ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки 09.04.02 «Информационные системы и технологии», профиль «Мобильные технологии»

КУРСОВОЙ ПРОЕКТ

Дисциплина: Мобильные операционные системы

Выполнил: студент группы 244-332

Курносова А.В.

Дата, подпись 06.01.25 _____

(Дата) (Подпись)

Проверила: к.т.н., ст. преп. Алпатова М.В._____

(Оценка)

Дата, подпись _____ _____

(Дата) (Подпись)

Замечания:

_____

_____

Москва

2024

Оглавление

Введение

В настоящий момент, большинство химических предприятий имеет большой объём задач. Задачи могут быть разделены между различными внутренними структурами предприятия, но это не исключает широкого спектра задач, выполняемых одной структурой.

Для наиболее эффективного распределения задач, на предприятиях существуют отдельные структуры, основной функций которых является анализ, структурирование и разделение обязанностей между другими структурными подразделениями. Вне зависимости от эффективности работы данных структур, вопрос сохранения строгой иерархии в разделении обязанностей остаётся актуальным, а спектр задач, выполняемых рабочими элементами – широким.

В работе рассматривается разработка приложения типа «планировщик» на начальном этапе. Данный этап разработки не включает в себя спецификацию приложения для химических производств и делает акцент на оптимизации приложения.

Корректная оптимизация приложения, даже на начальном этапе разработки, позволяет повысить эффективность его работы по различным параметрам.

Дальнейшая разработка приложения включает в себя разработку специфических элементов приложения, которые так же требуют дальнейшей оптимизации, на более зрелых этапах приложения.

В первой главе работы рассматривается аналитическая часть разработки приложения – предметная область, постановка цели и задачи работы.

Во второй главе рассматривается техническая разработка – разработка программного кода приложения, проведения анализа качества работы приложения, поиск путей оптимизации и возможных элементов оптимизации, проведение сравнительных характеристик полученных результатов до и после оптимизации приложения.

ГЛАВА 1. Аналитическая разработка

В первой главе рассматривается аналитическая разработка приложения – постановка цели и задач, актуализация поставленных задач, описание предметной области.

Цель

Разработка простого мобильного приложения и исследование его производительности. Определение параметров оптимизации приложения, проведение оптимизации и оценка влияния оптимизации на производительность.

Задачи

Общая разработка и оптимизация приложения может быть разбита на следующие задачи:

1. Разработка приложения «планировщик»;
2. Оценка производительности приложения;
3. Определение областей оптимизации приложения;
4. Проведение оптимизации приложения;
5. Оценка новых параметров производительности приложения;
6. Составление сравнительной характеристики работы приложения до и после оптимизации;
7. Составление отчётной документации по проведённой работе.

Предметная область

Приложение типа «планировщик» предполагает упрощение работы с задачами пользователя. Как правило, вносимая пользователем информация, является текстовой, но также, может вноситься и мультимедийный контент.

Основной задачей приложения является:

1. Создание списка дел;
2. Возможность градации информации по типу (начальная градация по типу «выполнено», «не выполнено» и дальнейшее разбиение информации опционально);

3. Возможность внесения изменения в список дел пользователя;
4. Возможность работы с календарём (добавление, удаление или изменение задач на определённый календарный день).

При добавление в приложение спецификации, функционал приложения расширяется. При более расширенном функционала, приложение имеет следующие функции:

1. Добавление, удаление или редактирование задач в списке;
2. Создание стабильных алгоритмов (под алгоритмами понимается постоянная, неизменная последовательность действий);
3. Создание списка задач на основании готовых алгоритмов;
4. Работа с календарём (добавление, удаление или редактирование задач на определённый календарный день);
5. Составление системы отчётных материалов (добавление на определённые слоты задач возможности прикрепления отчётных материалов – текстовых комментариев, фото или видео материалов);
6. Отправка отчётной документации (через функции синхронизации с другими устройствами и подключение к сети);
7. Работа с встроенными библиотеками (хранение текстовых документов, фото или видео материалов внутри приложения).

Более широкий функционал приложения подразумевает создание безопасной системы передачи и хранения информации и позволяет создать узконаправленный «мессенджер» на базе приложения «планировщика». Особенностями такого приложения является:

1. Создание безопасной внутренней сети хранения и передачи информации;
2. Создание системы алгоритмизации постоянных задач.

Под алгоритмизацией постоянных задач подразумевается переработка постоянных последовательностей действий в более компактные компоненты. Поскольку большинство постоянных последовательностей действий, как правило,

не нуждаются в ежедневном широком рассмотрении, их порядок может быть представлен в виде единого компонента – алгоритма.

(Например, в задачи пользователя ежедневно входит процесс заварки чая. Процесс может быть представлен в виде последовательности шагов: налить воду в чайник; включить чайник; дождаться, пока закипит вода в чайнике; взять чайный пакетик; положить чайный пакетик в кружку; налить горячую воду из чайника в кружку; подождать 3-4 минуты, пока чай не заварится. Данная последовательность действий может быть рассмотрена подробно на начальных этапах работы по данному алгоритму, но не требует ежедневного повторения. Наличие в списке дел каждого из этапов может быть излишним. Приложение позволяет сохранить всю последовательность действий в виде алгоритма, названного, например, «Заваривание чая». В текущий план на день, данное действие добавляется как алгоритм «заварить чай», но при этом, приложение позволяет раскрыть список шагов алгоритма и рассмотреть каждый из этапов выполнения данного действия более подробно, при необходимости).

Несмотря на возможное масштабирование приложения, добавление спецификации и изменение основных акцентов внутри приложения, его оптимизация требуется на всех стадиях разработки.

Оптимизация приложения может происходить различными путями и задействовать различные элементы самого приложения. Оптимизация может быть произведена по памяти или времени. При этом может быть затронута как программная часть приложения, так и мультимедийные компоненты.

Оптимизация приложения позволяет создать более эффективную версию того же приложения с тем же функционалом. Для проведения оптимизации могут быть задействованы различные ресурсы – как самостоятельный анализ компонентов приложения и поиск более эффективных путей решения реализованных задач, так и специальные программы, позволяющие произвести анализ компонентов приложения и выявить возможные элементы оптимизации.

ГЛАВА 2. Технологическая разработка

Во второй главе рассматривается поэтапная разработка приложения и его оптимизация. Происходит анализ и выбор технических путей реализации поставленных задач, рассмотрение полученных характеристик приложения. После разработки приложения, рассматривается поиск возможных путей оптимизации, реализация оптимизации и проведение сравнительных характеристик до и после оптимизации приложения.

Выбор технических инструментов

Разработка приложения происходит для мобильных устройств. Наиболее популярными языками для реализации приложений на Android являются Java и Kotlin. Java является более популярным языком разработки в сравнении с Kotlin, но, несмотря на это, Kotlin является конкурентноспособным языком программирования и набирает популярность среди разработчиков. Такая тенденция существует из-за особенностей языка программирования Kotlin. Он является более простым и совершенным по отношению к языку Java, имеет большее количество встроенных программных инструментов и позволяет существенно сократить объём программного кода.

Разработка самого приложения происходит на платформе Android Studio. Данная платформа позволяет использовать встроенные инструменты Android и имеет готовую библиотеку наиболее популярных элементов интерфейса. Данные элементы интерфейса позволяют реализовать на их базе любой функционал приложения, изменяя как функциональные значения элементов, так и их дизайн.

При разработке в Android Studio, необходимо первоначально определить версии Android, для которых будет актуально данное приложение. Приложение считается актуальным для выбранной версии Android и более поздних версий, относительно выбранной. В настоящий момент, наиболее актуальными считаются приложения, способные работать на версии Android5.0 и более поздних. Более ранние версии утратили свою поддержку Android и считаются устаревшими.

Исходя из этого, для разработки приложения были выбраны следующие параметры:

1. Язык программирования Kotlin;
2. Платформа Android Studio;
3. Версия Android5.0.

На базе выбранных инструментов происходит дальнейшая разработка приложения.

