The Speaker (Trabajo Práctico Nro. 1)

Juan Manuel Barrenche, *Padrón Nro. 86.152*
snipperme@gmail.com
Martín Fernández, *Padrón Nro. 88.171*
tinchof@gmail.com
Raúl Lopez, *Padrón Nro. 88.430*
rau_carpo@hotmail.com
Marcos J. Medrano, *Padrón Nro. 86.729*
marcosmedrano0@gmail.com
Federico Valido, *Padrón Nro. 82.490*
fvalido@gmail.com

Grupo Nro. 11 (YES) - 1er. Cuatrimestre de 2009
75.06 Organización de Datos - Titula: Arturo Servetto
Ayudante: Renzo Navas
Facultad de Ingeniería, Universidad de Buenos Aires

Domingo 5 de Abril de 2009

Resumen

El presente trabajo representa una aplicación de los conceptos vistos durante el módulo de *Organización de Archivos* del curso de *75.06 Organización de Datos* de la cátedra Servetto.

Consiste en implementar un sistema que permita la lectura de textos y la reproducción de los mismos en audio. El usuario carga los audios correspondientes a cada termino encontrado en un archivo de texto. El sistema debe ser capaz de persistirlos, recuperarlos y de reproducir de un texto, cada termino individual en base a las grabaciones cargadas previamente.
Este documento ha sido desarrollado en LaTeX.

Índice

1. Introducción

En las siguientes secciones trataremos de explicar el funcionamiento integral de nuestra solución para el sistema ** *The Speaker* **.

El problema planteado, para esta primera entrega, consta de dos partes o funcionalidades principales. El sistema debe ser capaz de cargar textos, durante esta carga, cada termino dentro del texto que no exista debe permitirle al usuario grabar el audio asociado al mismo. Es decir, que se registrará esa palabra y se la asociará al audio capturado por el micrófono del usuario y que el mismo confirme. La otra funcionalidad importante es la de reproducción de un archivo completo. Esta opción recorrerá todos los terminos del archivo que el usuario elija y reproducirá su audio asociado. Los terminos desconocidos no serán tenidos en cuenta para la reproducción del audio.

La solución fue implementada en Java. Intentando concentrarse en dejar la suficiente flexibilidad en el sistema para aplicar modificaciones para las siguientes entregas que requieren la ampliación de funcionalidad en el programa y optimización de la recuperación de la información manejada. Para lograr esto se realizó una división del sistema en módulos, como se puede ver en la definición de la arquitectura, con responsabilidades claramente delimitadas.

La división de trabajos fue en base a estos módulos (descritos en la sección de arquitectura). Los diferentes módulos fueron implementados por diferentes miembros del grupo en colaboración con los responsables de los módulos que tuvieran interacción con el modulo en cuestión. En una reunión inicial se bosquejaron las ideas para la solución. Que luego fueron separadas en módulos y asignados a cada uno de los integrantes.

Para el manejo de la información que administra ** *The Speaker* ** y la organización de los datos nos basamos en lo explicado durante las clases teóricas dictadas en la materia.

La documentación, exceptuando el manual del usuario, fue creado con LaTeX.

[1] Para la documentación del usuario se recomienda leer el documento *user.pdf* que se encuentra en la ruta docs/user/

2. Arquitectura del Sistema

Se busco encontrar una solución general, siempre que fuera posible, a los problemas particulares planteados. Para ello en primer lugar se dividió el problema principal en varios subproblemas de alcance acotado.

2.1. Módulos

Se definieron entonces los siguientes módulos:

- Parser: Se encarga de la normalización del texto y extracción de las diferentes palabras. Es utilizado por el programa principal, pidiéndole el listado de palabras normalizadas de un archivo.
- AudioService: Abstracción de las clases de manejo de audio proporcionada por la materia. Se incluye un InputStream para proporcionar a la biblioteca de audio que facilita la serializacion de ese InputStream. Es utilizado por el programa principal para la obtención y reproducción de audios.
- Buffers: Permiten el intercambio de datos con los archivos, funcionando como una memoria temporal donde se acumulan los datos leidos y los que se van a escribir. Son utilizados por el FileManager como herramienta para la lectura y escritura de datos en conjunto con los serializadores. Es la herramienta de intercambio de datos entre el FileManager y los Serializadores.
- Serializers: permiten realizar la transformación de los datos hacia un tipo unificado que pueda ser almacenado en los archivos y su posterior recuperación. Brinda una interfaz unificada para que el FileManager pueda realizar la conversión de los datos recibidos. Intercambia los datos con el FileManager mediante el uso de Buffers. Se hicieron implementaciones genéricas (para los datos primitivos) e implementaciones customizadas para nuestros registros que usan auxiliarmente a las genéricas.
- FileManager: Manejador de archivos de registros de longitud variable en bloques. Es una implementación generica que se customiza indicando el tamaño de los bloques y el serializador a usar. Permite el encadenado de bloques de manera tal que un registro serializado que exceda un bloque pueda ser guardado. Utiliza auxiliarmente a los Buffers para la acumulación de datos desde y hacia los archivos y para la interacción con los serializadores.
- PersistenceService: Es, principalmente, una customización (dos en realidad) del FileManager para la interacción con los archivos de los requerimientos; esta customización implica la definición de los serializadores (dos también) adecuados para los registros a usar. Además provee herramientas al principal para la realización a un nivel más alto de las acciones con los archivos requeridas.

