Содержание

[Введение 4](#_Toc468543769)

[1 Техническое задание 5](#_Toc468543770)

[2 Исследовательская часть 6](#_Toc468543770)

[2.1 Выявление заинтересованных сторон и их интересов 6](#_Toc468543771)

[2.2 Apache UIMA 7](#_Toc468543771)

[2.3 Google Cloud Speech API 8](#_Toc468543772)

[3 Конструкторская часть 10](#_Toc468543777)

[3.1 Выбор технических решений, удовлетворяющих интересам ЗС 10](#_Toc468543778)

[3.2 Диаграмма компонентов проекта 11](#_Toc468543778)

[3.3 Структура проекта 12](#_Toc468543779)

[3.4 Диаграмма классов 13](#_Toc468543780)

[4 Технологическая часть 15](#_Toc468543782)

[4.1 Запуск проекта 15](#_Toc468543783)

[4.2 Создание бинарной сборки системы 17](#_Toc468543784)

[4.3 Анализ исходного кода с помощью метрик качества 17](#_Toc468543785)

[4.4 Анализ зависимостей в коде системы 19](#_Toc468543786)

[4.5 Тестирование на корректность работы 21](#_Toc468543786)

[4.6 Оценка производительности 22](#_Toc468543786)

[Заключение 23](#_Toc468543787)

[Список литературы 24](#_Toc468543788)

[Приложение 1 25](#_Toc468543789)

# Введение

Целью курсовой работы является разработка Apache UIMA компонента, распознающего наличие речи и аннотирующего аудиоданные. Данная задача выполняется в два этапа: на первом этапе происходит запись аудиоданных, которыми является человеческая речь на русском языке, на втором – распознавание этих данных с помощью Google Cloud Speech API и вывод результатов распознавания.

Также в ходе курсовой работы необходимо реализовать объединение двух аннотаторов с помощью агрегатора, то есть выполнить объединение двух Analysis Engine (формирование Aggregate Analysis Engine) с помощью XML дескриптора.

1. Техническое задание

Разработка Apache UIMA компонента, распознающего наличие речи и аннотирующего аудио данные:

* изучить соответствующие системы;
* спроектировать интерфейс компонента;
* реализовать компонент;
* спроектировать JUnit тесты, провести тестирование;
* описать требования, конструкцию, особенности сборки и запуска в документации.

1. Исследовательская часть
   1. Выявление заинтересованных сторон и их интересов

В таблице 1 представлены результаты выявления и начального анализа заинтересованных сторон (ЗС) и их интересов по отношению к системе.

Таблица 1. Заинтересованные стороны и их интересы по отношению к системе

|  |  |
| --- | --- |
| Заинтересованные стороны | Интересы заинтересованных сторон |
| Пользователь системы умного дома | П1 Быстрое и точное распознавание речи.  П2 Стабильная работа системы.  П3 Близкая к линейной масштабируемость аннотатора при увеличении размера аудио файла и потока аудио файлов.  П4 Система осуществляет распознавание речи даже при зашумленном фоне или неточном произнесении слов. |
| Разработчик аннотатора (новый разработчик аннотатора) | Р1 Быстрое понимание принципа работы аннотатора, осуществляющего распознавание и аннотирование речи.  Р2 Быстрая сборка и запуск тестового пайплайна с примером.  Р3 Быстрое внесение изменений и проверка того, что они не нарушают работу существующего функционала. |
| Владелец опенсорсного проекта (project owner) | В1 Быстрая и полная передача исходного кода, настроек, документов.  В2 Возможность в дальнейшем усложнять реализацию аннотатора, например, распознавать речь, произносимую на разных языках. |
| Интегратор аннотаторов | И1 Быстрая настройка пайплайна UIMA из нескольких аннотаторов.  И2 Быстрое нахождение источника ошибок в системе с большим количеством аннотаторов. |

* 1. Apache UIMA

UIMA – это составной программный пакет, архитектура для разработки, исследования, построения и развертывания мультимодальной аналитики для анализа неструктурированной информации и её интеграции с поисковыми технологиями, разработанными IBM. Исходный код для эталонной реализации этого фреймворка был размещен на SourceForge, а позже на сайте Apache Software Foundation.

Analysis Engine – это программа, которая анализирует артефакты (т.е. документы) и вычленяет из них необходимую информацию.

Analysis Engines строятся на основе блоков, называемых аннотаторами (*annotators*). Аннотатор – это компонент, который содержит в себе логику анализа. Аннотатор анализирует артефакт (например, текстовый документ) и создает дополнительную информацию (так называемую metadata) об этом артефакте. Цель UIMA заключается в том, что аннотаторы не должны заботиться ни о чем, кроме своей логики анализа, например, подробности их развертывания или их взаимодействие с другими аннотаторами.

Analysis Engine (AE) может содержать один аннотатор (это называется Primitive AE), или состоять из других AE и таким образом содержать несколько аннотаторов (это называется Aggregate AE). Primitive и Aggregate AE реализуют один и тот же интерфейс и могут взаимозаменяемо использоваться приложениями.

Аннотаторы выдают результаты анализа в виде типизированных структур, называемых Feature Structures, которые представляют собой просто структуры данных, которые имеют тип и набор пар (атрибут, значение). Аннотирование – это особый тип Feature Structures, который привязан к области анализируемого артефакта (например, диапазон текста в документе).

Все Feature Structures, включая аннотации, представлены в UIMA Common Analysis Structure(CAS). CAS – это центральная структура данных, через которую взаимодействуют все компоненты UIMA. В комплект поставки UIMA SDK входит простой в использовании интерфейс Java для CAS, который называется JCas. JCas представляет каждую Feature Structure как объект Java. На рисунке 1 наглядно представлен принцип работы технологии UIMA.

