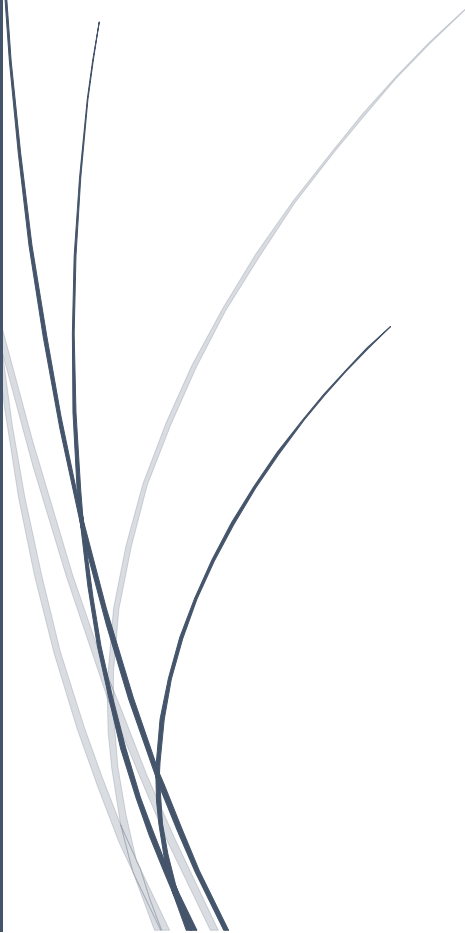


A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the year 2016.

2016

Trabajo Práctico Final

Diseño de Sistemas de Software

Several thin, curved lines in dark blue and light grey originate from the bottom left corner and curve upwards and to the right.

Matías Prado – Camila Robles

PROFESORES: ANDRÉS DIAZ PACE – ALEJANDRO RAGO

Contenidos

Glosario	2
Introducción	3
Contexto	4
Desarrollo	5
Background de la arquitectura.....	5
Vista general del sistema	5
Funcionalidad.....	5
Requerimientos significativos conductores del diseño.....	6
Background de la Solución	6
Enfoques arquitectónicos	6
Arquitectura	6
Decisiones de diseño.....	7
Decisiones de implementación	9
Implementación	9
Apache UIMA	10
UIMA Ruta.....	11
Estándares de programación	11
Versionado	15
Bibliotecas y frameworks utilizados.....	15
Aplicación	16
Test	20
Model	20
View.....	20
Controller	21
Conclusión	23

Glosario

AE: Del acrónimo inglés *Analysis Engine*. Son estructuras de UIMA encargadas de analizar el texto ubicando automáticamente los metadatos que se encuentran en él y etiquetándolos.

ANOTADOR: Son programas de JAVA que se utilizan en UIMA y que tienen elementos concretos de análisis para proceder con el etiquetamiento de una entidad determinada. **CAS CONSUMER:** es el componente de UIMA que analiza los datos de las entidades de interés de acuerdo a los análisis del AE.

CLOUD: Hace referencia a la nube de palabras creada a partir del análisis y procesamiento de una o varias clases Java. (Véase también Word Cloud)

COLLECTION READER: Es una interfaz del CPE de UIMA para acceder a los documentos que serán analizados.

DESCRIPTOR: Es un recurso utilizado por UIMA que describe el documento analizado por el sistema en formato XML.

JAVA: Es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems.

LEMMATIZE: Es un proceso lingüístico que consiste en hallar el lema correspondiente dada una forma flexionada (es decir, en plural, en femenino, conjugada, etc.). El lema es la forma que por convenio se acepta como representante de todas las formas flexionadas de una misma palabra. Es decir, el lema de una palabra es la palabra que nos encontraríamos como entrada en un diccionario tradicional: singular para sustantivos, masculino singular para adjetivos, infinitivo para verbos.

TAG CLOUD: Es una representación visual de texto, usualmente utilizado para visualizar metadata (tags). Nombre por el cual se conoce a lo que en este trabajo se llama Word Cloud.

UIMA: Del acronym inglés Unstructured Information Management Architecture. Es una arquitectura de componentes de software para el desarrollo de aplicaciones que faciliten el análisis de información no estructurada (como los textos ó audio) y la integración con otras tecnologías. Fue desarrollada por IBM. Actualmente es un proyecto Apache.

URI: Del acrónimo inglés Uniform Resource Identifier. Se trata de un conjunto de caracteres utilizados para identificar un recurso en Internet.

URL: Del acrónimo inglés Uniform Resource Locator. Se trata de un conjunto de caracteres utilizados para determinar la ubicación de un recurso determinado en Internet.

WORD CLOUD: La nube de palabras creada a partir del análisis y procesamiento de uno o varios paquetes en lenguaje Java. (Véase también Cloud)

XML: Del acrónimo inglés eXtensible Markup Lenguaje. Lenguaje de marcado extensible. Lenguaje de marcado derivado del SGML que permite crear estructuras de datos asociadas o no a documentos de texto.

Introducción

El análisis estático de software es un tipo de análisis que se realiza sin ejecutar el programa. El término se aplica, generalmente, a los análisis realizados por una herramienta automática. El análisis realizado por un humano es llamado comprensión de programas y, en algunos casos, se puede usar una combinación de ambos.

Los *patrones de diseño* son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

La existencia de patrones y de estándares de programación permite saber de qué se trata un paquete o una clase, sin la necesidad de leer el código completo. Con solo saber cuáles son las palabras más utilizadas, uno podría hacerse la idea de lo que hace una clase o un paquete.

El objetivo de este trabajo era realizar un sistema que analice paquetes de código java y, mediante el procesamiento de texto neurolingüístico, genere una *tag cloud* (o nube de palabras). Esto permitiría al usuario ver, de manera gráfica y clara, el contenido de un paquete o clase particular, donde el tamaño es mayor para las palabras que aparecen con más frecuencia. De esta manera es posible acceder al contenido más importante rápidamente.

Contexto

La principal característica de Java es la de ser un lenguaje compilado e interpretado. Como cualquier lenguaje de programación, tiene su propia estructura, reglas de sintaxis, convenciones y paradigma de programación. El paradigma de programación del lenguaje Java se basa en el concepto de programación orientada a objetos (OOP) que las funciones del lenguaje soportan.

Estructuralmente, el lenguaje Java comienza con *paquetes*. Un paquete es el mecanismo de espacio de nombres del lenguaje Java. Dentro de los paquetes se encuentran las clases y dentro de las clases se encuentran métodos, variables, constantes, entre otros.

Existen herramientas que permiten analizar texto para reconocer secciones y estructuras que son de importancia a la hora de comprender el código. Apache UIMA es una plataforma abierta que identifica componentes para cada función de análisis conceptualmente diferente, y garantiza que estos componentes se puedan reutilizar y combinar fácilmente.

Por otro lado, la herramienta *ClearTK* permite el procesamiento del texto segmentado por UIMA, agrupando palabras con la misma morfología o que comiencen de la misma manera. De esta forma, se obtiene un valor más realista de la cantidad de repeticiones de las palabras para generar la *tag cloud*.

