не такие мощные, как в настольных базах данных, есть поддержка foreign key, но по умолчанию она отключена. Однако, несмотря на малую функциональность данной БД, она полноценно используется многими разработчиками в популярных проектах (Рисунок 1.3.1).

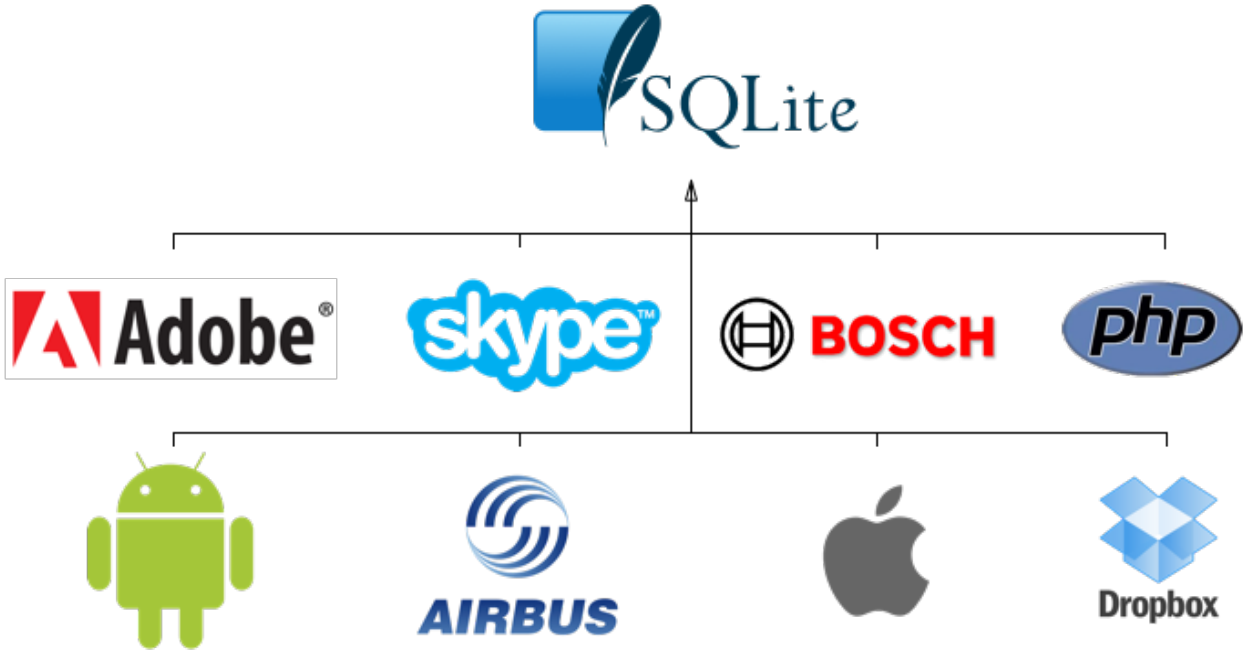

Рисунок 1.2.3.1 – Использование SQLite в разработке других программ и систем.

Так как база данных SQLite представляет собой файл, в котором хранятся и таблицы, и индексы, и сами данные, то по сути при работе с базой данных, приложение будет взаимодействовать с файлом. Поэтому операции чтения и записи могут быть довольно медленными. Этого можно избежать, используя асинхронные задачи или AsyncTask.

Класс AsyncTask в Android предназначен для выполнения какой-либо задачи в отдельном потоке, с последующем отображением результата в главном потоке. Таким образом, при использования асинхронных задач приложение не будет останавливаться и продолжит работать дальше независимо от того, что

происходит в AsyncTask. Данный способ работы приложения очень распространен в разработке Android-приложений и может использоваться не только при работе с SQLite, но и так же при обработке данных, полученных с сервера.

1.2.4. Сторонние библиотеки.

В последнее время все больше популярности получают сторонние библиотеки в android, которые позволяют быстро и эффективно использовать базы данных.

Realm – это мобильная база данных, которая называет себя «убийцей» SQLite (Android) и Core Data (iOS). Она позволяет создать реактивные и асинхронные мобильные приложения при работе с большими объёмами данных. А сама работа с ней, например, вставка, модифицирование или удаление записей делается в пару строк кода, что очень увеличивает эффективность разработки. Realm – довольно молодая библиотека, но уже успела стать одной из самых любимых у многих разработчиков. Она доступна на всех язык программирования для мобильных платформ: Java, Kotlin, Objective-C, React Native, Swift, Xamarin.

Room – библиотека, который представляет новый слой абстракции над SQLite, была представлена в мае на Google I/O 2017. В основе лежит три главных компонента. Сама база данных, с помощью которой выполняются все действия. Сущности, которые представлены в виде модели данных и объявляются с помощью аннотаций *@Entity, @PrimaryKey* и других. DAO – компонент, который может быть представлен в виде класса или интерфейса, объявляет методы для доступа к базе данных. Фактически, это та же самая SQLite, но с синтаксическим сахаром, которая упрощает разработку.

Не смотря на преимущества всех выше перечисленных библиотек, в данной курсовой работе будет использоваться SQLite, так как она используется по умолчанию не только в Android, но и в других успешных и крупных продуктах.

У нее большая документация и большое сообщество разработчиков, ежемесячные исправления ошибок и добавления новой функциональности.

2. Проектирование Базы Данных.

2.1. Разработка ER-модели базы данных больницы.

Прежде, чем приступать непосредственно к разработке, необходимо определить и сформировать понятия о предметах, фактах и событиях, которыми будет управлять сама система. Основой для этой информации будут выступать различные медицинские карты, которые находятся в свободном доступе. Далее их необходимо заменить информационными представлениями. Как правило, проектирование таких представлений сводится к разработке ER-модели данных.

Модель «сущность-связь» (Entity-Relationship) позволяет выделить ключевые сущности и определить связи, которые могут устанавливаться этими объектами. Каждая сущность определяется с помощью атрибутов, которые описывают свойства и в дальнейшем будут являться полями таблицы. Стоит отметить, что после создания ER-модели данных могут быть порождены все существующие модели данных: иерархическая, сетевая, реляционная и другие.

Так как при тщательном и детальном проектировании баз данных больницы могут возникнуть огромное количество сущностей, то для легкости использования обозначим только самые важные и значимые из них, которые не будут сказываться на большой ресурсоемкости мобильного приложения.

Первой из них будет создана «Person», которая будет общей для всех людей, включая весь персонал и пациентов. В качестве идентификатора служит обычное числовое значение, которое увеличивается на один при добавлении нового человека. Далее для остальных сущностей будет использоваться аналогичный подход, если не оговорено иное. Для данной сущности были выделены следующие атрибуты:

- идентификатор;
- полное имя;
- дата рождения;

- пол.

Далее определим сущность «Patient», которая будет дочерней сущностью к «Person» и иметь связь «один ко одному», так как один пациент определяется одним человеком. В качестве атрибутов будет служить краткая информация о самом человеке и его медицинские данные:

- идентификатор;
- группа крови;
- резус фактор;
- полный адрес;
- профессия;
- комментарии (может быть любая полезная для врача информация о пациенте).

Аналогично предыдущей выделим сущность «Doctor» с такой же связью к «Person», которая служит для определения всех врачей данной больницы, и связью «один ко многим» к «Patient», так как было выбрано, что у одного пациента может быть только один лечащий врач. Соответственно, один лечащий врач может быть у многих пациентов. Для атрибутов зададим следующие значения:

- идентификатор;
- специализация;
- дата выхода на работу.

