(75.06) Organización de Datos

Trabajo Práctico Speaker – Primera Entrega

Curso: Lic. Arturo Servetto

Ayudante Asignado: Renzo Navas

Grupo: Led Zeppelin

Integrantes:

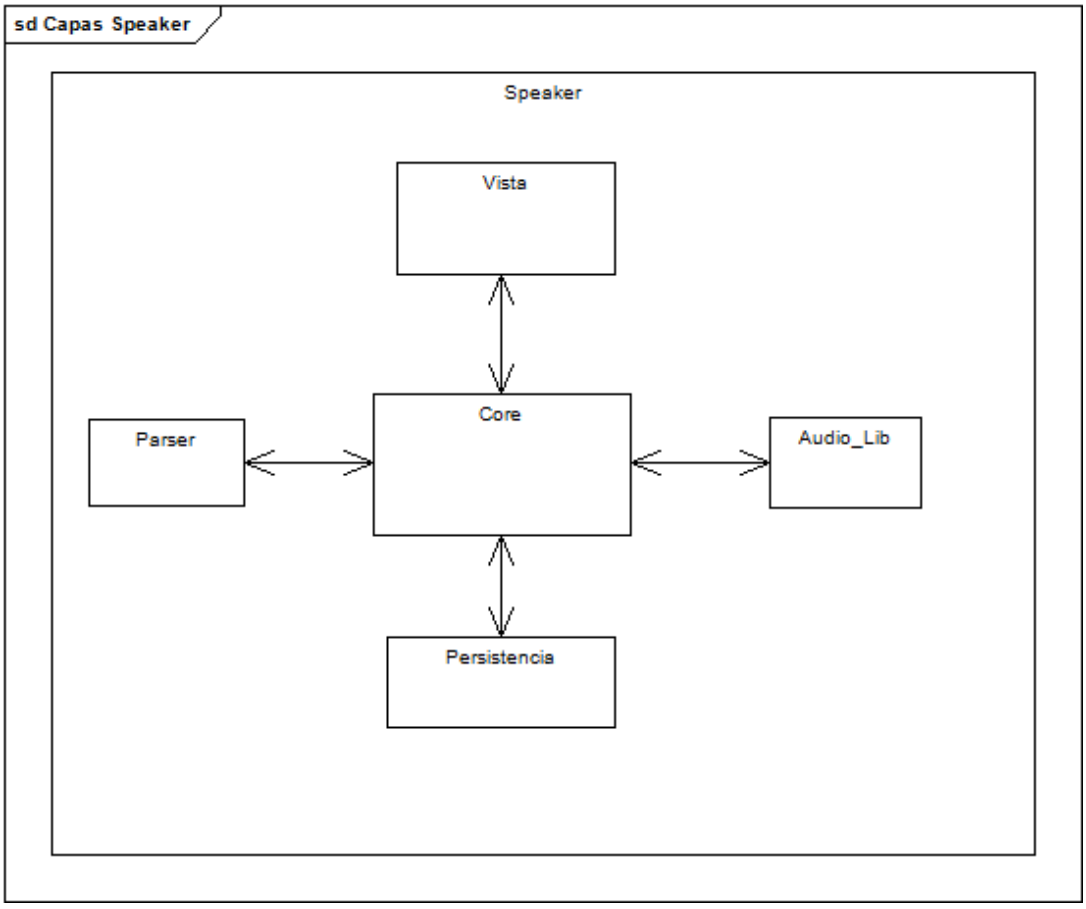
Gabriel Fusca	86521
Exequiel Leite	82631
Diego García Jaime	78938
Cristian Stügelmayer	82521
Ramiro Salom	83350

Índice de contenido

Arquitectura utilizada

La arquitectura utilizada se divide en 5 estructuras lógicamente diferenciadas:

1. Vista
2. Core
3. Parser
4. Persistencia
5. Audio_lib



Una vez comprendidos en profundidad los requerimientos de construcción de “TheSpeaker”, se presentó la primera y mas importante decisión en cuanto a la arquitectura respecta, la división del trabajo en unidades funcionales, que resulten relativamente independientes unas de otras. La decisión tomada, implicó tomar 5 bloques de diseño básicos y edificar la funcionalidad sobre ellos. Era claro, que no resultaba factible un esquema en el cual la totalidad de los bloques estuviesen desacoplados, necesitando de la existencia de un lazo o núcleo que delegue las acciones

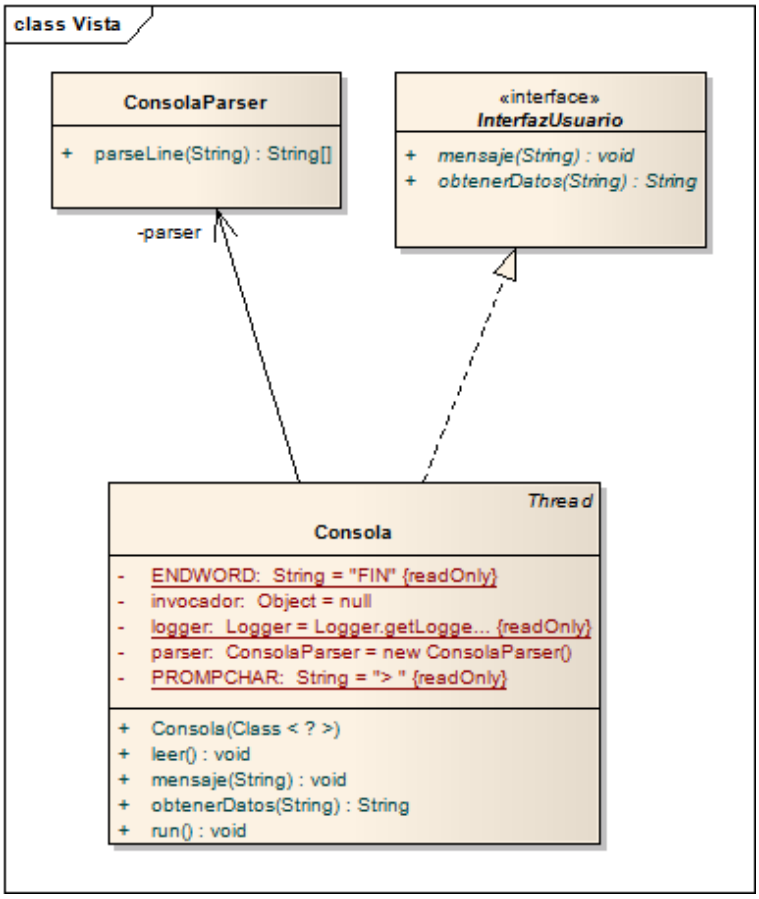
y coordine las tareas. A tal fin se creó el “Core” que resulta el punto de unión entre los diferente bloques.

Para que la distribución resulte eficiente, lo primero que se codificó fueron las respectiva interfaces, de modo que todos los actores puedan realizar sus prueba unitarias y establecer sus algoritmos sin depender estrictamente de otras partes del código.