Gracias a lo mencionado anteriormente, es posible la interpretación y visualización de manera gráfica de paquetes o clases de lenguaje Java sin la necesidad de hacer un análisis detallado del código.

Desarrollo

Background de la arquitectura

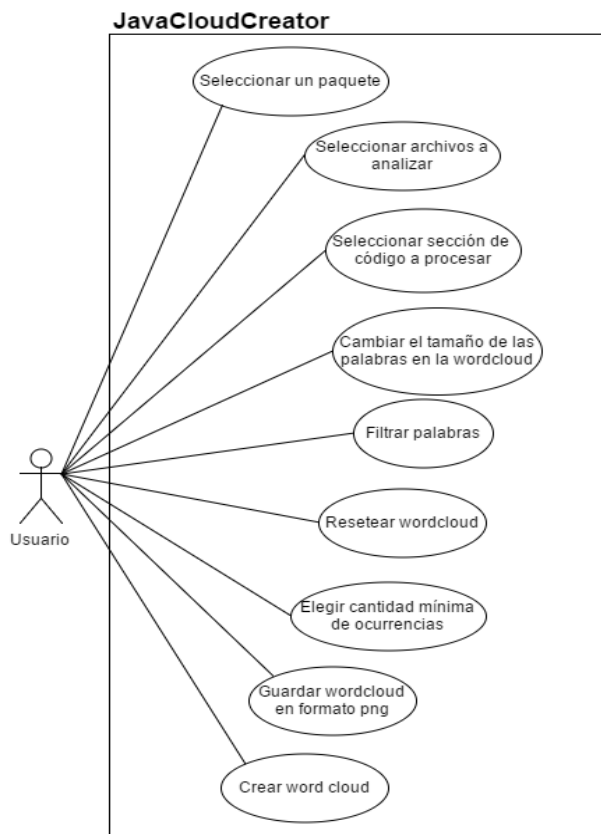
Vista general del sistema

El sistema **JavaCloudCreator** facilita el análisis de paquetes Java, permitiendo al usuario ver, de manera gráfica y clara, el contenido de un paquete o de una clase particular.

Funcionalidad

- **Seleccionar un paquete:** El usuario puede seleccionar un paquete para analizar su contenido.
- **Seleccionar archivos para analizar:** El usuario puede seleccionar los archivos que desea analizar.
- **Cambiar el tamaño de la cloud:** El usuario puede seleccionar el tamaño de las palabras en la *cloud*.
- **Filtrar palabras:** El usuario puede filtrar las palabras que no quiera que aparezcan en la *cloud*.
- **Elegir secciones de código a procesar:** El usuario puede seleccionar qué sección del código desea analizar: paquetes, *imports*, clases, métodos variables y comentarios.
- **Seleccionar la cantidad de mínima de apariciones:** El usuario tiene la capacidad de seleccionar el mínimo de ocurrencias necesarias para aparecer en la *cloud*.
- **Resetear la cloud:** El usuario puede resetear la *cloud*.
- **Guardar como png:** El usuario tiene la capacidad de guardar la *cloud* en formato png.
- **Crear la cloud:** El usuario puede crear una *word cloud* una vez configurado el sistema.

Diagrama de Casos de Uso



Requerimientos significativos conductores del diseño

La herramienta tiene como *architectural drivers* la modificabilidad, extensibilidad, adaptabilidad, eficiencia y portabilidad. Esto se debe a que el lenguaje podría tener modificaciones a lo largo del tiempo y es necesario dar soporte a dichos cambios. Por otro lado, si bien la aplicación hace uso del framework *ClearTK*, en el diseño realizado se contempla la posibilidad de utilizar otros frameworks de análisis del lenguaje natural o incluso analizar texto sólo con operaciones propias del lenguaje de implementación. Se eligió Java como lenguaje de implementación debido a que la aplicación debería funcionar en cualquier sistema operativo, y se hizo uso de Java Concurrency API para lograr mayor eficiencia en el procesamiento de los datos.

Background de la Solución

Enfoques arquitectónicos

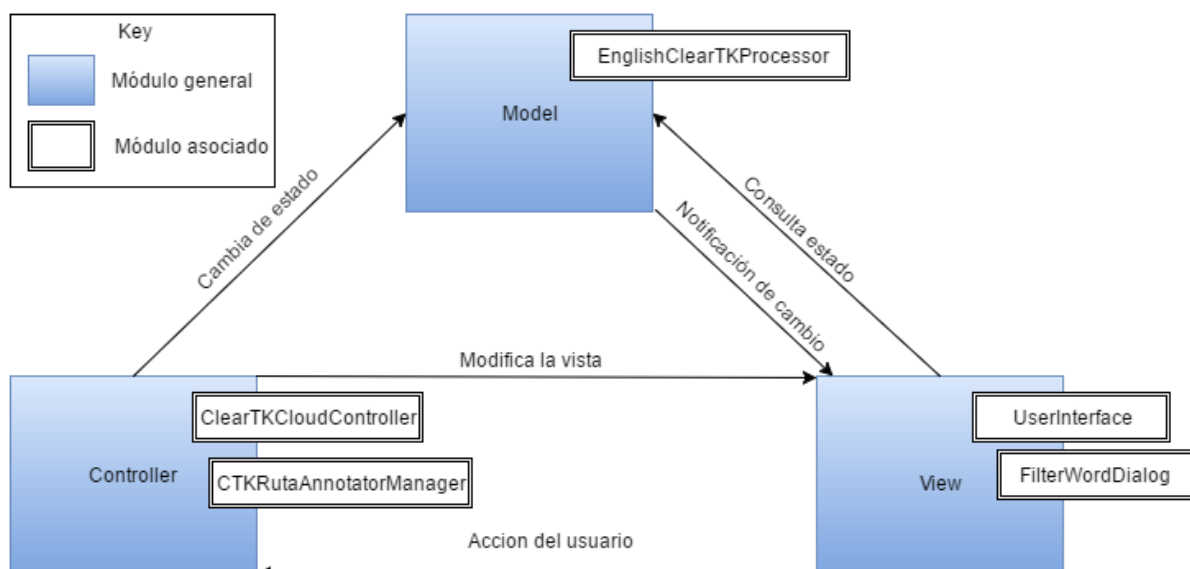
Para poder solventar la tarea de cumplir con los casos de uso mencionados anteriormente fue necesario utilizar patrones arquitectónicos, patrones de diseño y bibliotecas tanto internas como externas de Java.

Arquitectura

Modelo vista controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. En el caso particular de **Java Cloud Creator** la interfaz gráfica es una parte importante de la funcionalidad del software debido a que el mismo debe resultar amigable para el usuario.

Si bien el patrón no fue implementado siguiendo la arquitectura del mismo de manera estricta, se obtuvieron los beneficios del mismo, ya que le brinda al sistema una alta reusabilidad y una clara separación de conceptos, y a su vez tiene como ventaja mayor orden y prolijidad en el código fuente. Estas características facilitan la tarea de desarrollo de aplicaciones, la testeabilidad y su posterior mantenimiento.

Además, se aplican los conceptos de la programación orientada a objetos para lograr una abstracción de los comportamientos en común.