Когда пациент приходит на прием первый раз, то необходимо определить сущность «Diagnosis», которая определяет болезнь или его недуг. В дальнейшем диагнозы могут варьироваться в зависимости от медицинских обследований. В качестве идентификатора здесь было выбрано значение по МКБ-10 – международная классификация болезней 10-го пересмотра. Она состоит из 21 раздела, каждый из которых содержит рубрики с кодами заболеваний и состояний, чего вполне хватит для описания всех диагнозов. Данная сущность

будет дочерней к «Patient» и обладать связью «многие ко многим», поскольку один диагноз может быть у разных пациентов, и, соответственно, у одного пациента может быть много разных диагнозов. В данном случае определим три атрибута:

- код по МКБ-10;
- название диагноза;
- подтверждён.

После того, как пациенту поставлен диагноз, нужно определить его дальнейшее лечение, которое может состоять из нескольких этапов. Для этого будет создана сущность «Treatment», которая включает в себя несколько типов лечения, например, как медикаментозный, хирургический и другие. Она является дочерней к «Diagnosis», имеет связь «многие ко многим» и определяется атрибутами:

- идентификатор;
- название лечения;
- тип лечения.

Но как правило, лечение некоторых диагнозов может повезти за собой появление различных побочных эффектов. Для этого будет введена сущность «SideEffect», которая будет дочерней к «Treatment» и иметь связь «многие ко одному». Установим, что побочные эффекты от лечения одинаковых болезней могут отличаться друг от друга в зависимости от поставленного диагноза и противопоказаний у пациента. Для этого будет добавлен дополнительный атрибут, в котором это может быть отображено. Соответственно список атрибутов данной сущности выглядит следующим образом:

- идентификатор;
- название побочного эффекта;
- комментарии.

2.2. Преобразование базы данных из ER-модели в реляционную.

После того, как была спроектирована ER-модель базы данных, то следующим шагом будет её преобразование в реляционную модель. Она представляет собой наиболее распространённое представление данных, нежели остальные системы, поэтому разберём более подробно как оно происходит.

Для этого необходимо определить тип данных для каждой записи таблицы, возможность значений по умолчанию или значений null. Выделить первичные и внешние ключи. Затем отобразить получившуюся структуру данных в виде таблиц (Таблицы 2.2.1-2.2.6). В предыдущем разделе для каждой сущности были подробно описаны роли таких ключей и зависимостей между ними. Стоит отметить, что ниже будут указаны только те типы данных, которые существует в используемой базе данных SQLite.

Таблица 2.2.1 – Описание таблицы «Person»

Имя столбца	Тип	Ключ
Person Id	INTEGER	Primary Key
Полное имя	TEXT	-
Дата рождения	TEXT	-
Пол	TEXT	-

Таблица 2.2.2 – Описание таблицы «Patient»

Имя столбца	Тип	Ключ
Patient Id	INTEGER	Primary Key
Person Id	INTEGER	Foreign Key
Doctor Id	INTEGER	Foreign Key
Код по МКБ-10	TEXT	Foreign Key
Группа крови	TEXT	-
Резус фактор	TEXT	-

Работа	TEXT	-
Комментарии	TEXT	-

Таблица 2.2.3 – Описание таблицы «Doctor»

Имя столбца	Тип	Ключ
Doctor Id	INTEGER	Primary Key
Patient Id	INTEGER	Foreign Key
Специализация	TEXT	-
Дата выхода на работу	TEXT	-

Таблица 2.2.4 – Описание таблицы «Diagnosis»

Имя столбца	Тип	Ключ
Код по МКБ-10	TEXT	Primary Key
Patient Id	INTEGER	Foreign Key
Treatment Id	INTEGER	Foreign Key
Название диагноза	TEXT	-
Подтверждён	INTEGER	-

Таблица 2.2.5 – Описание таблицы «Treatment»

Имя столбца	Тип	Ключ
Treatment Id	INTEGER	Primary Key
Код по МКБ-10	TEXT	Foreign Key
SideEffect Id	INTEGER	Foreign Key
Название лечения	TEXT	-
Тип лечения	TEXT	-

Таблица 2.2.6 – Описание таблицы «SideEffect»

Имя столбца	Тип	Ключ
SideEffect Id	INTEGER	Primary Key
Treatment Id	INTEGER	Foreign Key
Название поб. эффекта	TEXT	-
Комментарии	TEXT	-

После составления всех таблиц можно переходить к построению диаграммы, в которой будут отражены все сущности и соответствующие им зависимости. В данной базе данных между некоторыми таблицами образуется связь «многие ко многим», а именно между «Patient» и «Diagnosis», «Diagnosis» и «Treatment». Поэтому для каждой из этих связей создаётся дополнительно ещё одна таблица, которая одной записи в первой таблице будет сопоставлять другую запись во второй таблице. Такие таблицы называются связующими, поскольку существуют только для образования связей и будут обозначены жёлтым цветом. Получившаяся диаграмма отображена на рис. 2.2.1.

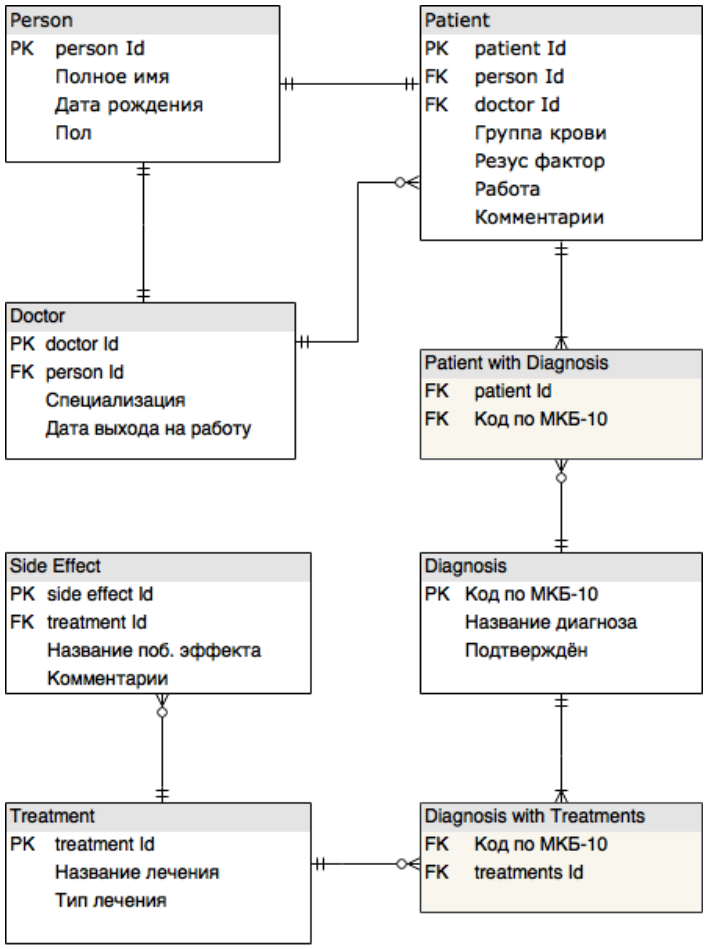

Рисунок 2.2.1 – Реляционная модель базы данных «Hospital».

2.3. Нормализация.

Последним шагом перед началом разработки самого приложением будет нормализация базы данных. Её целью является исключить избыточное дублирование данных, которое может привести к логическим ошибкам при выборке или изменении элементов таблиц. Всего существует 6 таких форм, каждая из которых предполагает некоторые условия, которым должна удовлетворять база данных. Целью этого раздела будет определить какой именно нормальной форме удовлетворяет получившиеся таблицы.

Первая нормальная форма предполагает, что все элементы внутри таблиц должны быть атомарными. В реляционной базе данных отношение всегда находится в 1НФ по определению понятия «отношение».