Для разработки дизайна приложения задействуются ресурсы, позволяющие создать макет приложения, разработать элементы дизайна с установленными расширениями экрана и реализовать функциональную модель приложения. Одним из таких инструментов является Figma.

Модель приложения

Перед началом разработки приложения, необходимо разработать его начальную модель. Начальная модель приложения позволяет определить необходимые окна приложения, структуру его интерфейса и общую логику приложения. Модель приложения не позволяет определить дизайн приложения или точное расположение окон, но позволяет отследить логику переходов между окнами и исключить тупиковые ветви приложения.

Для создания модели приложения задействован метод создания UX-моделей. Данная модель включает в себя примитивное обозначение элементов приложения, их примерное расположение (без точных обозначений и дизайна). На модели показаны переходы между окнами приложения и общая логика приложения (рис. 2.1).

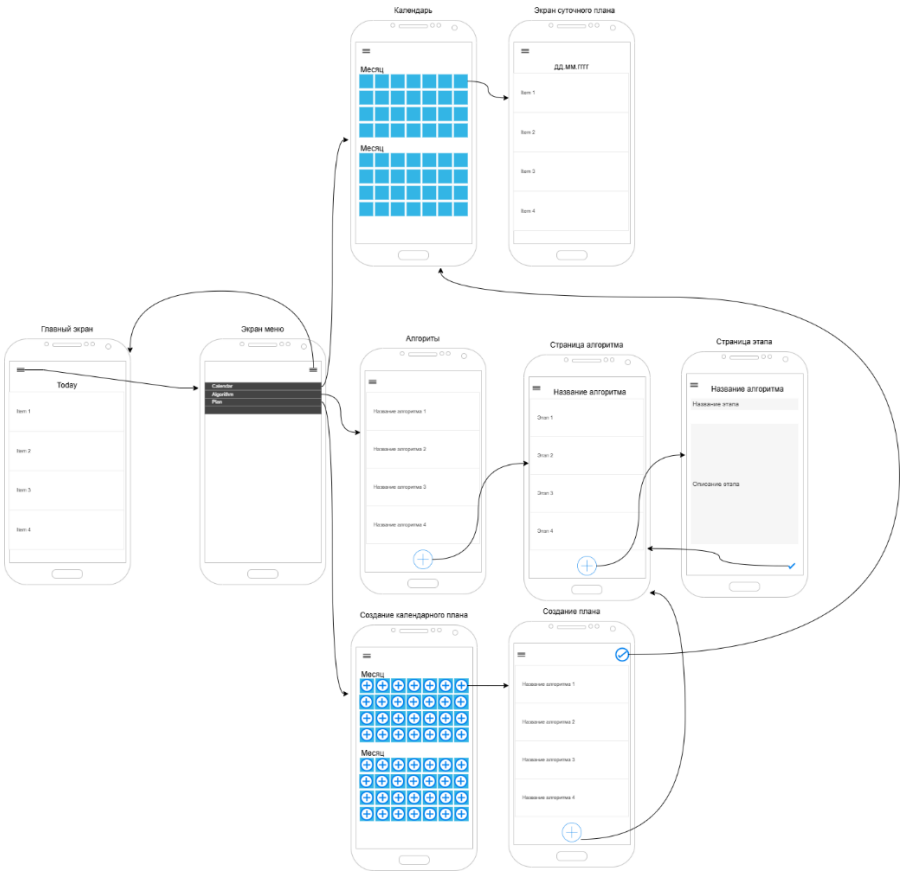


Рис 2.1. – UX-модель приложения.

Дизайн приложения

Для разработки дизайна приложения задействован инструмент Figma. При разработке созданы шаблоны каждого из элементов дизайна. При помощи шаблонов создаются все остальные окна приложения.

В приложении используются следующие элементы пользовательского интерфейса:

1. Текстовые фреймы для ввода пользовательской информации;
2. Текстовые фреймы для вывода пользовательской информации и создания списка задач;
3. Функциональные кнопки приложения;
4. Элементы «меню»;
5. Элементы «календаря»;
6. Фрейм фона приложения.

При создании дизайна приложения, каждый из элементов разработан в формате шаблона (рис. 2.2), после чего собраны окна приложения и функциональная модель приложения (рис. 2.3).

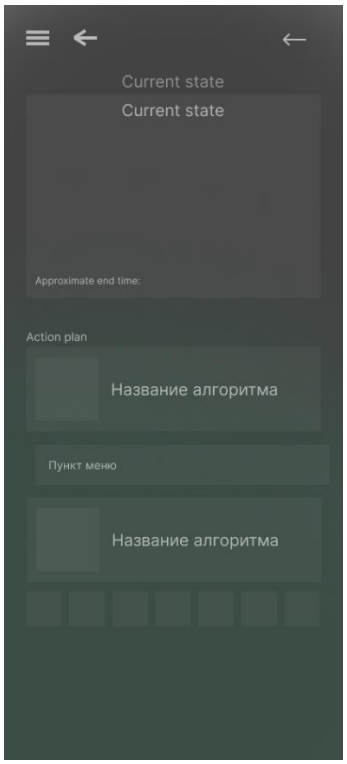


Рис. 2.2 – Шаблоны элементов приложения.

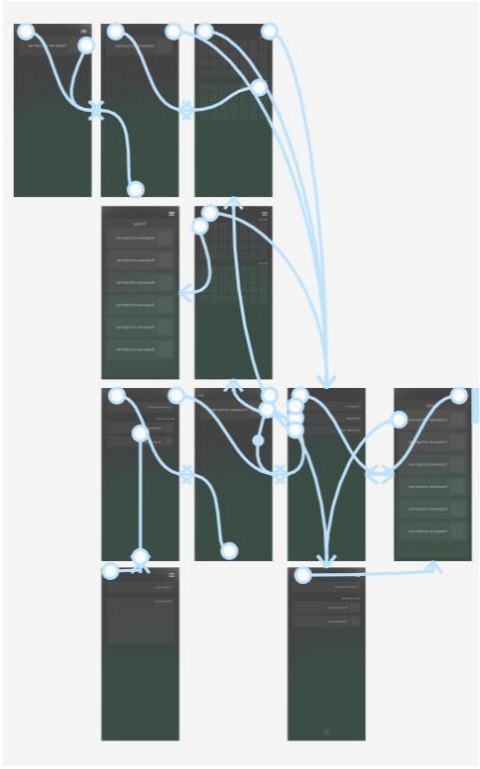


Рис. 2.3 – Функциональный прототип приложения.

При разработке дизайна приложения, важным аспектом является выбранное разрешение экрана. Поскольку приложения Android, вне зависимости от выбранной версии, могут быть использованы на различных устройствах с разным расширением экрана, необходимо учитывать изменяемый параметр каждого из элементов дизайна приложения. Для улучшения работы приложения на устройствах с различным разрешением экрана, используется создание двух шаблонов для двух интервалов разрешения экрана. Каждый из шаблонов используется в определённом интервале разрешения, а для всех промежуточных вариантов используется создание изменяемого параметра каждого элемента дизайна.

Программная реализация баз данных

Для разработки шаблона приложения, выбран встроенный шаблон NavigationDriwer. Данный шаблон имеет встроенную боковую шторку меню приложения и встроенную навигацию между окнами меню приложения. Данное решение позволяет упростить разработку приложения на начальном этапе и не пересоздавать механику работы меню с нуля. При этом, каждый элемент шаблона может быть настроен по дизайну, а функции элементов могут быть перенастроены или удалены (в случае отсутствия данных элементов в прототипе приложения).

Для создания элементов дизайна приложения, задействуется создание отдельных файлов дизайна. В данных файлах прописывается дизайн элементов, применяемых для различных окон приложения (например, дизайн фона). Такой метод позволяет не создавать дизайн каждого элемента с нуля для каждого окна, а создать дизайн единожды и в дальнейшем применять его к различным элементам.