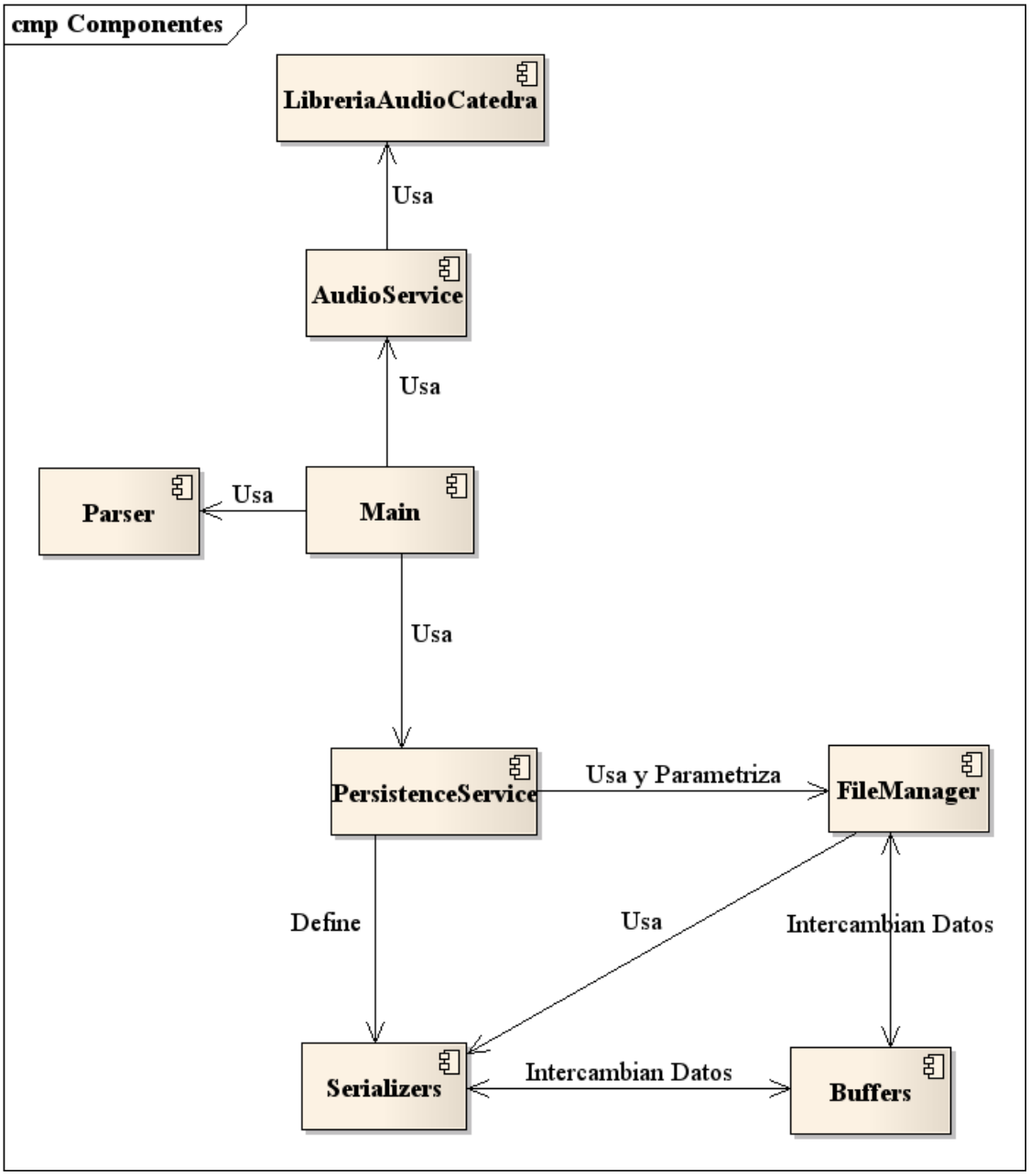

Figura 1: Diagrama de **Módulos**

2.2. Definición de los datos

La definición de los datos sigue la propuesta por los requerimientos, con la salvedad de que, dado que se usaron archivos con manejo de bloques, el offset del archivo de palabras fue dividido en dos partes; la primera indica el número de bloque que almacena el sonido correspondiente; la segunda, el número de objeto dentro de ese bloque. Físicamente, los bloques contienen información de control extra para el encadenamiento de bloques ya mencionado, pero esto se explica mejor en la sección correspondiente al FileManager

3. Manejadores de Archivo

Los datos que manejara **The Speaker** tenían, como requerimiento técnico, que ser persistidos en dos archivos separados y debían utilizar la siguiente estructura lógica.

1. `((palabra)i, offset)`
2. `(stream de audio)`

El primero está compuesto de la dupla palabra (que es un identificador) y offset/referencia. Simbolizando que para la *palabra* el audio se encuentra en *referencia*. El segundo archivo contiene los streams de audio capturados.

3.1. Análisis

Lo primero que se observa de ambos archivos es que sus registros son de longitud variable y que hay homogeneidad en los datos a almacenar, es decir, que en cada archivo se almacenan siempre los mismos tipos de datos. De manera que, ambos archivos, a pesar de tener naturalezas de datos diferentes requieren el mismo manejo. Por lo cual, admiten una misma solución de manejo del archivo mientras que la misma se mantenga indepen de la naturaleza de los datos a almacenar.

Por otra parte, las operaciones que se deben permitir son el agregado de registros y la consulta de los mismos. El agregado de registros no requiere, en ninguno de los dos casos, que se haga con un orden específico. Mientras que la recuperación de los datos, por otra parte, en el primer archivo debe poder ser secuencial (ya que se necesitan poder acceder a cada una de las palabras) y en el segundo caso debe poder accederse directamente a un registro que conozco su posición dentro del archivo. Por lo cual, estamos ante una organización secuencial de acceso relativo. Pudiendo recorrerse tanto secuencialmente como acceder a un registro específico (si se conoce previamente su dirección).