Si bien la cantidad de bloques utilizados es la misma que la cantidad de integrantes, no es estrictamente cierto que la implementación de una sección sea responsabilidad única de un solo participante, todos han aportado en la construcción de la algoritmia general pero siendo responsables del funcionamiento de la unidad asignada.

Se ha prestado especial interés en el diseño de la persistencia, siendo ese el objeto de estudio de la materia. Las premisas en su implementación, fueron conceptos tales como la reutilización del código y la generalización del mismo. Sabiendo que en futuras entregas se se modificaría las estructuras lógicas utilizadas, era imperativo que el código resulte fácilmente mantenible y escalable para no requerir de una reingenieria profunda llegado ese momento. Por otro lado, y con el objetivo en mente de reducir al máximo los accesos a disco, se han creado algoritmos en la sección de parseo que permiten una interpretación del texto desde un alto nivel, para intentar aplazar los accesos al medio de almacenamiento y en caso de ser necesario, que estos transporten la menor información posible.

Consola de Comandos



Esta clase es un implementación simple de un SHELL de comandos, implementa una interfaz “InterfazUsuario” que les permite a los que usan la clase, comunicarse con el usuario ya sea para pedirle datos o para informarle acerca del resultado de alguna operación.

La clase hereda de Thread, por que lo luego de construirla habrá que invocar al método start para que empiece su ejecución.

Esta clase, cuando se construye, instancia un objeto del tipo que fue pasado por parámetro al constructor, luego cuando se pone en ejecución con el método run, queda a la espera de que el usuario ingrese algún comando y presione enter.

Cuando un comando es ingresado, la consola intenta invocar un método de la clase que se pasó por parámetro en el constructor, por medio del uso de *reflection*, si no lo consigue, indica que no reconoció el comando.

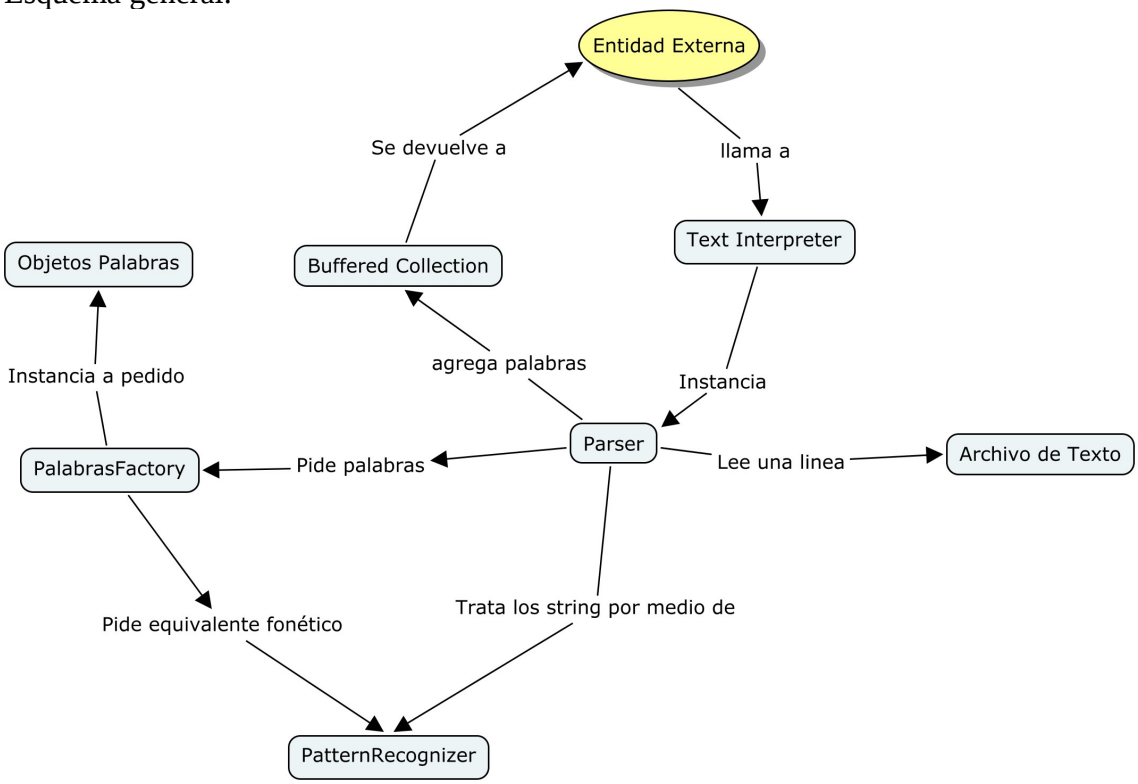
Para que la clase que implementa los métodos que ejecuta la consola pueda ser invocada por esta, solo debe declarar un método publico que cuyo primer parámetro sea del tipo “IntefazUsuario”, y luego si es necesario, tantos parámetros del tipo String como se necesiten, Solo con esto, la consola ejecutará los métodos deseados.

Para terminar la ejecución de la consola, basta tipear “fin” o alguna palabra que se reserve para este propósito y que es configurable, cuando esto suceda, se intentara invocar al método “quit” de la clase que implementa los comandos para que realice lo que crea conveniente y se terminará con la ejecución.

Interpretación de textos

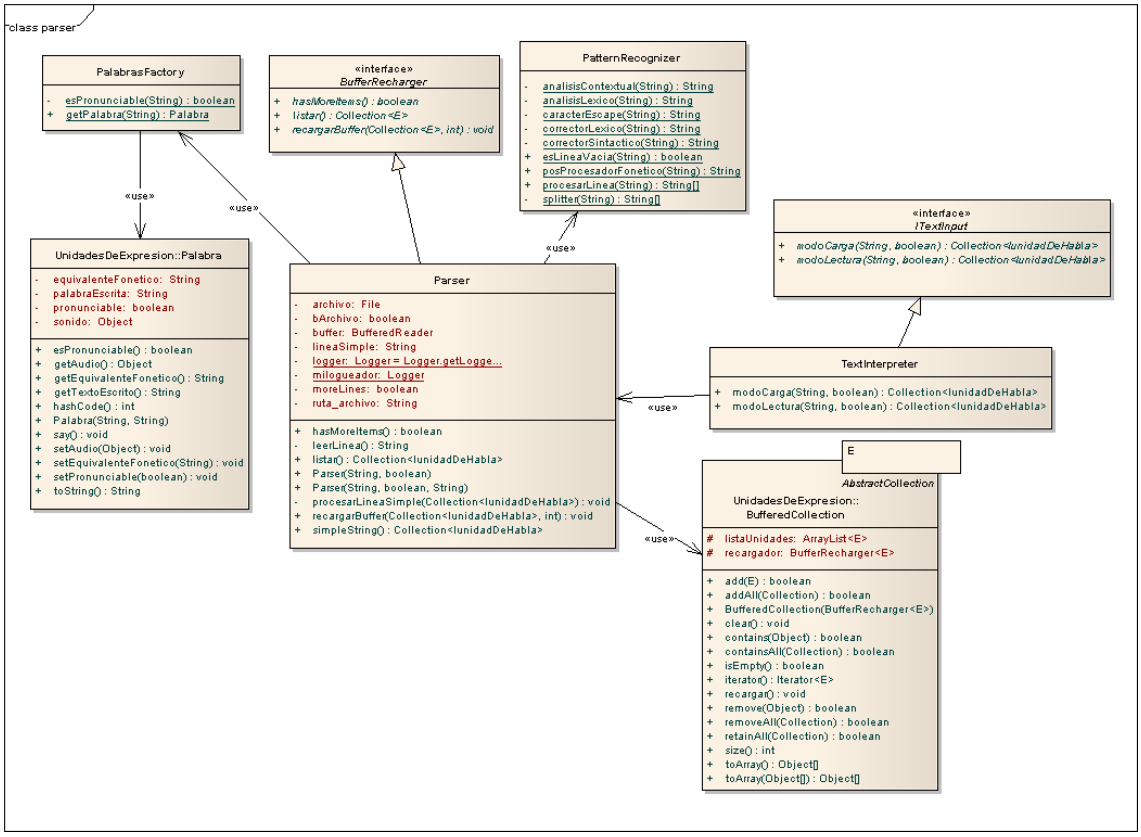
Funcionamiento general del Interpretador de Textos:

Esquema general:

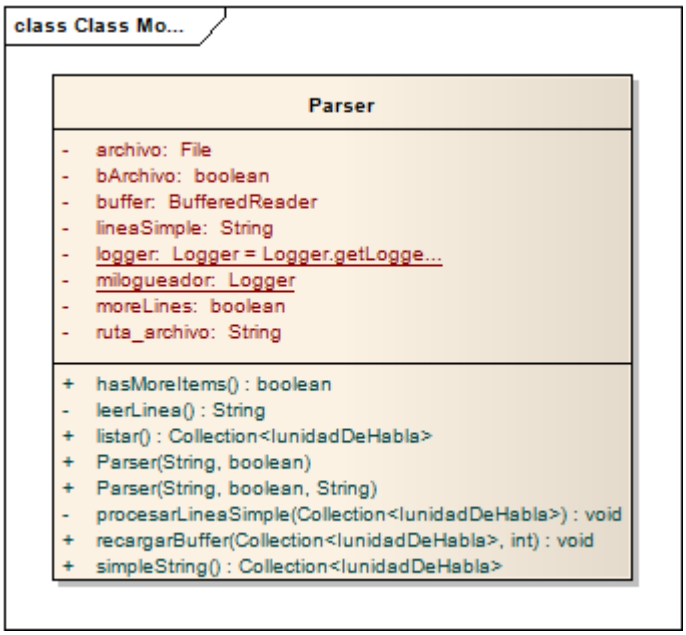


Cualquier entidad externa, puede instanciar un objeto del tipo TextInterpreter, que se encargará de realizar todas las tareas necesarias para devolver una colección de palabras listas para usar. El procedimiento implica instanciar un objeto parser con la ruta del archivo a interpretar, aunque también es posible invocar una instancia de parser para solo un string.
El parser se encargará de realizar las tareas necesarias para convertir el texto en una colección de palabras, empezando por leer una linea del archivo y entregársela al PatternRecognizer para su interpretación. Una vez finalizada esta tarea, se itera sobre las palabras obtenidas y se generan, mediante PalabrasFactory, nuevas instancias de las palabras en cuestión. Por ultimo los objetos que representan en el modelo a las palabras del texto, son almacenados en una colección especializada “BufferedCollection” para luego ser entregada a la entidad externa.

Diagrama de clases - Interpretador de Textos



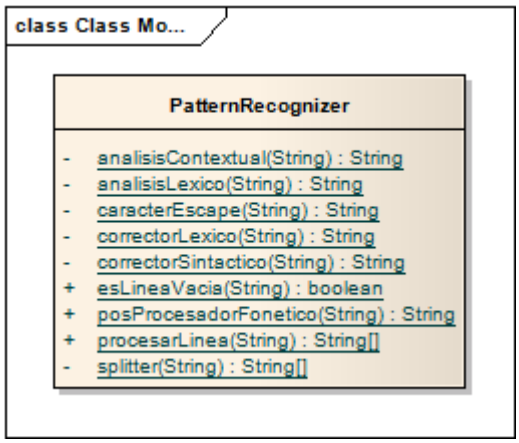
La clase Parser



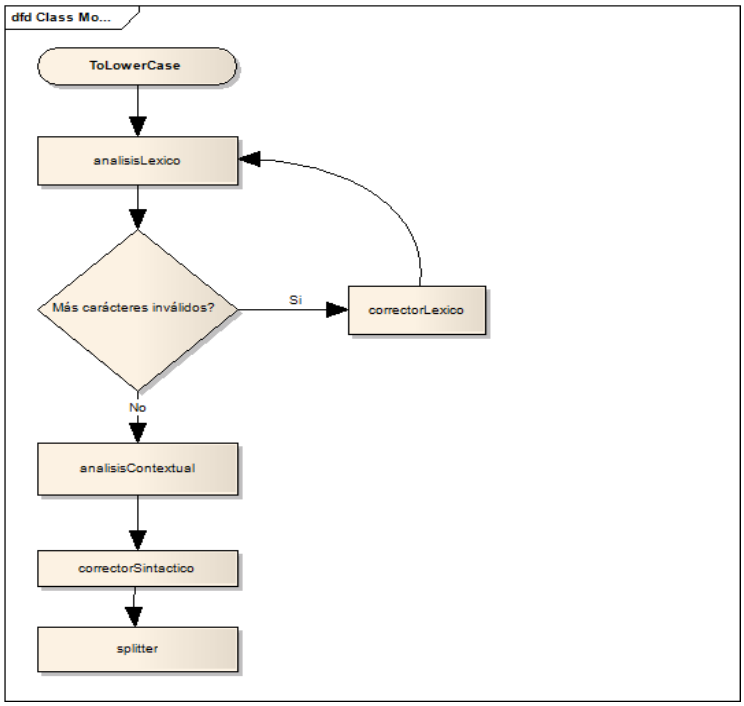
El núcleo del paquete se encuentra en la clase Parser, que se encarga de distribuir las tareas según se requiera, en primer lugar leyendo una linea del archivo de texto para luego pedir su procesamiento y por ultimo generar el objeto palabra para cada palabra obtenida.
Si bien su lógica no es compleja, presenta algunas características importantes como por ejemplo:

- Implementa la Interfaz “BufferRecharger”, esta interfaz le permite utilizar una colección especial desarrollada por el equipo la “BufferedCollection”, sobre la cual nos explayaremos en puntos posteriores, pero que a grandes rasgos, permite la paginación del contenido del archivo de texto. Esto se ha realizado pensando en la utilización de textos largos.
- Posee métodos especiales para que el usuario pueda definir el Charset utilizado por el archivo de texto. Como es sabido, no existe un método definido para obtener *on the fly* el juego de caracteres utilizado para codificar el texto que se pretende parsear, con lo cual debe pedirse por parametro. Por defecto se utiliza UTF-8.

