Санкт–Петербургский государственный университет

ШИМАНСКАЯ Ольга Олеговна

Выпускная квалификационная работа

Разработка редактора для работы с текстами на естественном языке

Уровень образования: бакалавриат
Направление 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа СВ.5005.2018 «Прикладная математика, фундаментальная информатика и программирование»
Профиль «Современное программирование»

Научный руководитель:
Старший преподаватель СПбГУ, к.ф.-м.н.
Шалымов Дмитрий Сергеевич

Консультант:
Программист ООО «ИнтеллиДжей Лабс»
Бахвалов Павел Ярославович

Рецензент:
Программист ООО «Интеллиджей Лабс»
Каргин Григорий Игоревич

Санкт-Петербург
2022 г.

Содержание

Введение

С момента появления персональных компьютеров пользователи работают с текстовыми файлами: создают, сохраняют и редактируют их. Для работы с текстовыми файлами были созданы текстовые редакторы — программы, предоставляющие возможность работы с текстом в интерактивном режиме.

С развитием технологий и появлением языков разметки функциональность текстовых редакторов расширялась для упрощения работы с новыми форматами файлов.

Существуют редакторы не только для работы с естественными языками, но и с языками программирования. Для языков программирования разрабатываются так называемые редакторы программ. Редакторы программ обычно входят в состав интегрированных сред разработки — программных средств, используемых программистами для разработки программного обеспечения. Редакторы программ являются более продвинутыми в сравнении с редакторами для текстов на естественном языке. Языки программирования обладают формальным набором синтаксических, лексических и семантических правил, по коду можно построить дерево абстрактного синтаксиса, и поэтому становится возможным выполнение следующих задач:

- проверка наличия ошибок до компиляции программ
- разбиение на синтаксические конструкции для создания удобной подсветки кода
- автодополнение названия функции или другой конструкции языка на основе первых введенных символов

Открытия в области обработки естественных языков сегодня позволяют создавать модели машинного обучения, способные выполнять различные действия над естественными текстами [1]: разбивать на предложения, выделять термины или морфемы, проверять орфографию и пунктуацию, производить перевод на другой язык, генерировать новый текст и так далее. Стало возможным работать с естественными текстами схожим с языками программирования образом. Однако существующие текстовые редакторы не предоставляют

возможность простого внедрения таких моделей, потому что они проектировались без учета возможности сложной обработки текста.

Создание редактора, способного внедрять сервисы, обрабатывающие естественный текст, позволит людям без опыта в программировании получить редактор с нужной им проверкой текста. Копирайтеры, журналисты, редакторы, писатели и другие люди, профессия которых связана с написанием текстов, получат возможность ускорить и автоматизировать свою работу, так как у них будет доступ к результатам анализа текста в самом редакторе.

Стоит также отметить, что многие современные редакторы являются кроссплатформенными и позволяют пользователям редактировать файлы с разных устройств: телефонов, планшетов, браузеров и т.д. Поэтому возможность работы редактора на разных платформах станет отдельным преимуществом.

Постановка задачи

Целью данной работы является разработка кроссплатформенного текстового редактора, который позволяет встраивать сервисы для анализа текста и поддерживает декорирование текста.

Таким образом, были поставлены следующие задачи:

1. Провести обзор и анализ существующих аналогов.
2. Разработать архитектуру собственного мультиплатформенного редактора и пользовательский интерфейс.
3. Реализовать редактор с использованием выбранных технологий.
4. Создать интерфейс и руководство для встраивания стороннего сервиса, анализирующего текст.

1. Обзор предметной области

В данной главе приведен обзор текстовых редакторов и процессоров, частично решающих поставленные задачи. Вместе с этим приводится обзор библиотек и технологий, используемых в реализации.

1.1. Текстовый редактор

Текстовый редактор — обобщенное наименование программ, предназначенных для создания, редактирования, вывода на экран и печать, а также сохранения в виде файлов различного рода текстовых документов и данных [2].

Текстовые редакторы можно разделить на 2 основные категории:

1. **Простые текстовые редакторы**

   Это редакторы, которые работают исключительно с текстовыми данными. Эти данные представляются в виде символов, без каких-либо графических элементов (изображений, текстовых стилей, и т.д.). Редакторы сохраняют файл в виде простого текста (англ. plain text).

   Примером такого текстового редактора является Блокнот (англ. Notepad) [3], являющийся частью операционной системы Windows.

2. **Текстовые процессоры**

   Отличие этих редакторов от простых в том, что они дополнительно позволяют форматировать текст, добавлять к нему различные стили (подсветка, шрифт, размер текста), визуальные элементы (картинки, таблицы, графики и так далее) и прочее.

   Многие из этих редакторов обладают свойством **WYSIWYG** ("What You See Is What You Get "Что видишь, то и получишь"), согласно которому редактируемое содержание текстового документа совпадает с конечным отображением или выглядит максимально близко к нему. CKEditor [4] — пример WSYIWYG-редактора.

Одна из наиболее широко представленных разновидностей текстовых редакторов — **редактор программ**. Редактор программ предназначен для создания и редактирования текста на каком-либо языке программирования. Обычно такой редактор встроен в Интегрированную Среду Разработки (ИСР).

Редакторы программ чаще всего обладают функциональностью, упрощающей написание кода:

- подсветка синтаксиса
- отображение подсказок (например, автодополнение)
- подсветка ошибок

Примером редактора программ является редактор, встроенный в IntelliJ IDEA [5].

1.2. Аналоги

Существует большое число редакторов, которые предоставляют возможность асинхронной проверки текста, автодополнения и декорирования текста.

Рассмотрим некоторые из самых широко используемых редакторов:

- Google Docs [6]
- Notepad++ [7]
- редактор в составе Visual Studio Code [8]
- редактор в составе Intellij IDEA [5]

Первые два из них является редакторами текста, а последние два являются редакторами программ.

Редакторы текста

Google Docs Редактор, разрабатываемый компанией Google, доступен для браузеров (на рисунке 1 представлена браузерная версия) и мобильных устройств.

Имеет встроенную проверку грамматики и подсказку-автодополнение. Однако, этот продукт имеет закрытый исходный код, и не дает возможность подключить сторонний сервис для проверки грамматики.

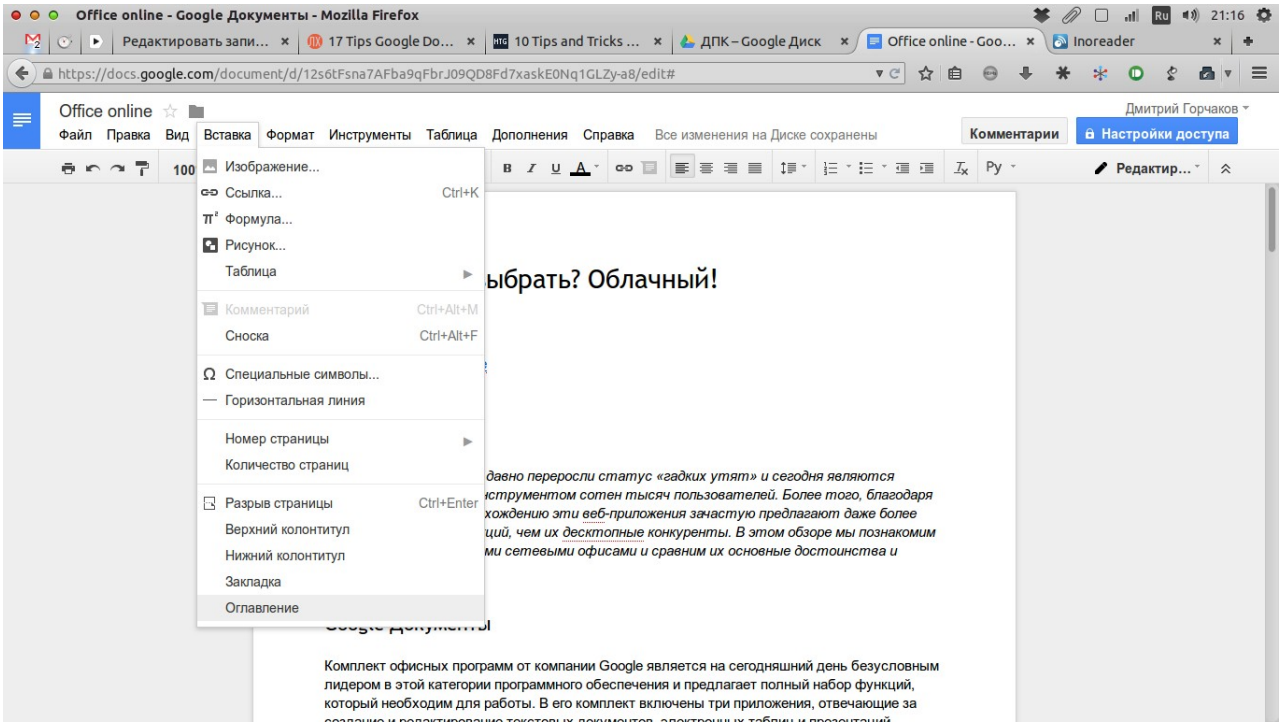

Рис. 1: Текстовый редактор Google Docs в браузере.