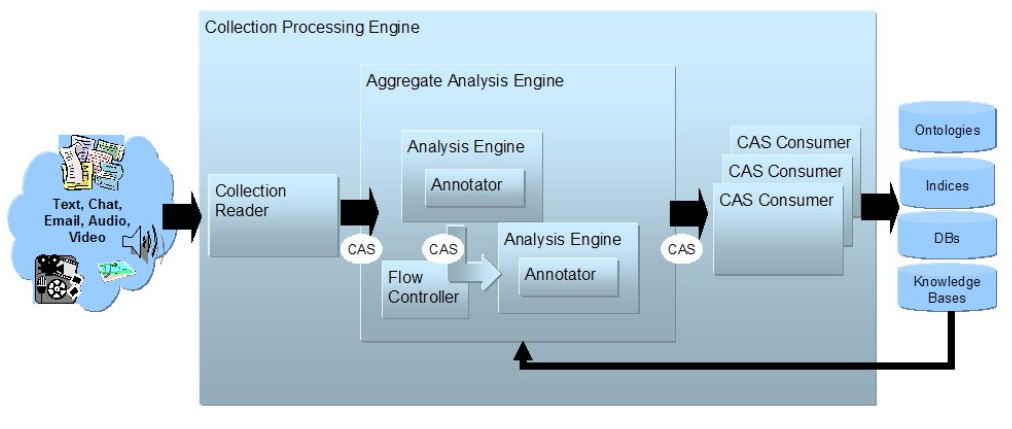


Рисунок 1 – Процесс анализа данных с помощью технологии UIMA

Каждое представление артефакта называется субъектом анализа, сокращенно Sofa (Subject of Analysis). Metadata, которые имеют явные обозначения областей артефакта, к которым они применяются, всегда связаны с конкретным Sofa. Например, аннотация по тексту указывает две функции – начало и конец, которые представляют смещения символов в анализируемой текстовой строке. Данные Sofa могут быть представлены в виде Java Unicode Strings, Feature Structure arrays примитивных типов или URI (Uniform Resource Identifier), который ссылается на удаленные данные, доступные через сетевое соединение.

* 1. Google Cloud Speech API

Google Cloud Speech API позволяет разработчикам преобразовывать аудио в текст, применяя мощные модели нейронных сетей в простом использовании API (Application programming interface). API поддерживает и распознает более 110 языков и диалектов для поддержки глобальной базы пользователей. Существует возможность транскрибировать текст, надиктованный пользователями приложения, задействовать голосовое управление или транскрибировать аудиофайлы. На рис.1 показан пример использования данного API.

Для того, чтобы использовать аутентификацию в Speech API необходимо получить ключ (Google Cloud Platform Console API key). Так же, для этих целей, можно использовать сервисный аккаунт. В данной работе применяется именно этот способ. После создания сервисного аккаунта с использованием персональных данных разработчика осуществляется получение ключа в виде JSON файла. При использовании client library для вызова Speech API необходимо задействовать механизм Application Default Credentials (ADC). Этот механизм лучше всего подходит для случаев, когда обращение должно иметь одинаковый уровень идентификации и авторизации для приложения независимо от пользователя. Это рекомендуемый подход для авторизации вызовов Google Cloud APIs.

,

где PATH\_TO\_KEY\_FILE необходимо заменить на директорию расположения полученного JSON файла.



Рисунок 2 – Пример, показывающий использование client library.

1. Конструкторская часть
   1. Выбор технических решений, удовлетворяющих интересам ЗС

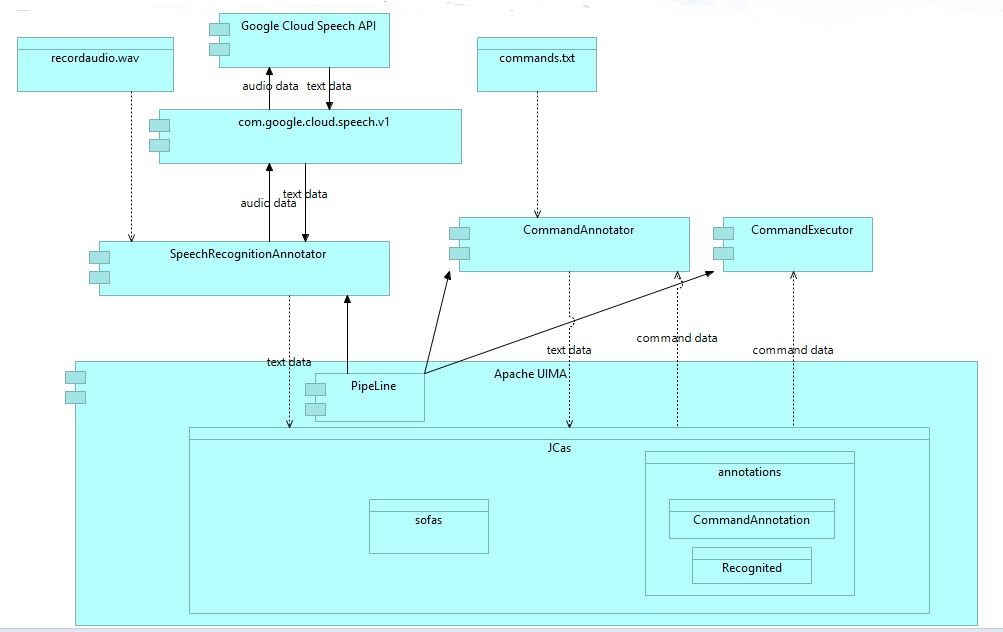
В таблице 2 представлены результаты выбора технических решений, позволяющие удовлетворить интересы заинтересованных сторон по отношению к системе.

