2016

Matías Prado – Camila Robles

profesores: Andrés diaz Pace – Alejandro rago

Trabajo Práctico Final

Diseño de Sistemas de Software

Contents

[Introducción 3](#_Toc460360585)

[Contexto 4](#_Toc460360586)

[Desarrollo 5](#_Toc460360587)

[Background de la arquitectura 5](#_Toc460360588)

[Vista general del sistema 5](#_Toc460360589)

[Funcionalidad 5](#_Toc460360590)

[Diagrama de Casos de Uso 6](#_Toc460360591)

[Requerimientos significativos conductores del diseño 6](#_Toc460360592)

[Background de la Solución 7](#_Toc460360593)

[Enfoques arquitectónicos 7](#_Toc460360594)

[Arquitectura 7](#_Toc460360595)

[Decisiones de diseño 8](#_Toc460360596)

[Decisiones de implementación 10](#_Toc460360597)

[Implementación 10](#_Toc460360598)

[Apache UIMA 11](#_Toc460360599)

[UIMA Ruta 12](#_Toc460360600)

[Estándares de programación 12](#_Toc460360601)

[Comentarios de comienzo 12](#_Toc460360602)

[Sentencias package e import 12](#_Toc460360603)

[Declaraciones de clases e interfaces 13](#_Toc460360604)

[Sentencia class o interface 13](#_Toc460360605)

[Variables 14](#_Toc460360606)

[Variables de clase 14](#_Toc460360607)

[Variables de instancia 14](#_Toc460360608)

[Métodos 14](#_Toc460360609)

[Comentarios de fin de línea 15](#_Toc460360610)

[Versionado 15](#_Toc460360611)

[Bibliotecas y frameworks utilizados 15](#_Toc460360612)

[ClearTK 15](#_Toc460360613)

[Apache UIMA 16](#_Toc460360614)

[UIMA Ruta 16](#_Toc460360615)

[UimaFIT 16](#_Toc460360616)

[OpenCloud 16](#_Toc460360617)

[Aplicación 16](#_Toc460360618)

[Test 21](#_Toc460360619)

[Model 21](#_Toc460360620)

[View 22](#_Toc460360621)

[Controller 22](#_Toc460360622)

[Conclusión 25](#_Toc460360623)

# 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 como un 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.

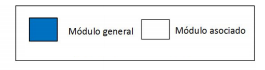
### Arquitectura

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 implementa los métodos de interacción con el usuario y la clase **FilterWordDialog** presenta un dialog que permite seleccionar las palabras que desean excluirse de la tag cloud.

El paquete **controller** tiene las clases encargadas de responder a eventos e invocar peticiones al 'modelo' cuando se hace alguna solicitud sobre la información.

Presenta dos interfaces **CloudController** encagarda de manejar las secciones del código Java que serán procesadas para generar la *word cloud*. Y **AnnotatorManager** encargada 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 ademas contiene todos anotadores los cuales se presentan en archivos .ruta y los engine y typesystem presentados en archivos .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.



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](https://es.wikipedia.org/wiki/Tiempo_de_ejecuci%C3%B3n) en diferentes [plataformas](https://es.wikipedia.org/wiki/Plataforma_%28inform%C3%A1tica%29) mediante la utilización del lenguaje Java.

Java es un [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) de [propósito general](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_de_prop%C3%B3sito_general), [concurrente](https://es.wikipedia.org/wiki/Computaci%C3%B3n_concurrente), [orientado a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los [desarrolladores](https://es.wikipedia.org/wiki/Desarrollador_de_software) de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el [código](https://es.wikipedia.org/wiki/C%C3%B3digo_fuente) 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.



## 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 incluídas 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.

### 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 detalla la implementación de las reglas utilizadas para procesar los archivos **.java**.

## Estándares de programación

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

* El 80% del coste 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 mucho más rápidamente y más a fondo.