En el diagrama se puede observar claramente la manera en que el modelo es accedido. Ante una petición del usuario se carga una vista en particular. Las vistas presentan los datos de la aplicación que se cargan según la selección del usuario. Estas se componen de diálogos que permiten realizar modificaciones de filtrado, selección de archivos a analizar, secciones de código a procesar y, por su parte, la vista principal **UserInterface** presenta el resultado total del análisis.

Decisiones de diseño

La programación orientada a objetos es un paradigma de programación que usa objetos en sus interacciones para diseñar aplicaciones. Está basada en varias técnicas incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

Se buscó lograr una abstracción que permitiera mejorar tanto la legibilidad del código como atributos de calidad de gran importancia como: modificabilidad, testeabilidad, portabilidad, performance y adaptabilidad.

Los módulos que componen la aplicación son model, view y controller. Estos se encuentran separados en paquetes que permiten una mejor abstracción y distribución más clara de las tareas de cada uno de ellos.

El paquete **view** presenta la vista y cuenta con la clase **UserInterface** que es la encargada de implementar los métodos de interacción con el usuario. La clase **FilterWordDialog** presenta un dialogo que permite seleccionar las palabras que deseen excluirse de la *word cloud*.

El paquete **controller** contiene las clases encargadas de responder a eventos e invocar peticiones al modelo cuando se hace alguna petición del usuario.

Presenta dos interfaces: **CloudController**, encargada de manejar las secciones del código Java que serán procesadas para generar la *word cloud*; y **AnnotatorManager** que se encarga de introducir los token procesados a la *word cloud*.

El paquete **model** contiene la interfaz **NLPAnalyzer**, encargada realizar la ejecución del pipeline, aplicando técnicas NLP y además contiene todos anotadores los cuales se presentan en archivos .ruta y los engine y typesystem contenidos en archivos formato xml.

El diagrama presentado a continuación muestra el diseño orientado a objetos en el que se incluyen paquetes, interfaces y clases. La correcta distribución de las responsabilidades permite al sistema ser más flexible ante posibles modificaciones.

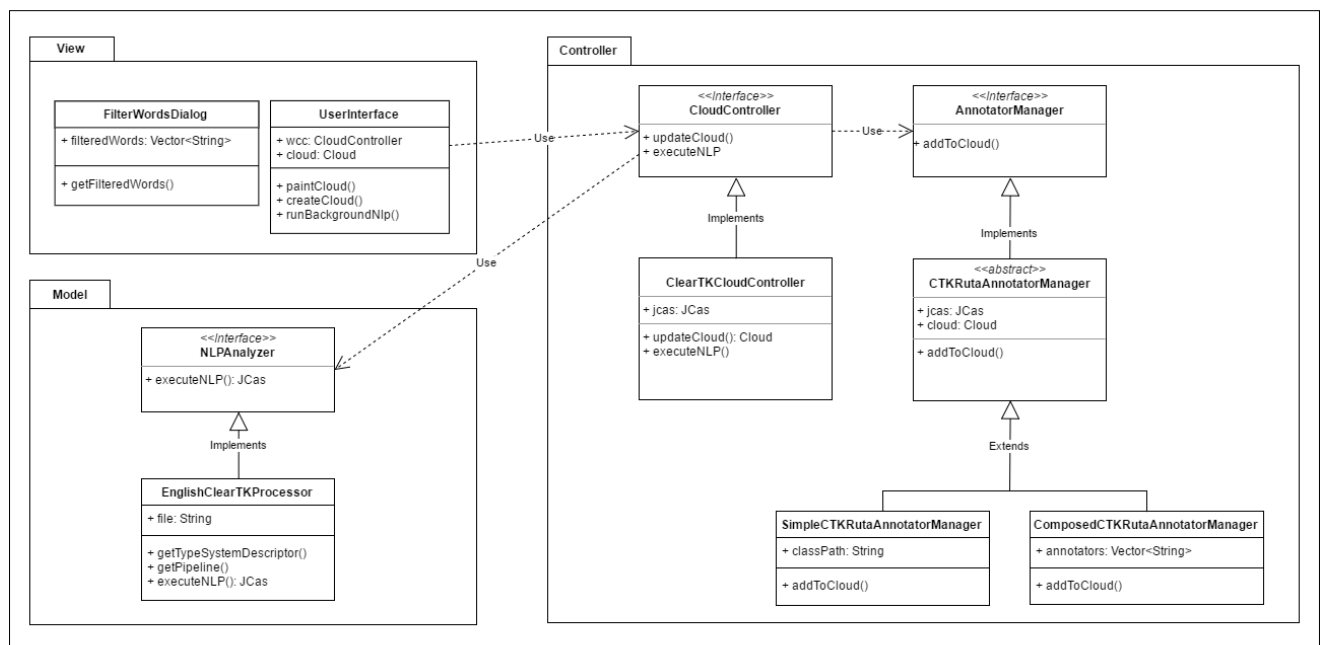


Diagrama de módulos y clases UML

Para la implementación del sistema se tuvieron en consideración los siguientes atributos de calidad:

Modificabilidad

Durante el desarrollo de un sistema pueden surgir cambios inevitables en el código debido a la necesidad de implementar nuevos requerimientos. El diseño de **JavaCloudCreator** se realizó teniendo en cuenta este tipo de situaciones, en las que un simple cambio en la funcionalidad puede implicar un costo muy grande. Las interfaces **NLPAnalyzer**, **CloudController** y **AnnotatorManager**, pueden ser implementadas de manera que el sistema pueda adaptar según los requerimientos sin necesidad de un gran costo de desarrollo.

Performance

Debido a que el procesamiento de código mediante NLP tiene una sobrecarga de recursos, se buscó mejorar el tiempo de respuesta mediante la utilización de **Java Concurrency API**. La misma brinda construcciones concurrentes eficientes, correctas y reutilizables. Por otro lado también proporciona mejoras concurrentes en escalabilidad, rendimiento, legibilidad, mantenibilidad y *thread-safety*. Al facilitar la implementación de aplicaciones concurrentes, se utilizó para crear threads por cada archivo a procesar. Esto permite la ejecución en paralelo de varios pipelines (uno por cada clase Java), mejorando en gran medida la eficiencia global del sistema.

Portabilidad

Se diseñó el sistema con la característica de poder ejecutarse en diferentes plataformas mediante la utilización del lenguaje Java.

Java es un lenguaje de programación de propósito general, concurrente, orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Extensibilidad

El sistema presenta la posibilidad de extender su funcionalidad fácilmente sin generar complicaciones gracias a la implementación de interfaces que abstraen el comportamiento. Es decir, se pueden agregar nuevos requerimientos teniendo como base el software existente sin mayores dificultades.

Adaptabilidad

Desde el comienzo del proyecto se ha hecho énfasis en este atributo de calidad. La idea se centra en tener una interfaz que posibilite el intercambio de técnicas NLP y que el funcionamiento siga siendo el esperado. Las interfaces anteriormente descritas permiten cambiar de procesador NLP sin necesidad de hacer una gran cantidad de cambios en el código de la aplicación.