Таблица 2. Технические решения, удовлетворяющие интересам ЗС

|  |  |
| --- | --- |
| Интересы заинтересованных сторон | Технические решения |
| П1 Быстрое и точное распознавание речи.  П2 Стабильная работа системы.  П3 Близкая к линейной масштабируемость аннотатора при увеличении размера аудио файла и потока аудио файлов.  П4 Система осуществляет распознавание речи даже при зашумленном фоне или неточном произнесении слов. | Распознавание осуществляется с помощью Google Cloud Speech API.  Проверка корректности работы осуществляется с помощью юнит-тестов в процессе разработки. |
| Р1 Быстрое понимание принципа работы аннотатора, осуществляющего распознавание и аннотирование речи.  Р2 Быстрая сборка и запуск тестового пайплайна с примером.  Р3 Быстрое внесение изменений и проверка того, что они не нарушают работу существующего функционала. | Самодокументируемый код с использованием Google Code Style.  С системой будет поставляться пример и ReadMe с инструкциями по сборке и запуску примера.  Исходный код будет структурирован по пакетам. Повторно-используемые методы будут вынесены в родительские классы. |
| В1 Быстрая и полная передача исходного кода, настроек, документов.  В2 Возможность в дальнейшем усложнять реализацию аннотатора, например, распознавать речь, произносимую на разных языках. | Код и настройки разрабатываемого аннотатора будут находиться в системе GitHub. Контроль версий будет производиться с использованием системы Git.  Для модульного тестирования будет использована система JUnit.  Будет использована система сборки Maven, интегрирующаяся в инструменты непрерывной интеграции.  Для обеспечения расширяемости код аннотатора и код примера тестового пайплайна будут разделены по разным проектам, зависимости между которыми будут только через стандартные интерфейсы Apache UIMA. |
| И1 Быстрая настройка пайплайна UIMA из нескольких аннотаторов.  И2 Быстрое нахождение источника ошибок в системе с большим количеством аннотаторов. | Будут использованы стандартные интерфейсы и дексрипторы аннотаторов и типов UIMA. Создание и управление жизненным циклом объектов аннотаторов будет производиться Apache UIMA.  Будут использованы стандартные средства логирования отладочной информации и сообщений об ошибках Apache UIMA. |

* 1. Диаграмма компонентов

Проект, в котором осуществляется распознавание наличия речи и аннотирование аудиоданных, используется в рамках более крупного проекта управления определенными устройствами с помощью голосовых команд. Диаграмма компонентов проекта голосового управления представлена на рисунке 3.

Рисунок 3 – Диаграмма компонентов для проекта голосового управления

Изначально запись с диктофона анализируется модулем SpeechRecognitionAnnotator, который позволяет транскрибировать аудиофайлы или текст, надиктованный пользователями приложения. Далее эти данные поступают на вход модуля распознавания команд, который возвращает список найденных текстовых команд, которые в последствии будут обработаны.

* 1. Структура проекта

В данном разделе рассматривается структура проекта, позволяющего распознавать наличие речи и аннотировать аудиоданные.

* + 1. Класс SpeechRecognitionAnnotator

По умолчанию для аннотаторов, использующих JCas, существует реализация интерфейса, называемая JCasAnnotator\_ImplBase, которая реализует все необходимые методы, за исключением метода process().

Таким образом, класс SpeechRecognitionAnnotator является аннотатором данных, позволяющим транскрибировать текст, надиктованный пользователями приложения, он наследует класс JCasAnnotator\_ImplBase и переопределяет единственный метод process(JCas aJCas).

В теле метода process(JCas aJCas) создается экземпляр класса SpeechRecordLibrary,и с помощью метода recognize() осуществляется перевод аудио файла в текст. Также внутри класса SpeechRecognitionAnnotator создается экземпляр класса Recognited, и с помощью метода addToIndexes() новая аннотация добавляется в xmi файл.

Для оценки производительности аннотатора используется метод System.nanoTime(), возвращающий время в наносекундах.

* + 1. Класс SpeechRecordLibrary

Класс SpeechRecordLibrary содержит метод recognize(). В теле метода recognize() инициализирован экземпляр класса RecognitionConfig.newBuilder(), в котором задаются параметры файла, и RecognitionAudio.newBuilder(), который содержит в себе байтовое представление файла.

* + 1. Класс JavaSoundRecorder

Класс JavaSoundRecorder содержит методы main(String[] args), start() и finish(). Метод main(String[] args) является запускающим, в нем создается экземпляр класса JavaSoundRecorder, создается поток, который необходим для того, чтобы подождать указанное время (6 с), в течение которого осуществляется запись. В методе start() осуществляется запись файла с помощью экземпляра класса AudioInputStream. Метод finish() останавливает запись.

* + 1. Класс Recognited и система типов

Класс Recognited автоматически генерируется инструментами технологии Apache UIMA, для этого необходимо создать систему типов. Аннотатору SpeechRecognitionAnnotator требуется только один тип аннотации Recognited. Необходимо создать файл SpeechAnnotator.xml, открыть его в редакторе Component Descriptor Editor и добавить тип аннотации: указать полное название класса и супер класс (рисунок 4).

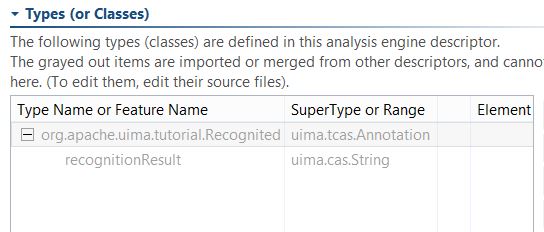


Рисунок 4 – Объявление системы типов файла SpeechAnnotator.xml

Тип Recognited наследует от uima.tcas.Annotation поле sofa (subject of analysis). Также необходимо добавить к этому типу поле recognitionResult. Чтобы обозначить, что работа будет осуществляться со строкой, uima.cas.String выбирается как супер класс.

* 1. Диаграмма классов

В данном разделе приведена диаграмма классов для той части проекта, которая позволяет аннотировать аудио данные. Класс SpeechRecognitionAnnotator создает аннотации в зависимости от входных аудиоданных, т.е. он может вызвать конструктор класса Recognited несколько раз. На диаграмме классов отражена связь автоматически сгенерированных классов Recognited и Recognited\_Type, а также связь класса SpeechRecognitionAnnotator и SpeechRecordLibrary.

