The Speaker (3era Entrega)

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

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

Lunes 15 de Junio de 2009

Resumen

Documentación de la arquitectura y compresores utilizados en la 3era entrega del trabajo práctico del curso de *75.06 Organización de Datos* de la cátedra Servetto.
Se detallan las clases e interfaces principales y su interacción, pero no se hace referencia a la arquitectura utilizada en las entregas anteriores ya que el punto de contacto con la nueva funcionalidad es mínimo. Este documento ha sido desarrollado en LaTeX.

Índice

1. The Big Picture

Se implementaron los compresores como **Serializadores**. En nuestra arquitectura actual, los serializadores son utilizados en todos los manejadores de archivos (VLFM, StraightFM,...) para hidratar y deshidratar objetos.
Implementando los compresores de esta manera obtuvimos la ventaja de no tener que modificar nuestra arquitectura actual, que actualmente está funcionando correctamente. Simplemente se reemplazan serializadores actuales por los nuevos serializadores que serán capaces de comprimir, en nuestro caso, los Documentos.

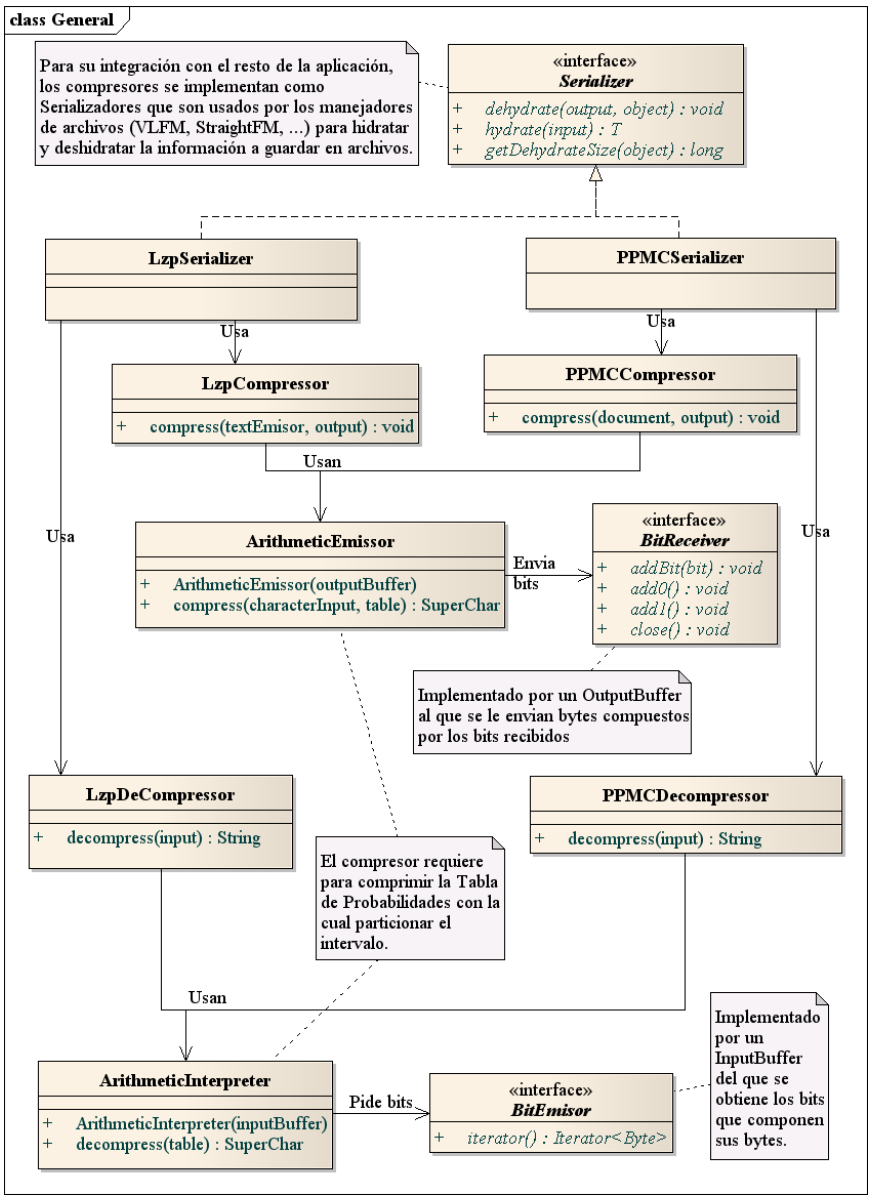

Figura 1: Diagrama de clases general de la arquitectura

Todos los serializadores que fue necesario implementar se apoyan, para su emisión, en el funcionamiento de un aritmético. Como se verá mas adelante, este proceso de emisión interactúa con otras dos clases claves de la solución implementada que son una tabla de probabilidades (*ProbabilityTable*) y una clase especial de (*SuperChar*) que permite la inclusión de símbolos específicos para los compresores (tales como **EOF** y **Escape**).

2. Probability Table

2.1. SuperChar

Se necesitaba una forma de representar símbolos, que fuera lo más amplia y flexible posible. Se inventó entonces el concepto de *SuperChar* que no es más que una abstracción colocada por sobre un entero. De esta manera pueden representarse los caracteres de un alfabeto de 16 bits junto con "caracteres" especiales como EOF y ESC. Además se puede utilizar para representar longitudes (es decir, que los símbolos sean números y no letras), lo cual era necesario para el compresor LZP.

La interface *SuperChar* define un método llamado *matches()* que permite saber si un SuperChar concuerda con otro. Concordar en este caso no es lo mismo que ser igual. Por ejemplo, existe un caracter especial llamado *ESC* que tiene la particularidad de concordar con cualquier otro SuperChar.

En adelante se usarán indistintamente los términos SuperChar y símbolo.

2.2. Interface ProbabilityTable