3.2. Solución propuesta

Se utilizarán instancias de una clase llamada **VariableLengthFileManager** para abstraer a cada uno de los archivo. Esta clase define el comportamiento tanto, de la carga de registros, como de las dos formas diferentes de recuperación de registros (completa y secuencial, y, de un único registro y direccionada).

Para que la misma clase pueda persistir archivos con registros de diferente naturaleza se implementaron los serializadores *(ver sección 5.)* El serializador es configurado en cada archivo a manejar y realiza las dos conversiones necesarias:

- la tira de bytes leida en un objeto (Mapeo)
- un objeto en una tira de bytes que será grabada (Serialización)

Esta clase no manejará directamente el acceso al archivo físico si no que delegará en un fino wrapper de la clase **RandomAccessFile** que se encargará del manejo de los datos en bloque. *Ver figura 2.*

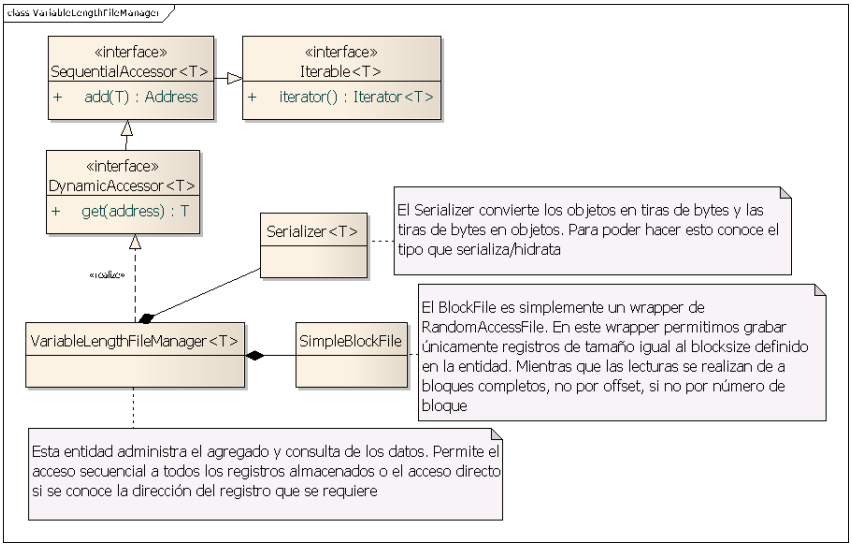

Figura 2: Diagrama de clases del **VariableLengthFileManager**

El manejo de este archivo es simple, a medida que se le solicita agregar objetos los serializa con el serializador con que fue configurado y los agrega al último bloque (esto no significa que se escriba en este momento). Si se le solicita algún objeto en particular, mediante la dirección del mismo, este manejador de archivo accede al bloque que indique la dirección y mapea los datos del objeto correcto utilizando el mismo serializador.

3.2.1. Agregado de registro

El agregado de registros, como se mencionó anteriormente, primero serializa el objeto y luego lo agrega al último bloque. Este último bloque siempre se encuentra cacheado ya que se graba (o regraba) cuando esa caché, de último bloque, desborda, es decir, su tamaño supera el tamaño designado para datos del bloque. En ese momento se graban todos los registros que estaban en caché menos el último agregado. Si, este último agregado, tuviera un tamaño mayor al tamaño de designado para datos del bloque el mismo es dividido en n bloques y todos esos bloques son grabados. *Ver figura 3.*

3.2.2. Lectura de todos los datos

Se implementó un iterador de todo el archivo que comienza en el bloque cero, mapea todos los datos de ese bloque a objetos y los va devolviendo de a uno. Luego de devolver todos los de ese bloque, pasa al siguiente bloque y realiza la misma operatori a. Esto se repite hasta que no queden mas datos por hidratar. *Ver figura 4.*

3.2.3. Lectura de un objeto dada una dirección

El manejador de archivo lee el bloque desde el archivo físico (excepto que el mismo esté en caché), y luego pasa los datos leidos por el serializador. Luego