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 la 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

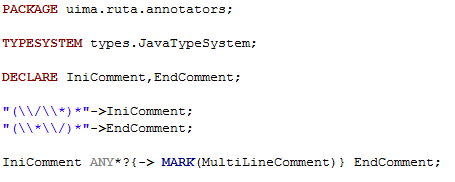
\* Información de la versión

\* Fecha

\* Copyright

\*/

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



### 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:

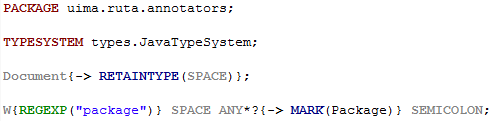
package java.awt;

import java.awt.peer.CanvasPeer;

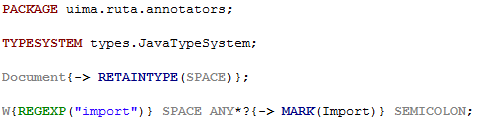
**Nota:** El primer componente del nombre de un paquete único se escribe siempre en minúsculas con caracteres ASCII y debe ser uno de los nombres de dominio de último nivel, actualmente com, edu, gov, mil, net, org, o uno de los códigos ingleses de dos letras que especifican el país como se define en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

Para dar soporte a este tipo de sentencias se utilizan el annotator package e Import respectivamente

#### Package



#### Import



### Declaraciones de clases e interfaces

#### Comentario de documentación

de la clase o interface

(/\*\*...\*/)

Este tipo de comentarios se pueden marcar con el annotator **MultiLineComments**  mencionado anteriormente .

### Sentencia class o interface

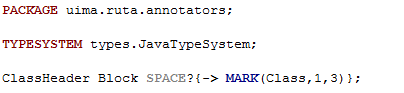
Los nombres de las clases deben ser sustantivos, cuando son compuestos tendrán la primera letra de cada palabra que lo forma en mayúsculas. Intentar mantener los nombres de las clases simples y descriptivos. Los nombres de las interfaces siguen la misma regla que las clases.

Por ejemplo:

public class UserInterface

modificador\_acceso interface NombreInterfaz

Para esto se utiliza el annotator **Class**



**Comentario de implementación de la clase o interface si fuera necesario**

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.

(/\*...\*/)

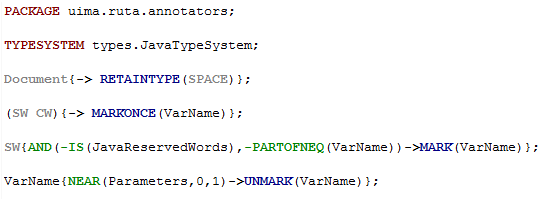
Se utiliza el annotator **MultiLineComments**  anteriormente mencionado.

### 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 no deben empezar con los caracteres subguión "\_" o signo del dolar "$", aunque ambos están permitidos por el lenguaje. Los nombres de las variables deben ser cortos pero con significado. La elección del nombre de una variable debe ser un mnemónico, designado para indicar a un observador casual su función. Los nombres de variables de un solo carácter se deben evitar, 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 annotator **VarName**.



### Variables de instancia

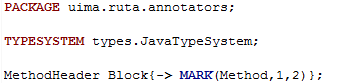
Al igual que en el caso anterior se puede usar el annotator **VarName.**

### Métodos

Los métodos deben ser verbos, cuando son compuestos tendrán la primera letra en minúscula, y la primera letra de las siguientes palabras que lo forma 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 utilizan el annotator Methods



### Comentarios de fin de línea

El delimitador de comentario // puede convertir en comentario una línea completa o una parte de una línea. No debe 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.

...

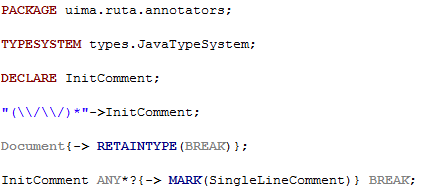
}

else {

return false; // Explicar por qué.

Son muy útiles para explicar qué hace una sección de código.

Para dar soporte a este tipo de comentarios se utilizan el annotator **SingleLineComments**



## 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 hace que sea fácil crear instancias de componentes UIMA sin utilizar archivos de descriptor XML en absoluto , proporcionando una serie de métodos de fábrica que permiten crear de manera sencilla componentes UIMA

**Licencia**: Opensource

**Sitio web:** https://uima.apache.org/uimafit.htmlhttps://uima.apache.org/uimafit.html

### OpenCloud

Es una biblioteca de java que permite el modelado de tags cloud.