Decisiones de implementación

Como se mencionó con anterioridad, el procesamiento de código mediante NLP tiene una sobrecarga de recursos, por lo tanto fue de vital importancia buscar un diseño en el cual se redujera al mínimo la cantidad de veces que se necesitaba procesar el mismo archivo.

La solución fue duplicar la *cloud* original. De esta manera la acción de filtrar nuevas palabras o cambiar el tamaño de la *word cloud* se pueden ver reflejadas al instante.

Si bien esta decisión tuvo un impacto en la memoria que consume la aplicación, el costo es ínfimo en comparación a la ganancia en eficiencia obtenida.

Para lograr lo anteriormente mencionado se implementó el método *paintCloud()* el cual es llamado cada vez que el usuario hace una modificación en las configuraciones de visualización de la *cloud*. Creando así la sensación de dinamismo y fluidez en la aplicación.

```
public void paintCloud() throws IOException {
    panel.removeAll();
    panel.repaint();
    int i = 0;
    filteredWords = fwd.getFilteredWords();
    Cloud aux = new Cloud();
    for (Tag a : cloud.tags()) {
        aux.addTag(a);
    }
    for (String remove : filteredWords) {
        aux.removeTag(remove);
    }
    for (Tag tag : aux.tags()) {
        if (tag.getScoreInt() > (int) (((SpinnerNumberModel) spinner
            .getModel()).getNumber())) {
            JLabel label = new JLabel(tag.getName());
            label.setOpaque(false);
            label.setFont(label.getFont().deriveFont(
                (float) tag.getWeight() * slider.getValue()));

            panel.add(label);
            i++;
        }
    }
    panel.revalidate();
    panel.repaint();
}
```

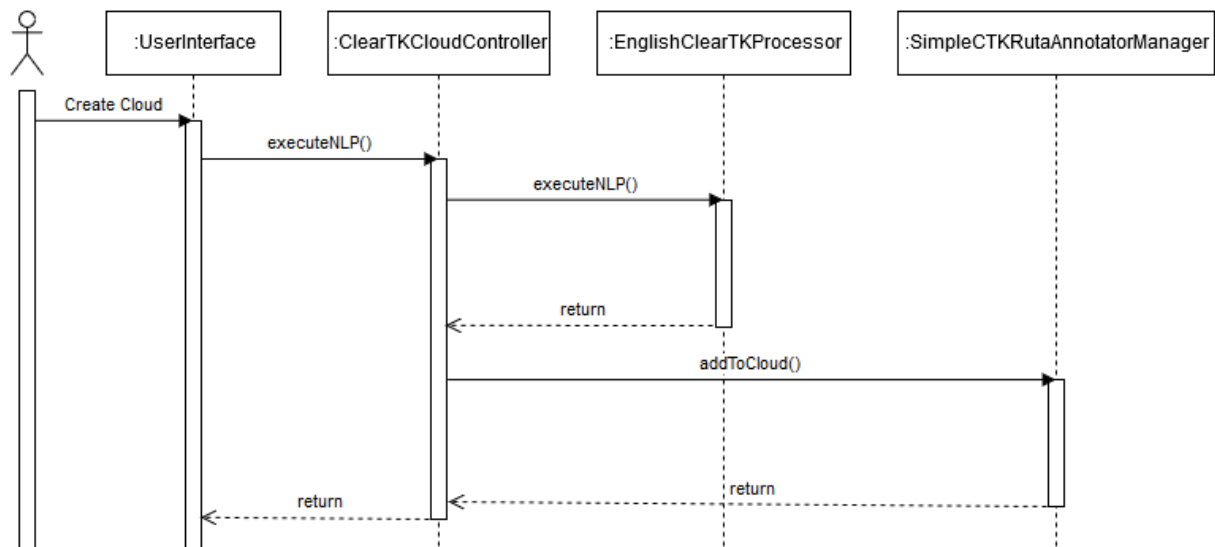
Implementación

La clase *EnglishClearTKProcessor* implementa la interfaz *NLPAnalyzer* utilizando, como su nombre lo dice, la herramienta *ClearTK* para generar los anotadores en inglés que son incorporados al pipeline. En caso de que se desee conservar la utilización de la herramienta *ClearTK*, realizando el análisis en otro idioma, sólo

habría que instanciar la clase `NLPAnalyzer`. A su vez, permite utilizar otro conjunto de anotadores, como puede ser *StanfordNLP* o *OpenNLP*.

`CloudController` es la interfaz encargada de manejar las secciones del código Java que serán procesadas para generar la *word cloud*. La misma se encuentra implementada en la clase `ClearTKCloudController` que utiliza `NLPAnalyzer` para realizar la ejecución del pipeline, aplicando técnicas NLP incluidas en el framework *ClearTK*.

`CTKRutaAnnotatorManager` implementa la interfaz `AnnotatorManager`. Es el encargado de introducir los token procesados en la *word cloud*, según la sección de código seleccionada por el usuario.



Apache UIMA

Unstructured Information Management Applications (UIMA), es un framework que sirve para el análisis de grandes volúmenes de información no estructurada con el fin de descubrir información relevante para el usuario final. Soporta la escritura, despliegue y reutilización de componentes de análisis en una variedad amplia de ajustes.

Los conceptos básicos que se utilizan en el proceso de análisis de texto incluyen anotadores, resultados del análisis, estructura de características, tipo, sistema de tipos, anotación y la estructura de análisis común.

Los **anotadores** contienen la lógica para analizar un documento y descubrir y registrar datos descriptivos sobre el documento completo (metadatos del documento) y sobre partes del documento. Estos datos descriptivos son los resultados del análisis. Los resultados del análisis anotan cualquier subserie contigua del documento de texto. En el mejor de los casos, los resultados del análisis corresponden a la información que el usuario desea.

Los anotadores forman una parte crucial del funcionamiento de **JavaCloudCreator**, ya que permiten analizar y seccionar el documento para su posterior procesamiento.

Una **estructura de características** es la estructura de datos subyacente que representa un resultado de análisis. Una estructura de características es una estructura atributo-valor. Cada estructura de características pertenece a un tipo y cada tipo tiene un conjunto especificado de características o atributos válidos (propiedades), similar a una clase Java. Las características tienen un tipo de rango que

indica el tipo de valor que la función debe tener, tal como String. Todos los anotadores de UIMA almacenan datos en estructuras de características. Por ejemplo, el texto "Juan Perez" se podría representar mediante una anotación de tipo Persona y las características nombre, edad, nacionalidad y profesión.

El **sistema de tipos** (TypeSystems) define los tipos de objetos (estructuras de características) que se pueden descubrir en un documento. El sistema de tipos define todas las estructuras de características posibles en términos de tipos y características (atributos), de forma similar a una jerarquía de clases en Java. Puede definir un número cualquiera de tipos diferentes en un sistema de tipos. Un sistema de tipos es específico del dominio y de la aplicación.

La mayoría de los anotadores de análisis de texto generan sus resultados de análisis en forma de anotaciones. Las anotaciones son una clase especial de estructura de características que se define para el proceso de análisis lingüístico. Una anotación abarca un fragmento de texto de entrada y se define en términos de sus posiciones inicial y final en el texto de entrada.