La tabla de probabilidades cumple con la interfaz requerida por el *Aritmético*:

1. *iterator()*: Permite iterar la tabla de probabilidades obteniendo en cada paso una tupla conteniendo un *SuperChar* y un Double correspondiente a la probabilidad de ese símbolo. El iterador será recorrido de manera que los símbolos más probables sean obtenidos primero, y, a igual probabilidad, se utilizará como criterio el valor entero del SuperChar (en el caso de SuperChars representando letras esto significará el uso del orden alfabético). La excepción a la regla de recorrida mencionada es *ESC* que en caso de existir en la tabla será devuelto siempre al final, cualquiera sea su probabilidad.

2. *getNumberOfChars()*: Permite obtener el número total de símbolos en la tabla.

3. *countCharsWithProbabilityUnder()*: Este método, junto con el anterior, permite responder a un problema posible del aritmético[1]. El método devuelve la cantidad de símbolos dentro de la tabla con una probabilidad inferior a la pasada.

2.3. Implementación de ProbabilityTable

La implementación de *ProbabilityTable* fue realizada mediante la representación de las frecuencias.

La clase *ProbabilityTableByFrecuencies* tiene:

1. Dos *SuperChar* correspondientes a un rango (los SuperChar son el mínimo y el máximo). Todos los símbolos de este rango, que no estén incluidos en los items siguientes, tienen una frecuencia de 1.

2. Un Set (conjunto, sin repeticiones [de SuperChar]) ordenado de tuplas formadas por un *SuperChar* y un Long correspondiente a la frecuencia, llamado *frequenciesTable*. El criterio de

[1] El aritmético debe garantizar que todos los símbolos tengan al menos una posición en el rango. Pero a veces, cuando varios símbolos tienen probabilidad muy baja y se encuentran en posiciones contiguas podría provocar que se les asigne a ambos el mismo intervalo (esto es debido a un problema de precisión). Lo que hace el aritmético para solucionarlo es calcular la probabilidad mínima para que a un símbolo le corresponda una posición del rango. Luego, con esa probabilidad mínima pide a la tabla de probabilidades la cantidad de símbolos con una probabilidad por debajo de ella. Esa cantidad la utiliza el aritmético para achicar el rango utilizado para calcular la cantidad de posiciones correspondientes a cada símbolo. Luego en las posiciones que restringió anteriormente pondrá los símbolos que estaban por debajo de la probabilidad mínima. La reducción del rango debe hacerse (repetirse) hasta que la cantidad de símbolos bajo la probabilidad mínima se "estabilice" (puesto que por cada reducción de rango la probabilidad mínima se hace mayor)