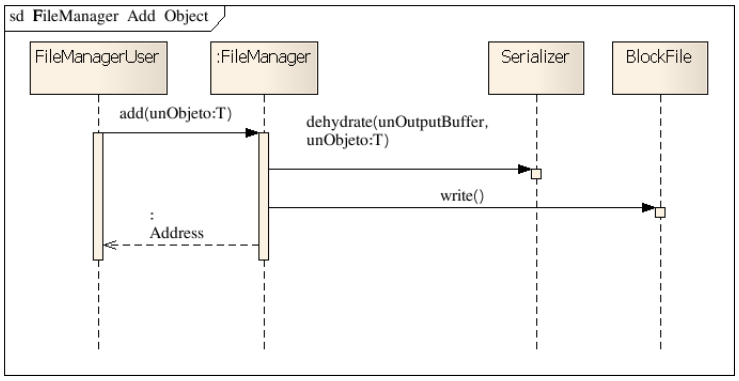

Figura 3: Diagrama de secuencia que muestra el agregado de un registro

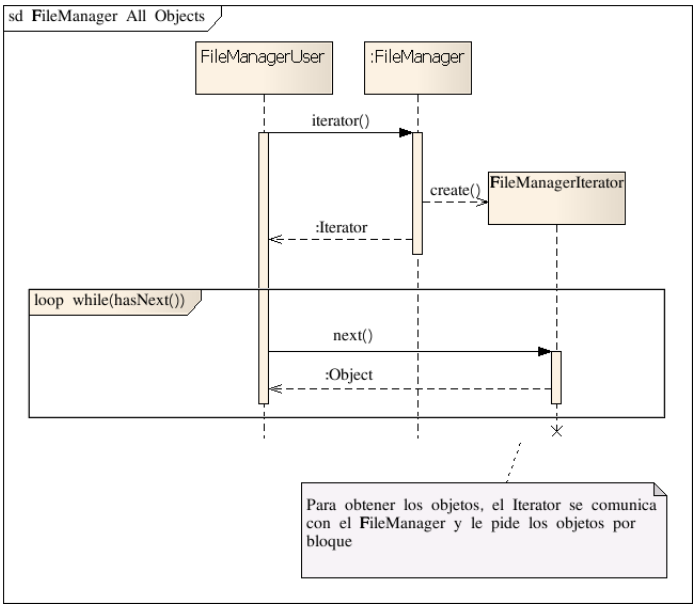

Figura 4: Diagrama de secuencia que muestra la iteración sobre todos los registros

busca entre los objetos creados el que tenga la posición indicada por la dirección para poder devolverlo a quién se lo haya solicitado. *Ver figura 5.*

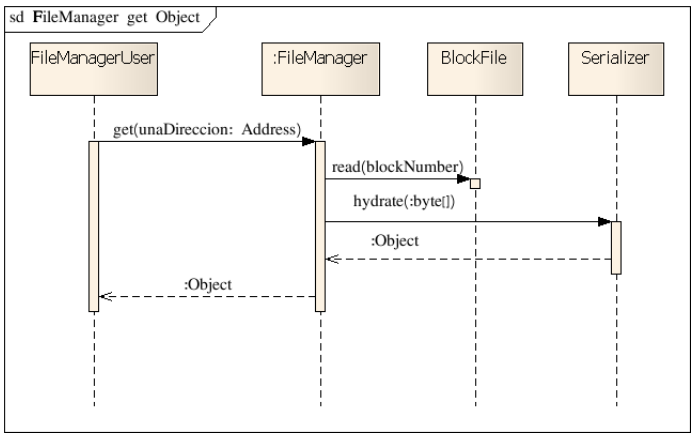

Figura 5: Diagrama de secuencia que muestra la recuperación de un único registro

3.3. Detalles técnicos

Los bloques cuentan con la siguiente información de control. Los últimos 2 bytes indican (almacenado como un Short signado) la cantidad de registros enteros que posee el bloque (esto se utiliza al momento de hidratar, para no intentar hidratar mas registros de los que se encontraban almacenados), para el caso que el registro esté en múltiples bloques este valor se marca en cero y se toman los 8 bytes anteriores para indicar la posición del próximo bloque que contiene datos del mismo registro.

Se implementaron 2 cachés, muy básicas, la primera, ya fue mencionada, contiene el bloque actual donde se están agregando registros. La segunda contiene el último bloque leido del disco (para disminuir accesos a disco)

4. Buffers de Entrada y Salida

El módulo de Buffers es quizás el mas sencillo de todos. Los buffers funcionan como una memoria de almacenamiento temporal de la información. Normalmente, se hace uso del buffer como una memoria intermedia, útil para almacenar información que esté por escribirse en un archivo o que haya sido leído de uno. Los buffers suelen mejorar el rendimiento en el intercambio de datos entre diferentes módulos de la aplicación.

4.1. Análisis

Los requisitos son claros: se necesita un Buffer de Entrada, del cual se leerán los datos cargados del archivo, y un Buffer de Salida que será utilizado para escribir los datos que necesiten ser persistidos:

- Buffer de Entrada: Debe poder ser cargado con los datos que se lean de los archivos y permitirá recuperar dichos datos para ser utilizados en la aplicación.
- Buffer de Salida: Debe poder ser cargado con los datos que necesiten ser persistidos.

4.2. Solución Propuesta

En principio, se crearon dos interfaces sencillas para cada buffer, `InputBuffer` y `OutputBufer`, y se realizaron dos implementaciones de estas interfaces que agregaron algo de comportamiento requerido por otros módulos de la aplicación. *Ver figura 6.* [1]

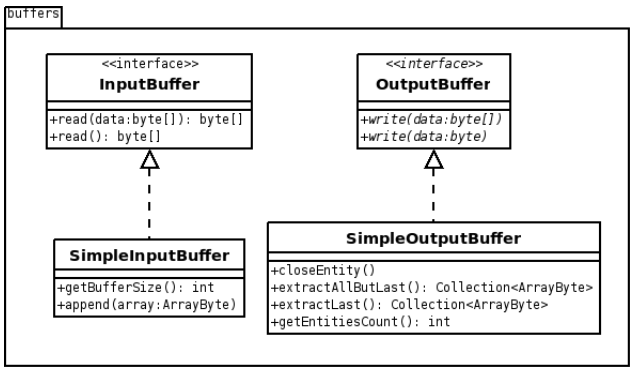

Figura 6: Diagrama de clases básico de los Buffers de Entrada y Salida

Durante el desarrollo de los otros módulos fué necesario agregar funcionalidad extra a los buffers.

[1] Las interfaces `InputBuffer` y `OutputBufer` se encuentran dentro del paquete `ar.com.datos.buffer`.

4.2.1. InputBuffer

La interfaz `InputBuffer` permite operaciones de lectura sobre la información cargada en el buffer.