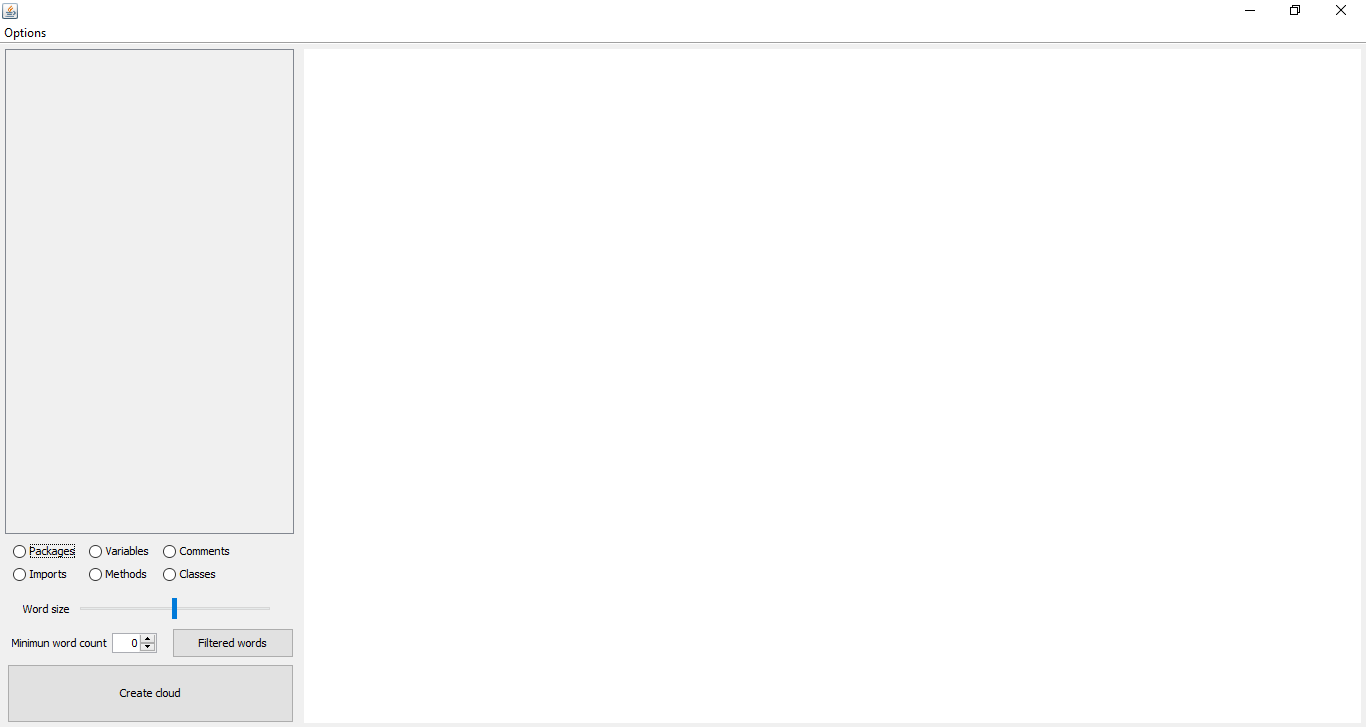
**Licencia**:Opensource

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

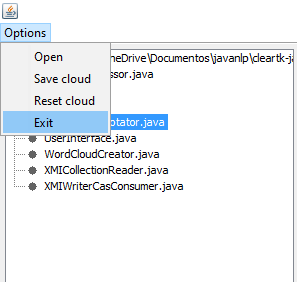
# Aplicación

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

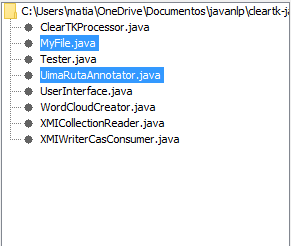
**Save cloud**: Permite guardar la *word cloud* en forma de imagen.

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

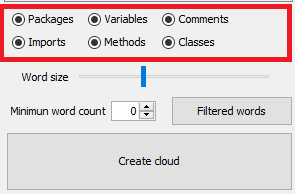
**Exit**: Para salir del programa.

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

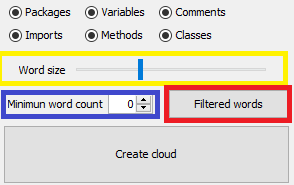
Una vez elegido el paquete, se pueden seleccionar las clases que se quieren analizar o bien analizar el paquete completo, seleccionando todas las clases.



Además de seleccionar qué archivos se quieren analizar hay que configurar que 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.