La clase PatternRecognizer



Gran parte de la lógica del paquete se encuentra resumida en esta clase. Básicamente permite realizar varias operaciones sobre un string, devolviendo un vector de palabras procesadas. La implementación está basada en el uso intensivo de Expresiones Regulares.



Desde su entrada hasta que finaliza el proceso, el string sufre varias modificaciones que pueden resumirse del siguiente modo:

1. Se pasa el string recibido a minúsculas, para evitar problemas de procesamiento, a nivel fonético no existen diferencias entre una palabra en mayúsculas y otra en minúsculas.
2. Análisis Léxico: Permite detectar caracteres mal utilizados y caracteres raros (caracteres que no son utilizados o no tienen sentido fonético) , suplantandolos por un equivalente o un espacio en caso de no existir el mismo, de esta forma se normaliza el texto para su manipulación
3. Análisis Contextual: Permite detectar aquellos signos de puntuación que deben ser pronunciados de aquellos que solo denotan pausas en la lectura. Una vez

detectados se utilizan caracteres de control para definirlos y poder interpretarse al momento de generar los objetos Palabra.

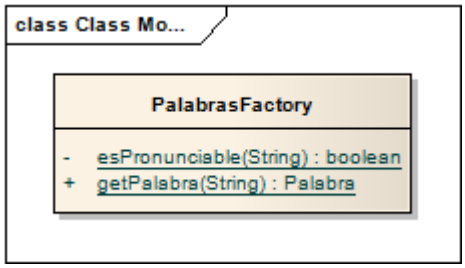
4. Corrector Sintáctico: Se encarga de separar palabras, numeros y signos de puntuación mediante espacios, para para que puedan ser correctamente procesados por el splitter
5. Splitter: Divide el string previamente procesado, en un vector de strings dejándolo listo para la generación de la palabra propiamente dicha.

Existe también un método denominado “posProcedorFonético” que permite, si esta activada la optimización al idioma español, obtener un equivalente fonético de la palabra escrita de modo tal que:

- Tanto los números, como los signos de puntuación son interpretados con su texto asociado, es decir, al ingresar el “1” para el sistema es equivalente a escribir “uno” ó al ingresar “.” para el sistema es equivalente a escribir “punto”
- La letra “h” es quitada de las palabras salvo que resulte antecedida por una “c”. fonéticamente se verifica que “ola” (la ola del mar) y “Hola” (el saludo) presentan conceptos completamente diferentes pero resultan homófonas.
- Diferentes combinaciones de letras son tomadas como equivalentes: mb|nv , ce|se , sa|za , je|ge , ll|y ,etc.

Todos estos procesos entre strings, fueron realizados con la premisa de disminuir al mínimo los accesos a disco mediante el aumento de índices que apuntan a igual sonido y la verificación lógica de aquellas palabras que deben pronunciarse de aquellas que no.

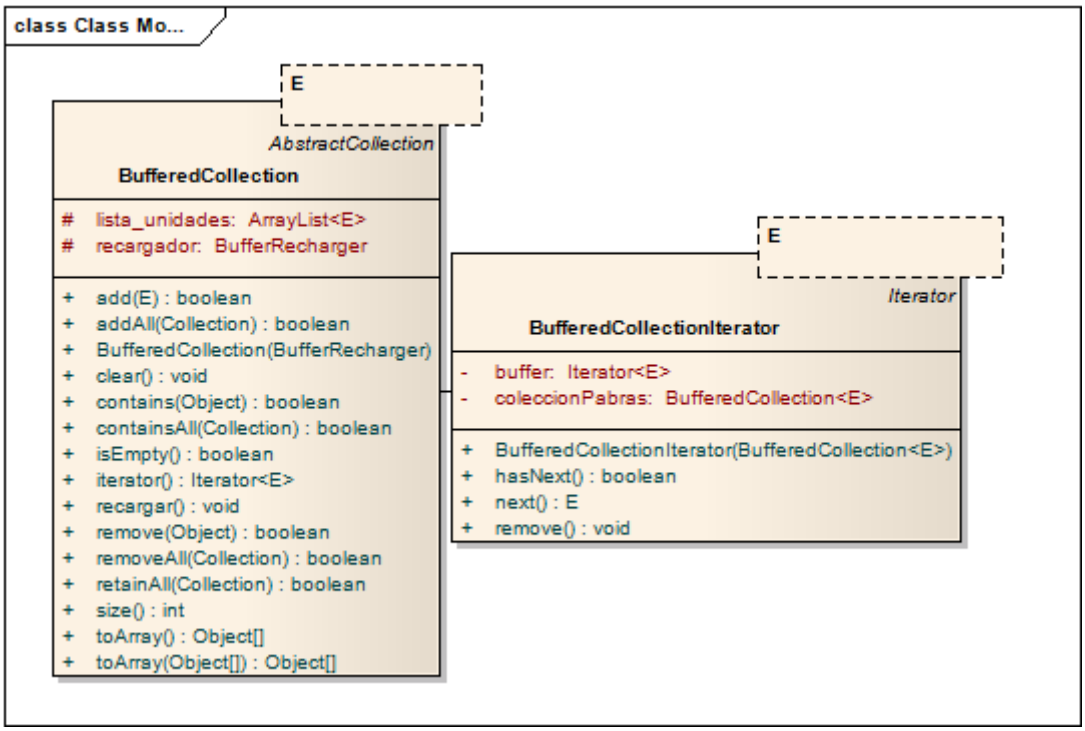
La clase PalabrasFactory



Esta clase, implementada según el patrón Factory, permite centralizar el proceso de creación de una nueva palabra. Simplemente recibe un string, busca su equivalente fonético (si esta activada la optimización de español), en caso de ser un signo verifica si debe pronunciarse y por ultimo instancia una nueva palabra agregándole todos sus atributos,
Se utilizó el patrón Factory, debido a que el proceso de creación de una nueva palabra es relativamente engorroso, insumiendo varias lineas de código para su desarrollo. Por otra parte, pensando en la extensibilidad del código es posible que para entregas

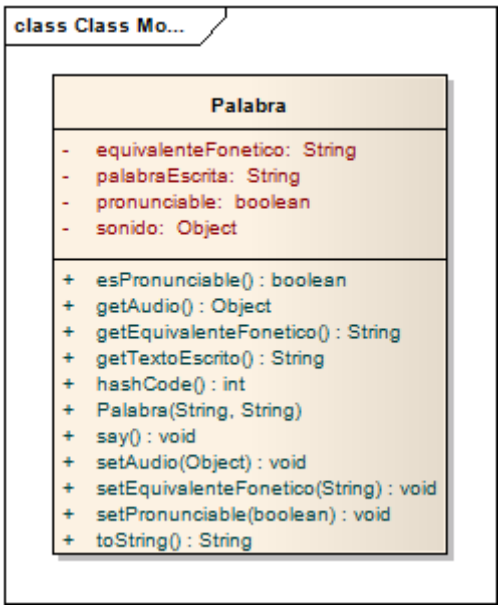
posteriores existan otras clases, por ejemplo la clase “signoDePuntuacion” ó la clase “numero” pudiendo siempre delegar el proceso de creación de objetos a esta clase.