Notepad++ Редактор с открытым исходным кодом, доступен для Windows и ReactOS. Имеет встроенное автодополнение. Функциональность может расширяться за счет создания и подключения плагинов — отдельно компилируемых программных модулей в дополнение к исходной программе. Например, проверка орфографии становится доступной при подключении плагина DSpellCheck [9].

К недостаткам данного редактора можно отнести следующее:

- Доступен на ограниченном числе платформ
- Отсутствие подробной документации для написания собственного плагина (для поиска ответов и подробностей API разработчики используют форум [10]).

- Для проверки грамматики в плагине нужно самостоятельно реализовать:

    1. Запрос повторного анализа текста.
    2. Функциональность для отображения подсказок/вариантов замены (сюда входит отслеживание позиции курсора в тексте).
    3. Обновление подчеркиваний ошибок (как отрисовка, так и удаление).

  Таким образом, несмотря на то, что редактор имеет возможность встраивания проверки текста, у него отсутствует специализированное API для такой функциональности, поэтому необходимо реализовывать большое количество сторонней логики.

  На рисунке 2 показан внешний вид редактора.

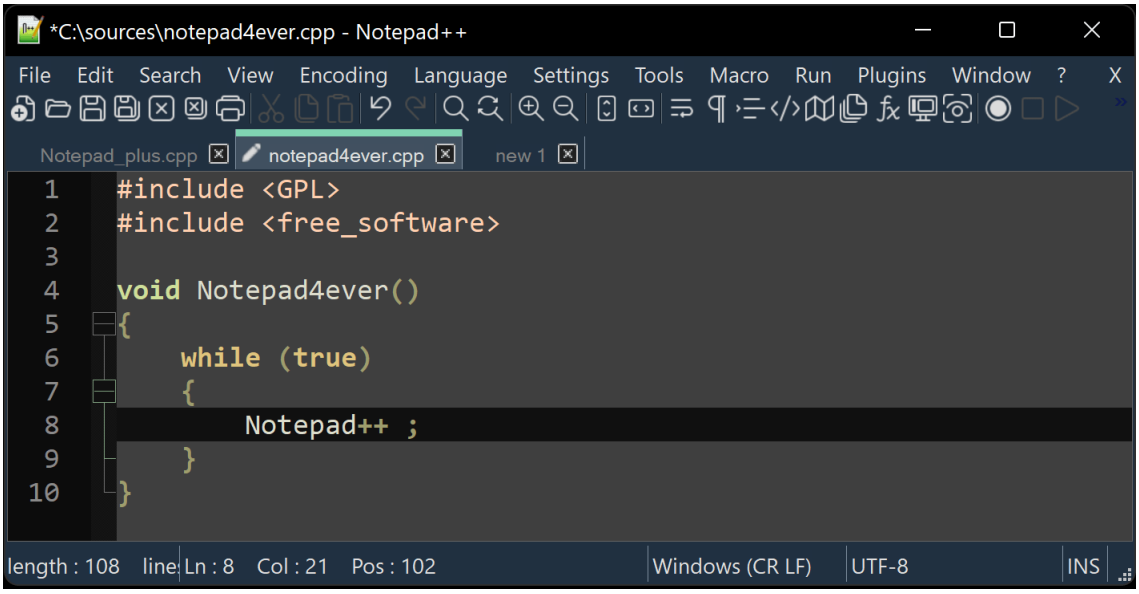

Рис. 2: Текстовый редактор Notepad++.

Редакторы программ

Intellij IDEA Это ИСР, разрабатываемая компанией JetBrains, для операционных систем Windows, Linux и MacOS. У неё есть поддержка плагинов, и можно установить плагин для анализа естественного текста, например, плагин

Grazie [23] от JetBrains. Можно реализовать собственный плагин и встроить нужный сервис для проверки текста, однако интерфейс Intellij IDEA [24] содержит множество графических элементов и функций, не требующихся для редактирования текстов на естественном языке, и они ухудшают пользовательский опыт. Примеры таких элементов можно увидеть на рисунке 3. Например, интерфейс Intellij IDEA содержит кнопки, связанные со сборкой и запуском проектов, меню для работы со сборщиками исходного кода, базами данных, системой управления версиями. Все эти элементы являются лишними для работы с текстами на естественном языке.

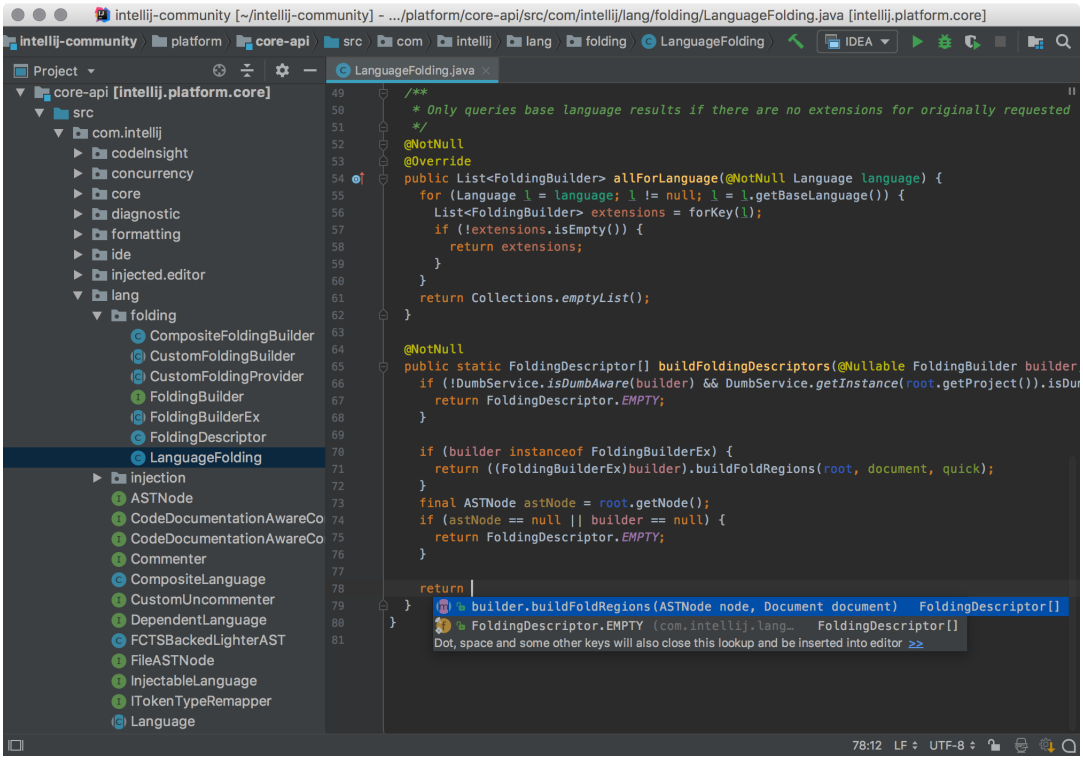

Рис. 3: Среда разработки Intellij IDEA.

Visual Studio Code Это редактор программ, разрабатываемый Microsoft. Он также является кроссплатформенным и доступен не только для операционных систем Windows, Linux и MacOS, но и для браузеров. У него также есть поддержка плагинов. В Visual Studio Marketplace можно получить доступ к плагинам, анализирующим текст (например, Code Spell Checker [11]). Также можно реализовать собственный плагин и с помощью VS Code API [12]

добавить подсветку ошибок, автодополнение и различные подсказки в текстовый документ. Недостаток данного подхода в том, что нужно реализовать проверку документа в плагине и вручную выделять ошибки, добавлять подсказки или автодополнение путём взаимодействия с VS Code API. Реализация простого плагина для проверки грамматики занимает около 500 строк [13]. Аналогично с Intellij IDEA, редактор ориентирован на работу в составе ИСР. Внешний вид данной ИСР представлен на рисунке 4.