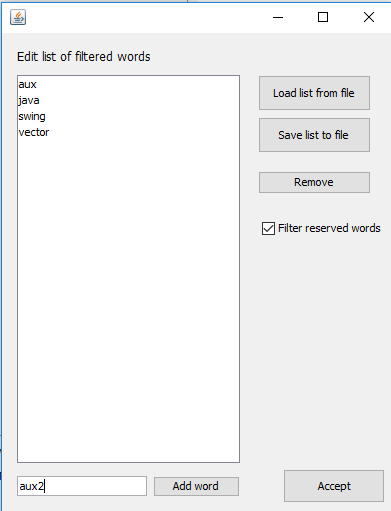


**Word Size**: Regula el tamaño de las palabras que se muestran en la cloud.

**Minimum cord 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, estas pueden ser agregadas tanto presionando el botón como tocando la tecla enter.

**Load list from file:** Permite cargar un txt, con una lista de palabras para filtrar, el txt debe tener una palabra por renglón.

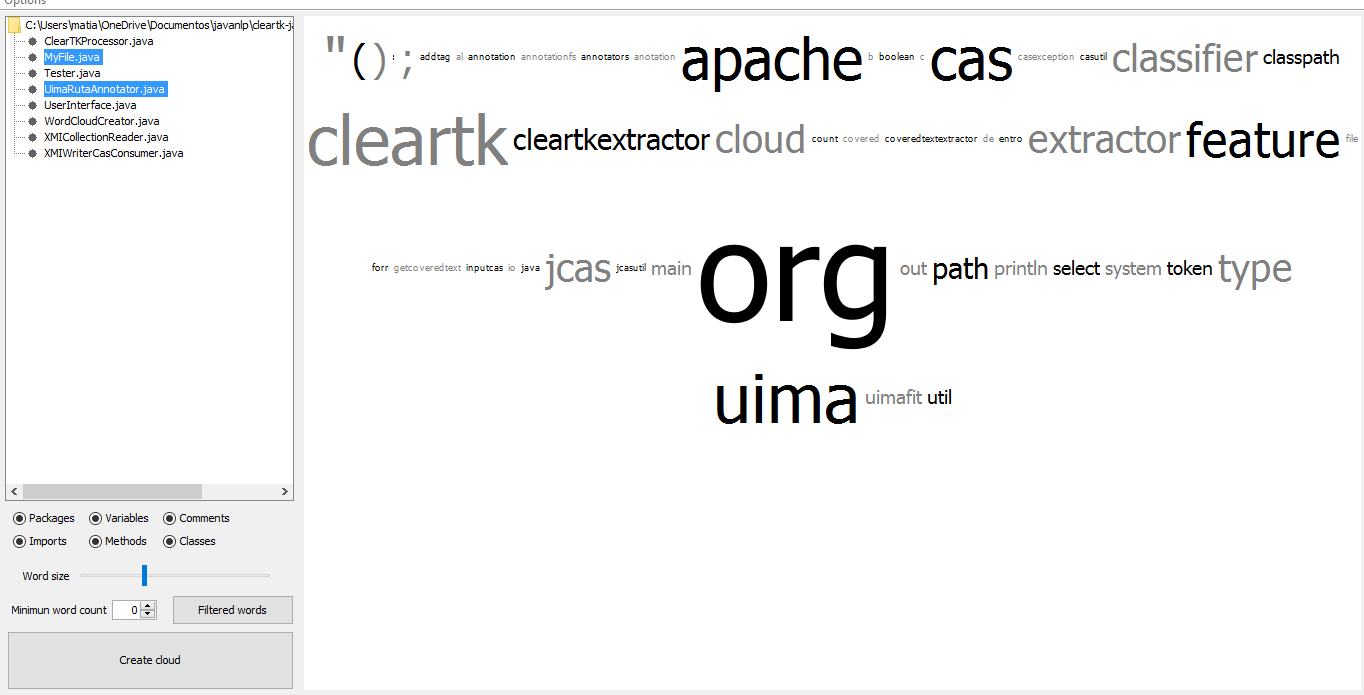
**Save list to file:** Permite guardar la lista como txt.

**Remove:** Sirve para eliminar una palabra de la lista.

**Filter reserved word:** Esta casilla debe estar tildada si se desean filtrar todas las palabras reservadas de java.

El botón **accept** debe ser presionado una vez que la lista está terminada.

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.