La clase BufferedCollection y BufferedCollection Iterator



La clase bufferedCollection, implementa la interfaz Collection; se trata de una colección que se manipula como una convencional pero tiene la capacidad implícita y transparente al usuario, de auto Paginarse, de modo que solo un subconjunto de elementos se encuentren accesibles a la vez. Su utilización es completamente genérica y puede emplearse con cualquier fin, pero está concebida para evitar cargar todas las palabras del archivo de texto al mismo tiempo en memoria. Por su parte el BufferedCollectionIterator, permite recorrer la colección desentendiéndose de las cuestiones internas a la paginación.

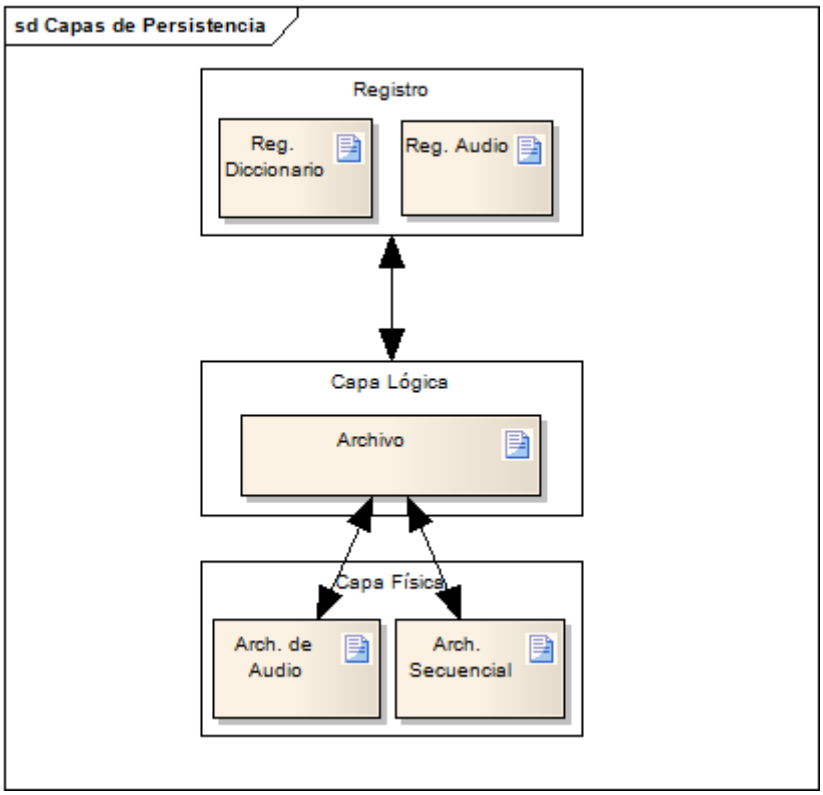
La clase Palabra



La clase palabra, representa el tipo de dato que utiliza este programa, permite almacenar el texto escrito por el usuario, su equivalente fonético, y su sonido asociado, permite saber si la misma es pronunciable y por ultimo, permite reproducir el sonido.

Persistencia

Para el manejo de persistencia, decidimos realizar una estructura lógica común, esto nos sirve para ganar extensibilidad y reutilización de código.
La idea general es manipular archivos en forma binaria. De esta forma, utilizamos registros de longitud variable y no importa el tipo de archivo que tratamos.
Además, leemos cadenas de bytes y luego las clases responsables de los registros lógicos se encargarán de entenderlos. Para ello contamos con la interfaz registro.
Cada tipo de archivo (audio y diccionario) tienen asociado un registro, el cual implementa los métodos de la interfaz registro. De esta forma, nos desentendemos de como se manejan los archivos a nivel físicos (desde el punto de vista del Core).



Estructuras físicas

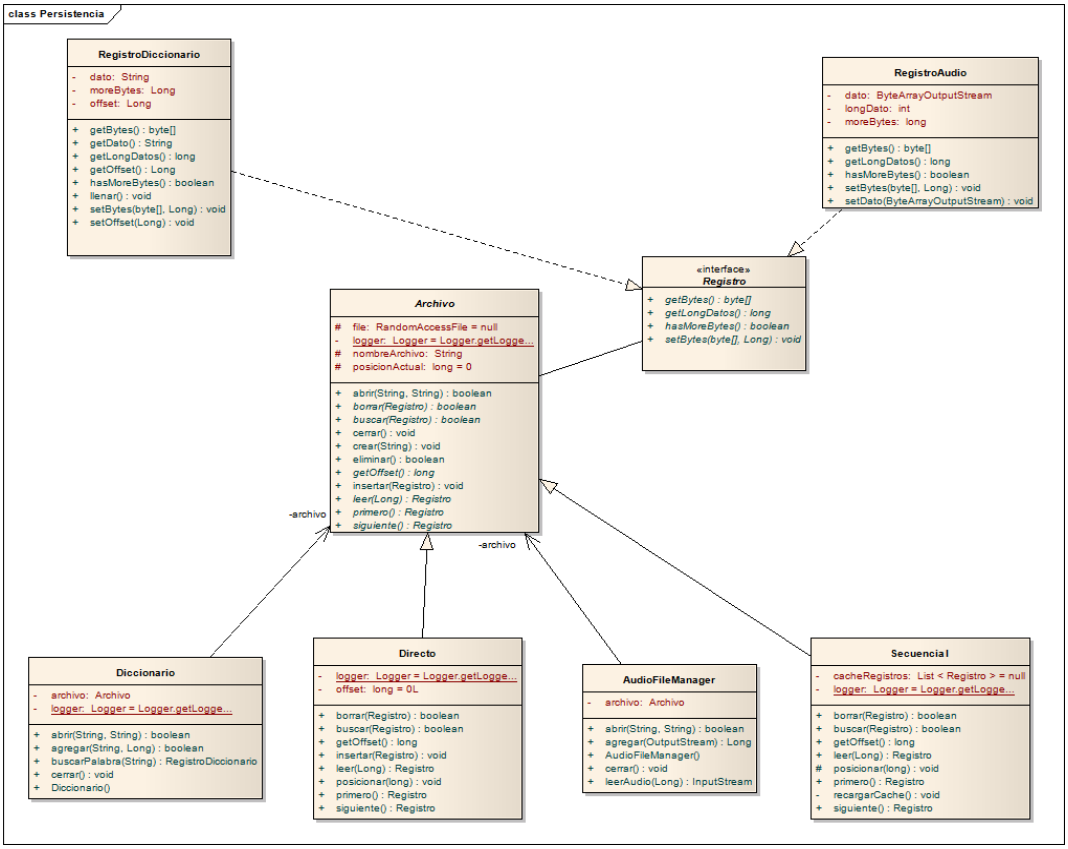
Estructura física del registro diccionario