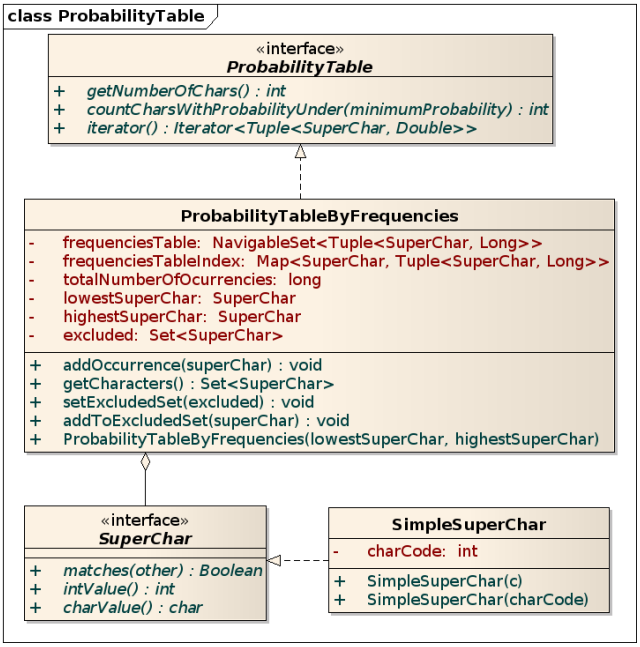

Figura 2: ProbabilityTable

ordenamiento es: primero por frecuencia descendente, y a igual frecuencia por orden de *SuperChar*. Esto permite recorrerlo secuencialmente usando este orden y encontrar subconjuntos con menos de cierta frecuencia. Pero no permite buscar, sin hacer una recorrida secuencial, un determinado SuperChar. Para solucionar esto, se tiene otra estructura: un Mapa cuya clave es un SuperChar y cuyo valor es una tupla de SuperChar y Long, siendo el SuperChar de la tupla el mismo que el de la clave; esta estructura se llama *frequenciesTableIndex*. Las tuplas correspondientes a los valores de *frequenciesTableIndex* son las mismas que están contenidas en el Set *frequenciesTable*.

3. Un Set (conjunto, sin repeticiones) sin orden de *SuperChar* llamado *excluded*. Los SuperChar contenidos en esta estructura no son contabilizados a la hora de calcular las probabilidades o de recorrer los elementos.

4. La sumatoria total de frecuencias de los items (se contemplan rango y excluidos). Usado para calcular la probabilidad de un símbolo.

Con estas estructuras, y algunos métodos explicados a continuación, puede lograrse la implementación de la interface *ProbabilityTable* para los diferentes casos requeridos por los compresores de manera muy rápida y eficiente.

A los métodos definidos por la interface *ProbabilityTable*, esta implementación agrega los siguientes:

1. *addOcurrence()*: Agrega 1 a la frecuencia de un símbolo.

2. *getCharacters()*: Obtiene los símbolos contenidos dentro de *frequenciesTableIndex*. Necesario para que PPMC pueda obtener la lista de exclusión para un modelo de orden inferior.

3. *setExcludedSet()*: Establece una nueva lista de exclusión. Usado por los órdenes 0-4 de PPMC.

4. *addToExcludedSet()*: Agrega un símbolo al Set *excluded*. Se hace notar que si nunca se llama a *setExcludedSet()* se arranca con un Set vacio. Esto es útil para el orden -1 de PPMC.

Además el constructor recibe el *lowestSuperChar* y el *highestSuperChar*, con lo que se establece el rango de SuperChars que tienen frecuencia inicial 1.

3. Procesado Aritmético

Todo el manejo de rangos propio del aritmético se concentró en la clase *ArithmeticProcessor*. La cual mantiene el estado de piso y techo y, además, conoce cuando deben generarse overflows y underflows. Cada vez que se le pide procesa un único paso de aritmético (sin importar si es compresión o descompresión) con una tabla que se le pasa por parámetro y un objeto que se encarga únicamente de decirle cual es la posición de la tabla de probabilidades que tiene que utilizar para quedarse como nuevo techo y piso (a este objeto se lo denomina matcher).