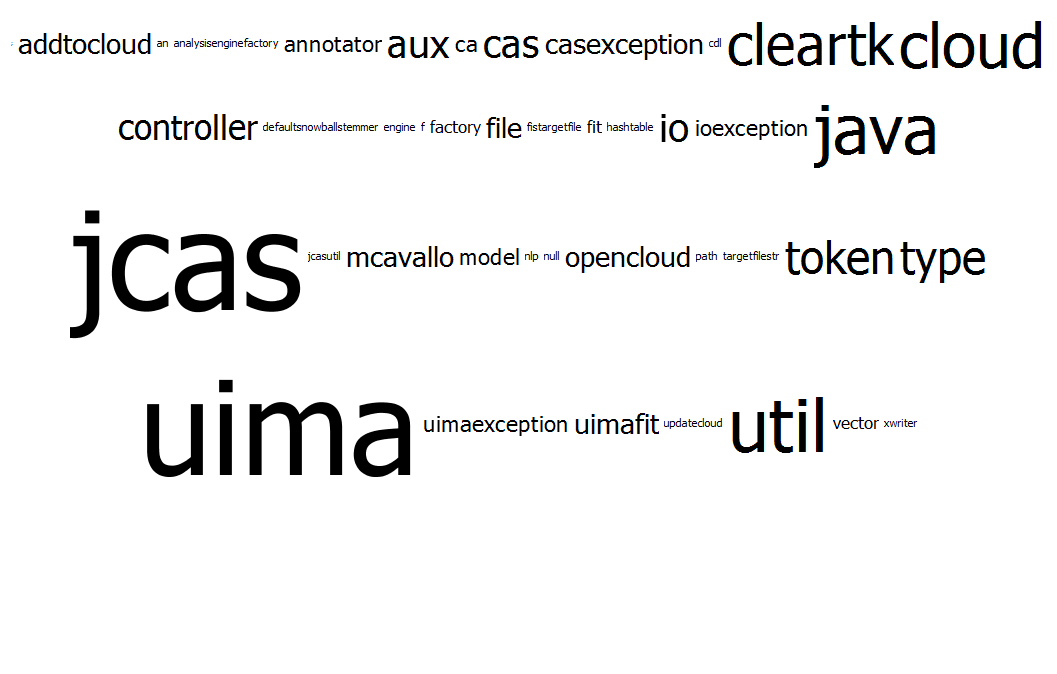
# Test

La aplicación tiene como finalidad ayudar al desarrollador a tener una idea general de código Java sin la necesidad de leer el código 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. Se filtraron las palabras que no aportan información relevante.

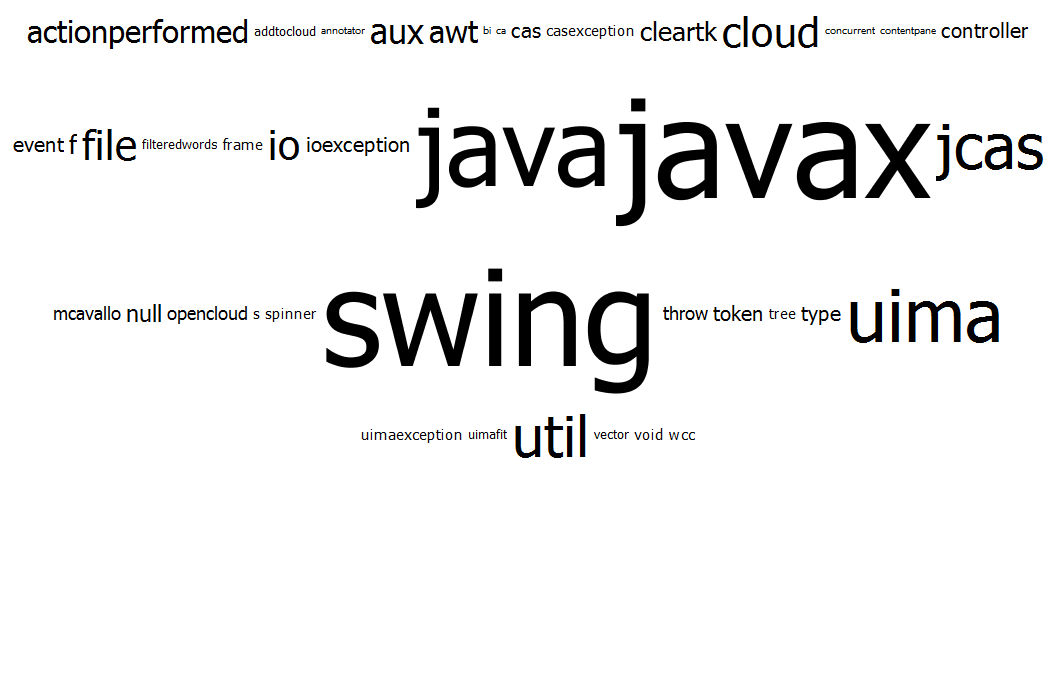
## Model

El análisis del paquete model luego de seleccionar todas las secciones de código (Packages, Classes, Methods ,etc), dio como resultado la siguiente cloud.