Todas las estructuras de características están representadas en una estructura de datos central denominada estructura de análisis común (CAS). Todos los intercambios de datos se manejan mediante la estructura de análisis común.

La estructura de análisis común contiene los siguientes objetos:

- El documento de texto
- La descripción del sistema de tipos que indica los tipos, subtipos, y sus características
- Los resultados del análisis que describen el documento o regiones del documento
- Un repositorio de índice que permite acceder a los resultados del análisis y realizar un proceso iterativo sobre ellos

Estos objetos fueron utilizados en la clase EnglishClearTKProcessor que, mediante el CAS, permite realizar diferentes anotaciones a partir del archivo .java a procesar.

Los anotadores que permiten el seccionado de código fueron realizados con el lenguaje Ruta. Este lenguaje permite la creación de reglas que, de forma sencilla, permiten la generación de *TypeSystems*. Los mismos fueron llamados desde *MainEngine*, que se encarga de ejecutarlos y realizar las anotaciones al texto.

UIMA Ruta

Se decidió utilizar el lenguaje Ruta para generar una segmentación del código que facilite su posterior análisis. Está basado en reglas, diseñado para el desarrollo ágil de aplicaciones de procesamiento de textos con UIMA. Mediante esta tecnología se pueden definir patrones de anotaciones de UIMA de manera intuitiva y flexible. Ruta cuenta con varios plugins para Eclipse construidos para facilitar la escritura de reglas incluyendo editores con correctores sintácticos del lenguaje y mecanismos para evaluar las reglas escritas.

A continuación se detallan algunos de los estándares de programación en lenguaje Java y las reglas Ruta utilizadas para procesar los archivos de acuerdo a los mismos.

Estándares de programación

Las convenciones de código son importantes para los programadores por varias razones:

- El 80% del costo del código de un programa va a su mantenimiento
- Casi ningún software lo mantiene toda su vida el autor original
- Las convenciones de código mejoran la lectura del software, permitiendo entender código nuevo rápidamente

Para que las convenciones funcionen, cada persona que escribe software debe seguir las reglas estipuladas. Debido a la existencia de las mismas, y a que gran parte de la comunidad que programa en Java las respeta, se puede hacer uso de UIMA para segmentar el código.

Comentarios de comienzo

Todo código fuente debe comenzar con un comentario en el que se lista el nombre de la clase, información de la versión, fecha y copyright:

```

/*
 * Nombre de la clase
 * Información de la versión
 * Fecha
 * Copyright
 */

```

El *annotator* **MultiLineComments** permite marcar este tipo de comentarios.

```

PACKAGE uima.ruta.annotators;

TYPESYSTEM types.JavaTypeSystem;

DECLARE IniComment, EndComment;

" (\\/(\\*) *" -> IniComment;
" (\\*\\/(\\) *" -> EndComment;

IniComment ANY*?{-> MARK(MultiLineComment)} EndComment;

```

Declaraciones de clases e interfaces

Comentario de documentación de la clase o interface

Este tipo de comentario debe contener cualquier información aplicable a toda la clase o interface que no era apropiada para estar en los comentarios de documentación de la clase o interface. Se pueden marcar con el *annotator* **MultiLineComments** mencionado anteriormente.

```

/**Coment**/)

```

Comentarios de fin de línea

El delimitador de comentario puede convertir en comentario una línea completa o una parte de una línea. No debería ser usado para hacer comentarios de varias líneas consecutivas; sin embargo, puede usarse en líneas consecutivas para comentar secciones de código.

```
if (foo > 1) {
// Hacer algo.
...
}
```

Son útiles para explicar qué hace una sección de código.

Para dar soporte a este tipo de comentarios se utiliza el anotador **SingleLineComments**:

```
return false; // Explicar por qué.

PACKAGE uima.ruta.annotators;

TYPESYSTEM types.JavaTypeSystem;

DECLARE InitComment;

" (\\|\\/)*"->InitComment;

Document{-> RETAINTYPE (BREAK) };

InitComment ANY*?{-> MARK(SingleLineComment)} BREAK;
```

Sentencias package e import

La primera línea no-comentario de los ficheros fuente Java es la sentencia **package**. Después de esta, pueden seguir varias sentencias **import**. Por ejemplo:

```
package java.awt;
import java.awt.peer.CanvasPeer;
```

Nota: El primer componente del nombre de un paquete único se escribe siempre en minúsculas con caracteres ASCII y debe ser uno de los nombres de dominio de último nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que especifican el país como se define en el ISO Standard 3166, 1981.

Para dar soporte a este tipo de sentencias se utilizan los anotadores **Package** e **Import** respectivamente.

Package

```
PACKAGE uima.ruta.annotators;

TYPESYSTEM types.JavaTypeSystem;

Document{-> RETAINTYPE (SPACE) };

W{REGEXP("package")} SPACE ANY*?{-> MARK(Package)} SEMICOLON;
```

Import

```
PACKAGE uima.ruta.annotators;

TYPESYSTEM types.JavaTypeSystem;

Document{-> RETAINTYPE (SPACE) };

W{REGEXP("import")} SPACE ANY*?{-> MARK(Import)} SEMICOLON;
```

Sentencia class o interface

Los nombres de las clases deben ser sustantivos, cuando son compuestos tienen la primera letra de cada palabra que lo forma en mayúscula. Los nombres de las interfaces siguen la misma regla que las clases.

Por ejemplo:

```
public class NombreClase { } -> modificador_acceso interface NombreInterfaz
```

Para esto se utiliza el anotador **Class**.

```
PACKAGE uima.ruta.annotators;  
public class UserInterface()  
  TYPESYSTEM types.JavaTypeSystem;  
  ClassHeader Block SPACE?{-> MARK(Class,1,3)};
```

Variables

Excepto las constantes, todas las instancias y variables de clase o método comienzan con minúscula. Las palabras internas que lo forman (si son compuestas) empiezan con su primera letra en mayúsculas. Los nombres de variables de un solo carácter se evitan, excepto para variables índices temporales. Nombres comunes para variables temporales son i, j, k, m, y n para enteros; c,d, y e para caracteres.

Variables de clase

Para dar soporte a este tipo de sentencias se utiliza el anotador **VarName**.

```
PACKAGE uima.ruta.annotators;  
TYPESYSTEM types.JavaTypeSystem;  
Document{-> RETAINTYPE(SPACE)};  
(SW CW*){-> MARK(VarName)};  
SW{AND(-IS(JavaReservedWords),-PARTOFNEQ(VarName))->MARK(VarName)};  
VarName{PARTOF(MultiLineComment)->UNMARK(VarName)};  
VarName{PARTOF(SingleLineComment)->UNMARK(VarName)};  
VarName{PARTOF(Import)->UNMARK(VarName)};  
VarName{PARTOF(Package)->UNMARK(VarName)};  
VarName{PARTOF(JavaReservedWords)->UNMARK(VarName)};  
VarName{NEAR(Parameters,0,2)->UNMARK(VarName)};  
VarName{NEAR(Parameters,0,3)->UNMARK(VarName)};
```

Variables de instancia

Al igual que en el caso anterior se puede usar el anotador **VarName**.