8 bytes | 4 bytes | Palabra a guardar en el Diccionario (long. variable)

Estructura física del registro de Audio

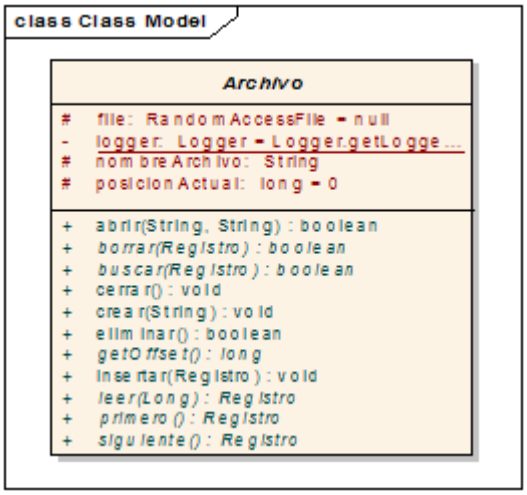
8 bytes | Cadena de bytes de Audio (long. variable)

Diagrama de Clases - Persistencia



Archivo

Esta es la superclase para manejo de archivos. La misma contiene los métodos comunes que utilizaremos para acceder y escribir (a nivel físico).

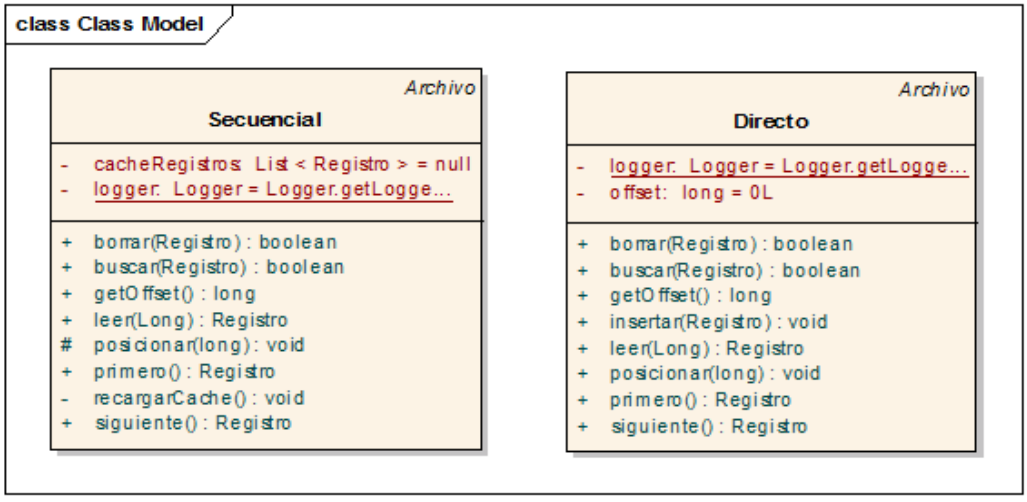


Secuencial

Esta clase sirve para manipular un archivo secuencial. En la misma se redefinen los métodos necesarios para poder implementar accesos secuenciales.

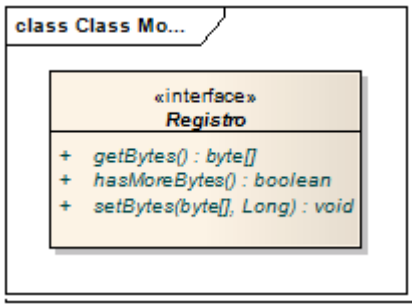
Directo

Esta clase sirve para manipular un archivo directo. En la misma se redefinen los métodos necesarios para poder implementar accesos directos. Luego, se podrá reutilizar parte de esta clase con manejo de índices.



Interfaz Registro

La misma tiene los métodos que utilizaremos para manipular los registros. Gracias a la misma podemos implementarla para todos los tipos de registros (lógicos) que necesitemos, y utilizarlos a nivel físico como cadenas de bytes.

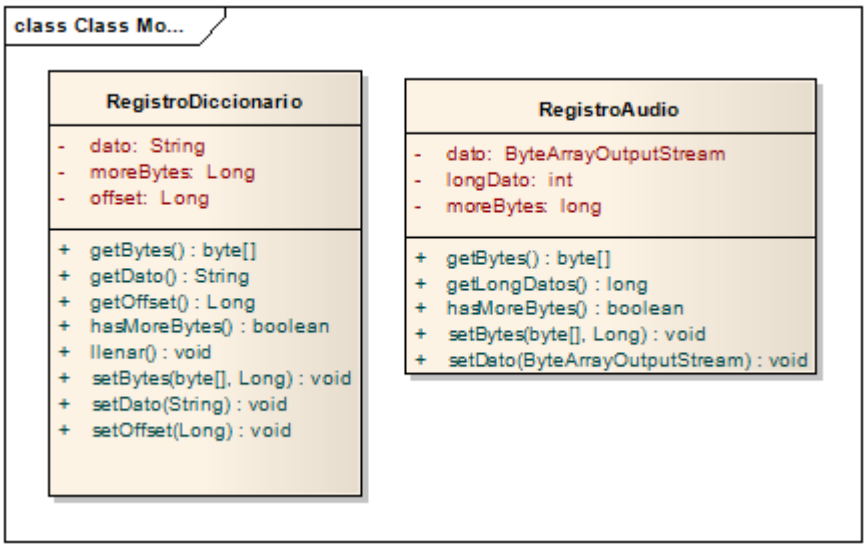


Registro Audio

Esta clase es la que maneja a nivel lógico la estructura de audio. Con ella convertimos el stream en un registro, que puede reproducirse, y viceversa.

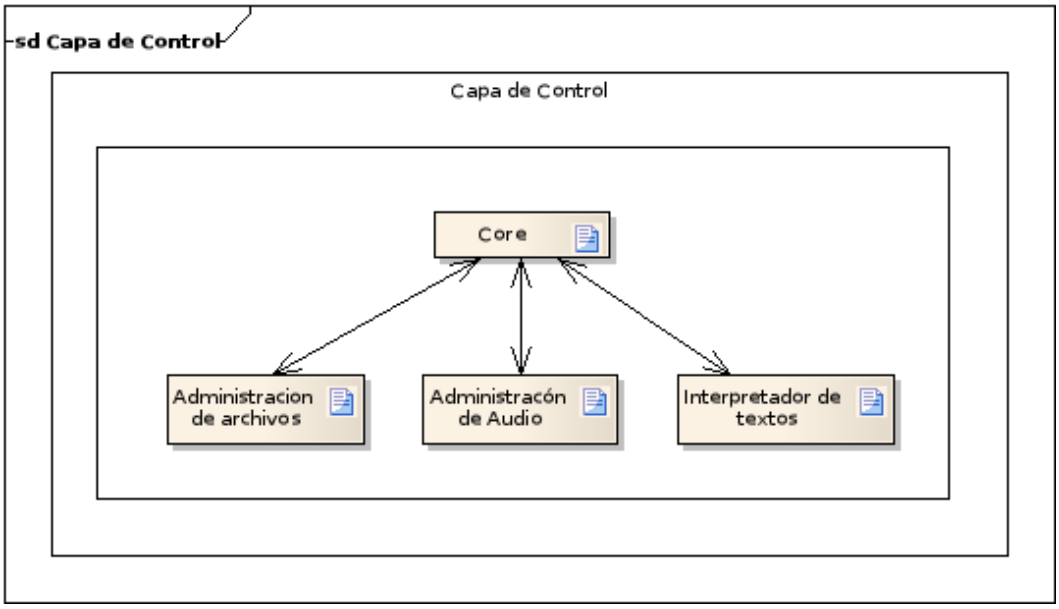
Registro Diccionario

Esta clase es la que maneja a nivel lógico la estructura del diccionario. Con ella convertimos la palabra (y la posición asociada al audio) en un registro, que puede leerse, y viceversa.