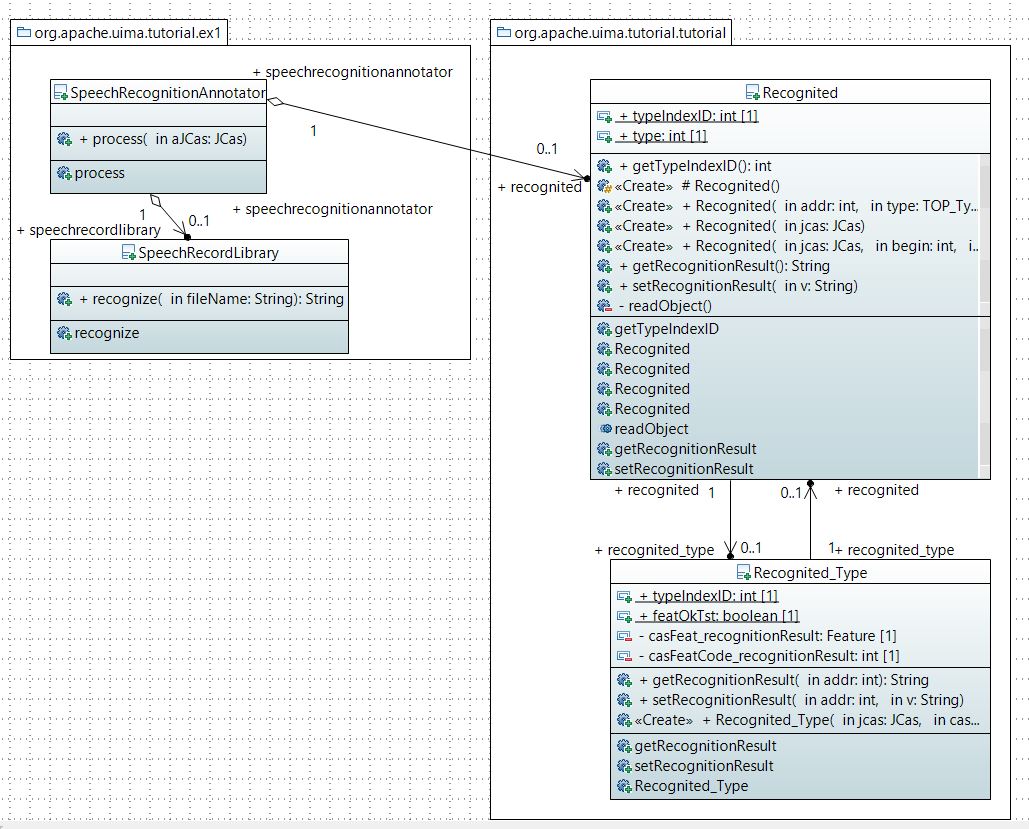


Рисунок 5 – Диаграмма классов

1. Технологическая часть
   1. Запуск проекта

Данный проект доступен по следующей ссылке: https://github.com/ekatmaximenko2016/final-project.

Сперва осуществляется запись аудио файла в формате wav с помощью класса JavaSoundRecorder. Записанный файл содержится в папке проекта src/test, содержимое аудио файла представляет собой речь на русском языке. Объединение аннотаторов было осуществлено с помощью агрегатора. Для этого в разделе Overview для Engine Type выбран тип Aggregate (рисунок 6), а также в разделе Aggregate добавлены аннотаторы SpeechRecognitionAnnotator и CommandAnnotator (рисунок 7).