Métodos

Los métodos compuestos de varias palabras deben tener la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forman en mayúscula.

Estos métodos se deben agrupar por funcionalidad más que por visión o accesibilidad. Por ejemplo, un método de clase privado puede estar entre dos métodos públicos de instancia. El objetivo es hacer el código más legible y comprensible.

Para dar soporte a este tipo de sentencias se utiliza el anotador **Method**:

```
PACKAGE uima.ruta.annotators;  
  
TYPESYSTEM types.JavaTypeSystem;  
  
MethodHeader Block{-> MARK(Method,1,2)};
```

Versionado

El proyecto se ha versionado desde su comienzo, incrementando la calidad del software y favoreciendo el trabajo en equipo. Para esto se utilizó **GitHub**.

Git Repository en Github: <https://github.com/roblescamila/javacloudcreator>

Bibliotecas y frameworks utilizados

ClearTK

Incluye herramientas de parsing, tokenizer, lemmatizer y stemmer, modelos de análisis para idioma inglés.

Licencia: Opensource.

Sitio web: <https://cleartk.github.io/cleartk/>

Apache UIMA

Es un framework para analizar información desestructurada, como texto en lenguaje natural. Soporta la escritura, despliegue y reutilización de componentes de análisis en una variedad amplia de ajustes. Creado en IBM y presentado en la incubadora en 2006, UIMA ha sido adoptado de-facto por una parte importante de la comunidad de procesamiento de lenguaje natural. Se graduó de la incubadora en marzo de 2010.

Licencia: Opensource

Sitio web: <https://uima.apache.org/>

UIMA Ruta

Apache UIMA Ruta es un lenguaje de tipo script basado en reglas. El lenguaje está diseñado para permitir el rápido desarrollo de aplicaciones de procesamiento de texto dentro de Apache UIMA. Un enfoque especial radica en el lenguaje específico de dominio intuitivo y flexible para definir patrones de anotaciones.

Licencia: Opensource

Sitio web: <https://uima.apache.org/ruta.html>

UimaFIT

UimaFIT facilita la instanciación de componentes UIMA sin utilizar archivos de descriptor XML, proporcionando una serie de métodos de fábrica que permiten crear de manera sencilla componentes UIMA.

Licencia: Opensource

Sitio web: <https://uima.apache.org/uimafit.html><https://uima.apache.org/uimafit.html>

OpenCloud

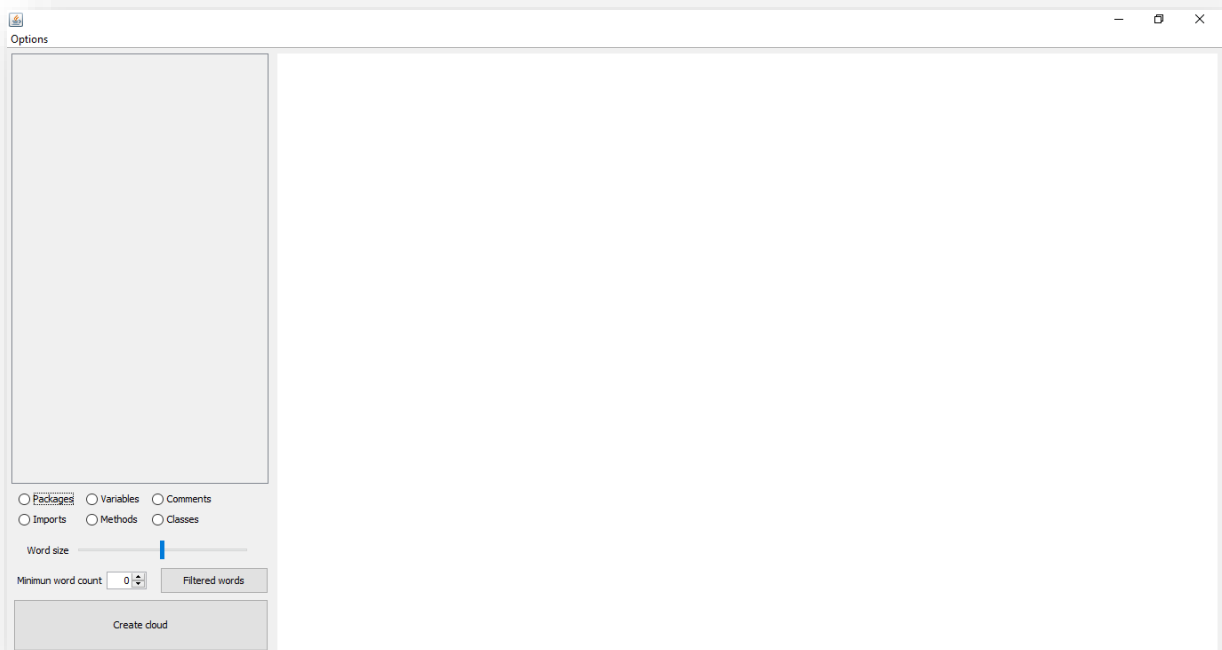
Es una biblioteca en lenguaje Java que permite el modelado de *tag clouds*.

Licencia: Opensource

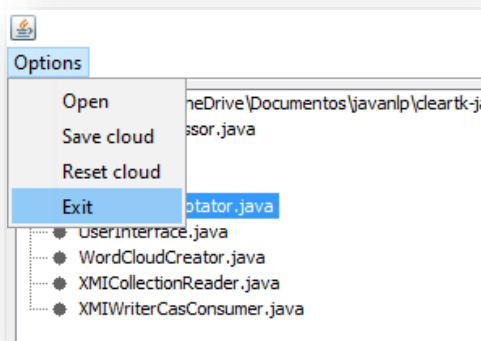
Sitio web: <http://opencloud.mcavallo.org/cgi-sys/defaultwebpage.cgi>

Aplicación

La aplicación fue diseñada para lograr un uso intuitivo y no requiere de un entrenamiento previo. Posee una sección de configuración a la izquierda y un panel a la derecha, donde se presenta la *word cloud* luego del procesamiento de los archivos seleccionados.



En la parte superior izquierda de la pantalla se puede ver el menú de opciones.



Open: Permite seleccionar el paquete que se desea analizar con la herramienta

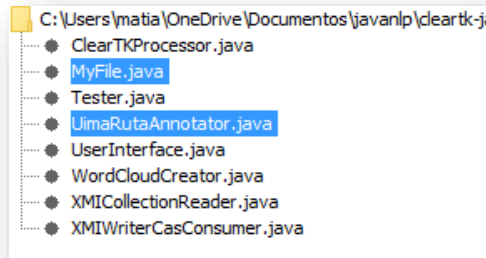
Save cloud: Permite guardar la *word cloud* en forma de imagen.

Reset cloud: Se utiliza para vaciar la *word cloud* y dejar en blanco el panel que la muestra.