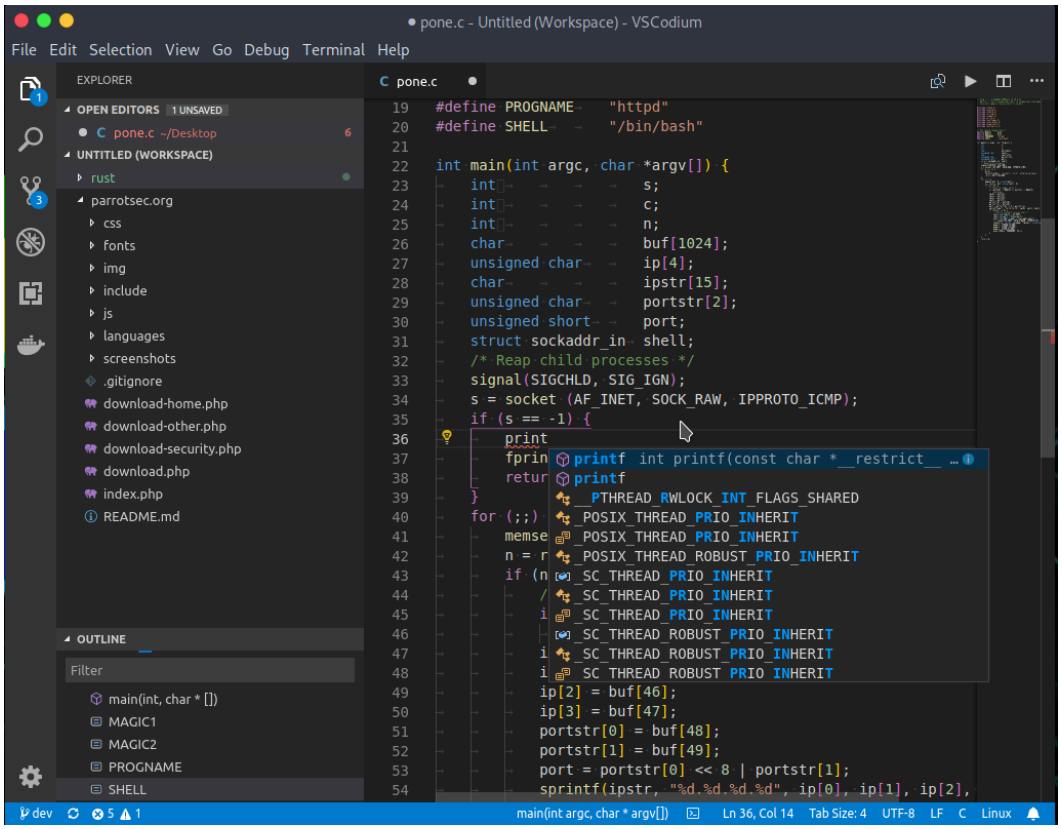

Рис. 4: Среда разработки Visual Studio Code.

Выводы

Текстовые редакторы, предназначенные для работы с текстами на естественном языке, не имеют функциональности для встраивания собственных сервисов анализа текста, и их архитектура не ориентирована на сложную обработку текста. Редакторы программ, однако, имеют более широкие возможности для работы с текстами (в частности, если это тексты программ). Некоторые ИСР позволяют создавать плагины, в которые можно встроить

свой сервис обработки текста, однако, это требует дополнительных знаний для создания плагина. Другим недостатком является то, что ИСР имеют большое количество графических элементов и функций, не требуемых для работы с текстами на естественном языке. Редактор, созданный в этой работе, избавляет разработчика от необходимости взаимодействовать с визуальными компонентами редактора и требует только реализации одного интерфейса для анализа текста.

1.3. Jetpack Compose и Compose Desktop

Jetpack Compose — декларативный UI фреймворк для Android [14], разрабатываемый компанией Google [15]. **Compose for Desktop** — UI фреймворк для разработки десктопных приложений на Kotlin, разрабатываемый компанией JetBrains [16]. **Десктопное приложение** — это программа, устанавливаемая на компьютер и работающая под управлением операционной системы. Compose for Desktop берет за основу Jetpack Compose, поэтому многие компоненты фреймворка являются общими для нескольких платформ, в частности для MacOS, Linux, Windows и Android. Оба этих проекта развиваются параллельно, и фактически Compose for Desktop является частью Jetpack Compose.

Реализация библиотеки построена вокруг Composable функций — функций, помечающихся аннотацией @Composable, говорящей компилятору, что данная функция принимает на вход данные и инициирует создание UI элементов. UI иерархия строится посредством вызова Composable функций из Composable функций.

Ниже приведен пример объявления Composable функции из библиотеки Jetpack Compose [17], отвечающей за создание UI элемента с ярлыком(badge) в его углу с дополнительной информацией:

```
@Composable
fun BadgedBox(
    badge: @Composable BoxScope.() -> Unit,
    modifier: Modifier = Modifier,
```

```
    content: @Composable BoxScope.() -> Unit,
) { ... }
```

Фреймворк придерживается принципов Slot API, то есть того, что каждый компонент библиотеки выполняет только одну обязанность и не отвечает за неявное поведение, например, связанное с самостоятельным UI компонентом, который передается в качестве параметра. В случае BadgedBox эта обязанность — правильно расположить ярлык(badge) относительно содержимого(content). Однако вся ответственность за создание ярлыка и всей связанной с ним внутренней логики лежит на других Composable функциях, которые будут переданы в качестве аргументов.

Обновление UI компонентов происходит путем рекомпозиции — процесса повторного вызова Composable функции с новыми данными. В Jetpack Compose разработчики стремятся сделать процесс рекомпозиции таким, чтобы происходила повторная композиция только измененных компонентов. Это нужно для оптимизации обновления UI. Компоненты также могут обладать состоянием, которое сохраняется между рекомпозициями, и управлять им самостоятельно. Изменение состояния вызывает рекомпозицию компонента, состояние можно объявить с помощью remember composable и функции mutableStateOf:

```
val mutableState = remember { mutableStateOf(default) }
```

Изменение переменной mutableState вызовет рекомпозицию компонента, в котором она содержится.

У данного фреймворка следующие преимущества для создания редактора:

- Доступность для различных платформ (в частности для Android, MacOS и Linux).
- Большое количество уже реализованных composable функций, предоставляющих Material компоненты [18]. Это избавляет разработчика от необходимости создания базовых элементов интерфейса (например, текстовых полей, кнопок, всплывающих окон и так далее).

- Кастомизируемость существующих компонентов. Это дает возможность усложнять и переиспользовать существующие компоненты без необходимости создания новых.

- Высокая производительность, поскольку механизм рекомпозиции действует выборочно, обновляя только те компоненты, которые изменились.

- Открытый исходный код и наличие документации.

  Можно выделить следующие недостатки:

- Фрейморк появился 2018 году, а стабильная версия вышла в 2021, поэтому сообщество разработчиков небольшое.

- Из-за того, что проект появился недавно, его производительность на Android уступает классическому подходу построения UI с использованием XML файлов [19]. Авторы статьи [20] выявили, что отрисовка UI в Jetpack Compose примерно в полтора раза дольше отрисовки интерфейса, разработанного с использованием XML файлов.

- Компонент BasicTextField, который будет использоваться в редакторе, не предназначен для работы с большими текстами (примерно для 30 страниц формата A4 отрисовка текста занимает заметное число времени). Однако, команда разработчиков планирует создать аналогичный компонент, который должен быть более производительным. В главе 3 приведена подробная информация и описание способов оптимизации.

1.4. Kotlin Multiplatform

Фреймворк Jetpack Compose написан на Kotlin [21] с использованием технологии **Kotlin Multiplatform** [22], позволяющей запускать и переиспользовать один и тот же код на разных платформах. Kotlin Multiplatform состоит из обычного языка Kotlin, включающего в себя язык, базовые библиотеки и инструменты, и платформо-специфичных версий Kotlin: Kotlin/Native (для

кода, не требующего виртуальной машины для запуска), Kotlin/JS (для браузеров) и Kotlin/JVM (для кода, запускающегося с использованием виртуальной машины). Таким образом, становится возможным переиспользовать основной код на разных платформах, и для того, чтобы реализовать редактор на новой платформе, достаточно добавить реализацию только платформо-зависимых методов (например, открытие ссылки в браузере).

Для реализации платформо-зависимых функций и классов в Kotlin Multiplatform были введены ключевые слова actual и expect. Ключевое слово expect обозначает объявление без реализации. В отличие от интерфейсов, наличие expect обязывает разработчика привести реализации для каждой из платформ с использованием ключевого слова actual.

Таким образом, общий код для всех платформ содержится в основном модуле, и к нему имеют доступ модули, отвечающие за платформы и содержащие платформо-зависимый код.

1.5. Grazie

Grazie — команда в компании JetBrains, занимающаяся разработкой библиотек, сервисов и плагинов для работы с естественными языками. Один из их продуктов — плагин Grazie [23] для Intellij IDEA, выполняющий проверку правописания комментариев в коде.

Для редактора будут использоваться их следующие библиотеки и сервисы:

- ai.grazie.gec — Grammar Error Correction (сервис по исправлению грамматических ошибок)
- Grazie Natural Language Completion — сервис для предсказания автодополнения текста.

2. Разработка редактора

В данной главе приведен обзор интерфейса и архитектуры приложения. Основные моменты реализации описаны в главе 4.