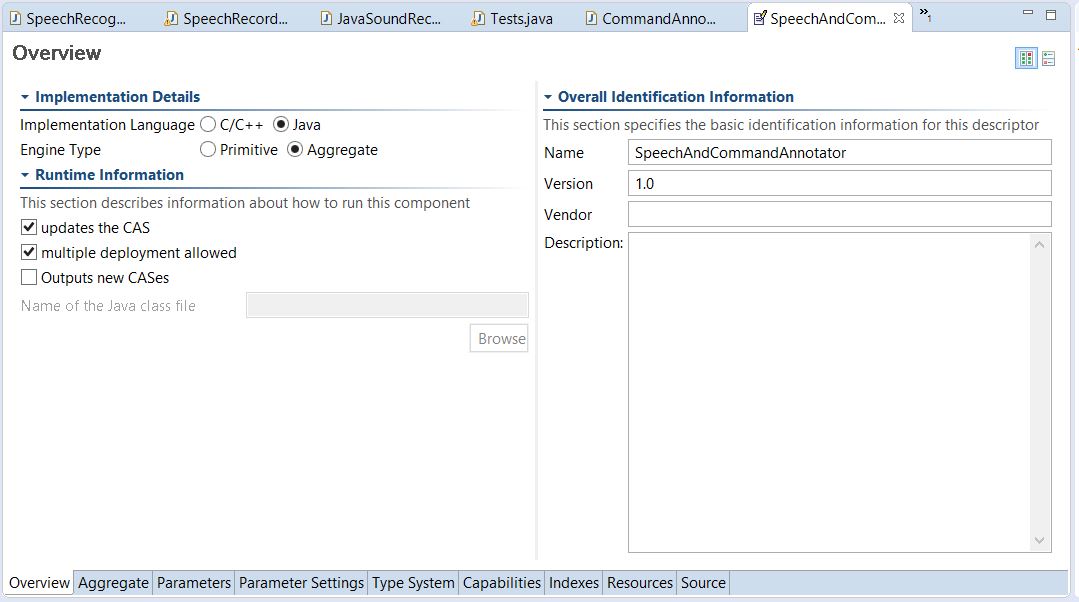


Рисунок 6 – Выбор типа Engine Type

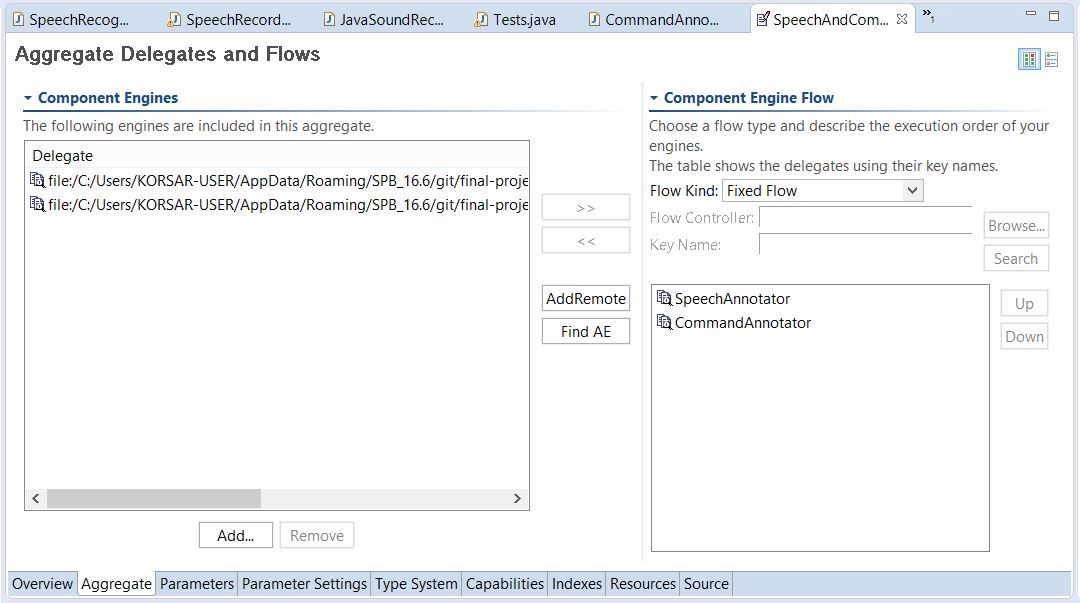


Рисунок 7 – Добавление аннотаторов в разделе Aggregate

Для запуска агрегатора необходимо выполнить первоначальную настройку: добавить Run Configuration для UIMA Analysis Engine (рисунок 8), а также в разделе Environment добавить переменную GOOGLE\_APPLICATION\_CREDENTIALS (рисунок 9):

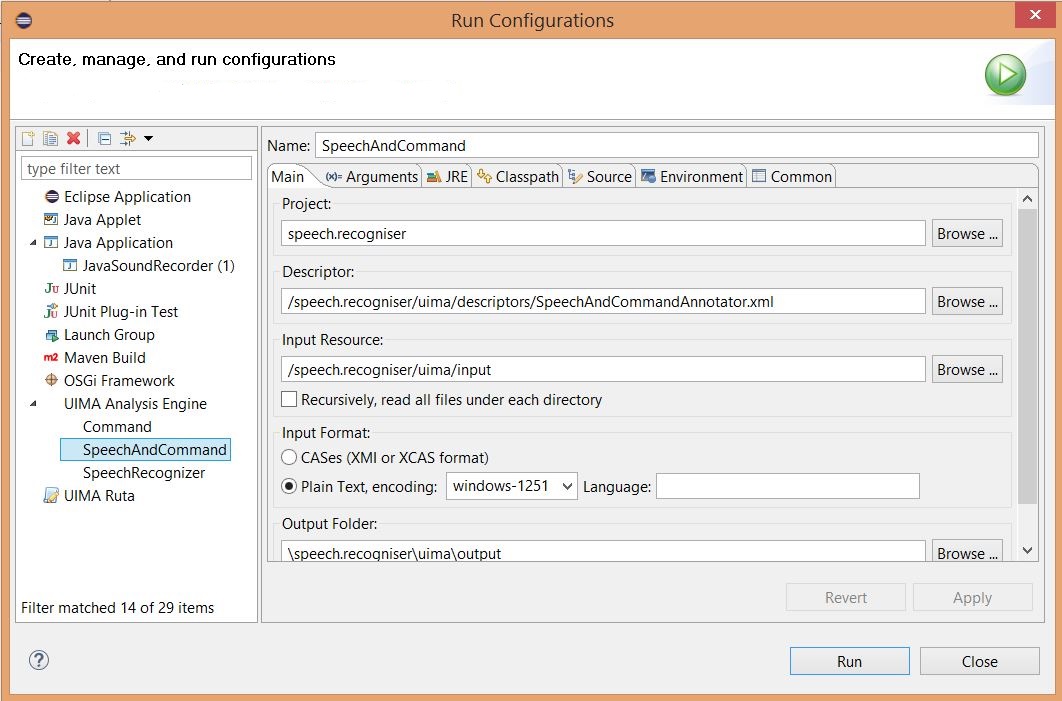


Рисунок 8 – Создание Run Configuration для UIMA Analysis Engine

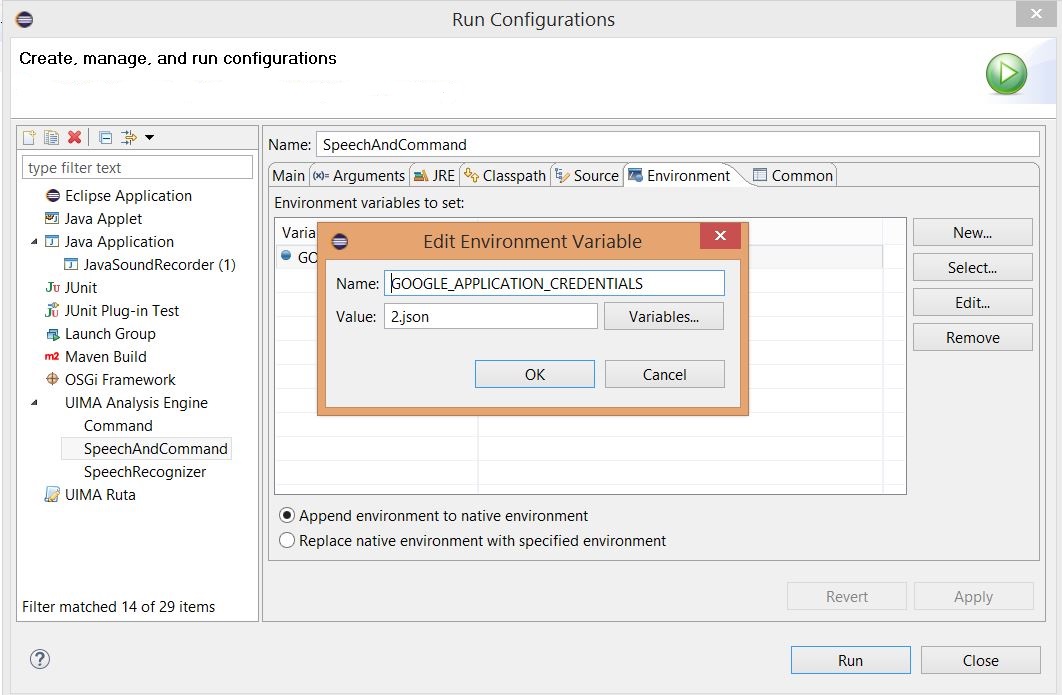


Рисунок 9 – Добавление переменной GOOGLE\_APPLICATION\_CREDENTIALS

* 1. Создание бинарной сборки системы

Для сборки проекта используется фреймворк для автоматизации сборки Maven. В корневой папке проекта speech.recogniser находится файл pom.xml, который содержит в себе все необходимые зависимости, поэтому не требуется подгружать другие библиотеки. Также в данном проекте используется фреймворк Apache UIMA.