Esta clase es abstracta para que sus subclasificaciones definan la acción a realizar (comprimir o descomprimir) ya que no define ninguna interfaz pública. Las subclases dirán que se hace con el resultado del proceso de la tabla. Este procesado invoca métodos templates para hacerle saber a sus descendientes que está ocurriendo en el process (por ejemplo, avisa cuando ocurren overflows y de que bits, etc.).

Por ejemplo, para el caso de compresión, el objeto de matcher definido se basa en el caracter a comprimir y cada vez que ocurre overflow o se "limpian"los underflows acumulados emite el bit correspondiente.

En cambio, el caso de descompresión, toma de una entrada los bits y el matcher se basa en que valores tiene el rango para decidir en que momento debe para el proceso de la tabla. Cuando ocurren los overflow o underflows el mismo descarta de su valor actual dichos bits y solicita mas bits a la entrada. También verifica que los bits ocultados cuando ocurrió la el underflow sean los opuestos al primer overflow que ocurra.

Esto también nos facilitó el agregado de *tracers* para el aritmético de Orden 1 que se implementó para la prueba de este módulo.

3.1. Emisión y lectura

Debido a que la arquitectura de los Serializadores se basa en emisiones y lecturas de bytes (InputBuffer y OutputBuffer) pero el funcionamiento del aritmético emite y lee de a bits se implementaron dos clases cuyo único fin es adaptar esta diferencia de tamaños de manejo de datos. La primera de ellas es el *BitEmisor*, el cual toma información en bytes de un *InputBuffer* y entrega bits (esto lo hace iterando sobre los bits de cada byte). Su contraparte es el *BitReceiver*, al cual se le puede pasar datos de a bits, los junta hasta obtener un octetos que luego emite en un único byte en el *OutputBuffer* que se le haya configurado de salida. La contra que tiene es que se le debe forzar el última emisión por si no llegó a obtener los 8 bits.

3.2. Compresor y Descompresor Aritmético

Como se comentó anteriormente, ambas clases extienden del proceso aritmético. Tratan de a un caracter por vez para que los compresores que se utilizan dentro de su forma de trabajo un aritmético puedan usarlo sin problema y nunca modifican la tabla de probabilidades que reciben ya que los usuarios de estas clases son los encargados de hacer mantenimiento de las mismas.

El Compresor, como se mencionó anteriormente, sólo emite los overflow que el proceso le indique. El matcher que define se basa en el caracter a comprimir. Mientras que el descompresor toma bits de la entrada cada vez que necesita nueva información (esto ocurre cuando se detectan overflows o underflows).

3.3. Trace del proceso aritmético

Parte de los requerimientos era implementar una interfaz de consola para verificar el funcionamiento del compresor aritmético. Para ello se implementó un aritmético dinámico de Orden 1 (que