Lectura del buffer. `InputBuffer` expone dos métodos cuyas firmas son:

```
public byte read() throws BufferException;
public byte[] read(byte[] data) throws BufferException;
```

El primer método permite leer el próximo byte desde el `InputBuffer`. El segundo método realiza una operación similar, pero en vez de leer el proximo byte, lee una cantidad **N** de bytes desde el buffer, siendo **N** el tamaño del byte array recibido por parámetro.

En ambos casos, si no hay suficientes datos para leer desde el `InputBuffer`, se lanza una excepción indicando dicha situación.

SimpleInputBuffer. La clase `SimpleInputBuffer` es una implementación simple de la interfaz `InputBuffer`. Básicamente consta de un array de bytes que representa el buffer interno y una posición que indica hasta donde se ha leído del buffer.

Esta implementación agrega una funcionalidad extra necesaria por el Manejador de Archivos de Longitud Variable: el método `append`. [2]

```
public void append(ArrayByte array);
```

Este método permite cargar los datos de array pasado por parámetro en el buffer. Debido a que el Manejador de Archivos es incapaz de conocer la cantidad de información que necesitará almacenar en el buffer, este metodo se encarga de ir agregando datos al buffer sin imponer ningún limite de datos.

4.2.2. OutputBuffer

La interfaz `OutputBuffer` permite almacenar información con el objetivo de poder persistirla luego.

Escritura del buffer. `OutputBuffer` expone dos métodos cuyas firmas son:

```
public void write(byte data);
public void write(byte[] data);
```

Ambos métodos permiten escribir en el OutputBuffer una cierta cantidad de bytes. Estas operaciones incrementan el tamaño del buffer de manera que pueda conocerse en cualquier momento la cantidad de bytes que se han escrito en el mismo.

[2] Ver Sección: Manejadores de Archivo

SimpleOutputBuffer. La clase `SimpleOuputBuffer` es una implementación de la interfaz `OutputBuffer`. Esta implementación se relaciona bastante con la clase `VariableLengthFileManager` ya que expone métodos que le son útiles al mismo. [3] A continuación se expone una descripción breve del funcionamiento de esta clase. *Ver figura 7.*

Funcionamiento del SimpleOutputBuffer. El `SimpleOutputBuffer` es el destinatario de los datos que el manejador de archivos intenta persistir. Este se encarga de ir agregando bytes de datos al buffer. Debido a que los registros a almacenar son de tamaño variable, necesita indicarle cuando se ha llegado al final de una entidad, de manera que el buffer pueda saber cuantas entidades se estan almacenando. [4]

La clase `SimpleOutputBuffer` precisa de un `BufferReleaser` que será el responsable de liberar el buffer en el caso de que el mismo haya sido desbordado. Cuando se agrega una entidad que sobrepasa el tamaño del buffer, este informa a su releaser para que el pueda liberar el buffer.

La operación de liberar el buffer se realiza luego de la operación que generó el desborde. De manera que el `SimpleOutputBuffer` tiene que llevar dos buffers internos. Uno con las entidades que han entrado perfectamente en el buffer y un buffer temporal en el que se irá almacenando la ultima entidad no cerrada. Al momento de cerrar dicha entidad si hay espacio en el buffer, se la pasa al buffer permanente, sino, se la deja en el buffer temporal, y se notifica del desborde.

Por último, la clase que utiliza el buffer, tiene que ser capaz de recuperar por un lado, las entidades que entraron perfectamente en el buffer, y por otro lado la entidad que no entró y que generó el desborde.
Con estas explicaciones, debería quedar claro el significado de cada método:

```
// Marca el fin de una entidad y avisa si hay exceso
public void closeEntity();

// Indica si ha habido un desborde en el buffer
public boolean isOverloaded();

// Cantidad de entidades que fueron agregadas
// (usando el metodo closeEntity)
public Short getEntitiesCount();

// Extrae del buffer los datos de la ultima
// entidad que se cerró.
public Collection<ArrayByte> extractLast();

// Extrae del buffer todos los datos que fueron
// escritos excepto los de la ultima entidad
// que se cerró
public Collection<ArrayByte> extractAllButLast();
```

[3] Ver Sección: Manejadores de Archivo
[4] Cuando hablamos de **entidad** estamos haciendo referencia a un **registro**

4.2.3. ArrayByte

La clase `ArrayByte` encapsula un byte array (`byte []`) y permite generar subarrays a partir de la misma. Es utilizada en los buffers y también por otros módulos ya que amplía las funcionalidades de un array de bytes.