2.1. Архитектура

Редактор реализован на Kotlin Multiplatform с использованием фреймворков Jetpack Compose и Compose for Desktop. Архитектура редактора представлена на рисунке 5.

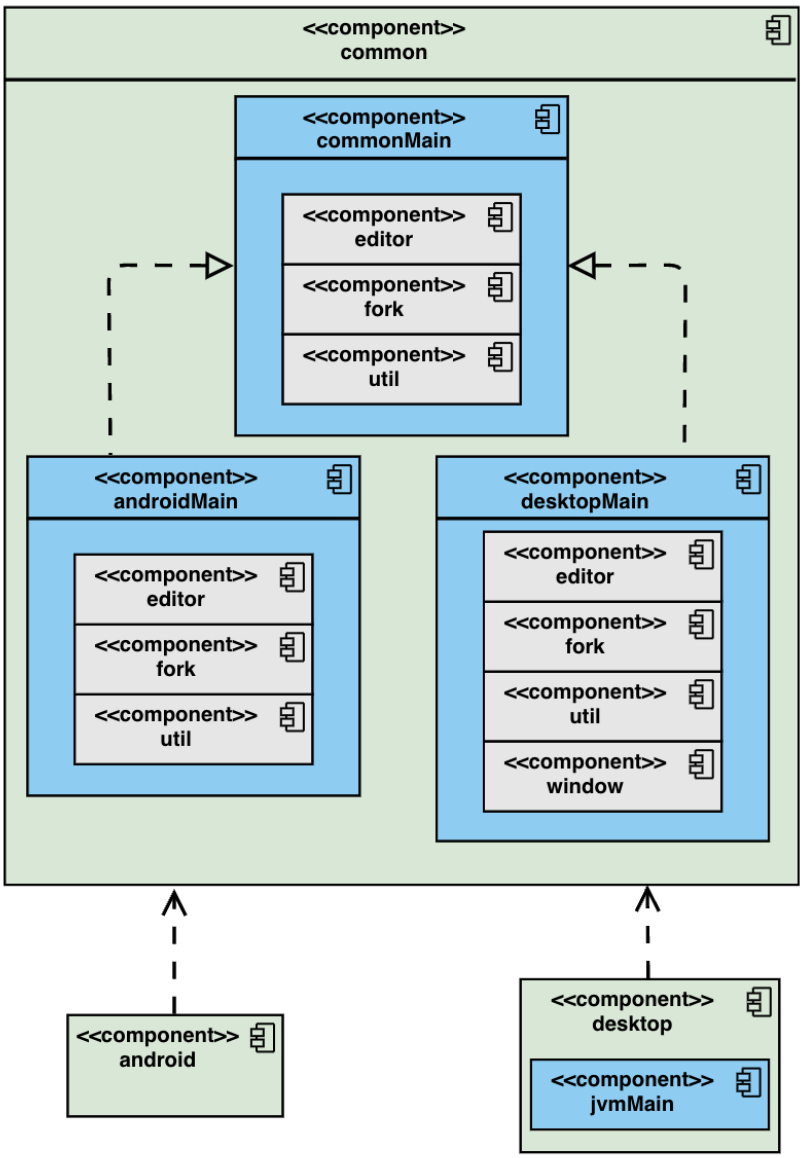

Рис. 5: Диаграмма компонентов приложения.

Кодовая база состоит из трех основных модулей: common, android, desktop:

- **Модуль common** включает в себя три модуля: commonMain, androidMain, desktopMain. Модуль commonMain содержит:
  - классы данных, использующиеся как в Android приложении, так и в десктопном
  - объявления платформо-зависимых компонентов и интерфейсов

  androidMain и desktopMain содержат фактическую реализацию платформо-зависимых компонентов и интерфейсов. Это реализовано с помощью механизма ожидаемых и фактических объявлений Kotlin, позволяющему подключаться к API конкретной платформы.
- **Модуль android** Содержит MainActivity Android приложения, которое запускает код редактора.
- **Модуль desktop** состоит из модуля jvmMain, включающего в себя функцию main для запуска десктопного приложения.

Основной код редактора реализован в пакете com.highlightEditor, который включает в себя следующие пакеты:

- **editor**. Включает в себя следующие директории:
  - **diagnostics**. Содержит классы данных с состояниями автодополнения и найденных ошибок в тексте. Также содержит ожидаемое объявление компонента DiagnosticPopup.
  - **docTree**. Содержит методы для работы с документной моделью текстового файла.
  - **draw**. Содержит классы, отвечающие за декорирование текста.
  - **text**. Содержит класс TextState, в котором находятся:
    * состояния, хранящие текущий текст, его документную модель и информацию о состоянии компонента BasicTextField (TextFieldLayoutResult)

    * методы для обновления состояний после ввода пользователя
    * методы для нахождения позиции в тексте и элемента документной модели по положению курсора

- **fork**. Содержит копию компонента BasicTextField из библиотеки Jetpack Compose, в API которого были внесены необходимые для редактора изменения (подробней в главе 3).

2.2. Пользовательский интерфейс

Графические компоненты приложения предоставлялись Jetpack Compose и Compose for Desktop. Они представляют собой реализацию компонентов из Material Design [18].

Для приложения использовались следующие графические компоненты Material Design:

- Табы [25]. Используются для переключения между типами элементов документной модели. Пользователь, выбрав тип элемента, начинает вводить текст, соответствующий данному элементу. Элементы одного типа имеют уникальный стиль (размер шрифта, цвет, декорация).

- Индикатор загрузки [26]. Используется для того, чтобы показать пользователю, что анализ текста происходит в данный момент времени.

- Snackbar [27]. Используется для отображения сообщения об отсутствии Интернет соединения.

Также из библиотеки Compose UI используется Composable функция Popup [28]. С помощью нее можно отобразить плавающий контейнер в заданной позиции с любым содержимым. В редакторе он используется для отображения:

- **информации об ошибке в тексте** и предлагает варианты замены. При нажатии (или при двойном нажатии, в случае Android версии редактора) на выбранный вариант происходит замена выделенного текста на предложенный. Этот элемент интерфейса представлен на рисунке 6:

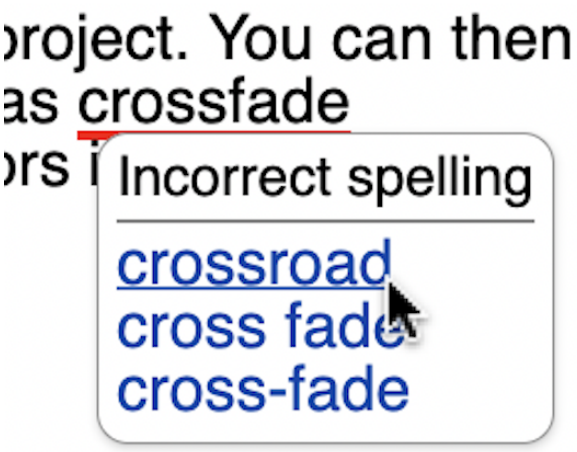

Рис. 6: Плавающий контейнер с информацией об ошибке и вариантами замены.

- **элемента для открытия ссылок**. При наведении (или при двойном нажатии, в случае Android версии редактора) на элемент документной модели, представляющий ссылку, отображается Popup с подсказкой «Open link in a browser» (рус. «Открыть ссылку в браузере»), при нажатии на которую открывается выбранная ссылка в браузере на персональном компьютере (или мобильном телефоне). Внешний вид элемента представлен на рисунке 7:

Рис. 7: Плавающий контейнер для открытия ссылки в браузере.

Редактор был реализован для платформы Android (представлен на рисунке 8) и для платформ MacOS, Linux, Windows (представлен на рисунке 9).