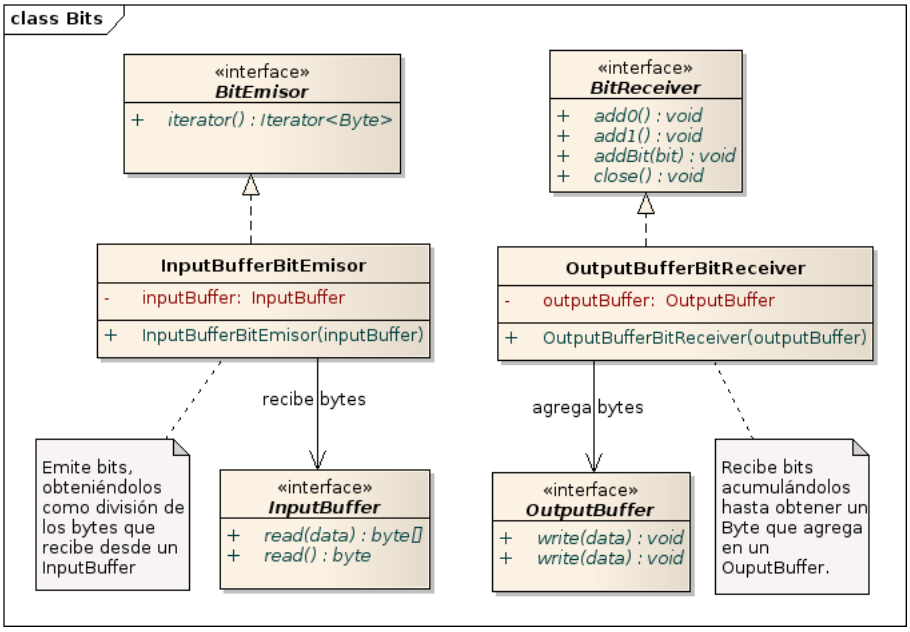

Figura 3: Diagrama de clases para BitEmisor y BitReceiver

maneja los contextos por medio de HashMaps) y que puede ser configurado con un PrintStream al que se le enviará, a modo de Log, información sobre que está sucediendo dentro del aritmético.

Estas capacidad de trace fue agregada en subclasificaciones de las clases de compresión y descompresión para manetener limpio el código de las mismas.

4. Compresión PPMC

Como en los demás compresores implementados, se realizó una separación entre el compresor y el descompresor. Las clases y las interfaces son relativamente simples. Existe sin embargo funcionalidad exactamente igual en ambos, con lo cual algunas cosas se generalizaron.