Capa de Control

El la capa de control esta basada en la relación del core con el bloque de administración de archivos, el de administración de audio y el interpretador de textos.



Bloque de Administración de archivos

Este bloque se encarga del control de la persistencia de los datos. Las clases que lo componen son:

1. AudioFileManager.
2. Diccionario

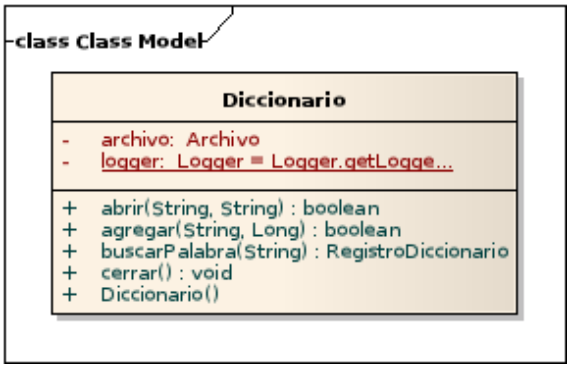
A continuación explicaremos dichas clases.

AudioFileManager

Esta clase es la que se encarga de la administración del archivos de audio, la misma permite manejar las acciones básicas sobre el archivo que contiene los datos de audio. Entre sus funciones mas importantes tenemos las que permiten agregar y leer los datos de audio.

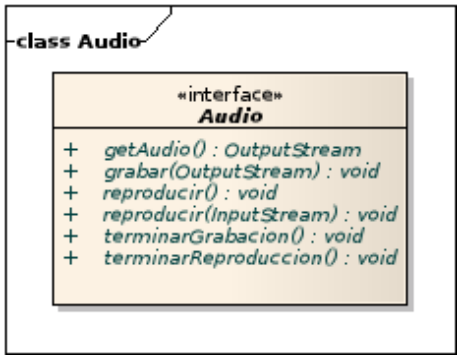
Diccionario

Esta clase es la que se encarga de la administración del archivo que contiene el léxico del sistema, la misma permite manejar todas las acciones referidas a la búsquedas e inserción de palabras utilizando un archivo secuencial.



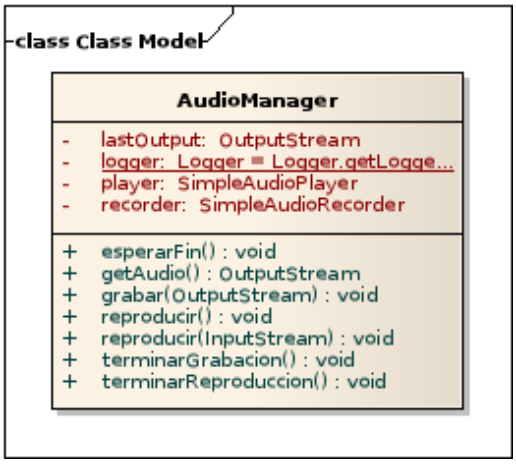
Bloque de administración de audio

Este bloque se encarga de la grabación en memoria y de la reproducción del audio grabado. Para poder manipular el audio se diseño una interfaz de audio, la cual contiene los métodos necesarios para grabar y reproducir. La implementación de dicha interfaz se da en la clase AudioManager.



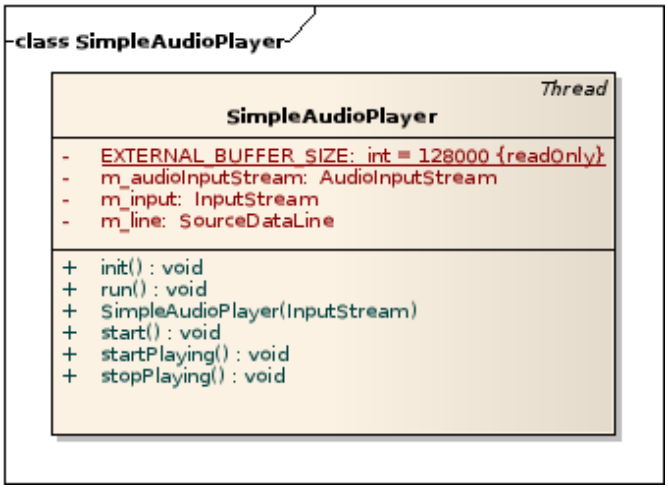
AudioManager

Esta clase es la que se encarga de realizar la grabación del audio de una palabra y así mismo también se encarga de reproducir una palabra. En esta clase solo se manipulan datos de audio, ya que la grabación del audio a un medio físico es tarea del AudioFileManager.

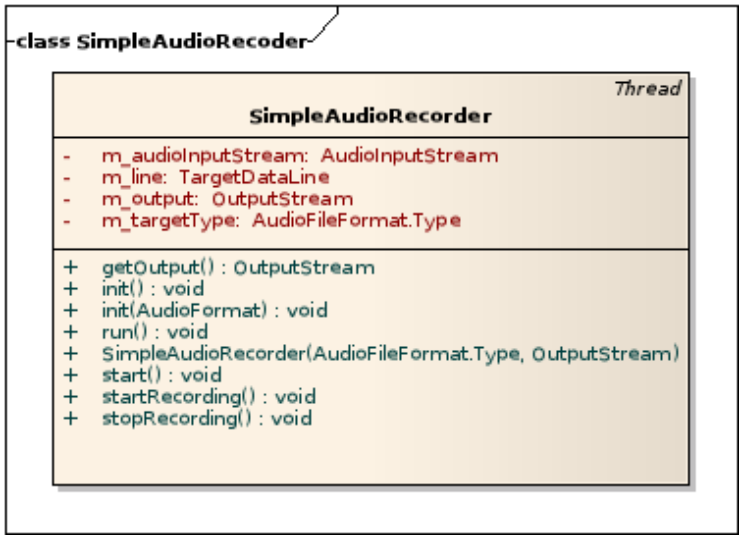


AudioManager para llevar a cabo la grabación y reproducción de audio utiliza la librería de audio. La misma tiene las siguientes clases:

SimpleAudioPlayer



SimpleAudioRecorder



Cuando se requiere grabar el audio que entra por el micrófono se utiliza la clase SimpleAudioRecorder ejecutando un hilo que es lanzado cuando el usuario ingresa el comando 'i', y es detenido cuando el usuario ingresa el comando 'f', dejando a disposición del core el audio grabado.