Основным элементом при работе приложения является создание списков. Элемент списка в приложении позволяет пользователю работать с задачами и составлять примитивный список задач. Для создания самого элемента списка используется встроенный элемент списка в Android Studio. Пополнение списка происходит через отдельное текстовое поле, принимающее текстовую информацию и передающее его с писок. Элементы списка отображаются в пользовательском интерфейсе в последовательности внесения задач пользователем. Для хранения внесённых пользователем задач, используется база данных SQLite.

Для передачи данных в базу данных, используется следующий метод:

1. Текстовое поле для внесения новых записей в список, принимает значение, введённое пользователем;
2. Информация из текстового поля принимается приложением и переводится в формат String для дальнейшей работы. При этом, происходит очищение строки от лишних элементов (пробелов), при помощи метода trim();

3. Информация, принятая текстовым полем, передаётся в базу данных. При этом, происходит определение уникального ID записи и сохранение введённой пользователем информации;
4. Информация из базы данных передаётся в список. При этом, каждому уникальному ID (каждой уникальной записи), соответствует собственная строка списка в пользовательском интерфейсе.

Список является масштабируемым, а каждая новая запись в список добавляется под уникальным идентификатором. Этот метод позволяет создавать неограниченное количество записей в списке. С точки зрения интерфейса – список ограничивается в рамках окна приложения. При заполнении списка количеством строк, превышающей по объёму размеры экрана, элемент списка позволяет прокрутить список ниже, не включая при этом другие элементы окна приложения (в ограниченных элементом списка рамках). Этот метод позволяет реализовать список любой длины, при этом, сохраняя стабильный фон приложения и стабильное положение других элементов окна на экране.

Аналогичные методы используются для создания системы авторизации пользователя. Данные системы авторизации хранятся в базе данных пользователей. При этом, при регистрации или авторизации пользователя, происходит сравнение с имеющимися в базе данных значениями. Для приёма данных регистрации или авторизации, происходит обращение к текстовым полям, отвечающим за получение пользовательских данных (имя и пароль пользователя). Значения текстовых полей так же обрабатываются, принимаясь приложением как значения типа String и очищаются от излишних символов (листинг 2.1).

Листинг 2.1 – Создание базы данных и внесение полученных значений

```
class DBUsers (val context: Context, val factory: SQLiteDatabase.CursorFactory?)
:
    SQLiteOpenHelper(context, "LabLabUsers", factory, 1){
    override fun onCreate(p0: SQLiteDatabase?) {
        TODO("Not yet implemented")
        val query = "CREATE TABLE users (" +
                "id INTEGER PRIMARY KEY AUTOINCREMENT DEFAULT NULL, " +
                "name TEXT, " +
                "pass TEXT)"
        p0!!.execSQL(query)
    }
```

```
    override fun onUpgrade(p0: SQLiteDatabase?, p1: Int, p2: Int) {
        TODO("Not yet implemented")
        p0!!.execSQL("DROP TABLE IF EXISTS users");
        onCreate(p0);
    }

    fun addUser(user: User){
        val values = ContentValues()
        values.put("name", user.name)
        values.put("pass", user.pass)

        val db = this.writableDatabase
        db.insert("users", null, values)

        db.close()
    }

}
```

Для оценки качества работы приложения на этапах его разработки, используется встроенный эмулятор в Android Studio. Встроенный эмулятор позволяет оценить корректность работы приложения на устройствах с различными версиями Android, различными разрешениями экрана и при разных сопутствующих параметрах работы устройства. Запуск на встроенном эмуляторе, также позволяет оценить корректность работы всех элементов интерфейса приложения и через командную строку оценить корректность текущих процессов в приложении (рис. 2.4).

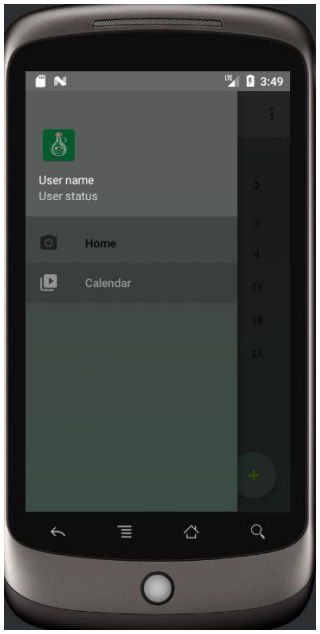


Рис. 2.4. – Запуск приложения на встроенном эмуляторе.

Запуск приложения на эмуляторе также позволяет отследить ошибки компиляции приложения. Некоторые из ошибок кода могут быть не заметны при самом написании кода и не иметь выделений. Запуск на эмуляторе позволяет оценить допущенные ошибки и найти их решение по коду ошибки и документацию.

Программная реализация календаря

Значимым инструментом в работе приложения является календарь. Работа с календарём может осуществляться различными методами: обращение к сторонним программам, предоставляющим доступ к календарю; обращение к встроенному календарю Android Studio; написание собственного календаря с использованием встроенных инструментов Java.

При решении задачи путём обращения к сторонним приложениям, предоставляющим доступ к календарю, возможно возникновение следующих ошибок:

1. Некорректная работа приложения, вследствие некорректной работы приложения-источника;
2. Отсутствие получения доступа к сторонним ресурсам со стороны пользователя;
3. Зависимость работы приложения от сторонних ресурсов (необходимость передачи данных от сторонних приложений, невозможность самостоятельной, обособленной работы приложения).

Для приложения, основой развития которого является безопасное подключение между устройствами, хранение данных и обособленная работа, последний недостаток в методе обращения к сторонним ресурсам является наиболее значимым. Несмотря на это, в ходе работы были рассмотрены методы обращения и получения данных от календаря Google. Данные методы были применены на практике и рассмотрены на примере работы приложения, но, в дальнейшем, выбор был осуществлён, в пользу самостоятельного получения данных календаря приложением.

Для получения данных календаря обособлено, внутри приложения, задействован метод CalendarView. Данный метод позволяет получать дату внутри приложения, а также отображать календарь в пользовательском интерфейсе. Для вывода календаря в пользовательском интерфейсе и обработки взаимодействий пользователя с календарём, был добавлен метод работы календаря (листинг 2.2).

Листинг 2.2. – Метод работы с календарём

```
<LinearLayout
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_marginTop="80dp">

    <CalendarView
        android:id="@+id/calendarView"
        android:layout_width="match_parent"
        android:layout_height="400dp"
        android:onClick="onClick"
        android:selectedWeekBackgroundColor="#ff0000"
        android:weekNumberColor="#0000ff"
        android:weekSeparatorLineColor="#00ff00" />

</LinearLayout>
```

Метод позволяет обрабатывать действия пользователя с календарём, принимать и выделять выбранные пользователем даты. Сам интерфейс помещается в разделе «меню» - «календарь». В данном разделе происходит формирование новых задач на определённую дату календаря. Для этого пользователю необходимо обратиться к дате календаря, перейти в раздел создания задачи и добавить задачу на выбранную дату.

Пользовательский интерфейс календаря расположен в документе активности, относящемуся к странице «меню» - «календарь». Документ активности обрабатывает события, происходящие на данной странице, при переходе из раздела «меню» (рис. 2.5).