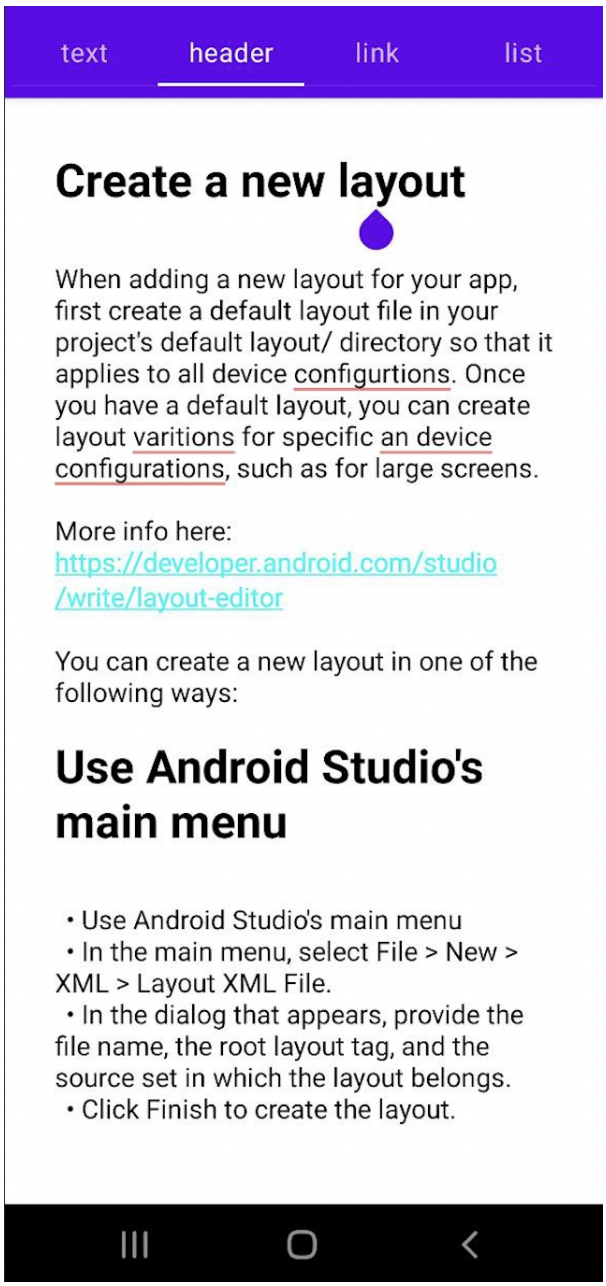

Рис. 8: Редактор в Android реализации.

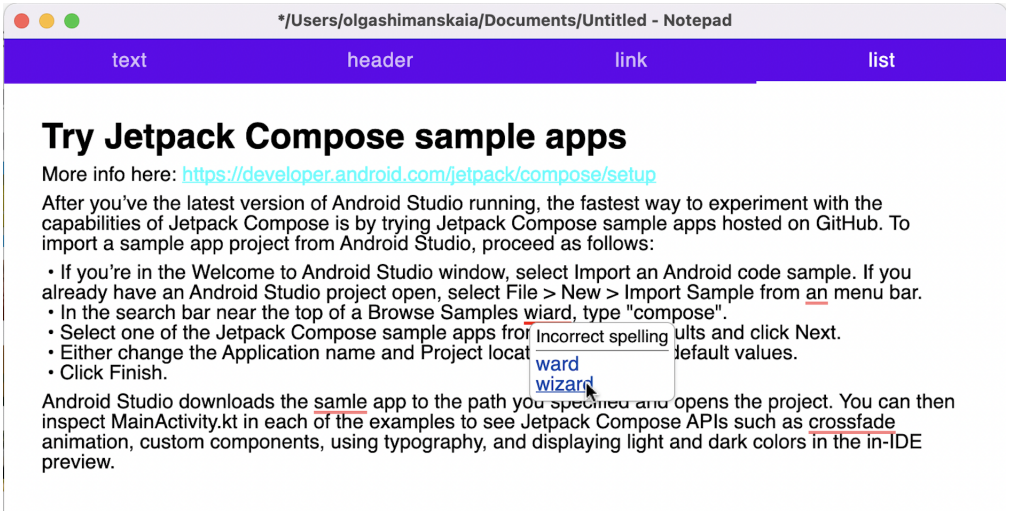

Рис. 9: Редактор в MacOS реализации.

В десктопной версии редактора также доступна работа с файловой системой, доступно:

- **открытие** файлов
- **сохранение** файлов
- **экспорт** в следующие форматы:
  - **Markdown** [32]
  - **LaTeX** [33]

Интерфейс реализован с помощью компонента Menu из Compose for Desktop. На платформе MacOs данное меню выглядит так, как представлено на рисунке 10:

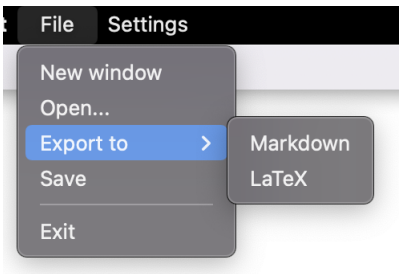

Рис. 10: Меню редактора в MacOS версии.

2.3. Взаимодействие со встроенными сервисами

Редактор предоставляет возможность встроить собственный сервис для проверки грамматики и автодополнения. Для этого нужно реализовать интерфейс с двумя методами:

```
interface TextAnalyzer {
    suspend fun analyze(text: List<Sentence>):
    ↪ List<DiagnosticElement>
    suspend fun autocomplete(context: String, prefix: String):
    ↪ List<String>
}
```

1. Метод **analyze** — принимает на вход список предложений и возвращает список объектов DiagnosticElement.

   Такой интерфейс удобен тем, что многие сервисы для проверки грамматики проверяют не весь текст целиком, а отдельные предложения. Например, Grazie Grammar Error Correction в методе correct запрашивает не текст, а список предложений, и возвращает список исправлений для каждого из переданных предложений.

   Предложение в редакторе — объект класса Sentence.

   ```
   data class Sentence(
       val text: String,
       val range: IntRange
   )
   ```

   Он состоит из двух полей:

   - текст предложения
   - интервал предложения в целом тексте

   Разбиение на предложения происходит внутри редактора (подробней в 4.2).

   Код класса DiagnosticElement выглядит следующим образом:

```
data class DiagnosticElement(
    val offset: Int,
    val length: Int,
    val message: String,
    val suggestions: List<String> = listOf()
)
```

Объект DiagnosticElement состоит из следующих полей:

- **offset** — отступ в переданном тексте, с которого начинается ошибка. Несмотря на то, что в метод передаётся список предложений, поле range в объекте Sentence позволяет из отступа ошибки в самом предложении определить отступ в целом тексте путем прибавления range.first (начала интервала).
- **length** — длина текста ошибки
- **message** — сообщение с информацией об ошибке
- **suggestions** — список возможных исправлений данной ошибки

2. Метод **autocomplete** — принимает на вход контекст и префикс. На основе контекста может определяться дополнение для префикса. По умолчанию контекст — весь текст, и он передается в метод, в самом методе можно использовать какую-то его часть.

   Такой интерфейс удобен тем, что позволяет пользователю настраивать длину контекста, который будет передан в сервис. Префикс нужен для того, чтобы предсказывать его продолжение, для многих API автодополнения это обязательный параметр (например, для Grazie Natural Language Completion). Его также можно не использовать, если сервис его не требует.

   Префикс — суффикс текста, начинающийся после последнего пробела (в случае если пробел в конце текста — пустая строка)

   На выход метод должен вернуть список возможных дополнений префикса. В текущей реализации редактора отображается только первый элемент из списка следующим образом, представленным на рисунке 11:

Рис. 11: Автодополнение в редакторе.

Анализ текста реализован таким образом, что пользователю не нужно беспокоиться о том, что его проверка занимает большое время или доступ к сервису нестабилен:

- Редактирование и анализ текста происходят асинхронно. Для этого код вызова analyze и autocomplete находится в отдельном CoroutineScope [21].
- Повторный анализ запрашивается только в том случае, если текст в редакторе совпадает с текстом, переданным на анализ, и если предыдущая проверка завершилась.
- В случае отсутствия Интернет соединения будет отображено уведомление в редакторе.

3. Компонент CoreTextField в Jetpack Compose

Основной компонент фреймворка Jetpack Compose, на котором разрабатывался редактор — BasicTextField, реализация которого состоит из вызова внутреннего компонента CoreTextField. CoreTextField отвечает за отрисовку и редактирование текста, отрисовку и за обновление расположения текстового курсора или выделения в тексте. После начала разработки редактора было обнаружено, что на больших текстах (от 30 страниц формата A4) редактирование происходит с задержкой, которая заметна пользователю. Оказалось, что это известная проблема [29] и разработчики библиотеки планируют в одной из будущих версий фреймворка изменить архитектуру компонента, чтобы он смог стабильно работать на бóльших объемах текста.

3.1. Состояние компонента и отрисовка текста

Одними из основных аргументов, принимаемых компонентом CoreTextField, являются следующие:

- **value** типа **TextFieldValue**. Содержит информацию о тексте, который будет нарисован, и о выделении в тексте (в случае если выделение нулевой длины, то это положение текстового курсора).
- функция **onValueChange**, вызываемая при редактировании текста, изменении положения курсора или выделения.
- **onTextLayout**, предоставляющий объект **TextLayoutResult**, содержащий информацию о стиле текста и его разбиении на параграфы.

Причиной медленной работы редактирования текста является архитектура и реализация компонента. После каждого изменения **TextFieldValue** происходит перерисовка всего текста (содержащегося в TextLayoutResult), даже если он выходит за пределы видимой области (DecorationBox).

Визуальное представление того, как устроена отрисовка текста в компоненте CoreTextField, представлено на рисунке 12.