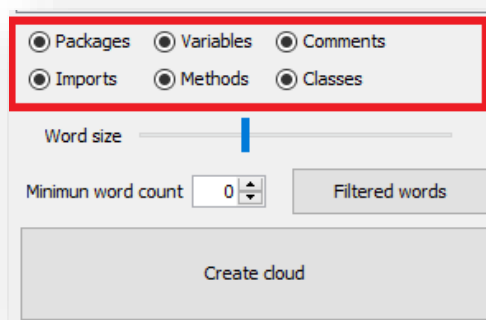
Exit: Para salir del programa.

Al presionar *Open* se abre una ventana que permite seleccionar el paquete a analizar.

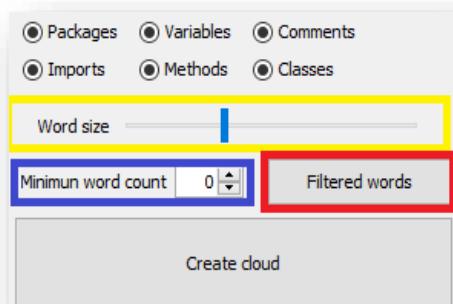
Una vez elegido el paquete, se pueden seleccionar las clases que se quieren analizar o bien analizar el paquete completo, seleccionando todas las clases.



Además de seleccionar qué archivos se quieren analizar hay que configurar qué seccionar del código se quieren tener en cuenta en dicho análisis. Para esto hay que tildar las opciones **Packages**, **Variables**, **Comments**, **Imports**, **Methods** y **Classes**.



Tanto **word size**, **minimum word count** y **filtered words**, pueden ser seteados antes o después de crear la cloud, en caso de setearlos después de crear la cloud los cambios se verán reflejados instantáneamente.

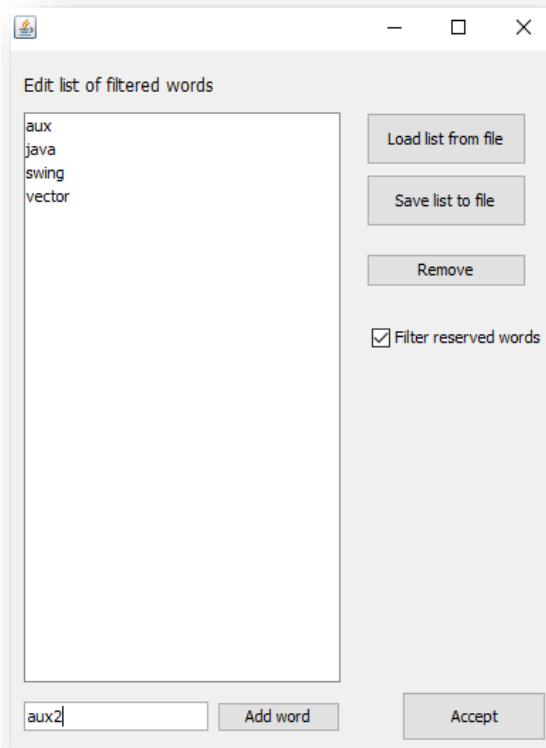


Word Size: Regula el tamaño de las palabras que se muestran en la cloud.

Minimum word count: Indica la cantidad mínima de apariciones que debe tener la palabra para ser tenida en cuenta.

Filtered words: Sirve para filtrar palabras, para que estas no aparezcan en la cloud, es útil para reservar palabras que no aportan información de un paquete.

Al presionar el botón *Filtered words* aparece el siguiente panel de configuración:



Add word: Permite agregar palabras, a la lista de palabras filtradas.

Load list from file: Permite cargar un archivo en formato txt con una lista de palabras para filtrar. El archivo debe tener una palabra por renglón.

Save list to file: Permite guardar la lista en formato txt.

Remove: Sirve para eliminar una palabra de la lista.

Filter reserved words: Esta casilla debe estar tildada si se desean filtrar todas las palabras reservadas del lenguaje Java.

Una vez que todas las configuraciones están terminadas se debe presionar el botón **Create word cloud**, para dar comienzo a la creación de la nube de palabras.

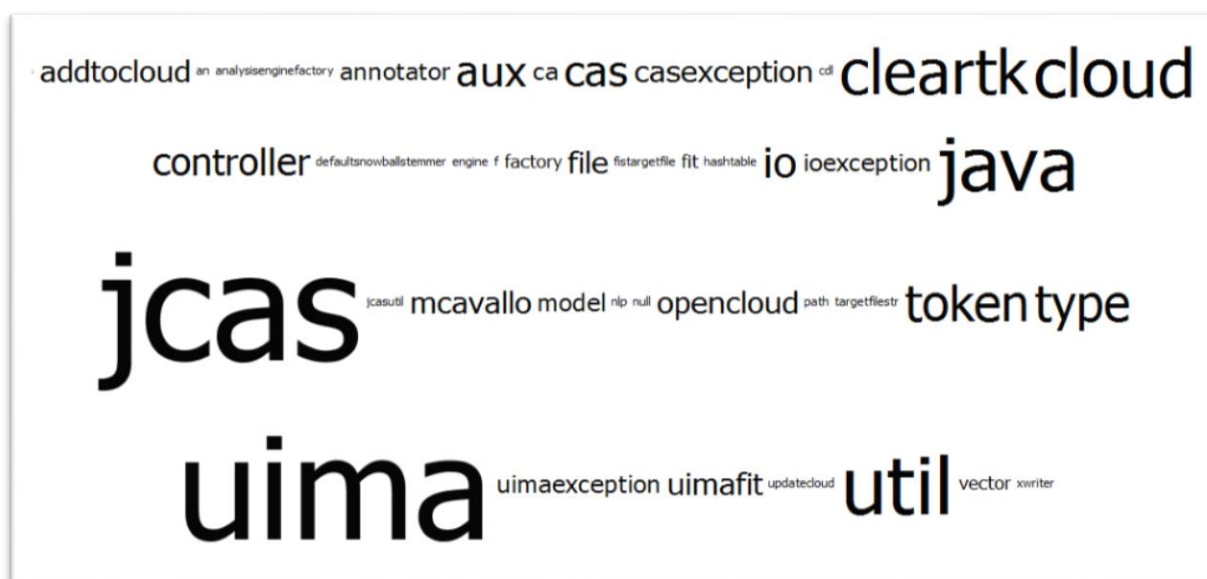
Test

La aplicación tiene como finalidad ayudar al desarrollador a tener una idea general de código Java sin la necesidad de leerlo por completo.

Como modo de prueba se utilizaron los paquetes de *Java Cloud Creator*. Si bien se pueden elegir las secciones de código a analizar, para realizar las pruebas se hizo un procesamiento completo, es decir, de todas las secciones de código (Packages, Classes, Methods, etc.). Para lograr mejores resultados se filtraron aquellas palabras que no aportan información relevante.

Model

A priori se puede observar que el paquete **model** utiliza los frameworks *ClearTK* y *UIMA*, que son utilizados para el análisis NLP. También se puede observar el uso la biblioteca *JCas* para hacer dicho análisis.