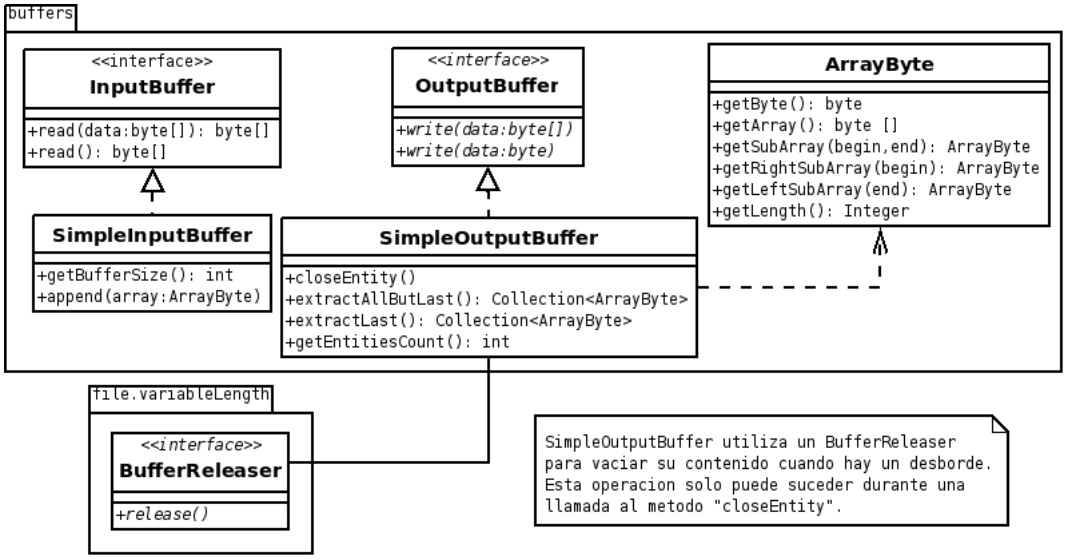

Figura 7: Diagrama de clases de los Buffers de Entrada y Salida

5. Serializadores

El intercambio de datos con los archivos se realiza en forma de bytes (es decir, de una manera unificada) a traves de InputBuffer y OutputBuffer. Debe existir, entonces, una forma de "serializar"los datos hacia bytes y de poder recuperarlos luego a partir de ellos. Para lograrlo, esta serialización debe incluir, además de las conversiones a byte, los datos de control necesarios, también en forma de byte, para poder luego rearmar los objetos (datos) originales. Para todo esto se definió la interface Serializador con tres métodos:

- dehydrate: Recibe un objeto a deshidratar y un OutputBuffer de destino. La implementación debe convertir a una tira de byte el objeto recibido, incluyendo los datos y la información de control. Luego esa tira es informada al OutputBuffer.
- hydrate: Recibe un InputBuffer del cual obtendrá una tira de bytes que usará para armar un objeto. La tira de bytes que obtendrá desde el InputBuffer contiene los datos y la información de control necesaria para la hidratación del objeto.
- getDehydrateSize: Obtiene de manera rápida (es decir, no realiza ninguna transformación, solo el cálculo) la cantidad de bytes que ocupará el objeto deshidratado.

5.1. Implementaciones

Se definieron implementaciones de los tipos más comunes de datos a serializar:

- En principio la de tipos simples primitivos: usando el contrato definido por el equipo de Java en las interfaces DataInput y DataOutput, y su implementación en RandomAccessFile como modelo, se definieron las operaciones de conversión entre tipos primitivos y tiras de bytes. Se definió una clase con métodos estáticos para estas conversiones llamada PrimitiveTypeSerializer, y Serializadores para cada uno de los tipos primitivos que delegan su comportamiento en la clase mencionada.
- Un serializador de colecciones para objetos de una clase que puedan ser serializados individualmente. La colección será serializada como CantObjetosObjeto1...ObjetoN . Para la serialización de cada objeto (1..N) se usa un serializador que parametriza de esa manera el serializador de colecciones. Por defecto la cantidad de objetos se serializa mediante el ShortSerializer, pero esto puede ser cambiado mediante el uso de un cardinalitySerializer diferente (y con esto variar la cantidad de bytes requerida para la serialización de CantObjetos). Las cantidades se manejan como unsigned siempre.
- Serializadores de String. Se hicieron dos implementaciones:
  - La primera pone la cantidad de caracteres al principio y luego la serialización de cada caracter. Para la implementación se delegó todo