A priori se puede observar que el **model**, utiliza los framework *ClartTK* y *UIMA*, los cuales son utilizados para el análisis NLP. También se ve que se usa la biblioteca *JCas* para hacer dicho análisis. Y que tiene conexión con el controlador de la aplicación.

## View

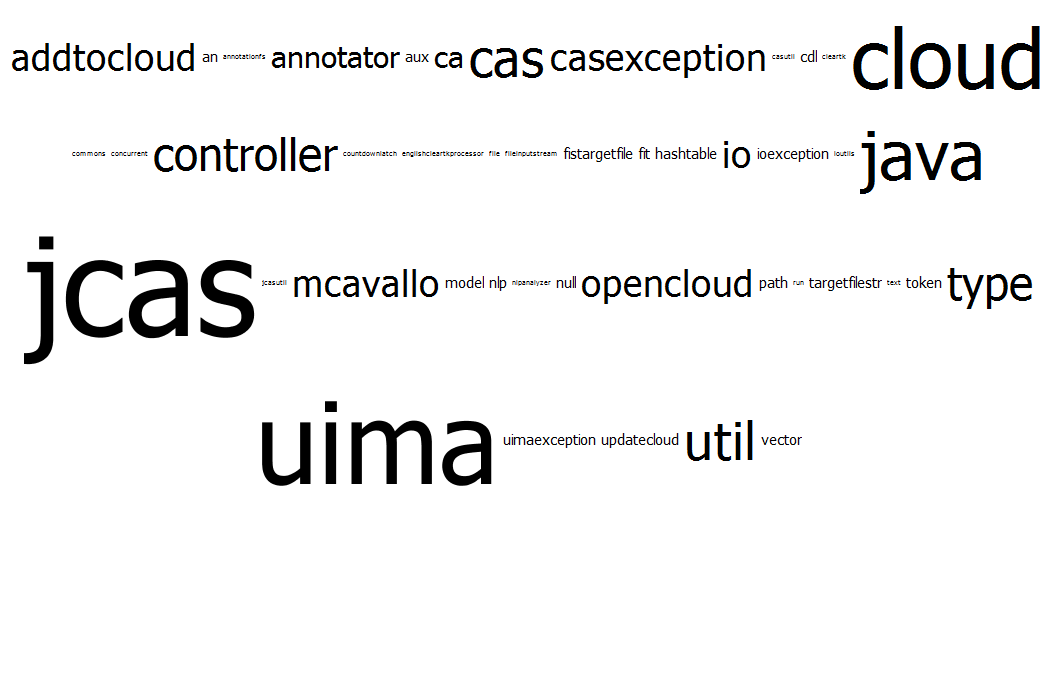


En el caso del paquete **view**, mirando el word cloud uno puede deducir que la herramienta que se utilizó para hacer la interfaz fue *Swing,* que se tiene acceso a los dispositivos de entrada y salida (*IO*).

Al igual que en el caso anterior aparecen las palabras *UIMA JCas y ClearTK.* Lo que nos hace dar una idea que el procesamiento de 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 toda la información que es procesada por el *ClearTK*. Por último lugar se ve que la vista tiene conexion con el controlador.

También es importante destacar que aparecen palabras que dan poca información, las cuales podrían haber sido filtradas, como es el caso de JAVA JAVAX y UTIL. En este caso no fueron filtradas a modo de ejemplo.

## Controller



Al mirar la word cloud que pertenece al procesamiento del paquete que le corresponde al controlador se pueden divisar palabras como JCas, CAS y UIMA. Lo cual es lógico porque formaban parte del modelo. También se puede leer AddToCloud y Cloud, lo que nos da una idea que aquí es donde se agregan los tag a la WordCloud.

En síntesis, luego de analizar la aplicación WordCloudCreator y mirar las WordCloud correspondientes a cada paquete de la aplicación, en vistas generales se pudo obtener una idea de a qué apunta la herramienta , que 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 nuevos 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 puedo 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.