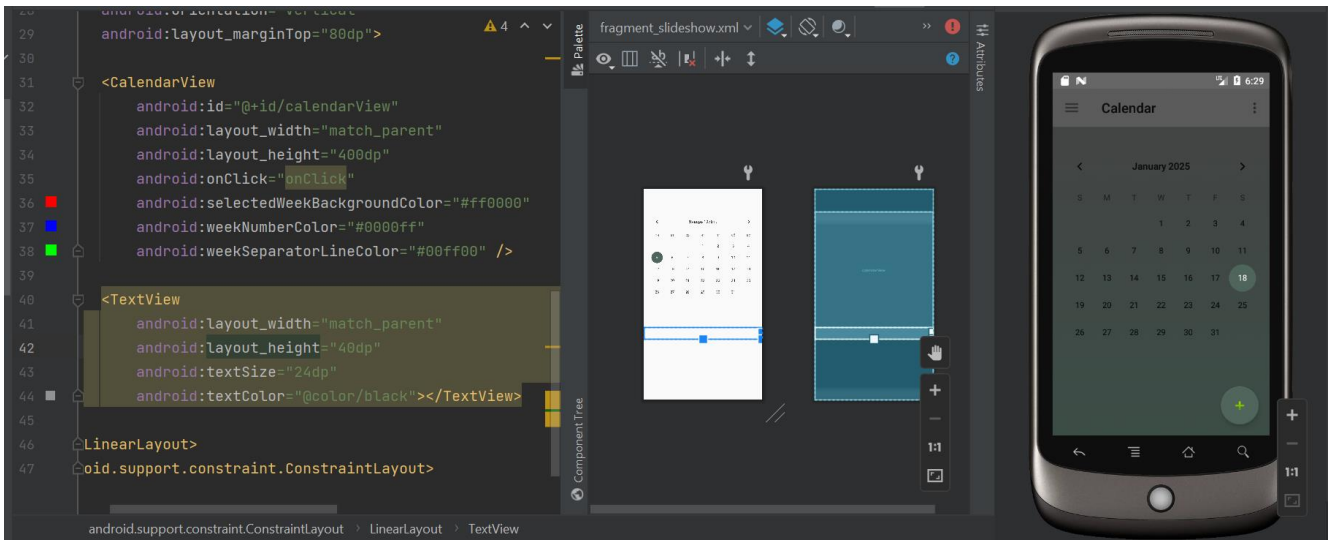


Рис. 2.5 – Разделение интерфейса на окнах приложения.

Тестирование и компиляция приложения

После разработки всех компонентов приложения, важным элементом проверки работоспособности, является оценка качества работы приложения на встроенном эмуляторе Android Studio. При работе на встроенном эмуляторе, приложение не должно иметь существенных ошибок компиляции или запуска приложения.

Несмотря на постоянную оценку качества работы приложения на различных этапах его разработки, при создании исполняемого файла, необходимо оценить не только ошибки, нарушающие работу приложения на устройстве, но и ошибки, потенциально влекущие некорректную работу при определённых состояниях устройства пользователя.

Удовлетворительным результатом в оценке качества работы считается отсутствие ошибок и рекомендаций при компиляции приложения. Данный параметр может быть проверен через терминал (рис. 2.6).

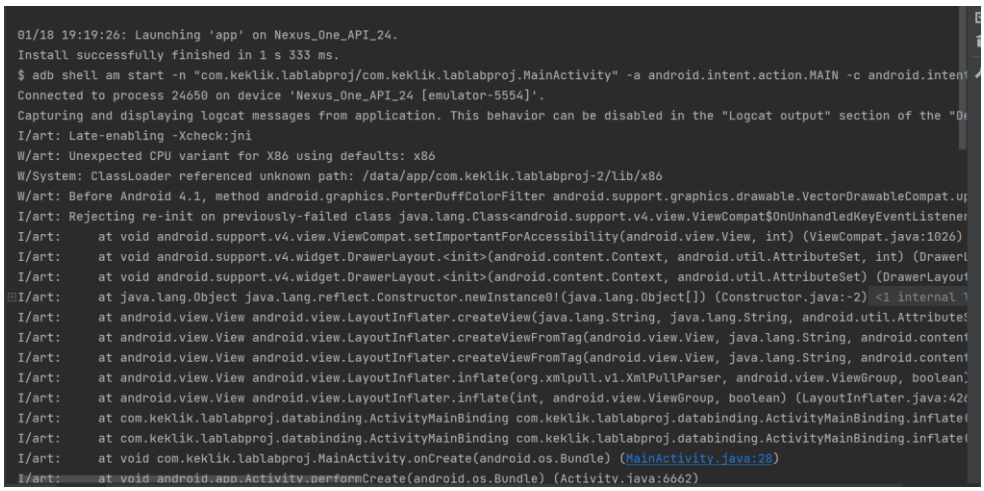


Рис. 2.6. – Оценка корректности компиляции через терминал.

После проверки на отсутствие ошибок при компиляции приложения, может быть произведена сборка приложения в исполняемый файл APK. Для этого задействуются встроенные компоненты Android Studio.

Сборка файла APK происходит при помощи билда приложения в APK формате через параметры билда приложения. Перед осуществлением сборки файла, необходимо указать настройки иконки приложения и добавить файл изображения (рис. 2.7). Для настройки иконки приложения, используется инструмент Image Asset. Данный инструмент позволяет настроить иконку приложения и установить её параметры. Для корректной работы иконки приложения, необходимо указать отображение иконки при всех вариантах работы приложения: main, debug и release.

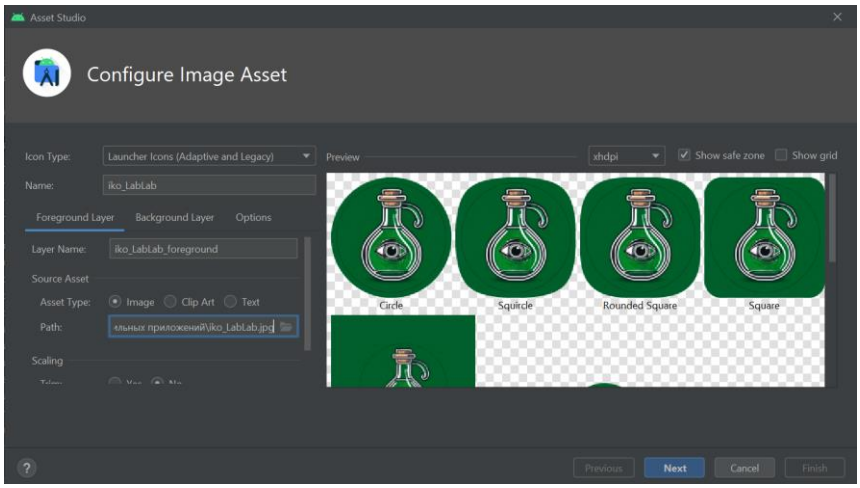


Рис.2.7. – Иконка приложения.

После создания иконки приложения, может быть собран файл APK. При создании файла, создаётся ключ приложения. Данный ключ позволяет использовать приложение на различных платформах и сохранять при этом права разработчика. Сам исполняемый файл сохраняется в отдельной папке.

Оптимизация приложения по потреблению ОЗУ

Оптимизация приложения может быть произведена по различным параметрам. Так, оптимизация приложения может привести к более эффективной работе по времени (задачи в приложении могут выполняться быстрее) или по памяти (приложение затрачивает меньше ресурсов на устройстве-носителе).

Наиболее частая ошибка работы приложения, связанная с некорректной (или отсутствующей) оптимизацией приложения – OutOfMemoryError. Данная ошибка может возникать при продолжительной работе приложения и связана с переполнением стека памяти. Данная ошибка, как правило, приводит к принудительному завершению работы приложения. Для выявления возможный проблем и оптимизации приложения произведено:

1. Оценка потребления оперативной памяти приложением;
2. Проведение оптимизации приложения по потреблению оперативной памяти;
3. Оценка полученного результата потребления оперативной памяти после проведения оптимизации приложения.

Для оценки потребления оперативной памяти приложением, используется инструмент Android Studio Profiler. Этот инструмент предназначен для мониторинга производительности приложения. Запуск инструмента производится через панель инструментов Android Studio. При запуске открывается окно параметров, которые позволяют оценить затраты процессора, ОЗУ и энергии с динамикой по времени (рис. 2.8).