Вторая нормальная форма предполагает, что база данных находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от первичного ключа. В данном случае у каждой строки всех таблиц установлен уникальный идентификатор, что сразу исключает возможность найти меньшее подмножество атрибутов, от которого можно вывести нефункциональную зависимость.

Третья нормальная форма предполагает, что база данных находится в 2НФ и каждый не ключевой атрибут не транзитивно зависит от первичного ключа. Спроектированная модель в предыдущем разделе не удовлетворяет последнему правилу, поскольку для таблицы «Patient» атрибуты «группа крови» и «резус фактор» предполагают всего четыре и две возможных записи соответственно. Для того, чтобы получившаяся база данных находилась в 3НФ, необходимо выносить эти атрибуты в отдельные таблицы и в таблице «Patient» ссылаться на них. В самом начале было обговорено, что будут использованы самые базовые сущности и понятия, поэтому в данных момент нет смысла усложнять модель базы данных.

Так как каждая последующая нормальная форма непрерывно зависит от предыдущих, то нет смысла рассматривать их дальше. Поэтому можно сказать, что база данных «Hospital» находится во второй нормальной форме.

3. Реализация Базы Данных и приложения.

3.1. Выбор языка программирования и среды разработки.

Самой первой средой разработки под Android была Eclipse. Для создания приложения необходимо было подключить плагин Android Development Tools (ADT). Однако 9 декабря 2014 года Google заменила данную среду разработки на собственный проект – Android Studio. Эта IDE основана на программном обеспечении IntelliJ IDEA от JetBrains и доступна на следующие платформы: Windows, OS X и Linux. Разработка приложения введется на языке Java и XML. После анонсирования Android Studio Google прекратила поддержку плагина ADT для Eclipse.

Разберемся, почему же Android Studio лучше Eclipse. Во-первых, Android Studio более интеллектуальна. В ней присутствует очень удобная отладка приложения, доступен Debug режим. Прямо во время разработки можно запустить приложение на реальном устройстве, подсоединенным кабелем к компьютеру, и проверить его работу. Интерфейс намного проще и удобнее для пользователя. Доступно огромное количество форм и различных инструментов, для добавления их на экран устройства. Так же стоит отметить, что именно на Android Studio выходит большинство обучающих уроков как на русском, так и на английском языках, что делает проще знакомство с Android начинающим разработчикам.

Во-вторых, Google постоянно выпускает обновления, чтобы поддерживать основные тенденции дизайна и самых новых технологий. Самое последнее обновление было выпущено 16 сентября 2017 года, и теперь для разработки доступна версия Android Studio 3.0 beta 6. Из всех обновлений можно выделить самые значительные добавления:

- сборка приложений, основанная на Gradle;
- различные виды сборок, генерация нескольких .apk файлов и их анализ;

- рефакторинг кода;
- статический анализатор кода (Lint), который позволяет находить проблемы производительности, несовместимости версий и другое;
- шаблоны основных макетов и компонентов Android;
- поддержка разработки приложений для Android Wear и Android TV;
- версия Android Studio 2.1 получила улучшенную поддержку Java 8 и многое другое;

Таким образом, нет сомнений, что на данный момент Android Studio является лучшей средой разработки под Android, которая одновременно сочетает в себе удобство, скорость, огромный инструментарий и частые обновления.

Что касается языка программирования, в мае на Google I/O 2017 было объявлено об официальной поддержке языка Kotlin от JetBrains для Android. Kotlin – статически оптимизированный язык программирования, работающий поверх JVM, способен также компилироваться в JavaScript и на другие платформы через инфраструктуру LLVM.

Язык подразумевается, как замена существующей Java, но при этом способен существовать с ним параллельно в одном проекте. В частности, в Android, язык встраивается с помощью Gradle, что позволяет для разработчика внедрять новые функции на Kotlin без переписывания приложения целиком. Кроме прочего, Kotlin уже давно используется в больших компаниях и крупных проектах. Он имеет приятных синтаксис, очень большую документацию и что самое главное, он безопасный и позволяет избежать одну из самых популярных ошибок в Java – «null pointer exception».

Таким образом, для курсовой работы будет выбран именно Kotlin, но с тем условием, что класс для работы с SQLite будет написан на Java, так как изначально он предназначался именно для него, что позволит избежать ошибок в дальнейшем. Также можно будет проследить, как будет работать Koltin рядом с Java в одном проекте.

В итоге, разработка приложения будет производиться на операционной системе macOS High Sierra 10.13 с помощью среды разработки Android Studio 3 Beta 2 на языках программирования Java и Kotlin.

3.2. Создание класса для работы с SQLite.

Для того, чтобы работать с базой данных в операционной системе Android, необходимо создать класс, который наследуется от абстрактного *SQLiteOpenHelper*. Он позволяет переопределить методы *onCreate, onOpen, onClose* и другие, на которых основана работа SQLite. Так же необходимо написать свои методы для работы с данными. Далее будет показана работа только с одной таблицей, так как работа с другими однотипна и меняются только названия полей и методов.

Для того, чтобы в приложении можно было обращаться к SQLite, необходимо учесть тот факт, что для всего приложения должен существовать только одна база данных, для чего идеально подходит паттерн программирования Singleton (Листинг 3.2.1). Он гарантирует, что в приложении будет только один объект некоторого класса и обеспечивает глобальный доступ к нему.

Листинг 3.2.1 – использование паттерна Singleton при работе с SQLite

```
1.  public class DBHandler extends SQLiteOpenHelper {
2.
3.      private static DBHandler sInstance;
4.      …
5.      public static synchronized DBHandler getInstance(Context context) {
6.          if (sInstance == null) {
7.              sInstance = new DBHandler(context.getApplicationContext());
8.          }
9.      return sInstance;
10.     }
11.
12.     private DBHandler(Context context) {
13.         super(context, DATABASE_NAME, null, DATABASE_VERSION);
14.     }
```

В конструкторе класса вызывается родительский метод, который создает базу данных с именем *DATABASE_NAME* и версией *DATABASE_VERSION*. Стоит обратить внимание на первый параметр – это контекст.

Контекст в Android приложении представляет текущее состояние объекта или приложения и доступен отовсюду. Это одна из самых важных частей в этой области программирования, поэтому необходимо чётко представлять, что это и как это использовать правильно, так как неправильное использование контекста может легко привести к утечкам памяти. Как правило, различают два основных типа.

Application context – это Singleton, который доступен из любой части приложения с помощью метода *getApplicationContext()*. Он привязан к жизненному циклу приложения и должен применяться только к тем объектам, которые используют паттерн программирования Singleton и должны быть доступны все время, пока приложение запущено.

Activity context – это объект, который привязан к жизненному циклу Activity, который представлен в виде отдельного экрана. То есть, если требуется создать какой-либо объект, который привязан к текущему Activity, то строго рекомендуется использовать именно этот тип контекста.

Таким образом, при работе с базой данных необходимо использовать именно Application context (Листинг 3.2.1 строка 7), так как она должна быть доступна в процессе работы всего приложения.

Далее необходимо определить поля таблицы (Листинг 3.2.2), которые будут инициализированы как *private final String*, что определяет их, как неизменяемые объекты и соответственно создать саму таблицу (Листинг 3.2.3) обычным запросом SQL.