* 1. Анализ исходного кода с помощью метрик качества

На рисунке 10 отображен список всех метрик по разделам. Всего имеется четыре раздела:

* метрики количества (Count);
* метрики сложности (Complexity);
* метрики Роберта Мартина (Robert C. Martin);
* метрики Чидамбера-Кемерера (Chidamber & Kermerer).

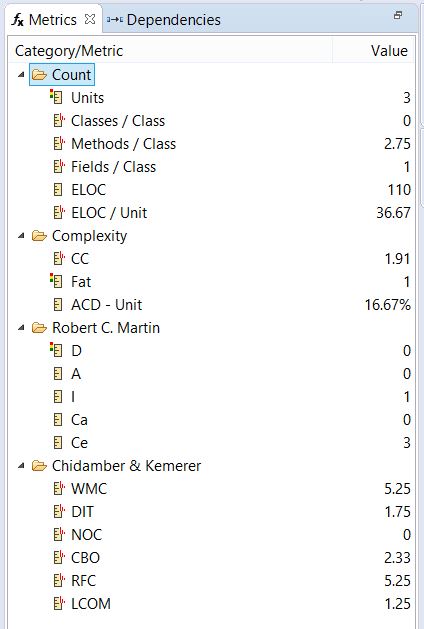


Рисунок 10 – Значения метрик

Первый раздел с метриками количества (Count) содержит следующие метрики:

* количество классов верхнего уровня (Unit);
* среднее число внутренних классов на класс (Classes / Class);
* среднее число методов в классе (Methods / Class);
* среднее число полей в классе (Fields / Class);
* число строчек кода (ELOC);
* число строчек кода на модуль (ELOC / Unit).

Второй раздел с метриками сложности (Complexity) содержит следующие метрики:

* средняя циклическая сложность (CC);
* метрика Fat (Fat);
* средняя зависимость компонентов между модулями (ACD - Unit).

Третий раздел с метриками Роберта Мартина содержит следующие метрики:

* нормализованное расстояние от основной последовательности (D);
* абстрактность (A);
* нестабильность (I);
* число афферентных соединений (Ca);
* число эфферентных соединений (Ce).

Последний раздел с метриками Чидамбера-Кемерера содержит следующие метрики:

* средняя длина метода на класс (WMC);
* средняя глубина наследования (DIT);
* среднее количество классов-наследников (NOC);
* среднее число соединений класса (CBO);
* среднее число методов, которые потенциально могут быть выполнены в ответ на сообщение, полученное объектом этого класса (RFC);
* отсутствие единства методов (LCOM).

На основании полученных значений метрик можно сделать вывод о том, что программа имеет наилучшую сбалансированность между абстрактностью и нестабильностью, а число методов в каждом классе не превышает оптимально допустимого количества. Остальные величины метрик также не превышают допустимых значений.

На рисунке 11 показано соотношение классов проекта по их размеру. Видно, что самый большой размер имеет класс JavaSoundRecorder.

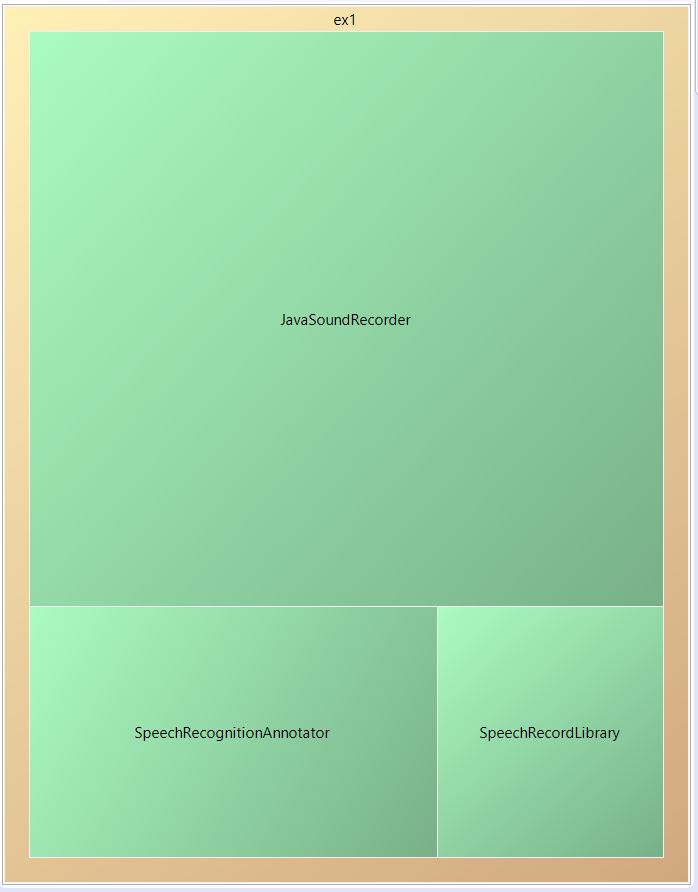
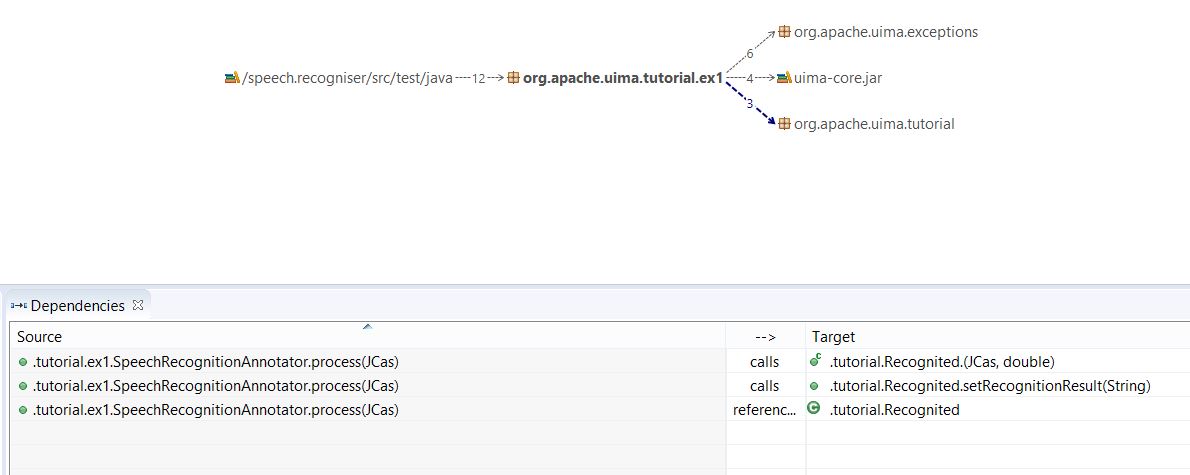