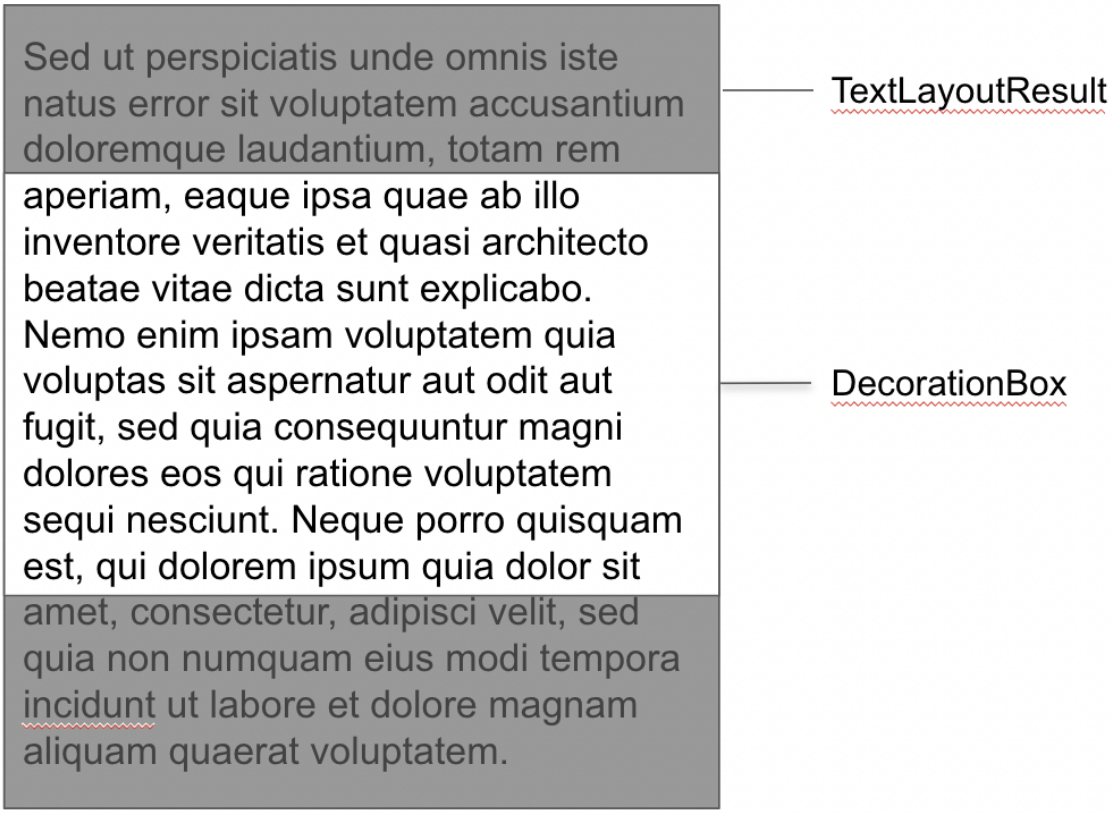

Рис. 12: Устройство компонента CoreTextField. Текст с белым фоном видим пользователем, текст с серым фоном не видим и попадет в видимую область после соответствующей вертикальной прокрутки.

3.2. Оптимизация отрисовки текста

Были предприняты некоторые попытки ускорить отрисовку текста в компоненте CoreTextField. Было предложено 2 идеи:

1. Передавать в качестве параметра **value** только видимый текст.

   Это не лучшее решение, так как придется дополнительно:

   - Реализовать интерфейс и поведение вертикальной прокрутки текста.
   - Учесть работу с выделением текста (выделить весь текст с помощью сочетания клавиш или текст, выходящий за пределы видимой области).

2. Предварительно разбивать текст на параграфы, и в реализации компонента отрисовывать только те параграфы, которые пересекаются с видимой областью.

   Изначально в реализации компонента происходит отрисовка всех параграфов, хранящихся в представителе класса **MultiParagraph**, который в свою очередь входит в состав **TextLayoutResult**. Параграфы определяются на основе параметра annotatedString типа AnnotatedString в TextFieldValue, объект которого передается в CoreTextField. В AnnotatedString содержится параметр **paragraphStyles**, являющийся списком стилей параграфов. Каждый параграф задается на основе интервалов в тексте.

   Таким образом, можно разбить текст на параграфы по символу переноса строки, а затем передать полученные значения интервалов в качестве параметра paragraphStyles.

   Метод paint, отвечающий за отрисовку текста, находится в объекте TextPainter. Этот метод вызывается при каждом изменении TextFieldValue, и в нем находится вызов метода paint класса MultiParagraph, в котором происходит отрисовка текста всех параграфов:

```
/** Paint the paragraphs to canvas. */
fun paint(
    canvas: Canvas,
    color: Color = Color.Unspecified,
    shadow: Shadow? = null,
    decoration: TextDecoration? = null
) {
    canvas.save()
    paragraphInfoList.fastForEach {
        it.paragraph.paint(canvas, color, shadow, decoration)
        canvas.translate(0f, it.paragraph.height)
    }
    canvas.restore()
}
```

Можно изменить этот метод так, чтобы отрисовывать только те параграфы, которые попадают в видимую область на экране, взятую с отступом. Отступы сверху и снизу размером с высоту видимой области достаточны для того, чтобы пользователь при вертикальной прокрутке текста не замечал, что происходит перерисовка видимого текста.

Для реализации были добавлены дополнительные параметры в API методов paint, содержащие информацию о том, какая высота видимой области и с какого отступа по оси Y текст попадает в эту область. У самих параграфов есть API, позволяющее узнать отступ по оси Y сверху в TextLayoutResult, в котором содержится весь текст. Итого можно понять входит ли данных параграф в видимую область с отступом или нет.

Был проведен анализ работы метода TextPainter.paint на тексте длины 120000 символов. На основе 20 редактирований текста было выявлено, что средняя скорость работы данного метода уменьшилась с 110 миллисекунд до 13 миллисекунд.

От внедрения данного подхода пришлось отказаться по следующим причинам:

- Разбиение текста на параграфы влияет на работу текстового курсора перед переносом строки. Поведение текстового курсора не совпадает с ожидаемым.
- Компонент содержит другие тяжеловесные операции, которые всё еще влияют на скорость работы редактирования больших текстов. Например, было выявлено, что операция для вычисления объекта типа TextLayoutResult по тексту занимает примерно то же время, что и отрисовка.

3.3. Изменения в API CoreTextField и BasicTextField

Внутри реализации CoreTextField содержится информация о вертикальной прокрутке текста — текущем отступе сверху до DecorationBox. При про-

крутке текста происходит обновление отступа. Однако нельзя получить этот отступ из компонента, так как это его внутренняя переменная.

Было решено изменить API, введя следующий параметр:

```
onScroll: (Float) -> Unit = {}
```

Такое наименование и тип метода соответствуют стилю JetpackCompose, в данном случае имя метода говорит о том, что он будет вызван, когда изменится позиция прокрутки (Scoll), и в него будет передана информация о текущем отступе сверху.

Сама функция ничего не возвращает, поскольку этот метод не должен влиять на внутреннюю работу компонента и его состояние. Этот метод нужен для того, чтобы получить информацию о внутреннем состоянии компонента.

Метод onScroll теперь позволяет получить отступ до DecorationBox, эта информация нужна для:

- правильного расположения DiagnoticPopup в видимой области
- отрисовки подчеркивания ошибок в тексте

В объекте TextLayoutResult можно получить позицию символа в тексте относительно этого объекта, однако вся отрисовка происходит в координатной системе видимой области DecorationBox, поэтому нужно смещать эти отступы по Y относительно текущего состояния вертикальной прокрутки.

4. Работа с текстом

В этой главе содержится обзор основного кода и архитектуры, связанных с обработкой текста в редакторе.

4.1. Документная модель

4.1.1 Определение

Документная модель — способ описания структуры текстового документа в виде дерева элементов.

В реализации редактора были использованы следующие элементы:

- заголовок (DocumentType.HEADER), вид в редакторе представлен на рисунке 13.

Рис. 13: Внешний вид элемента типа заголовок в редакторе.

- текст (DocumentType.TEXT), вид в редакторе представлен на рисунке 14.

Рис. 14: Внешний вид элемента типа текст в редакторе.

- ссылка (DocumentType.LINK), вид в редакторе представлен на рисунке 15.

Рис. 15: Внешний вид элемента типа ссылка в редакторе.

- элемент списка (DocumentType.LIST), вид в редакторе представлен на рисунке 16.

Рис. 16: Внешний вид элемента типа элемент списка в редакторе.

4.1.2 Внешний вид элементов документной модели в редакторе

Поскольку текущая реализация ограничена и элементы модели самостоятельные (например, нет списка и подсписка), то было решено хранить их не в виде дерева, а в виде последовательности.