Листинг 3.2.2 – Инициализация полей для таблицы Patient

```
1.  // Table name
2.  private static final String TABLE_PATIENT = "Patient",
3.
4.  // TABLE_PATIENT – column names
5.  private static final String KEY_PATIENT_ID = "patientID",
6.          KEY_BLOOD_TYPE = "bloodType",
7.          KEY_RH_FACTOR = "rhFactor",
8.          KEY_LOCATION = "location",
9.          KEY_JOB = "job",
10.         KEY_COMMENTS = "comments";
```

Листинг 3.2.3 – Создание таблицы Patient

```
@Override
public void onCreate(SQLiteDatabase db) {
    final String createTablePatient = "CREATE TABLE "+TABLE_PATIENT+ "("+
        KEY_PATIENT_ID + " INTEGER PRIMARY KEY, " +
        KEY_PERSON_ID + " INTEGER, " +
        KEY_DOCTOR_ID + " INTEGER, " +
        KEY_BLOOD_TYPE + " TEXT, " +
        KEY_RH_FACTOR + " TEXT, " +
        KEY_LOCATION + " TEXT, " +
        KEY_JOB + " TEXT, " +
        KEY_COMMENTS + " TEXT, " +
        "FOREIGN KEY (" + KEY_PERSON_ID + ") REFERENCES " +
        TABLE_PERSON + "(" + KEY_PERSON_ID + "), " +
        "FOREIGN KEY (" + KEY_DOCTOR_ID + ") REFERENCES " +
        TABLE_DOCTOR + "(" + KEY_DOCTOR_ID + "))";
    …
    db.execSQL(createTablePatient);
    …
}
```

Следующим шагом необходимо определить методы на основные действия с таблицей «Patient» – добавление нового пациента (Листинг 3.2.4), получение всех пациентов (Листинг 3.2.5) и получение его по идентификатору (Листинг 3.2.6), поиск по имени пациента (Листинг 3.2.7) и его удаление (Листинг 3.2.8).

Листинг 3.2.4 – Добавление новой записи в таблицу Patient

```
public Integer addPatient(Patient patient, Person person) {
    SQLiteDatabase db = this.getWritableDatabase();

    Integer personId = addPerson(person, db);

    ContentValues patientValues = new ContentValues();
    patientValues.put(KEY_PERSON_ID, personId);
    patientValues.put(KEY_DOCTOR_ID, -1);
    patientValues.put(KEY_BLOOD_TYPE, patient.getBloodType());
    patientValues.put(KEY_RH_FACTOR, patient.getRhFactor());
    patientValues.put(KEY_LOCATION, patient.getLocation());
    patientValues.put(KEY_JOB, patient.getJob());
    patientValues.put(KEY_COMMENTS, patient.getComments());

    Integer patientId = (int)db.insert(TABLE_PATIENT, null, patientValues);
    db.close();
    Log.d("Add new patient", patient.toString());

    return patientId;
}
```

При каждом обращении к базе данных необходимо получать его экземпляр с помощью методов *getWritableDatabase()* и *getReadableDatabase()*. Если

необходимо добавить или изменить какие-либо данные, то используется первый метод, а при чтении записей соответственно второй. Это используется для обеспечения целостности данных и защищает от случайных действий, которые в дальнейшем могут привести к ошибкам. В данном случае в базу данных добавляется новый пациент, поэтому используется *getWritableDatabase()* (Листинг 3.2.4 строка 2).

При каждом взаимодействии с базой данных можно использовать стандартные SQL запросы, но в данном контексте неудобно конкатенировать строки для больших запросов, поэтому можно использовать стандартные методы объекта *SQLiteDatabase* такие как *insert(), delete(), update()* и другие. Но ради справедливости стоит отметить, что при появлении сложных и длинных запросов придется писать его вручную (Листинг 3.2.7 строка 5), потому что не всегда получится составить его аналог с помощью существующих инструментов.

Как правило, методы объекта *SQLiteDatabase*, которые перечислены выше, имеют разную сигнатуру, поэтому разберем, как именно работает вставка новой записи (Листинг 3.2.4 строка 15). В качестве первого параметра передается название таблицы, в которую производится вставка. Второй параметр *nullColumnHack* обозначает, что в таблицу вставляется запись, у которой все поля *null* или *default*, но это случается настолько редко, что как правило, в качестве этого параметра передается просто *null.* Третьим параметром передается экземпляр класса *ContentValues*, который содержит все поля новой записи как ключ – значение. Метод *insert()* возвращает идентификатор новой созданной записи.

Для отслеживания процесса работы Android приложения используются логи, которые могут быть прочитаны в процессе отладки. В данном случае (Листинг 3.2.4 строка 17) лог сообщает о том, что был создан новый пациент и отображает его данные.

Листинг 3.2.5 – Получение всех записей из таблицы Patient

```
1.  public List<Patient> getAllPatients() {
2.      List<Patient> patientList = new ArrayList<>();
3.      SQLiteDatabase db = this.getWritableDatabase();
4.
5.      Cursor cursor = db.query(TABLE_PATIENT, new String[] {
6.              KEY_PATIENT_ID, KEY_PERSON_ID, KEY_DOCTOR_ID, KEY_BLOOD_TYPE,
7.              KEY_RH_FACTOR, KEY_LOCATION, KEY_JOB, KEY_COMMENTS},
8.              null, null, null, null, null, null);
9.
10.     if (cursor.moveToFirst()) {
11.         do {
12.             Patient patient = new Patient(
13.                     cursor.getInt(0),       // KEY_PATIENT_ID
14.                     cursor.getInt(1),       // KEY_PERSON_ID
15.                     cursor.getInt(2),       // KEY_DOCTOR_ID
16.                     cursor.getString(3),    // KEY_BLOOD_TYPE
17.                     cursor.getString(4),    // KEY_RH_FACTOR
18.                     cursor.getString(5),    // KEY_LOCATION
19.                     cursor.getString(6),    // KEY_JOB
20.                     cursor.getString(7));   // KEY_COMMENTS
21.             patientList.add(patient);
22.
23.         } while (cursor.moveToNext());
24.     }
25.
26.     cursor.close();
27.     db.close();
28.     return patientList;
29. }
```

Для отображения всех пациентов необходимо получить их из базы данных, для этого необходимо создать курсор (Листинг 3.2.5 строка 5). В нем указывается имя таблицы, из которой происходит выборка, все поля данной таблицы и последовательно параметры для фильтрации полученных записей, а именно:

1. selection
2. selectionArgs
3. groupBy
4. having
5. orderBy
6. limit

В данном случае требуется просто получить все записи, поэтому в качестве всех параметров передается *null.* Далее с помощью созданного курсора создаются

экземпляры класса *Patient* и кладутся в список (Листинг 3.2.5 строки 10-24). В конце выполнения всех действий с базой данных и курсором необходимо их закрывать с ними соединение (Листинг 3.2.5 строки 26-27), так как могут возникнуть утечки памяти.

Листинг 3.2.6 – Получение записи из таблицы Patient по идентификатору

```
1.  public Patient getPatient(int id) {
2.      SQLiteDatabase db = this.getReadableDatabase();
3.
4.      Cursor cursor = db.query(TABLE_PATIENT, new String[] {
5.              KEY_PATIENT_ID, KEY_PERSON_ID, KEY_DOCTOR_ID, KEY_BLOOD_TYPE,
6.              KEY_RH_FACTOR, KEY_LOCATION, KEY_JOB, KEY_COMMENTS},
7.              KEY_PATIENT_ID + "=?",
8.              new String[] { String.valueOf(id) }, null, null, null, null);
9.      if (cursor != null) {
10.         cursor.moveToFirst();
11.     }
12.
13.     Patient patient = new Patient(
14.             cursor.getInt(0),       // KEY_PATIENT_ID
15.             cursor.getInt(1),       // KEY_PERSON_ID
16.             cursor.getInt(2),       // KEY_DOCTOR_ID
17.             cursor.getString(3),    // KEY_BLOOD_TYPE
18.             cursor.getString(4),    // KEY_RH_FACTOR
19.             cursor.getString(5),    // KEY_LOCATION
20.             cursor.getString(6),    // KEY_JOB
21.             cursor.getString(7));   // KEY_COMMENTS
22.
23.     cursor.close();
24.     db.close();
25.     return patient;
26. }
```

Если при просмотре списка пациентов пользователь захочет узнать более подробную информацию о человеке, то при нажатии на него открывается новый экран с предоставленными данными. Для их получения снова создается курсор, но уже в параметре *selection* и *selectionArgs* передаются название поля, по которому должна производится выборка, и сам идентификатор соответственно. Далее все происходит аналогично предыдущему методу за исключением того, что возвращается один пациент, а не их список.