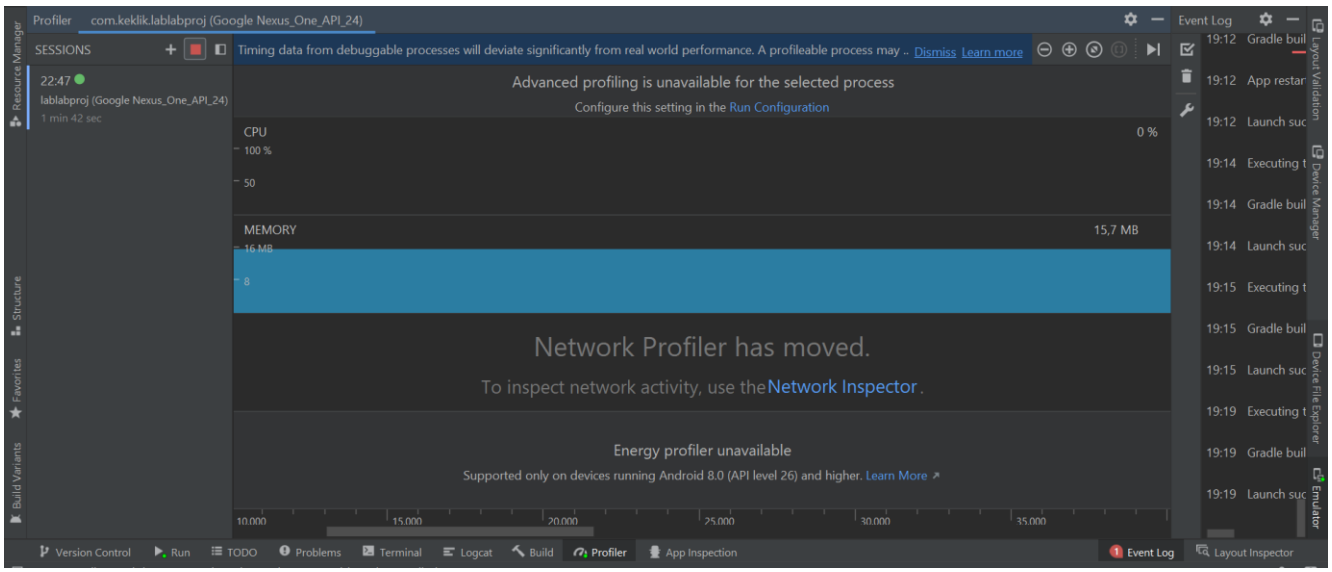


Рис. 2.8. – Затраты приложения по процессору, ОЗУ и энергии с динамикой по времени.

Поскольку оптимизация приложения происходит по параметру памяти, наиболее актуальным показателем является «Memory». При раскрытии данного параметра, возможно рассмотреть детализированный график, в котором содержится информация по потреблению памяти приложением (рис. 2.9).

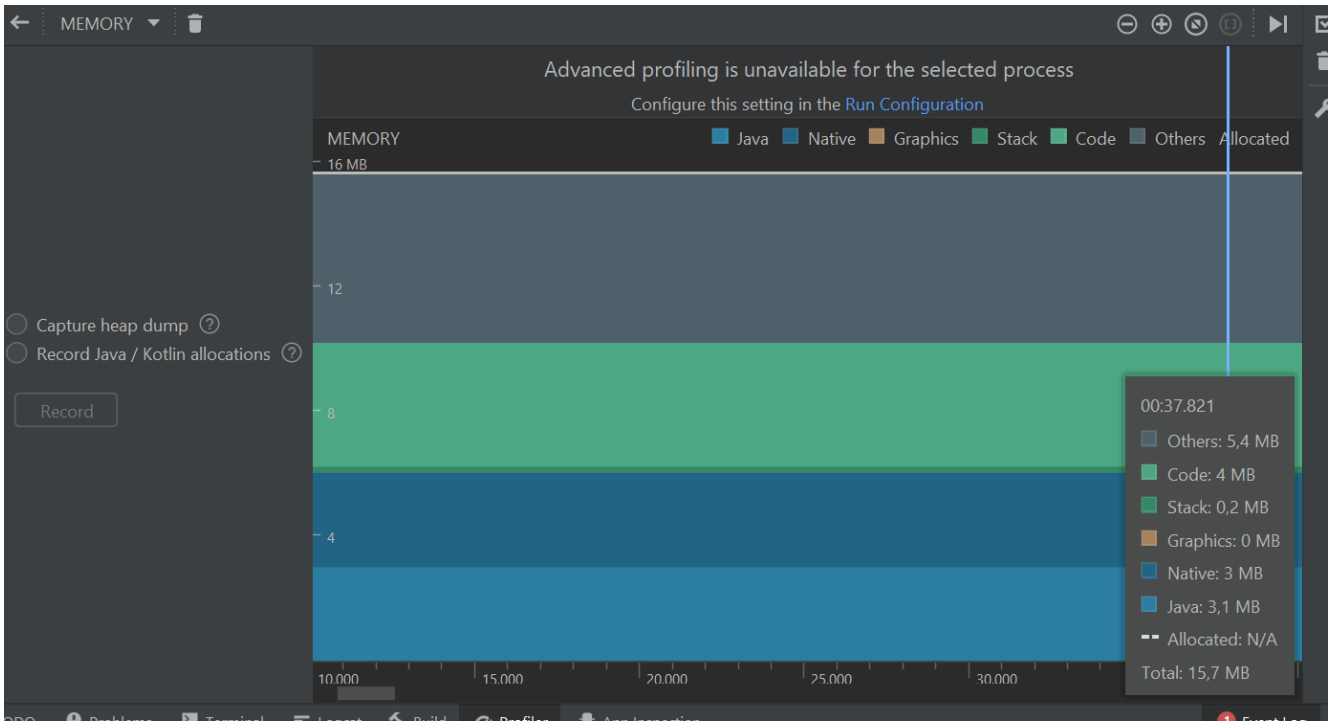


Рис. 2.9. – График потребления памяти приложением.

Для проведения оптимизации приложения, необходимо рассмотреть и дать определение каждому параметру на представленном графике.

Total: Общее количество памяти, которую использует приложение. Параметр включает в себя все остальные параметры:

- **Java**: Память, используемая объектами Java или Kotlin;
- **Native**: Память, используемая объектами, созданными на языках программирования C или C++. Этот раздел связан с библиотеками или фреймворками, которые используются в приложении;
- **Graphics**: Память, используемая графическими ресурсами (например, текстуры);
- **Stack**: Память, выделенная для стека вызовов приложения. Каждый поток имеет свой собственный стек, который содержит информацию о текущем состоянии выполнения;
- **Code**: Память, используемая исполняемым кодом приложения. Это включает в себя байт-код Java, нативный код и другие ресурсы;
- **Other**: Память, используемая различными другими ресурсами, которые не попадают ни в одну из перечисленных категорий.
- **Allocated**: Общее количество памяти, которое было выделено приложением. Это может отличаться от общего использования из-за того, что часть памяти может быть освобождена сборщиком мусора.

После получения данных о памяти приложения, был произведён анализ затрат памяти приложением при его работе. Для этого, Внутри приложения были добавлены новые записи, относящиеся к «спискам дел» (внесены записи в базу данных), произведена работа с календарём и совершены переходы между различными экранами приложения.

В ходе анализа затрат памяти приложением, было выявлено:

1. Добавление новых записей в базу данных влияет на затраты памяти, но данные затраты являются естественными, поскольку добавляется новая, хранимая в приложении информация;

2. При удалении записи из списка дел, затрачиваемая память уменьшается, поскольку уменьшается объём хранимой в приложении информации;
3. При переходе между окнами приложения через раздел меню, память, затрачиваемая приложением, увеличивается. После увеличения объёма памяти, снижение затрачиваемого объёма не происходит (рис. 2.10).