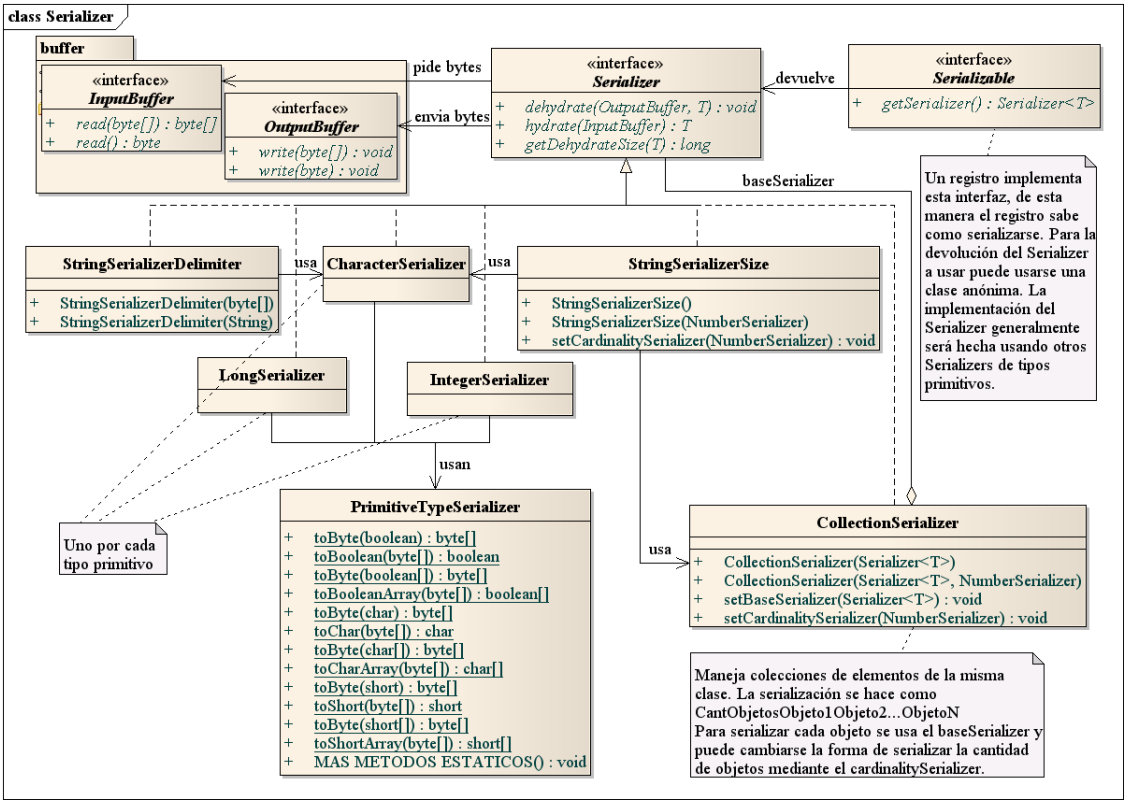

Figura 8: Diagrama de clases del **Serializer**

hacia un CollectionSerializer parametrizado con un CharacterSerializer y un ByteSerializer para la cantidad de caracteres (por tanto valen las mismas consideraciones de CollectionSerializer). El serializador de cantidad de caracteres puede ser intercambiado por otro.

- La segunda serializa cada caracter, apoyándose en CharacterSerializer, y pone una secuencia de bytes predeterminada (que puede ser intercambiada) al final de la tira de bytes.

5.1.1. Uso con registros: Alternativas y solución

Un concepto con el que se trabajó (y luego fue descartado) fue el de armar serializadores de manera dinámica: Para ello pensamos en una clase llamada DynamicSerializer permitía serializar colas de objetos de distintas clases. Se construye pasando un serializador base que será usado para serializar el primer elemento de la cola. Además se recibe un serializador para el siguiente elemento que será englobado ("wrappeado") como un DynamicSerializer y devuelto (lo que permite usarlo estableciéndole otro serializador como "siguiente", wrappeado y devuelto; así con cada tipo de elemento de la cola). De esta manera cada DynamicSerializer realiza la serialización de un elemento de la cola y delega la serialización del resto al siguiente, repitiéndose el proceso hasta el final. De esta manera un objeto complejo, compuesto por varios elementos simples para los que existe un serializador, pueden ser serializados sin necesidad de crear una nueva clase/implementación de serializer. Para ello primero deben tomarse los elementos simples que conforman el elemento compuesto y agregarlos a una cola en un orden preestablecido. Al hidratar debe tomarse la cola hidratada y tomando adecuadamente cada elemento de esta recomponer el objeto original. Es este armado/desarmado del elemento en la cola lo que nos llevó a abandonar está opción pues, a pesar de quitar la necesidad de crear una clase por cada nuevo serializador, ensuciaba en muchos lados el código por la necesidad de trabajar con la cola.

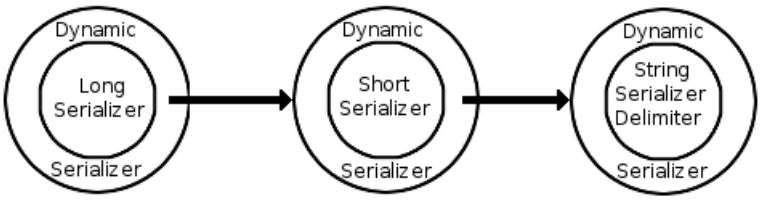

Figura 9: Ejemplo de **DynamicSerializer**

Fue entonces cuando se decidió definir otra interfaz llamada Serializable que define un método (getSerializer()) que permite conocer el serializador para un objeto en particular (el concepto sería ün serializable sabe como serializarse"). La implementación del serializador puede hacerse en forma de una clase anónima incluida en la clase a serializar dejando, entonces, el código mucho más limpio.

Una alternativa para lograr la serialización en forma dinámica era, usando como base la interface Serializer y sus implementaciones comunes, el uso de metadata a través de annotations. La idea sería marcar, mediante una annotation creada especialmente, los atributos a serializar y el serializador (esto último [especificar el serializador] puede, incluso, ser opcional pués puede poseerse un Serializer por defecto para cada clase) a usar para dicho atributo. Dicho serializador puede ser uno de los de tipos primitivos mencionados más arriba (u otra implementación de Serializer). También debe marcarse -mediante la misma annotation- el orden de serialización, es decir el orden que ocupará cada atributo deshidratado en la tira de bytes (y entonces, el orden en que debe procesarse la tira para hidratarla). Por último habrá un Serializador especial, que recibe una clase parametrizada, capaz de interpretar la annotation mencionada para realizar la deshidratación/rehidratación de cualquier objeto que use use adecuadamente dicha annotation. El problema con esta opción es que hace un uso intensivo de reflection, lo cual hace mucho más lento el proceso, por lo cual fue descartada.

6. Servicios de Persistencia

Si bien para la persistencia de los datos creamos los manejadores de archivos, es necesario un objeto que se encargue de administrar el acceso y la recuperación de los mismos a un nivel mayor de abstracción, es decir, a nivel de palabras y sonidos. Es necesario, además, de acuerdo a los requerimientos de entregas posteriores, que su interfaz sea independiente de la implementación de los archivos.

6.1. Requerimientos del servicio de persistencia

El servicio de persistencia encapsula el manejo de los dos archivos: el de palabras y el del audio. Debe presentar una interfaz que permita agregar palabras con su respectivo audio, consultar si una palabra está registrada y recuperar un audio a partir de su correspondiente palabra. Decidimos que una vez que la palabra fue registrada, no sólo no se pueda eliminar, sino que además, no se puede modificar su registro de audio. En este caso, el servicio debe informar que no es una acción válida.