Листинг 3.2.7 – Осуществление выборки из таблицы Patient и Person

```
1.  public List<Patient> getPatientBySearch(String text) {
2.      List<Patient> patientList = new ArrayList<>();
3.      SQLiteDatabase db = this.getReadableDatabase();
```

```
4.
5.      String selectQuery = "SELECT " +
6.              TABLE_PERSON + "." + KEY_PERSON_ID + ", " +
7.              TABLE_PATIENT + "." + KEY_PATIENT_ID + ", " +
8.              TABLE_PERSON + "." + KEY_FULL_NAME + " FROM " +
9.              TABLE_PERSON + " INNER JOIN " + TABLE_PATIENT +
10.             " WHERE (" +
11.             TABLE_PERSON + "." + KEY_PERSON_ID + " = " +
12.             TABLE_PATIENT + "." + KEY_PERSON_ID + " and " +
13.             TABLE_PERSON +"."+ KEY_FULL_NAME + " LIKE \'" + text + "%\')";
14.     Cursor cursor = db.rawQuery(selectQuery, null);
15.     if (cursor.moveToFirst()) {
16.         do {
17.             patientList.add(getPatient(cursor.getInt(1)));
18.             Log.d("DB", cursor.getString(2));
19.         } while (cursor.moveToNext());
20.     }
21.     cursor.close();
22.     db.close();
23.     return patientList;
24. }
```

Одним из самых сложных запросов в данной базе данных получился поиск по имени пациента. Если перед пользователем представлен огромный список из более, чем ста записей, то найти определенного человека уже является трудной задачей. Для этого была создана кнопка поиска по имени человека (Рисунок 3.2.1). В процессе ввода букв в строку поиска, в базу данных посылается запрос, который отбирает только те записи, которые удовлетворяют этому поиску.

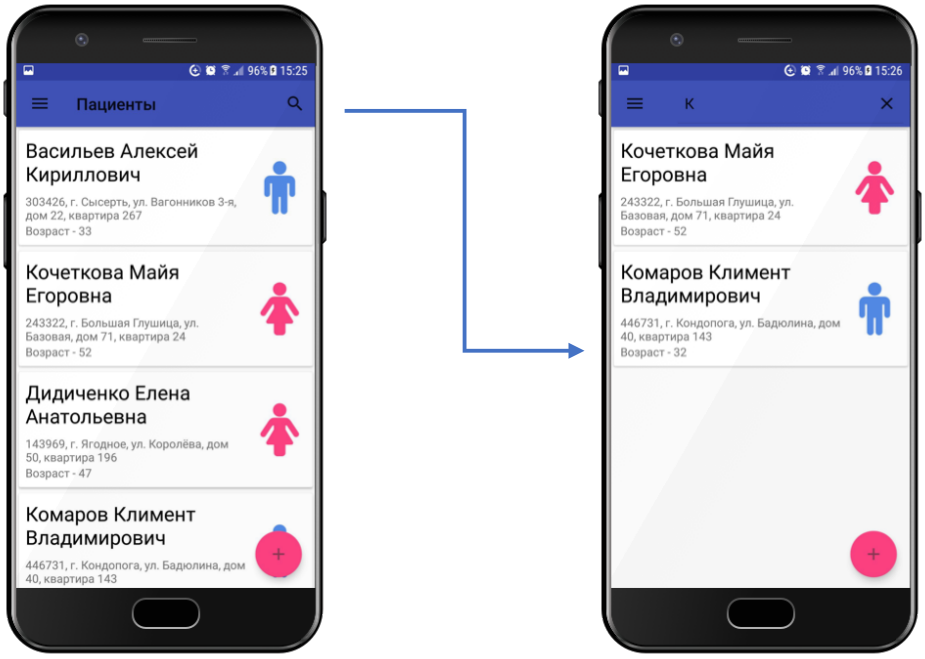

Рисунок 3.2.1 – Поиск по имени пациента

Нельзя выполнять запрос поиска только по таблице «Person», которая содержит имя человека, так как она является родительской к таблицам «Doctor» и «Patient» и, следовательно, в результат попадут все люди из базы данных. В данном случае нужно осуществлять поиск только по тем людям, у которых идентификатор *personId* из таблицы «Person» содержится в записях таблицы «Patient». Для этого идеально подходит запрос INNER JOIN (Рисунок 3.2.2), который отбирает записи из обоих таблиц с одинаковым идентификатором.

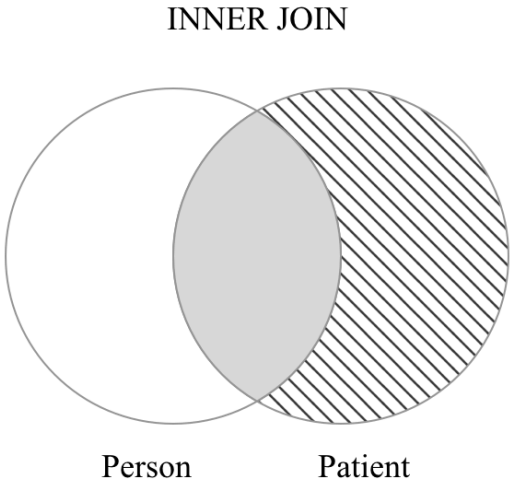

Рисунок 3.2.2 – Визуализация запроса INNER JOIN

Как видно на рисунке, оставшаяся часть круга «Patient» заштрихована наклонными линиями. База данных спроектирована таким образом, что все записи «Patient» имеют ссылку на родительскую таблицу «Person» и правильнее было бы отобразить выше, что один круг вложен в другой. Но для удобства восприятия, круг «Patient» расположен отдельно, а его заштрихованная часть говорит о том, что она пуста.

Соответственно поиск по имени осуществляется с помощью оператора LIKE (Листинг 3.2.7 строка 13), который выполняет отбор по первым символам. Аналогичная функция была реализована для таблиц «Doctor» и «Diagnosis».

Листинг 3.2.8 – Удаление пациента

```
1.  public void deletePatient(int patientId) {
2.      SQLiteDatabase db = this.getWritableDatabase();
3.
4.      int personId = getPatient(patientId).getPersonID();
5.
6.      this.getWritableDatabase().delete(TABLE_PATIENT, KEY_PATIENT_ID +
7.              " =?", new String[]{Integer.toString(patientId)});
8.      this.getWritableDatabase().delete(TABLE_PERSON, KEY_PERSON_ID +
9.              " =?", new String[] {Integer.toString(personId)});
10.     this.getWritableDatabase()
11.             .delete(TABLE_PATIENT_WITH_DIAGNOSIS, KEY_PATIENT_ID + " =?",
12.             new String[]{Integer.toString(patientId)});
13.
14.     db.close();
15. }
```

Если в процессе пользования приложения потребуется удалить какого-нибудь пациента, то при просмотре информации о нем, пользователь может нажать на иконку удаления (Рисунок 3.2.3). Для удаления информации о пациенте из всех таблиц базы данных было решено использовать каскадное удаление, но как было выяснено, в Android до сих пор это не очень хорошо работает или работает через раз. От чего во многих источниках опытные программисты советуют делать это вручную написанием запросов, которые напоминают собой тригеры. В данном случае удаляется вся информация из таблицы «Patient» (Листинг 3.2.8 строка 6), «Person» (Листинг 3.2.8 строка 8) и «Diagnosis» (Листинг 3.2.8 строка 10).