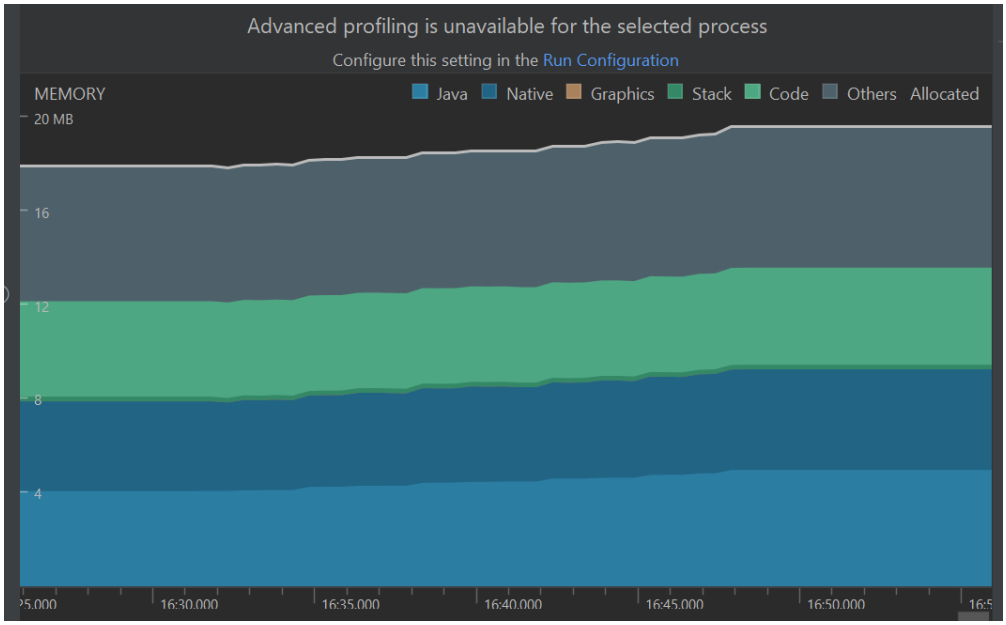


Рис. 2.10. – График затраты памяти при многократном переходе между окнами меню приложения.

Последний выявленный пункт оказался наиболее интересным для рассмотрения.

Поскольку переходы между окнами меню происходят постоянно, а затраты памяти при этом не снижаются до исходного состояния, многократный переход между окнами меню приложения, может приводить к остановке работы приложения. Данное событие может происходить в перспективе, если пользователь будет совершать многочисленные переходы между окнами (что не является естественным использованием приложения). Несмотря на это, многократная остановка работы приложения может приводить к некорректным процессам при работе приложения.

Для решения данной проблемы, был рассмотрен принцип роста затрачиваемой памяти при переходах между окнами. Для этого, было совершено несколько переходов между окнами и сравнение затрат памяти по пунктам до и

после совершения переходов. Результат сравнения представлен на графиках (рис. 2.11 и рис. 2.12).

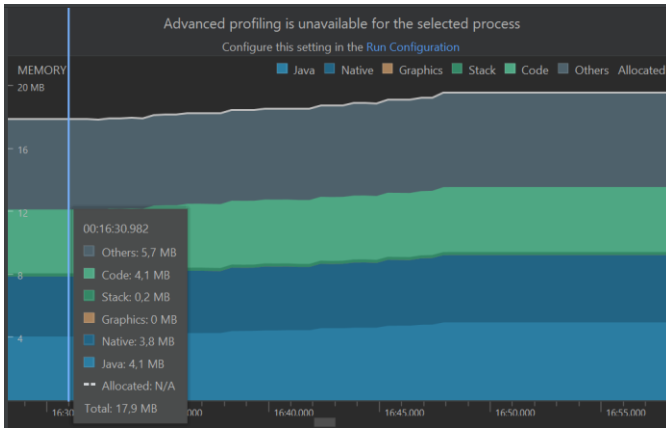


Рис. 2.11 – Показатели памяти до нескольких переходов между окнами

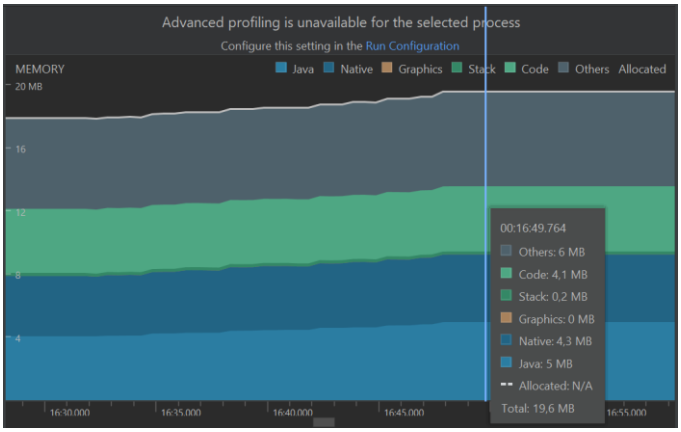


Рис. 2.12. – Показатели памяти после нескольких переходов между окнами

Из графиков видно, что общая затрачиваемая память растёт из-за роста двух показателей – native и java. Иными словами, проблема может быть связана с работой файлов Java или Kotlin, или с файлами на C и C++ (фреймворками).

Большие затраты памяти происходят по параметру Java. Чуть меньшие затраты памяти происходят па параметру Native. Это значит, что наибольшие проблемы работы приложения связаны со стеками хранения объектов (Java) и чуть менее значимые проблемы связаны с фреймворками (Native).

Java

Для начала оптимизации приложения, необходимо решить вопрос с параметром, вызывающем больший рост затрачиваемой памяти – Java.

Для решения данной проблемы, необходимо более подробно рассмотреть механику перехода между окнами приложения.

При переходе между окнами, происходит отображение нового окна следующим методом (листинг 2.3).

```
public void onClick(View view) {
    Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
            .setAction("Action", null).show();
```

Данный метод включает выбранное окно для пользователя, используя при этом метод show(). Данный метод делает видимым окно для пользователя, при этом, старое окно, если оно было открытым, остаётся работать в фоновом режиме. Данный метод приводит к существованию множества окон приложения в стеке объектов, которые не задействуются ни приложением, ни пользователем.

Для решения данной проблемы, может быть изменён метод обработки нажатий на пункты меню и открытий окон приложения, или добавлен метод очистки стека.

Для более корректного перехода между окнами, применён метод, который позволяет заменять выбранное окно на новое (листинг 2.4).

Листинг 2.4. – Метод замены окон.

```
FragmentManager fragmentManager = getSupportFragmentManager();
```

Данный метод позволяет не отображать новое окно поверх старого, а заменять отображаемое окно на новое. В таком случае, старое окно прекращает свою работу и не находится в стеке объектов.

При заполнении стека объектов на 500Мб – 700Мб, происходит переполнение стека объектов приложения и принудительное завершение работы. При масштабировании приложения, переполнение стека может происходить значительно быстрее, чем на данном этапе, что приведёт к принудительному завершению работы приложения уже после нескольких переходов между окнами.

Изменение метода перехода между окнами приложения, приводит к стабильному существованию ограниченного количества окон приложения. В таком случае, стек не перегружается, а переходы между окнами приложения не приводят к постоянному возрастанию памяти.

Native

Вторым аспектом возрастающей памяти приложения является параметр Native. Данный параметр связан с работой фреймворков. В качестве основы системы сборки приложений, в Android Studio используется Grable. Данный плагин

может работать не совершенно, что приводит к возрастанию затрачиваемой приложением памяти.

Для решения данной проблемы, могут быть рассмотрены инструменты оптимизации приложений – ProGuard, D8, R8. Для определения того, как именно данные инструменты могут помочь в оптимизации приложения, рассмотрены принципы сборки приложения в Android Studio (рис. 2.13).