Информация и методы для работы с документной моделью содержатся в классе DocumentModel.

Были реализованы следующие методы, необходимые для работы с документной моделью в редакторе:

- **addElement** — метод для добавления элемента в документную модель.

  При вводе текста мы добавляем элемент определенного типа в известный интервал в тексте. Поскольку элементы хранятся в отсортированном по интервалу виде, то мы можем найти позицию, куда вставить данный элемент, с помощью функции binarySearch из стандартной библиотеки Kotlin.

- **removeRange** — метод для удаления подстроки в заданном интервале из документной модели.

  При удалении символа или выделения мы должны удалить какой-то интервал относительно всего текста. Для этого мы находим элементы, в которые попадают начало и конец интервала соответственно. После происходит удаление элементов между ними и самих этих элементов (или их части, если начало/конец попадает в середину элемента).

- **getContent** — возвращает аннотированную строку (AnnotatedString), построенную по текущим элементам модели. Параметр spanStyles будет содержать стили для каждого элемента из документной модели.

- **concatElements** — вспомогательная функция, соединяющая последовательные элементы одного типа.

- **getElementByOffset** — нужен для того, чтобы в интерфейсе отобразить тип элемента, на котором находится текстовый курсор.

4.1.3 Экспорт в форматы Markdown и LaTeX

Большинство редакторов позволяют пользователю производить экспорт текста в сторонние форматы. Удобство документной модели для экспорта в том, что текст уже разделен на элементы (заголовок, ссылка и т.п.), поэтому достаточно пройтись по ним и каждый из них перевести в аналогичный элемент стороннего формата.

В десктопной версии редактора текст можно сохранить в расширении **dm**, представляющий собой строковое представление объекта формата JSON [30], где есть следующие ключи:

- **text**. Его значение — простой текст, который был сохранен.
- **elements**. Его значение — список элементов документной модели, где каждый элемент является представлением объекта DocumentElement в формате JSON (данная функциональность реализована с помощью библиотеки Kotlinx serialization [31]).

Данный формат позволяет сохранять информацию о документной модели, а при открытии файла такого формата — легко восстановить текст нужного вида в редакторе.

Был реализован экспорт документной модели, представленной расширением **dm**, в два известных и широко используемых формата:

1. **Markdown** [32] — облегчённый язык разметки для обозначения форматирования в простом тексте.
2. **LaTeX** [33] — самый широко известный и используемый набор макрорасширений системы компьютерной вёрстки TeX, облегчающий набор сложных документов.

В классе DocumentModel находятся методы toMarkdown и toLatex, которые переводят документную модель в Markdown и LaTeX соответственно.

Экспорт в Markdown

Перевод элементов происходит следующим образом:

- **Заголовок** переводится в строку вида “# текст заголовка”, в языке разметки Markdown так обозначается главный заголовок.
- **Текст** переводится в без изменений.
- **Ссылка** переводится в строку вида “[текст ссылки]”. Квадратными скобками в Markdown выделяется текст ссылок.
- **Элемент списка** переводится в строку вида “* текст элемента списка”. Одним из способов обозначения списка в Markdown является символ * перед элементом списка.

Экспорт в LaTeX

Перевод элементов происходит следующим образом:

- **Заголовок** переводится в строку вида “\section*{текст заголовка}”, в языке разметки LaTeX так обозначается главный заголовок без нумерации.
- **Текст** переводится в без изменений.
- **Ссылка** переводится в строку вида “\url{текст ссылки}”.
- **Элемент списка** переводится в строку вида “\item текст элемента списка”. В LaTeX дополнительно нужно обозначать начало и конец списка, для этого в начале последовательности элементов типа DocumentModel.LIST и в конце добавляется “\begin{itemize}” и “\end{itemize}” соответственно. Ключевое слово itemize означает ненумерованный список.

4.2. Класс TextState

В реализации редактора текст хранится в классе TextState. В этом классе находятся методы, вызываемые при каждом обновлении текста, и методы, достающие из объекта класса TextLayoutResult информацию о позиции в тексте, на которую указывает курсор, и о позиции отрисованного символа по его позиции в тексте.

TextState хранит и поддерживает состояние следующих объектов:

- **text** — текущий TextFieldValue, передаваемый в редактор.
- **textLayoutResult** — TextLayoutResult редактора, используемый для перевода координат курсора в отступы в тексте, а также определения позиции линии или символа в TextLayoutResult.
- **prevSelection** — предыдущее положение текстового курсора/выделения в тексте.

  При редактировании текста в компоненте BasicTextField разработчик получает новое значение TextFieldValue с помощью функции onValueChange, на вход которой оно подается. Однако, нет никакой информации о том, как изменилось значение относительно предыдущего. Для этого можно использовать положение текстового курсора или выделения. Сравнивая старое и новое положение выделения, а также длину старого и нового текста можно узнать, какое изменение произошло.

  Положения выделения и длин текста соотносятся с изменением следующим образом:

  – Предыдущий и новый текст совпадают.

    Это означает, что поменялось положение выделения, поэтому нужно обновить prevSelection и text, больше ничего делать не требуется.

  – Оба выделения длины 0 (это означает изменение положения текстового курсора) и новый текст длиннее предыдущего.

Это означает, что был вставлен новый текст с позиции предыдущего текстового курсора до нового. Вставленный текст можно таким образом определить, взяв подстроку из нового текста.

- Оба выделения длины 0 и новый текст короче предыдущего.

  Это означает, что был удален текст с позиции предыдущего текстового курсора до нового. Удаленный текст можно таким образом определить, взяв подстроку из предыдущего текста.

- Предыдущее выделение ненулевой длины и текущее нулевой.

  Это означает, что был удален выделенный текст и вставлен новый. Удаленный текст можно определить по прошлому выделению, а вставленный по новому тексту (это подстрока с начала прошлого выделения до конца текущего).

- Новое выделение ненулевой длины.

  Такая ситуация в компоненте может произойти, когда пользователь выделил часть текста, в таком случае предыдущий и новый текст совпадут.

- **documentModel** — документная модель текста.

  На основе приведенных выше наблюдений с выделениями мы можем контролировать произошедшие изменения и обновлять документную модель.

- **sentences** — текст, разбитый на предложения.

  Для разбиения на предложения был реализован обработчик, работающий на регулярных выражениях [35].

  Регулярное выражение — это формальный язык поиска и манипуляций с подстроками текста, построенный на использовании шаблонов, представляющих собой строки, состоящие из символов и метасимволов (то есть символов, обозначающий какие-либо символы и их последовательности).

  В проекте содержится список правил в формате **srx** [34], в котором описаны условия, при которых предложение заканчивается и начинает-

ся новое, и список исключений, которые могли бы подойти под условие окончания предложения, например, если это сокращение названия месяца Nov. (сокращенно от November), то это не обязательно означает, что предложение завершилось. Список правил был предоставлен командой Grazie компании JetBrains.

Объект **SentenceTokenizer** содержит реализацию обработчика и читает правила из файла формата **srx**.

Метод **tokenizeText** объекта **SentenceTokenizer** принимает на вход текст типа String и возвращает список объектов Sentence. Sentence состоит из текста и его позиции в исходном тексте.

Для разбиения на предложения был реализован проход по правилам, представленных в виде регулярного выражения (тип RegExp в языке программирования Koltin).

Каждое сопоставление с правилом заменяется на символы с кодом \uE001, не имеющим стандартного определения в Юникоде.

Для того чтобы сохранить позиции предложений в исходном тексте, сопоставление заменяется на последовательность символов \uE001 той же длины, что и совпадение. Таким образом, итоговый текст будет иметь ту же длину, и его можно разбить на предложения с помощью регулярного выражения “ \uE001+”, обозначающее последовательность символов \uE001 ненулевой длины.

Полученные предложения в последствии отправляются в метод analyze реализации интерфейса TextAnalyzer.

4.3. Отрисовка результатов анализа

Отрисовка грамматических ошибок

Для отрисовки грамматических ошибок в тексте был реализован абстрактный класс HighlightDrawer, содержащий следующий метод:

```
abstract fun draw(position: Position, drawScope: DrawScope)
```

Параметр position обозначает позицию текста, который надо выделить относительно TextLayoutResult.

Position это сокращенная сигнатура от List<TextSegment>. Поскольку иногда нужно выделить текст, который начинается на одной линии, а заканчивается на другой, то для этого его расположение разбивается на такие сегменты для каждой линии. TextSegment состоит из следующих полей:

- line — номер линии в тексте.
- boundingRect — координаты прямоугольника, обрамляющего текст на данной линии относительно TextLayoutResult.

Второй аргумент функции обозначает DrawScope, в котором будет происходить отрисовка. В редакторе это объект DrawScope, приходящий на вход функции Modifier.drawBehind, использующейся в компоненте BasicTextField.