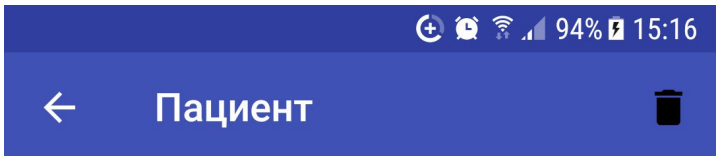

Рисунок 3.2.3 – Иконка для удаления текущего пациента

Как было написано в начале данного раздела, работа с другими таблицами проводится аналогично и разбирать подробно их нет смысла, однако всегда стоит внимательно проверять все выполняемые действия с базой данных, так как на этом построена работа данного приложения.

3.3. Пример создания нового пациента.

Разберем более подробно процесс создания нового пациента. При открытии приложения перед пользователем открывается сразу список текущих пациентов в больнице (Рисунок 3.3.1). Слева в строке меню расположена иконка «сендвич», получившая свое название из-за трех линий, расположенных друг на друге, при нажатии на которую открывается список названий других списков, такие как доктор, диагноз и лечение. С помощью данного приема пользователь может легко перемещаться между разными разделами одного приложения, что успешно используют многие крупные компании в своих приложениях.

На главном же экране располагается список всех пациентов, чья краткая информация находится в карточках, которые является основой Material Design в Android. Внизу присутствует кнопка для добавления нового пациента. Стоит отметить, что такой элемент называется Floating Action Button, которая так же является основным элементом дизайна, предложенным Google. Дословно эта фраза переводится как «парящая кнопка», что говорит о том, что она все время будет висеть сверху всех элементов списка. Такой прием очень точно объясняет принцип работы материальных поверхностей при проектировании элементов интерфейса приложения.

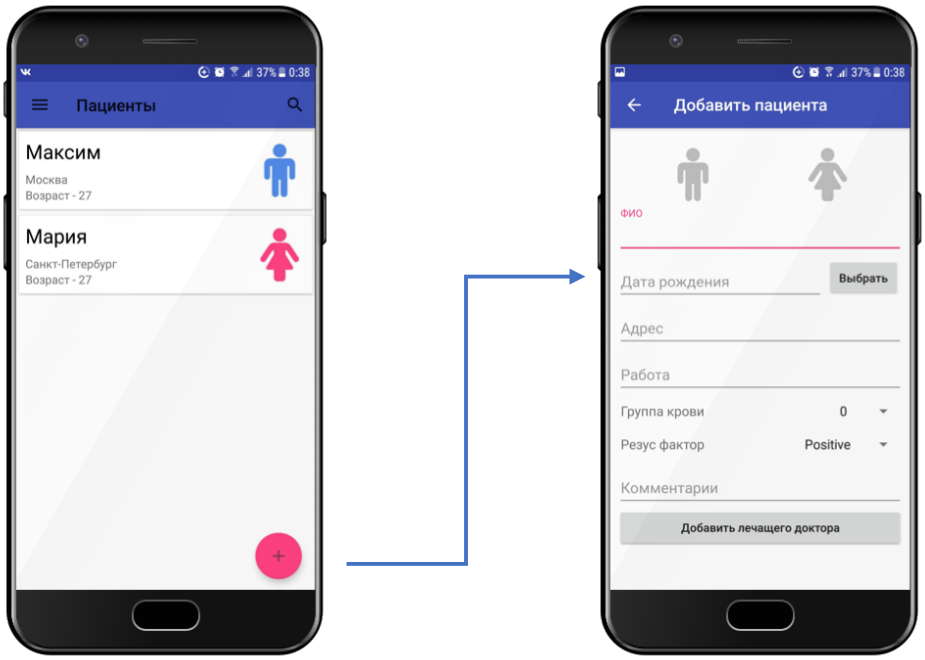

Рисунок 3.3.1 – Появление окна для добавления нового пациента

При нажатии на эту кнопку появляется новое окно, где пользователь может заполнить всю информацию о новом пациенте. Все поля в точности повторяют таблицы «Person» и «Patient». Наверху располагаются иконки для выбора пола нового пациента, которые при нажатии на мужчину и женщину загораются голубым и розовым соответственно.

Так как в первую очередь приложение создается для человека, то он должен видеть все действия и процессы, которые выполняет само приложение. Соответственно каждое взаимодействие с ним должно отображаться на экране телефона. Поэтому было выбрано, что вся фильтрация входящих данных будет происходить не на стороне базы данных, а на стороне самого интерфейса. Для этого при нажатии на кнопку добавления лечащего доктора происходит валидация всех введенных данных, и если какое-либо поле не заполнено, то под ним красным цветом выводится ошибка, что оно пустое. Стоит отметить, что пустым полем может быть только комментарий. Для ввода даты был использован стандартный элемент интерфейса – фрагмент *Date Picker*, который возвращает выбранную дату в заданном формате. Если пользователь захочет ввести вручную

дату, то он может допустить ошибку, для чего была создана еще одна проверка на корректность данных (Рисунок 3.3.2). Введенная дата проверяется с помощью заданного регулярного выражения.

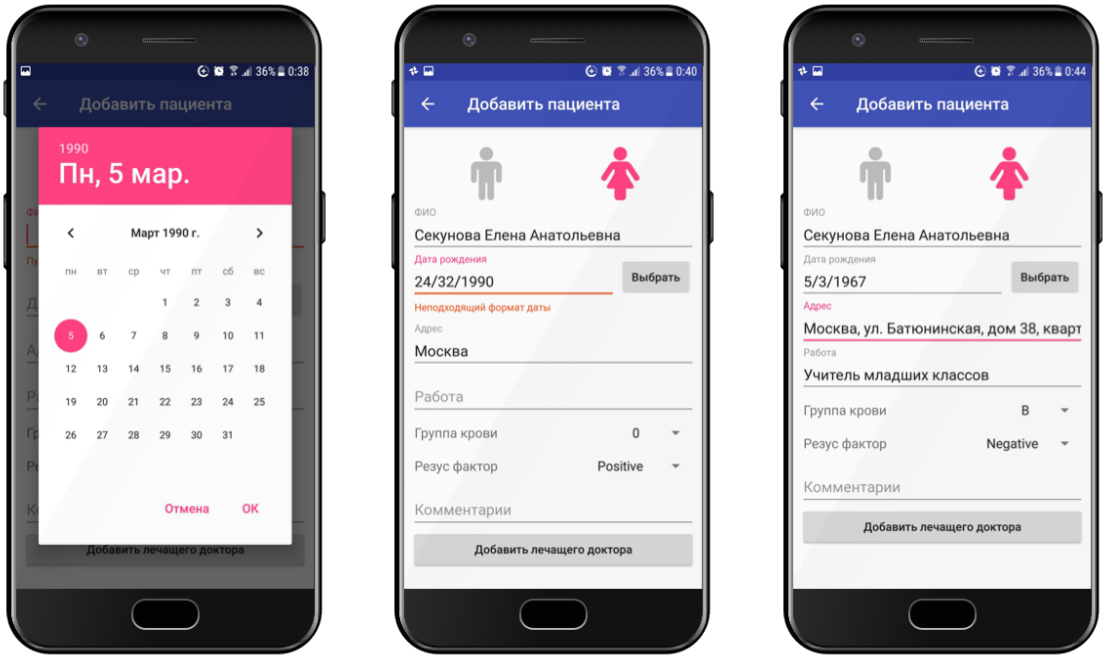

Рисунок 3.3.2 – Процесс создания нового пациента

Если пользователь правильно заполнил все поля данного экрана, то он может перейти к добавлению лечащего доктора (Рисунок 3.3.3), которым может быть только один человек. На этом экране отображаются все врачи, которые находятся в базе данных. Соответственно пользователь с помощью нажатия на карточку может выбрать любого человека, идентификатор которого сразу же отправится в запись только что созданного пациента.