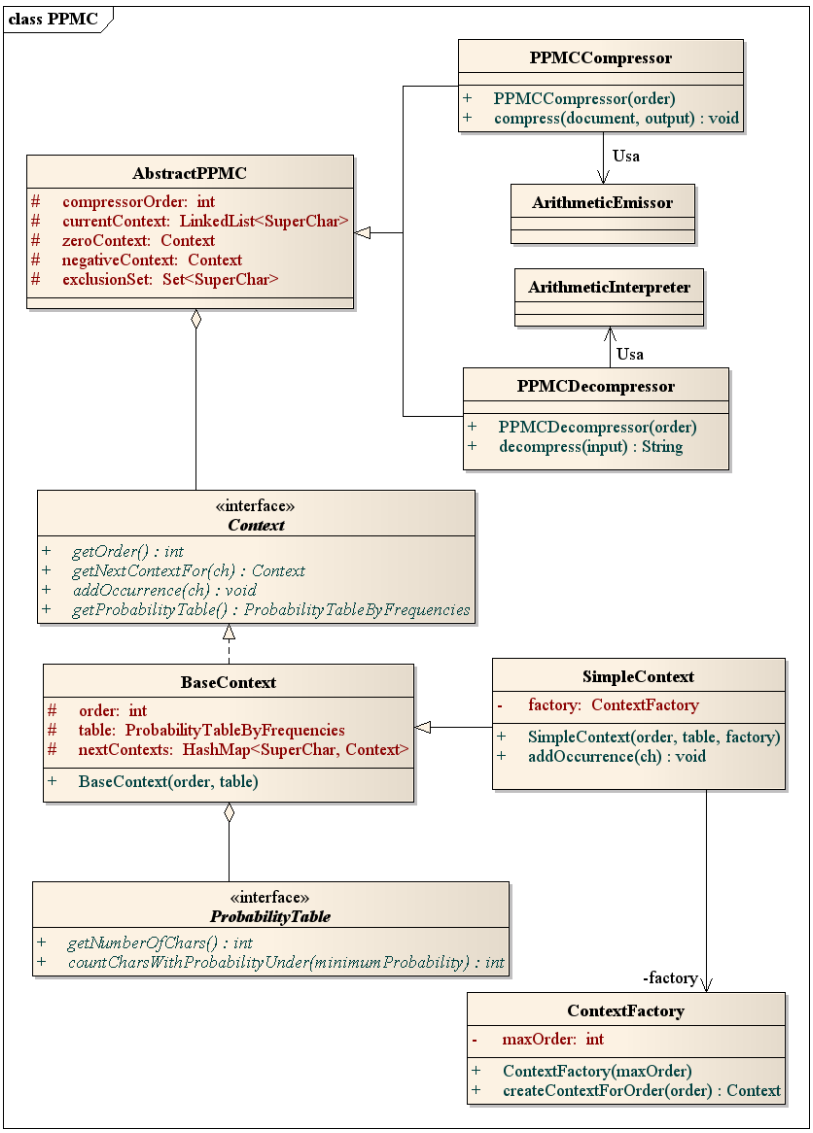

Figura 4: Diagrama de clases general de la arquitectura

4.1. Compresor PPMC

Se crea un compresor indicando el orden a utilizar (hasta 4) y opcionalmente puede recibir un PrintStream en el que irá logueando información de debugging que puede ser útil (que se está emitiendo en cada caso, y como quedan los contextos). Para comprimir un documento, se le debe

pasar al compresor un *Document* y un *OutputBuffer* en el que irá guardando el resultado de la descompresión. Al igual que en el LZP, el buffer es utilizado para crear un compresor aritmético. El aritmetico va comprimiendo los caracteres que el PPMC le vaya pasando, junto con la tabla de probabilidad del contexto adecuado (de esto último obviamente se encarga el Compresor PPMC). El aritmético envía los bits correspondientes a la compresión al buffer.

4.2. Descompresor PPMC

El descompresor se crea de la una manera identica al compresor, es decir, se le puede pasar el orden a utilizar y un PrintStream. Para la descompresión se le debe pasar un *InputBuffer* que contenga el mensaje a descomprimir. Se utiliza un Aritmético que interpreta los bits segun las tablas de probabilidad que le pase el descompresor PPMC. El aritmético lee los bits desde el InputBuffer recibido. Después de realizar la descompresión de todos los caracteres (y encontrar un *End-Of-File*) se devuelve un String que representa el mensaje original.

4.3. Contextos

Tanto el compresor como el descompresor utilizan una estructura idéntica de contextos para mantener un estado coherente. La forma de implementar esto, como se mostró en la arquitectura propuesta, fue la de crear una estructura de contextos encadenados. El compresor/descompresor PPMC tienen a su alcance los contextos de orden 0 y de orden -1. A partir del orden 0, se puede encontrar cualquier contexto utilizando una List que indique el contexto buscado (Por ejemplo puede contener [A,B,C] y se buscara el contexto correspondiente a .ABC"). Cada contexto tiene una tabla de probabilidad que registra las ocurrencias de caracteres para ese contexto. Se pueden agregar ocurrencias de caracteres al contexto, que repercutiran en agregar ocurrencias de caracteres en la tabla.

4.3.1. Tipos de Contextos

Se crearon 2 tipos de contextos, un contexto base con una implementación básica y un contexto que permite encadenar más contextos. Si bien no difieren demasiado entre sí, tienen la diferencia importante de poder o no encadenar contextos. El contexto base es utilizado en el orden -1 y en el ultimo orden posible. El otro tipo de contexto es utilizado en los ordenes 0 a n-1. Este último permite que al agregar una ocurrencia de un caracter al contexto, si el caracter no existía previamente en dicho contexto, se crea un nuevo contexto del tipo adecuado, y se linkea contra este contexto.

4.4. Algoritmo de Compresión y Descompresión

El algoritmo utilizado para la compresión y la descompresión es bastante similar en esencia. Se tiene una representación en caracteres del contexto actual y se utiliza para saber que contextos será necesario acceder. Se parte del contexto de orden 0 y se recorren los contextos recursivamente. Generalmente se llega al último contexto (de orden n) y se realiza el proceso de emisión recorriendo los contextos, finalizada está parte se hacen actualizaciones en las tablas si es necesario. El contexto de orden -1 es generalmente tratado de manera distinta, ya que aquí no hay que agregar ocurrencias de caracteres sino eliminar caracteres una vez que fueron emitidos.

4.4.1. Exclusión completa

Para aplicar exclusión completa simplemente se crea un *Set* (*HashSet*) al iniciar la descompresión y se van agregando los caracteres que necesitan excluirse. De esta manera cada vez que se va a

emitir algo, se aplica antes a la tabla de probabilidades este set de exclusión.