С помощью данного класса можно реализовать собственную отрисовку для выделения текста. В редакторе на данный момент есть реализация UnderlineDrawer, которая подчеркивает текст красным цветом. Линия, подчеркивающая текст, который попал под курсор мыши или который был нажат (в случае Android версии), имеет большую яркость. Внешний вид подчеркивания представлен на рисунке 17.

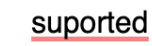

Рис. 17: Отрисовка в реализации UnderlineDrawer.

Отрисовка автодополнения

Для отрисовки автодополнения используется реализация интерфейса VisualTransformation фреймворка Jetpack Compose.

VisualTransformation содержит единственный метод:

```
fun filter(text: AnnotatedString): TransformedText
```

Этот метод принимает на вход аннотированную строку и возвращает объект типа TransformedText, содержащий в себе новую аннотированную строку и объект типа OffsetMapping. В OffsetMapping содержится два метода:

```
fun originalToTransformed(offset: Int): Int

fun transformedToOriginal(offset: Int): Int
```

Эти методы нужны для перевода отступа в исходном тексте в трансформированный и наоборот соответственно.

В случае автодополнения трансформированный текст получается следующим образом: к исходной аннотированной строке добавляем суффикс, содержащий предсказанное сервисом дополнение, и задаем ему тот же стиль, что и у элемента, который он дополняет, и меняем цвет текста на серый, показав таким образом нередактируемость данной части текста.

OffsetMapping строится следующим образом: из оригинального текста в трансформированный отображение идентичное, а из трансформированного в оригинальный мы переводим отступ в конец оригинального текста, если он попадает на дополнение, в противном случае отображение также идентичное.

Внешний вид автодополнения в редакторе представлен на рисунке 11 в главе 2.

Заключение

В рамках работы был проведен обзор аналогов как среди редакторов текста, так и среди редакторов программ. Было выявлено, что они либо не решают поставленную задачу, либо предоставляемые ими возможности являются неудобными для конечного пользователя. Для реализации мультиплатформенного редактора была выбрана технология Kotlin Multiplatform. Реализованный редактор доступен на следующих платформах: MacOs, Linux, Windows, Android. Интерфейс для первых трёх платформ был реализован с использованием фреймворка Compose for Desktop, а интерфейс для Android и общая для всех платформ функциональность и компоненты интерфейса были реализованы с помощью фреймворка Jetpack Compose. Архитектура редактора, таким образом, состоит из общего модуля и модулей для Android и десктопа, её диаграмма представлена в 2.1.

Основная функциональность редактора — возможность встроить собственный сервис для проверки грамматики и автодополнения. Реализация интерфейса TextAnalyzer дает пользователям возможность встроить собственные сервисы. Руководство для встраивания сервиса также приведено в репозитории [36].

Дополнительно редактор позволяет экспортировать текст в такие известные форматы как Markdown и LaTeX, а так же есть возможность открывать ссылки, написанные в редакторе, в браузере.

В будущем планируется расширить множество элементов, поддерживаемых документной моделью. Вдобавок хочется предоставить пользователям возможность дополнительно декорировать текст (менять размер шрифта и его тип, и тому подобное), а также позволить редактировать языки разметки и анализировать их в самом редакторе.

Реализация редактора представлена в репозитории [36].

Список литературы

[1] Joseph S. R. et al. Natural language processing: A review //International Journal of Research in Engineering and Applied Sciences. – 2016. – Т. 6. – №. 3. – С. 207-210.

[2] Воройский Ф.С. Информатика. Новый систематизированный толковый словарь-справочник. (Введение в современные информационные и телекоммуникационные технологии в терминах и фактах). 3-е изд, 2003, стр. 368

[3] Windows Notepad. URL: https://www.microsoft.com/ru-ru/p/windows-notepad/9msmlrh6lzf3 (дата обр. 25.04.2022).

[4] CKEditor. URL: https://ckeditor.com/ckeditor-5 (дата обр. 25.04.2022).

[5] IntelliJ IDEA. URL: https://www.jetbrains.com/ru-ru/idea (дата обр. 25.04.2022).

[6] Текстовый редактор Google Docs. URL: https://www.google.ru/intl/ru/docs/about (дата обр. 25.04.2022).

[7] Текстовый редактор Notepad++. URL: https://notepad-plus-plus.org (дата обр. 25.04.2022).

[8] ИСР Visual Studio Code. URL: https://code.visualstudio.com (дата обр. 25.04.2022).

[9] Плагин DSpellCheck для Notepad++. URL: https://github.com/Predelnik/DSpellCheck (дата обр. 05.05.2022)

[10] Форум для Notepad++. URL: https://community.notepad-plus-plus.org/ (дата обр. 05.05.2022)

[11] Code Spell Checker (plugin for VS Code).
URL: https://marketplace.visualstudio.com/items?itemName=streetsidesoftware.code-spell-checker (дата обр. 25.04.2022).

[12] VS Code API. URL: https://code.visualstudio.com/api/references/vscode-api (дата обр. 25.04.2022).

[13] Spell and Grammar Checker. URL: https://github.com/microsoft/vscode-spell-check (дата обр. 25.04.2022).

[14] Платформа Android. URL: https://www.android.com/intl/ru_ru/ (дата обр. 05.05.2022)

[15] Smyth N. Jetpack Compose Essentials: Developing Android Apps with Jetpack Compose, Android Studio, and Kotlin. (n.p.): Payload Media, Incorporated, 2022.

[16] Compose for Desktop. URL: https://www.jetbrains.com/ru-ru/lp/compose-mpp дата обр. 25.04.2022).

[17] BadgeBox компонент библиотеки Jetpack Compose. URL: https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/material/material/src/commonMain/kotlin/androidx/compose/material/Badge.kt;l=58 (дата обр. 25.04.2022).

[18] Material компоненты от компании Google. URL: https://material.io/components (дата обр. 05.05.2022)

[19] Vasic M. Mastering Android Development with Kotlin: Deep dive into the world of Android to create robust applications with Kotlin. – Packt Publishing Ltd, 2017.

[20] Fjodorovs I., Kodors S. JETPACK COMPOSE AND XML LAYOUT RENDERING PERFORMANCE COMPARISON //HUMAN. ENVIRONMENT. TECHNOLOGIES. Proceedings of the Students International Scientific and Practical Conference. – 2021. – №. 25. – С. 49-54.

[21] Jemerov D., Isakova S. Kotlin in action. – Simon and Schuster, 2017.

[22] Moore K.D.,Taheri S.,Mota C. Kotlin Multiplatform by Tutorials (First Edition): Build Native Apps Faster by Sharing Code Across Platforms. (n.p.): Razeware LLC, 2022.

[23] Плагин Grazie для Intellij IDEA. URL: https://plugins.jetbrains.com/plugin/12175-grazie (дата обр. 05.05.2022)

[24] Krochmalski J. IntelliJ IDEA Essentials. – Packt Publishing Ltd, 2014.

[25] Tabs, Material Design Component. URL: https://material.io/components/tabs (дата обр. 25.04.2022).

[26] Circular Progress Indicator, Material Design Component. URL: https://material.io/components/progress-indicators#circular-progress-indicators (дата обр. 25.04.2022).

[27] Snackbar, Material Design Component. URL: https://material.io/components/snackbars#usage (дата обр. 25.04.2022).

[28] Popup, Compose UI. URL: https://developer.android.com/reference/kotlin/androidx/compose/ui/window/package-summary#Popup(androidx.compose.ui.Alignment,androidx.compose.ui.unit.In (дата обр. 25.04.2022).

[29] Запрос на исправление ошибки, связанной с редактированием большого текста, в фреймворке Jetpack Compose . URL: https://issuetracker.google.com/issues/181332856 (дата обр. 05.05.2022)

[30] Bassett L. Introduction to JavaScript object notation: a to-the-point guide to JSON. – "O'Reilly Media, Inc. 2015.

[31] Репозитории kotlinx.serialization. URL: https://github.com/Kotlin/kotlinx.serialization (дата обр. 05.05.2022)

[32] Gruber J. Markdown: Syntax. URL: http://daringfireball.net/projects/markdown/syntax. Retrieved on June. – 2012. – T. 24. – C. 640.

[33] Helmut Kopka and Patrick W. Daly. Guide to LATEX. Fourth edition, Addison-Wesley, Boston, MA, USA, 2004. xii+597 pp. ISBN 0-321-17385-6, p. 3

[34] Mi lkowski, Marcin Lipski, Jaroslaw. (2011). Using SRX Standard for Sentence Segmentation.

[35] Фридл Дж. Регулярные выражения. Библиотека программиста. — СПб.: Питер, 2001. 352 с.

[36] Репозиторий с кодом работы. URL: `https://github.com/jeinygroove/highlight-editor-multiplatform` (дата обр. 23.05.2022)