Рисунок11 – Соотношение классов по размеру

* 1. Анализ зависимостей в коде системы

Рисунки 10, 11, 12 характеризуют зависимости между пакетами проекта и внешними пакетами.

Рисунок 12 – Зависимости между пакетами org.apache.uima.tutorial.ex1 и org.apache.uima.tutorial



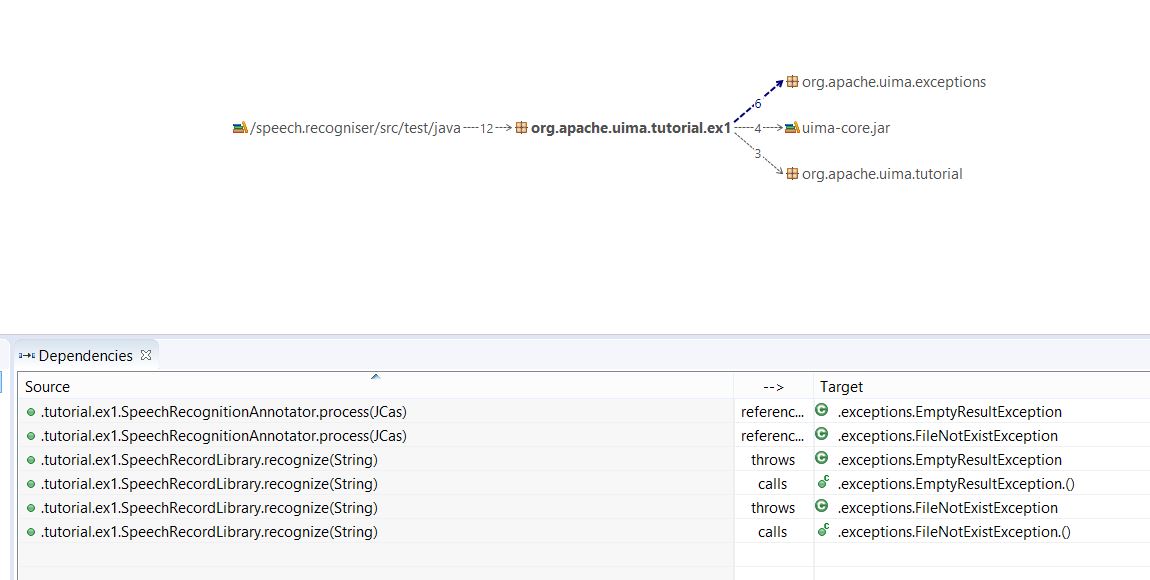


Рисунок 13 – Зависимости между пакетами src/test/java org.apache.uima.tutorial.ex1 и org.apache.uima.tutorial.ex1