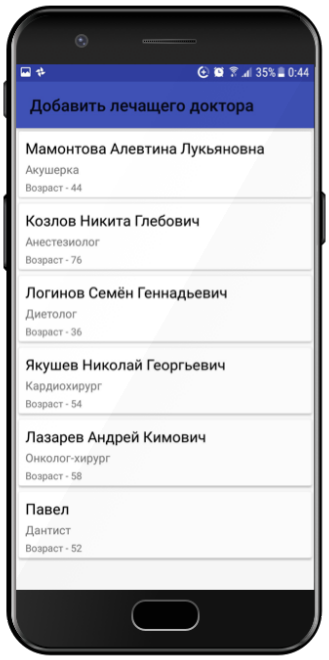

Рисунок 3.3.3 – Добавление лечащего доктора к пациенту

Далее необходимо выбрать диагноз, который необходимо поставить этому пациенту (Рисунок 3.3.4).

Рисунок 3.3.4 – Добавление диагноза к пациенту

На этом экране аналогично предыдущему отображаются все данные о диагнозах в базе данных. Можно выбрать от одного до нескольких диагнозов для одного пациента. Стоит обратить внимание на строку меню, а именно если выбран хотя бы один диагноз, то наверху появляется белая галочка для завершения добавления пациента. Для реализации данной функции был введен обычный счётчик нажатий на карточки, который отслеживает количество выбранных элементов.

После того, как пациент был успешно добавлен, он появляется в общем списке (Рисунок 3.3.5). При нажатии на него открывается отдельных экран, который позволяет узнать более подробную информацию о нём. В самом низу расположены карточки с лечащим доктором и диагнозами. Сначала может показаться, что они не активны и при нажатии на них ничего не произойдет, но это не так. Каждая из них открывает новый экран с более подробной информацией о выбранном элементе. В приложении аналогично организованы функции для добавления новых врачей и диагнозов, просмотр информации и остальное взаимодействие с ними.

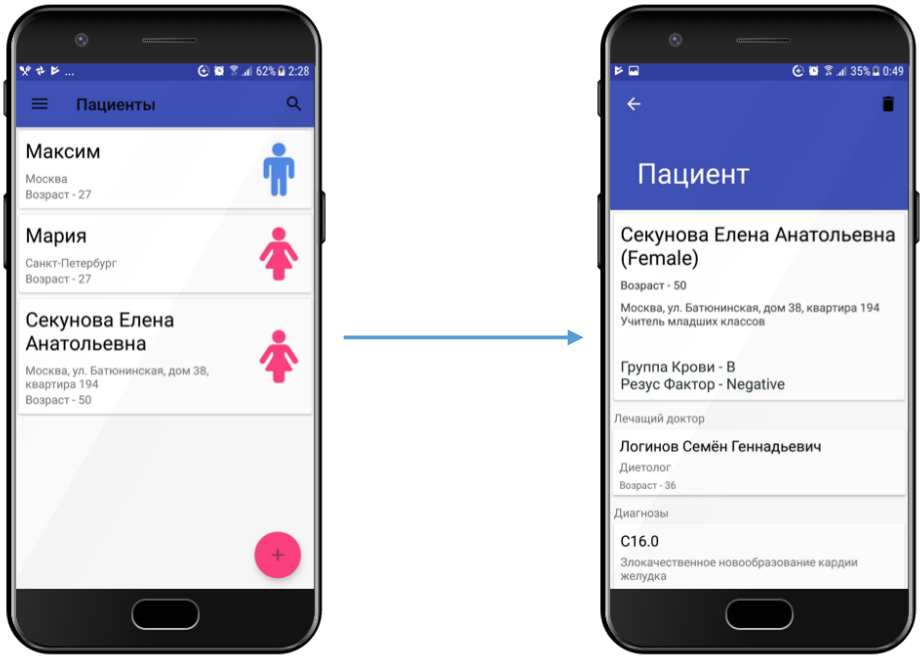

Рисунок 3.3.5 – Получение информации о пациенте

3.4. Заполнение данными базы данных.

При первом взаимодействии с приложением пользователь увидит только пустые списки и ничего более. Соответственно, придётся либо добавлять все новые данные вручную, что займет непозволительно много времени, а можно изначально заложить в приложение некоторые данные, чтобы можно было сразу приступить к работе. В этом пункте я подробнее рассмотрю именно второй вариант.

Всё начинается с класса *App*, который наследуется от класса *Application*. Он создается при открытии приложения до создания всех экранов, сервисов, фрагментов и других элементов. Данный класс поддерживает глобальное состояние всего приложения и имеет свой жизненный цикл, который активен все время, пока приложение видно на устройстве. Как правило, с помощью него инициализируется все данные, которые должны быть доступны в любой момент времени и из любой части приложения. Конечно можно это делать при открытии самого первого экрана, но это считается плохой практикой, так как Activity будет содержать дополнительный слой кода, который к нему никак не относится. Но если сразу при открытии приложения начинать записывать данные, то при дальнейших открытиях они будут дублироваться, а этого допустить нельзя. Для того, чтобы избежать такой ошибки, используется *PreferenceManager* (Листинг 3.4.1), который проверяет, открыто ли приложение первый раз, и если ответ положительный, то выполняются методы для заполнения данными базы данных.

Листинг 3.4.1 – Заполнение данными при первом открытии приложения

```
override fun onCreate() {
    super.onCreate()

    val prefs = PreferenceManager.getDefaultSharedPreferences(this)
    if (!prefs.getBoolean("firstTime", false)) {
        addDoctors()
        addDiagnoses()
        addTreatments()
        addSideEffects()

        val editor = prefs.edit()
        editor.putBoolean("firstTime", true)
```

```
13.         editor.apply()
14.     }
15. }
```

Разберём более подробно, как работает метод для добавления новых диагнозов. Так как язык реализация приложения – это Kotlin, то с помощью него можно значительно сократить количество вызываемых методов и общий объем кода.

Изначально в методе для добавления диагнозов создаётся экземпляр объекта базы данных (Листинг 3.4.2). После чего последовательно используется метод *addDiagnosis()* для добавления всех элементов в базу данных. В конце нужно вызвать метод *close(),* чтобы не возникло утечек памяти. Как можно заметить, в коде идет постоянное обращение к объекту *dbHandler,* что вызывает дополнительные затраты по производительности, поэтому этого надо постараться избежать.

Листинг 3.4.2 – Метод для добавления диагнозов

```
1.  private fun addDiagnoses() {
2.      val dbHandler = DBHandler.getInstance(this)
3.
4.      dbHandler.addDiagnosis(Diagnosis(
5.              "C16.0",
6.              "Злокачественное новообразование кардии желудка"))
7.
8.      dbHandler.addDiagnosis(Diagnosis(
9.              "C69.6",
10.             "Злокачественное новообразование глазницы"))
11.
12.     dbHandler.addDiagnosis(Diagnosis(
13.             "A15.2",
14.             "Туберкулез лёгких, подтвержденный гистологически"))
15.
16.     ...
17.     dbHandler.close()
18. }
```

Язык программирования Koltin позволяет избежать многочисленных обращений ко одному объекту с помощью функции *apply{ }* (Листинг 3.4.3). Для этого необходимо обратиться напрямую к классу *DBHandler* и применить к нему данный функционал. После чего внутри фигурных скобок можно обращаться к любому методу и полю вызванного объекта. Это значительно упрощает читаемость кода и увеличивает производительность.