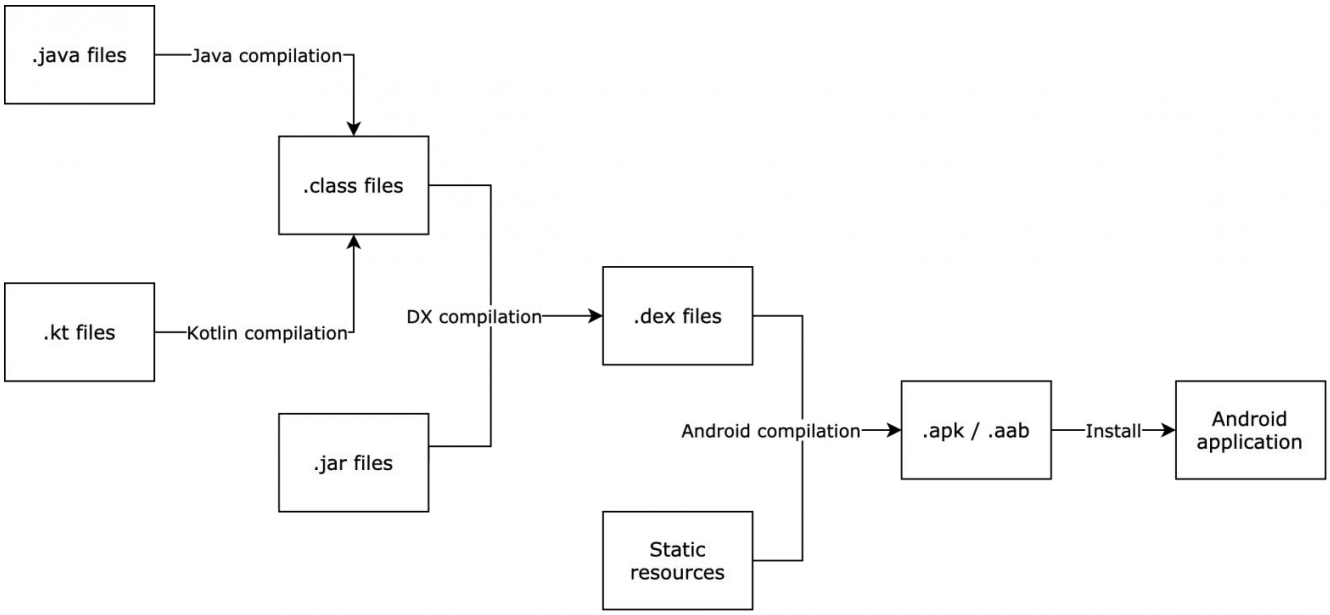


Рис. 2.13. – Порядок сборки приложения Android Studio.

Для оптимизации приложения по параметру Native, порядок сборки может быть изменён. Поскольку порядок сборки определяется Android Studio, без вмешательства в данный процесс со стороны разработчика, для оптимизации могут быть применены инструменты, которые влияют на процесс сборки приложения, но при этом не являются элементами программного кода приложения, написанным разработчиком. Наиболее популярным инструментом для проведения оптимизации приложения является ProGuard. Данный инструмент позволяет позволяет повлиять на процесс сборки приложения перед этапом dex компиляции. В таком случае dex компилятор принимает на вход файлы .class, а на выходе – файлы .dex. Данное вмешательство не нарушает общей структуры сборки приложения, но при этом происходит оптимизация приложения по Native (рис. 2.14).

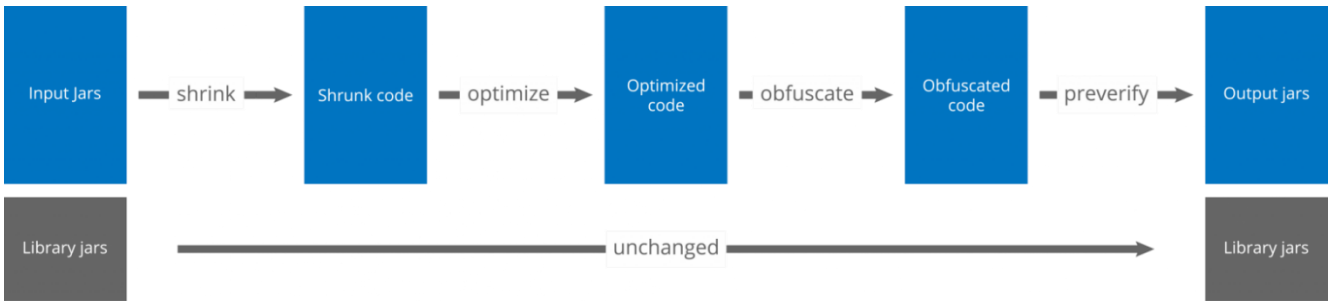


Рис. 2.14. – Схема оптимизации приложения ProGuard.

Для применения инструмента необходимо воспользоваться его файлами. ProGuard — это open-source утилита, которая работает с java-кодом. Директории проекта позволяют рассмотреть более подробно каждую из возможных функций инструмента (рис. 2.15).

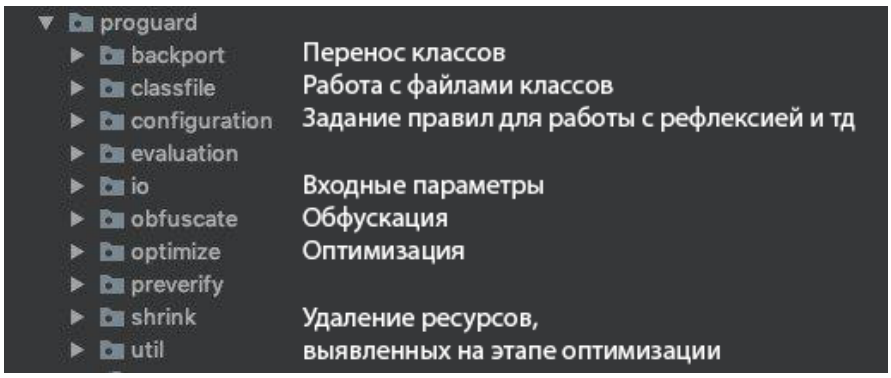


Рис 2.15 – Директория ProGuard.

Поскольку приложение необходимо оптимизировать, наиболее актуальным для рассмотрения является раздел Optimize.

Оптимизация приложения запускается из класса ProGuard (листинг 2.5).

Листинг 2.5. – Класс оптимизации ProGuard.

```
/**
     * Performs the optimization step.
     */
    private boolean optimize(int currentPass,
                             int maxPasses) throws IOException
    {
        if (configuration.verbose)
        {
            System.out.println("Optimizing (pass " + currentPass + "/" + maxPasses +
")...");
        }

        // Perform the actual optimization.
        return new Optimizer(configuration).execute(programClassPool,
                                                    libraryClassPool,
                                                    injectedClassNameMap);
    }
```

Поскольку во время оптимизации удаляются незадействованные элементы кода, для более корректного процесса оптимизации (и точного сохранения необходимых элементов кода), возможно установить определённые правила проведения оптимизации через класс оптимизации ProGuard. Но даже в случае отсутствия прописанных правил, не исключено корректно выполнение оптимизации утилитой.

После проведения оптимизации приложения, также проведены повторные тестирования затрат оперативной памяти приложения при его работе. При смене окон приложения, затрачиваемая память незначительно повышается, после чего приходит к первоначальному значению. Общая затрачиваемая память приложения снижена (рис. 2.16).

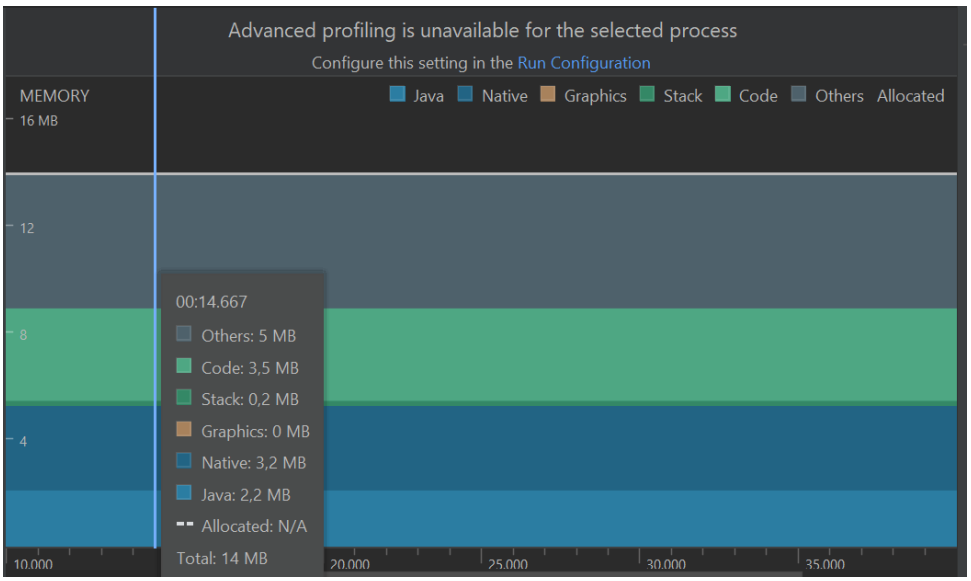


Рис. 2.16. – Конечные показатели приложения по затратам оперативной памяти.