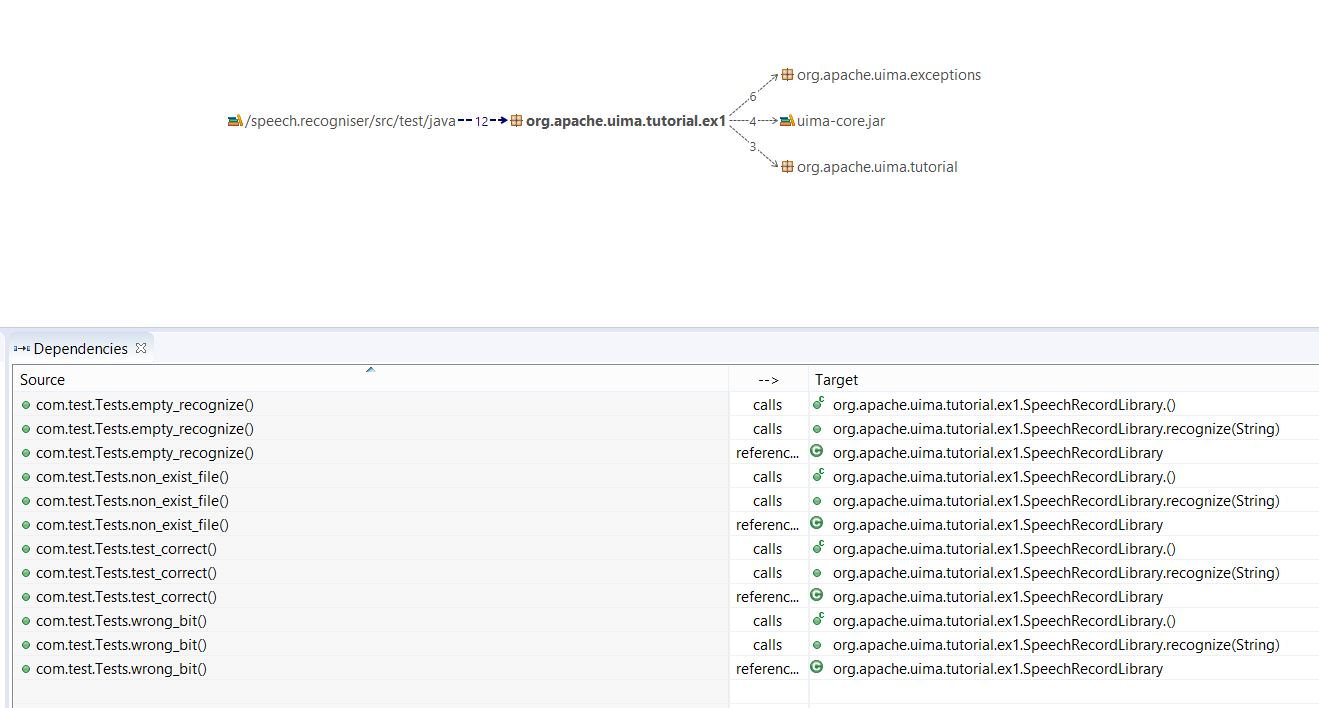


Рисунок 14 – Зависимости между пакетами org.apache.uima.tutorial.ex1 и src/test/java

* 1. Тестирование на корректность работы

Модульное тестирование, или юнит-тестирование (unit testing) – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы. Тесты реализуются для каждой нетривиальной функции или метода.

В рамках курсовой работы было реализовано четыре юнит-теста, позволяющих проверить, что программа верно работает как на корректных, так и на некорректных данных.

* Юнит-тест test\_correct(), позволяющий проверить, не привело ли очередное изменение кода к возникновению ошибок. Он обрабатывает аудио файл с заранее известным текстом и сравнивает распознанный текст с тем, что должен был получиться.
* Юнит-тест empty\_recognize(), обрабатывающий аудио файл, в котором не содержится ни одного слова, и позволяющий проверить, будет ли корректно работать аннотатор, если на вход будет подан пустой аудио файл.
* Юнит-тест non\_exist\_file(), позволяющий проверить, будет ли корректно работать аннотатор, если файл для аннотирования данных не будет существовать.
* Юнит-тест wrong\_bit(), обрабатывающий аудио файл неверного формата и позволяющий проверить, будет ли корректно работать аннотатор, если на вход будут переданы некорректные данные.

На рисунке 15 видно, что все тесты были выполнены успешно. Это значит, что аннотатор работает корректно при всех условиях, описанных выше.

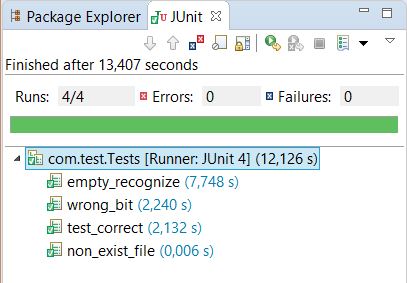


Рисунок 15 – Результат работы тестов для проверки корректности работы аннотатора

* 1. Оценка производительности

Для вычисления зависимости времени работы аннотатора SpeechRecognitionAnnotator от размера файла (количества слов, содержащихся в файле) применяется метод System.nanoTime(), использующий специальные счетчики, не связанные с системными часами и возвращающий время в наносекундах.  Производительность этого метода достаточно сильно зависит от используемого «железа». Для оценки производительности используются файлы, записанные в течение 6000, 12000, 18000, 24000, 30000, 36000, 42000, 48000, 54000, 60000 мс, в среднем за каждые 6 секунд человек произносит 4 слова. Для 10 выбранных значений было определено время работы аннотатора и построен график ( рисунок 16):



Рисунок 16 – Зависимость времени работы аннотатора SpeechRecognitionAnnotator от размера файла

По результатам тестирования (рисунок 16) видно, что зависимость времени работы аннотатора SpeechRecognitionAnnotator от размера файла можно назвать линейной. Кривизну графика можно объяснить погрешностью вычисления времени работы программы из-за неточности метода System.nanoTime(), а также качества интернет соединения.

Тестирование производительности аннотатора CommandAnnotator в зависимости от числа команд было осуществлено ранее.

Заключение

В ходе курсовой работы была изучена технология Apache UIMA, были рассмотрены ее преимущества и особенности. В рамках курсовой работы был спроектирован, реализован и протестирован аннотатор, позволяющий распознавать в любом входном аудио файле слова на русском языке и представлять распознанные слова в виде текста.

Также в ходе курсовой работы было осуществлено объединение двух аннотаторов, то есть формирование Aggregate Analysis Engine с помощью XML дескриптора.

Список литературы

[1] UIMA Tutorial and Developers' Guides: [Электронный ресурс]. Режим доступа: <http://uima.apache.org/d/uimaj-2.4.2/tutorials_and_users_guides.pdf> – Дата обращения: 27.10.2017

# Приложение 1

Листинг 1 - класс CommandAnnotator

package org.apache.uima.tutorial.ex1;

import org.apache.uima.analysis\_component.JCasAnnotator\_ImplBase;

import org.apache.uima.exceptions.EmptyResultException;

import org.apache.uima.exceptions.FileNotExistException;

import org.apache.uima.jcas.JCas;

import org.apache.uima.tutorial.Recognited;

/\*\*

\* Example annotator that detects room numbers using Java 1.4 regular expressions.

\*

\*/

public class SpeechRecognitionAnnotator extends JCasAnnotator\_ImplBase {

/\*\*

\* @see JCasAnnotator\_ImplBase#process(JCas)

\*/

public void process(JCas aJCas) {

long start = System.nanoTime();

//System.out.println(System.nanoTime());

// get annotation indexes

String fileName = "src/test/recordaudio.wav";

System.out.println("Speech worked");

SpeechRecordLibrary speech = null;

try {

speech = new SpeechRecordLibrary();

} catch (Exception e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

String resRecog = null;

try {

resRecog = speech.recognize(fileName);

} catch (EmptyResultException | FileNotExistException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

Recognited recogRes = new Recognited(aJCas, 1.0);

recogRes.setRecognitionResult(resRecog);

recogRes.addToIndexes();

long end = System.nanoTime();

long elapsedTime = end - start;

double seconds = (double)elapsedTime / 1000000000.0;

System.out.println(seconds);

//System.out.println(System.nanoTime());

}

}