6.2. Implementación del servicio de persistencia

De acuerdo a los requerimientos mencionados anteriormente, definimos que el servicio de persistencia debe implementar la interfaz **SoundPersistenceService**. En particular, para esta entrega, la clase **SoundPersistenceServiceVariableLengthImpl** lo hace utilizando archivos con registros de longitud variable organizados en bloques.

En la siguiente imágen podemos observar las características de esta interfaz y, posteriormente, analizaremos la implementación:

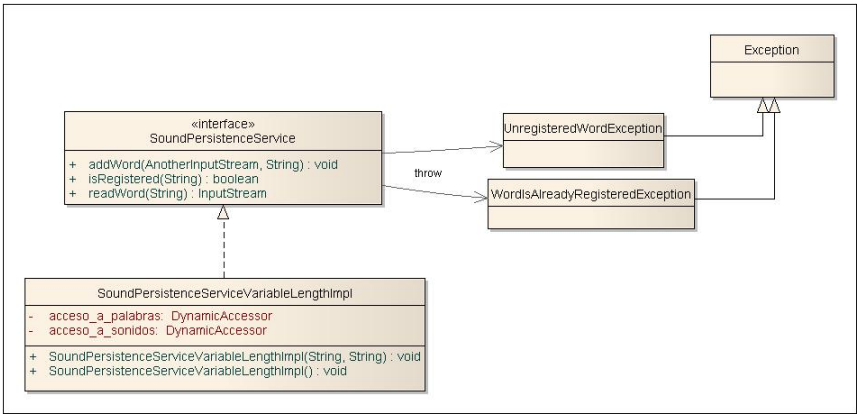

Figura 10: Diagrama de clases del SoundPersistenceService

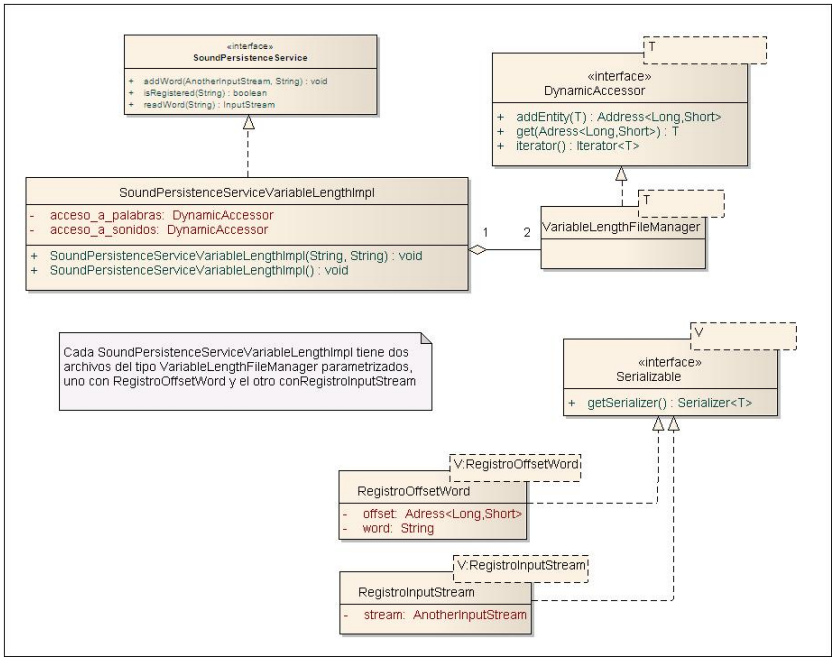

Figura 11: Diagrama de clases de implementación de SoundPersistenceService

6.2.1. Acceso a los archivos

De acuerdo al enunciado, existen dos archivos: uno que guarda como registros el par (palabra,offset) y otro que tiene sólamente (datosaudio). Cada registro está representado por las clases **RegistroOffsetWord** y **RegistroInputStream** respectivamente.

Cada uno de estos registros implementa la interfaz Serializable y, por lo tanto, no solo guardan datos sino, además, la información de como serializarse. Así, al momento de ser creado el archivo, se le pasa como parámetro el serializador correspondiente al tipo de registro que guarda. Ésto hace que el servicio de persistencia trabaje directamente con objetos serializados y no con cadenas de bytes.

Si bien los archivos tienen registros diferentes, el acceso a ellos es idéntico, y se hace mediante la interfaz **DynamicAccessor**.Es decir, que se tienen dos referencias del tipo **DynamicAccessor** a objetos que son instancias de **VariableLengthFileManager**.

6.2.2. Inserción de nuevas palabras

Para la inserción de una palabra, con su respectivo audio, primero se recorre todo el archivo de palabras verificando que no haya sido guardada antes. Si no está, se inserta el audio en el archivo de sonidos, se recupera la dirección en la que ha sido guardado y se agrega el registro (palabra,offset) en el archivo de palabras. En caso de que la palabra ya exista, y se quiera volver a insertar, independientemente de si el audio es el mismo o no, se lanza una excepción.

6.2.3. Recuperación de audio

Para la recuperación de un audio, se recibe como parámetro la palabra y se la busca recorriendo, en forma secuencial, el archivo correspondiente. Una vez que se la encuentra, se toma su offset y, a partir de él, se recupera el audio accediendo en forma relativa al archivo de sonidos. Al igual que en la inserción, en caso de que la palabra no esté, se lanza una excepción. Un tercer método del servicio de persistencia es ver si una palabra está guardada o no. Al igual que antes, se recorre el archivo de palabras y se devuelve el resultado.