Исходя из полученных данных, Общая затрачиваемая память приложения равняется 14Мб, снижены затраты памяти по показателям Java и Native. Данный результат получен засчёт оптимизации приложения по хранению объектов в стеках памяти и изменению системы работы с фреймворками.

Для наглядной оценки полученных результатов, составлена сравнительная таблица (таблица 2.1).

Таблица 2.1. – Затраты памяти до и после оптимизации

	До оптимизации		После оптимизации	
Параметр	Статика, MB	Работа, MB	Статика, MB	Работа, MB
Total	17,9	19,6	14,1	14,8
Others	5,7	6	5	5,3
Code	4,1	4,1	3,5	3,5
Stack	0,2	0,2	0,2	0,2
Graphics	0	0	0	0
Native	3,8	4,3	3,2	3,4
Java	4,1	5	2,2	2,4

Также, на основании таблицы, построен график. График отражает скорость роста затрат памяти приложением до и после оптимизации приложения в статике и в работе (график 2.1).

График 2.1. – Общие затраты памяти.

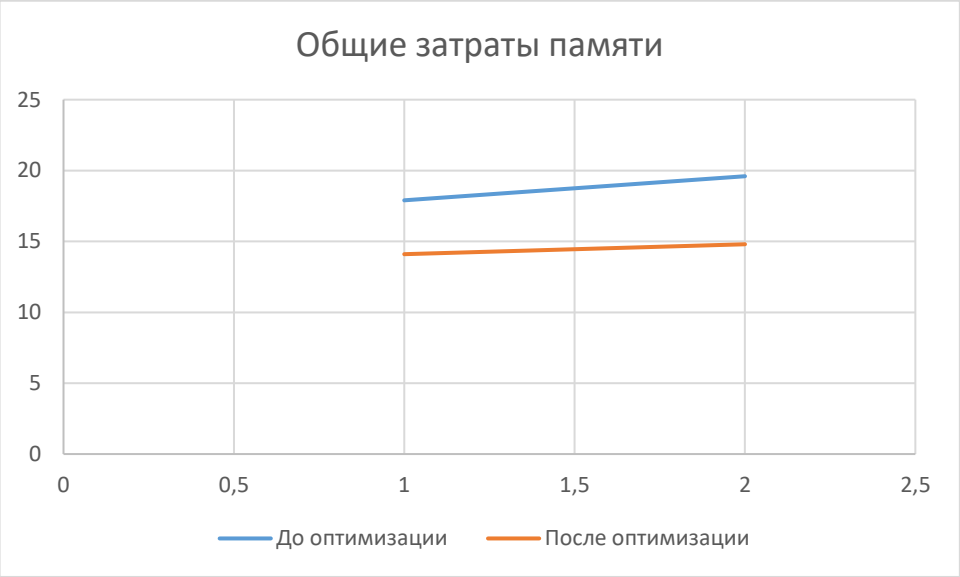
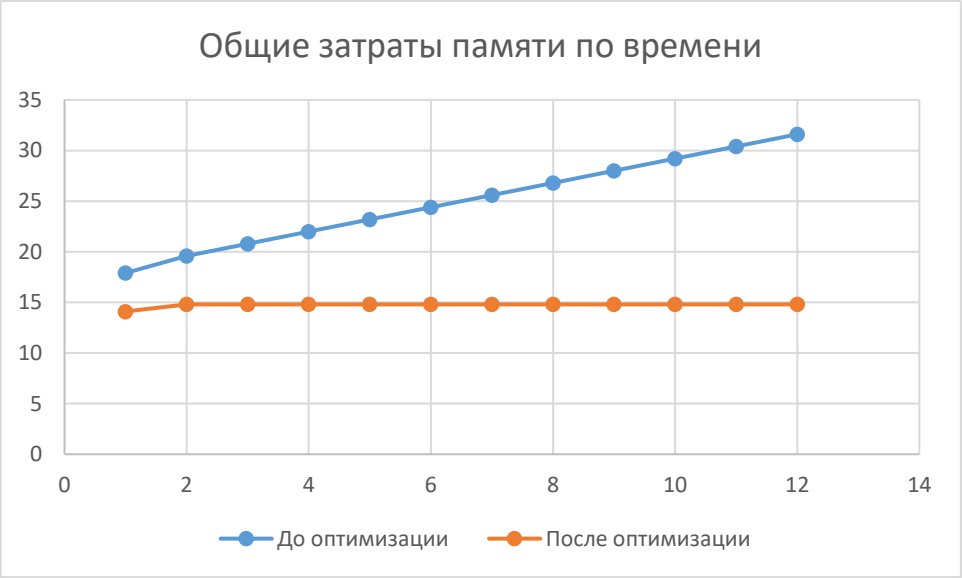


График построен на основании общих затрат памяти приложением. На графике видно, что помимо общего более высокого потребления памяти приложением до его оптимизации, так же происходит и более быстрый рост затрачиваемой приложением оперативной памяти. После проведения оптимизации приложения, снижены как общие затраты памяти приложения, так и скорость роста затрачиваемой памяти.

Кроме этого, анализ затрат оперативной памяти Android Studio Profiler выявил, что после оптимизации приложения, общие затраты памяти не поднимаются выше, чем в конечной точке построенного графика, в то время как до

оптимизации приложения, затрачиваемая память приложения росла постоянно, с каждым новым обращением к интерфейсу приложения. Поэтому, в перспективе по времени, график выглядит следующим образом (график 2.2).

График 2.2 – Общие затраты памяти по времени.



Первая точка графика – состояние приложения в покое. Вторая и последующая точки – приложение в работе. Линия графика оптимизированного приложения находится ниже линии графика неоптимизированного – это отображает общие более низкие затраты оперативной памяти. Так же, график оптимизированного приложения более пологий, что говорит о более высокой эффективности приложения по памяти в перспективе времени пользования приложением.

Заключение

В ходе работы выполнены:

1. Разработка приложения;
2. Оценка производительности приложения;
3. Анализ полученных характеристик производительности и анализ путей оптимизации;
4. Реализация оптимизации приложения;
5. Оценка новых характеристик производительности приложения;
6. Сравнительный анализ полученных характеристик до и после оптимизации приложения.

Для оптимизации приложения, была проведена оптимизация по затратам оперативной памяти приложением в статическом и рабочем состоянии. Оптимизация по памяти проведена на основании анализа полученных данных о затратах памяти приложением и выявления неэффективной работы параметров, в перспективе влекущей принудительную остановку приложения.

Оптимизация по памяти позволила:

1. Снизить общий объём потребляемой приложением оперативной памяти;
2. Снизить расширение приложения по затратам оперативной памяти в процессе работы.

Для оценки полученных результатов и сравнения характеристик приложения до и после оптимизации, составлены таблицы и графики. Полученные графики отражают эффективность проведённой оптимизации приложения и уменьшение затрат оперативной памяти.

Список литературы

1. Обращение к календарям Google и другим сторонним приложениям календаря [Электронный ресурс] https://habr.com/ru/articles/664876/
2. Оптимизация приложения на Android при помощи стандартных инструментов [Электронный ресурс] https://habr.com/ru/articles/768636/
3. Анализ производительности React Native приложений [Электронный ресурс] https://habr.com/ru/companies/kuper/articles/803755/
4. Оптимизация сборок Android приложений: ProGuard, D8, R8. [Электронный ресурс] https://habr.com/ru/articles/533578/
5. Принцип работы ProGuard [Электронный ресурс] https://habr.com/ru/articles/436564/
6. Обфускация, как метод защиты программного кода [Электронный ресурс] https://habr.com/ru/articles/533954/
7. Назначение и применение обфускации программного кода [Электронный ресурс] https://habr.com/ru/articles/735812/