Листинг 3.4.3 – Улучшенная версия метода для добавления диагнозов

```
1.  private fun addDiagnoses() {
2.      DBHandler.getInstance(this).apply {
3.          addDiagnosis(Diagnosis(
4.                  "C16.0",
5.                  "Злокачественное новообразование кардии желудка"))
6.
7.          addDiagnosis(Diagnosis(
8.                  "C69.6",
9.                  "Злокачественное новообразование глазницы"))
10.
11.         addDiagnosis(Diagnosis(
12.                 "A15.2",
13.                 "Туберкулез лёгких, подтвержденный гистологически"))
14.
15.         …
16.         close()
17.     }
18. }
```

Аналогично были реализованы остальные методы в классе *App* для добавления элементов в текущую базу данных.

4. Тестирование.

4.1 Использование библиотеки Stetho.

Сразу стоит отметить, что тестирование приложения происходило в течении всего процесса разработки. После того, как добавлялся какой-то новый функционал, он проверялся на реальном устройстве и дорабатывался в случае неверной работы. Но тут возникает другая потребность в просмотре и проверке всех данных из базы данных, которые могут быть доступны не только на экране устройства. Именно для таких целей была создана библиотека под названием Stetho.

Stetho – это инструмент, который позволяет отлаживать все значимые процессы, которые происходят в приложении с помощью Chrome Developer Tools. С помощью нее можно наблюдать за всеми взаимодействиями с сетью, напрямую работать с базой данных на устройстве, сопоставлять иерархию элементов определенного экрана и многое другое. Для этого необходимо именно в браузере Google Chrome перейти по адресу *chrome://inspect/#devices,* что откроет страницу, в которой будут указаны все устройства, которые подсоединены к компьютеру по USB. Отладка возможно только в том случае, если телефон подсоединен с помощью провода. Далее необходимо выбрать именно то приложение, которое хотим использовать (Рисунок 4.1.1). Оно будет обозначено с помощью имени пакета, которое у всех приложение должно быть уникальным и номера текущей версии.

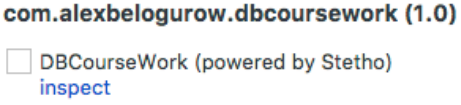

Рисунок 4.1.1 – Приложение в Chrome Dev Tools

При нажатии на надпись *inspect* откроется отдельно окно для отслеживания всех процессов, но нас интересует только работа с базой данных, которая

доступна по вкладке слева (Рисунок 4.1.2). Как видно на рисунке, открывается список всех таблиц базы данных.

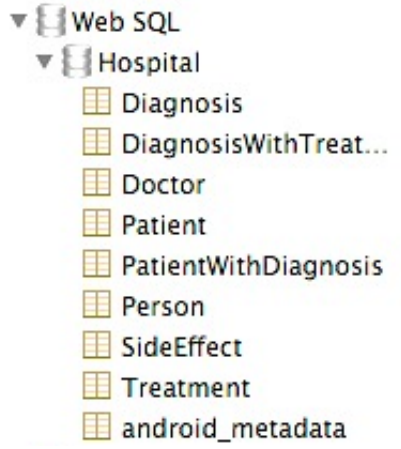

Рисунок 4.1.2 – Список таблиц базы данных

Для работы с определенной таблицей её необходимо выделить, и справа отобразится полная информация о ней (Рисунок 4.1.3), которая включает название всех полей и их значения. Такой способ просмотра информации значительно упрощает взаимодействие со всей базой данных.

rowid	ICD	diagnosisName
1	C16.0	Злокачественное новообразование кардии желудка
2	C69.6	Злокачественное новообразование глазницы
3	A15.2	Туберкулез лёгких, подтвержденный гистологически
4	A16.2	Туберкулез лёгких без упоминания о бактериологическом или гистологическом подтверждении
5	A17.8	Туберкулез нервной системы других локализаций
6	A83.6	Болезнь, вызванная вирусом Роцио
7	A84.1	Центральноевропейский клещевой энцефалит
8	B01.9	Ветряная оспа без осложнений
9	J20.0	Острых бронхит, вызванный Mycoplasma pneumoniae
10	Z01.0	Обследование глаз и зрения
11	Z03.4	Наблюдение при подозрении на инфаркт миокарда
12	Z04.6	Общее психиатрическое обследование по запросу учреждения
13	Z10.0	Профессиональное медицинское обследование
14	Z13.1	Специальное скрининговое обследование с целью выявления сахарного диабета

Рисунок 4.1.3 – Список записей в таблице «Diagnosis»

Кроме всего, Stetho представляет неограниченную возможность работы с помощью запросов. При нажатии на строку Hospital (Рисунок 4.1.2) открывается окно, которое представляет собой некое подобие терминала. Далее можно писать любые SQL запросы и работать с базой данных напрямую. Такая возможность позволяет сразу попробовать некоторые задумки с работой приложения прежде,

чем писать сам код, который сразу может не заработать. Это позволяет сэкономить много времени и сил.

ЗАКЛЮЧЕНИЕ

В ходе написания курсового проекта была спроектирована база данных больницы с помощью SQLite на платформе Android.

В рамках работы над проектом:

- была изучена возможность хранения данных в операционной системе Android и выбран SQLite, освоены приемы написания запросов и общей работы с ней;
- в качестве языка реализации было выбрано совмещение двух JVM языков программирования: Java и ранее неизвестный Kotlin, последний из которых показал себя как достойную замену для многих существующих языков;
- были изучены некоторые приёмы применения материального дизайна в Android, которые пользуются огромной популярностью и делают приложения визуально приятнее, а именно *CardView* и *FloattingActionButton;*
- была освоена работа с библиотекой Stetho, которая является одним из основных инструментов для отладки приложений на устройствах.

Кроме прочего стоит отметить, что SQLite хорошо себя показал для создания и манипулирования базой данных из 8 таблиц, которые связаны между собой различными отношениями. Это означает, что хоть она и не обладает всеми преимуществами полноценных баз данных, но вполне успешно может конкурировать с ними на для реализации на устройствах, которые не требует большой производительности.

Получившееся приложение можно без проблем использовать для работы с данными реальной больницы, так как оно успешно показало себя во всех тестах, имеет возможность расширения и редактирования, что необходимо для роста числа пользователей и устройств.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Харди Б., Филлипис Б., Стюарт К., Марсикано К. Android программирование для профессионалов: Питер. 2016. 640 с.
2. Официальная документация SDK Android: Available at: https://developer.android.com/index.html, accessed 15.10.2017.
3. Официальная документация библиотеки Stetho: Available at: http://facebook.github.io/stetho/, accessed 15.10.2017.
4. Официальная документация SQLite: Available at: https://www.sqlite.org/docs.html, accessed 15.10.2017.
5. Официальная документация языка программирования Kotlin: Available at: https://kotlinlang.org/docs/reference/, accessed 15.10.2017.
6. Международная классификация болезней МКБ-10. Электронная версия: Available at: http://www.mkb10.ru, accessed 15.10.2017.
7. К.Ю. Китанина, И.В. Рублевкая, Т.В. Честнова, В.А. Хромушин. Сборник медицинских документов (часть 1): Учебное пособие. Тула: ТулГу. 2013. 451 с.