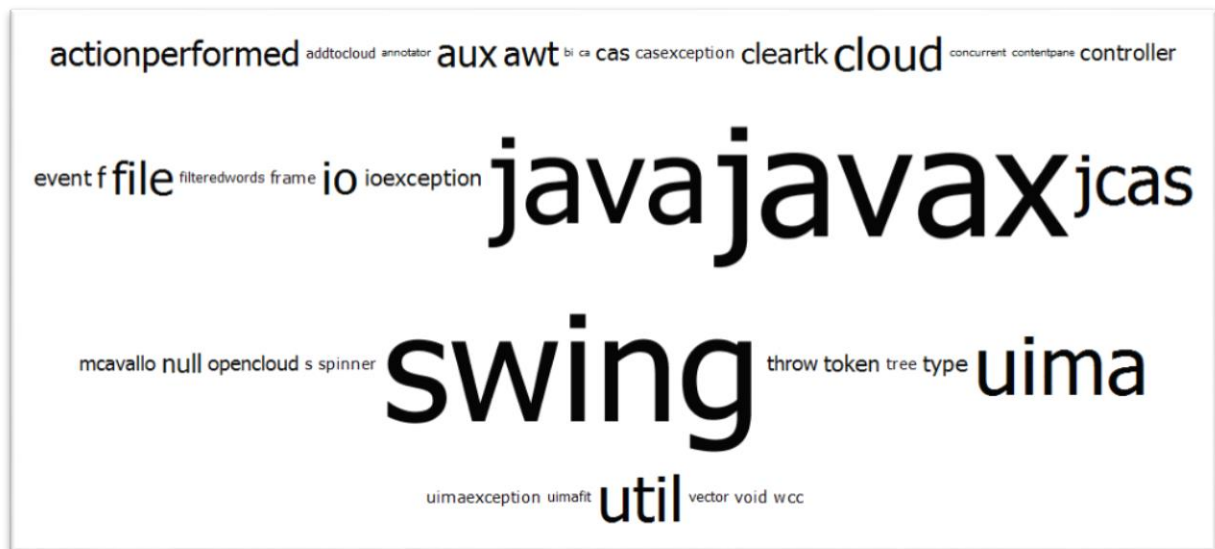


View

En el caso del paquete **view**, a partir de la nube de palabras se puede deducir que la herramienta que se utilizó para hacer la interfaz fue *Swing*, y que se tiene acceso a los dispositivos de entrada y salida (IO).

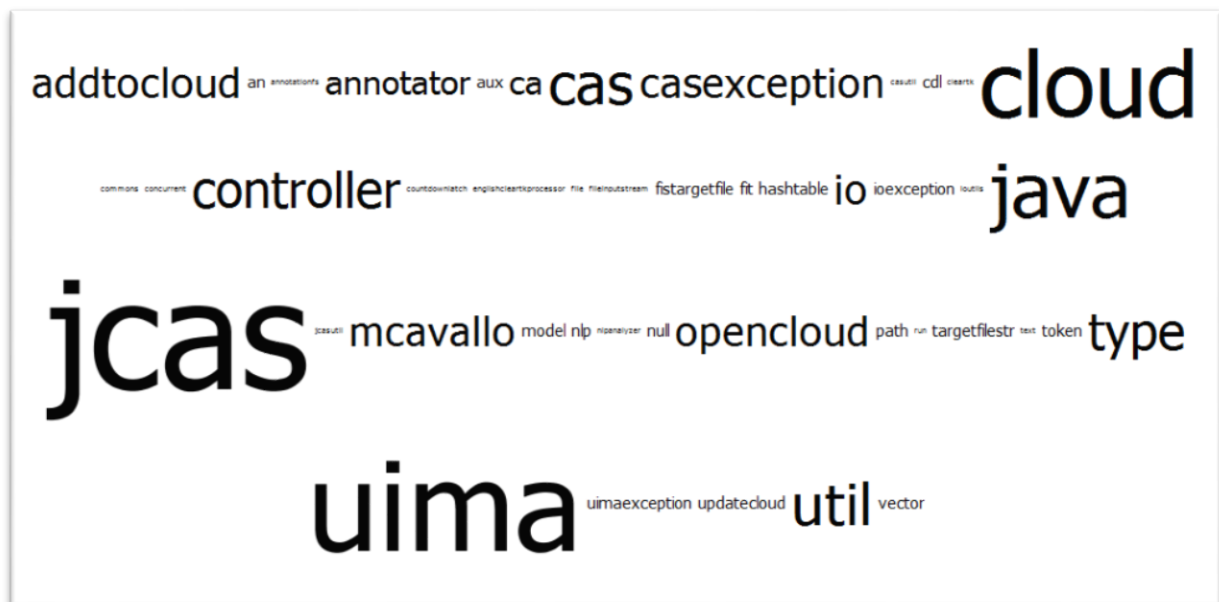
Al igual que en el caso anterior aparecen las palabras *UIMA*, *JCas* y *ClearTK*. Lo que nos hace dar una idea que el procesamiento NLP cumple un rol muy importante en la aplicación. También aparece la palabra *Cloud*, que nos podría dar indicios de cómo se va a presentar la información procesada por *ClearTK*. Por último, se ve que la vista tiene relación con el controlador, como era esperado.

También es importante destacar que aparecen palabras en la nube que aportan poca información, y que podrían haber sido filtradas sin afectar negativamente al resultado. Este es el caso de Java, Javax y Util. En este caso no fueron filtradas a modo de ejemplo.



Controller

Al observar la nube de palabras perteneciente al paquete **Controller**, se pueden divisar palabras como JCas, CAS y UIMA. Lo cual es lógico porque forman parte del modelo. También se puede leer AddToCloud y Cloud, lo que nos da una idea de que aquí es donde se agregan los *tags* a la *WordCloud*.



En síntesis, luego de analizar la aplicación **WordCloudCreator** y mirar las nubes de palabras correspondientes a cada paquete de la aplicación, en vistas generales se pudo obtener una idea de a qué apunta la herramienta, qué tecnologías se utilizaron y una idea de la arquitectura de la misma.

Trabajos futuros

- Detección automática de patrones de diseño y arquitectónicos a partir de la *tag cloud*.
- Análisis comparativo de distintas herramientas de NLP.
- Añadir anotadores en lenguaje Ruta para incorporar nuevas secciones de código para procesar de manera dinámica.
- Permitir agregar nuevas técnicas NLP de manera dinámica.

Conclusión

La realización del trabajo sirvió para finalizar la comprensión de los temas vistos no sólo en la materia sino también en la carrera. También que al hacer un sistema de estas magnitudes es importante realizar un diseño adecuado desde el principio, ya que todas las decisiones tomadas afectan a la totalidad del proyecto, impactando fuertemente en los atributos de calidad significativos para el mismo.

Además el trabajo sirvió para poner en práctica la capacidad de comprender frameworks y la importancia de su utilización, si bien la curva de aprendizaje de los mismos fue bastante plana al comienzo luego de investigar y realizar varias pruebas, se pudo tener un conocimiento que permitió la utilización de los mismos. Los cuales fueron de mucha ayuda y alivianaron el trabajo en enormes magnitudes.

Por otro lado, la elección de las bibliotecas es determinante a la hora de diseñar una aplicación. Estas elecciones no eran tenidas en cuenta antes de la realización de este trabajo.