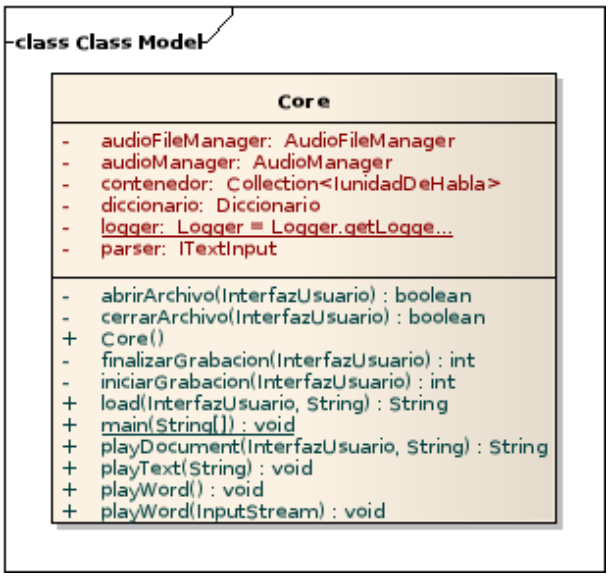
Cuando se requiere reproducir el audio de una palabra, recientemente grabada o leída del archivo de audio, se utiliza la clase SimpleAudioPlayer. Esta clase corre en un hilo, por lo tanto lo único que se hace es esperar que termine de correr lo que equivale a que termine de reproducir el audio.

Bloque Interpretador de textos

Es el encargado de analizar los textos que el core indica. Este bloque esta explicado con detalles en la sección “Interpretación de textos”

Core

Esta clase es el núcleo de la aplicación, es la que se encarga de vincular todos los módulos para llevar a cabo la grabación de palabras y la reproducción de textos.



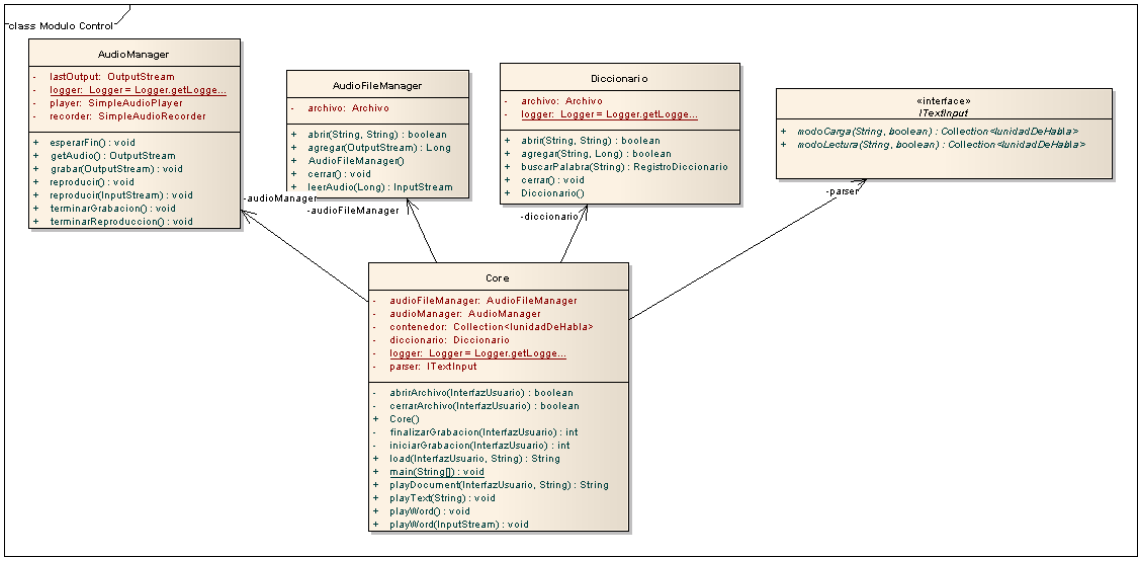
Grabación de Audio

Para llevar a cabo la grabación del audio el core trabaja con el parser para obtener una colección con todas las palabras distintas en el archivo de texto fuente sin importar el orden, las cuales son candidatas a ser grabadas al diccionario. Para determinar cuales, el core busca en el diccionario las palabras una por una, y si no las encuentra solicita al usuario que las grabe, una vez grabada se reproduce el audio y se espera la confirmación de parte del mismo. Si la grabación es correcta se procede a agregar la palabra al diccionario y la grabación al archivo de audio, luego se continua con la siguiente palabra y así hasta agotar las palabras candidatas.

Reproducción de palabras

Para llevar a cabo la reproducción de un texto el core trabaja con el parser para obtener una colección de todas las palabras que existen en el archivo respetando el orden, las cuales serán reproducidas. Para realizar la reproducción busca la palabra en el diccionario y luego, si la palabra existe, busca el audio a reproducir y lo reproduce, en caso de no existir la palabra no será reproducida. También se puede reproducir un conjunto de palabras ingresadas en la consola, la única diferencia que existe es que al trabajar con el parser no se le pasa el nombre del archivo sino la cadena con todas las palabras ingresadas.

Diagrama de clases - Core



Archivo de Configuración

A los fines prácticos de no tener que recompilar y poder darle más flexibilidad al programa, implementamos un archivo de configuración en XML, en el cual se especifican los parámetros necesarios. El mismo está configurado con los valores por defecto, como se muestra a continuación (algunos parámetros pueden no estar en este ejemplo, dado que solamente ejemplificador).

```
<?xml version="1.0" encoding="UTF-8"?>
<configuracion>

    <TAMANIO_BUFFER_LECTURA>128</TAMANIO_BUFFER_LECTURA>

    <TAMANIO_BUFFER_ESCRITURA>128</TAMANIO_BUFFER_ESCRITURA>

    <BUFFER_LECTURA_TEXT_INPUT>30</BUFFER_LECTURA_TEXT_INPUT>

    <TAMANO_CACHE>10</TAMANO_CACHE>

    <SPANISH_OPTIMIZATION_ACTIVATED>true</SPANISH_OPTIMIZATION_ACTIVATED>

    <DEFAULT_TEXT_INPUT_CHARSET>UTF-8</DEFAULT_TEXT_INPUT_CHARSET>

    <ARCHIVO_DICCIONARIO>/home/pepito/Speaker/diccionario.data</ARCHIVO_DICCIONARIO>

    <ARCHIVO_AUDIO>/home/pepito/Speaker/audio.data</ARCHIVO_AUDIO>

    <ESCAPES_REGEX>
        <![CDATA[((l{2}))|([b])|((?<!c)h)|((je))|((ji))|((mb))|((ce))|((ci))|((za))|((zu))]]>
    </ESCAPES_REGEX>

</configuracion>
```

Este archivo debe encontrarse en la misma carpeta del Archivo binario.