4.5. Serializer

Para integrar el compresor con nuestra arquitectura, fue necesario crear un *PPMCSerializer* que utiliza el compresor y descompresor PPMC para hidratar y deshidratar Documentos.

5. Compresión LZP

Se separó el compresor y el descompresor en clases aparte. El compresor recibe un *TextEmisor* (ver luego) y un *OutputBuffer* donde dejar los bits de la compresión. Mientras que el descompresor recibe un *InputBuffer* desde donde obtener los bits a descomprimir y devuelve un String representando la descompresión realizada.

El compresor utiliza el *OutputBuffer* recibido para crear un *ArithmeticEmissor* que recibirá las emisiones del LzpCompressor, y dejará estas emisiones representadas en bits dentro del *OutputBuffer*.

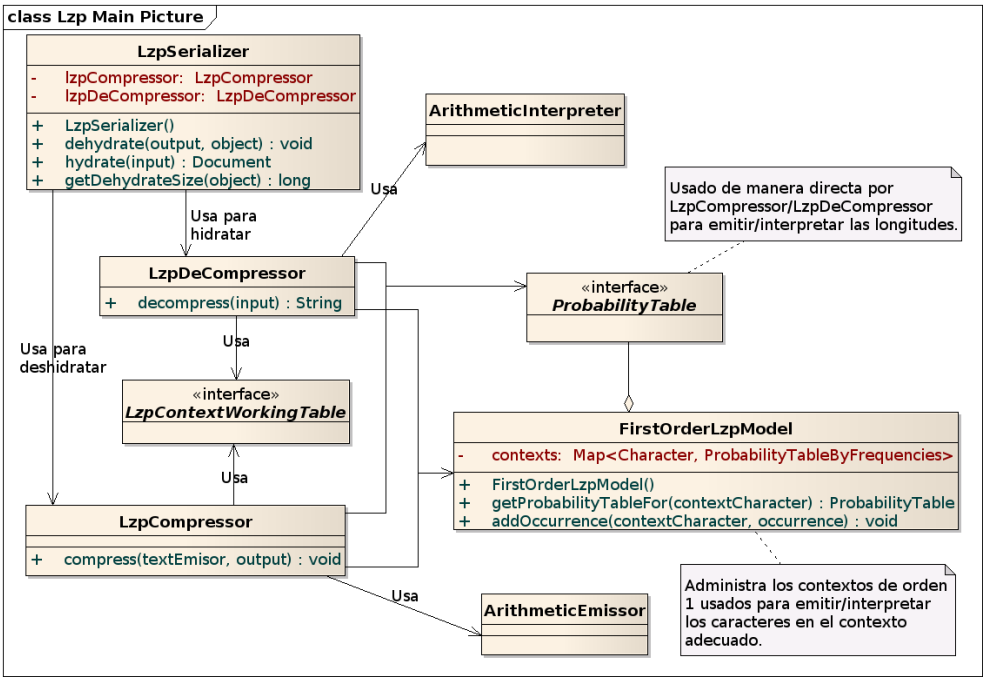

Figura 5: LzpMainPicture

El descompresor utiliza el *InputBuffer* recibido para crear un *ArithmeticInterpreter* que le brindará, a partir de los bits obtenidos desde el *InputBuffer*, emisiones al descompresor que utilizara para obtener la salida.

Tanto el compresor como el descompresor tiene una *ProbabilityTable* para manejar las probabilidades de las longitudes, y un *FirstOrderLzpModel* que les permite administrar las tablas de probabilidades de cada contexto para los caracteres. Cada vez que el compresor/descompresor hace uso del aritmético envia, junto con el símbolo (çaracter"[+EOF] o longitud) a comprimir/descomprimir, la tabla de probabilidades correspondiente. *FirstOrderLzpModel* mantiene las tablas de probabilidades para cada contexto, o las crea de manera lazy si no existían al momento de ser pedidas.

Además tanto el compresor como el descompresor utilizan una *LzpContextWorkingTable* para almacenar la última aparición de cada digrama.

5.1. TextEmisor y TextReceiver

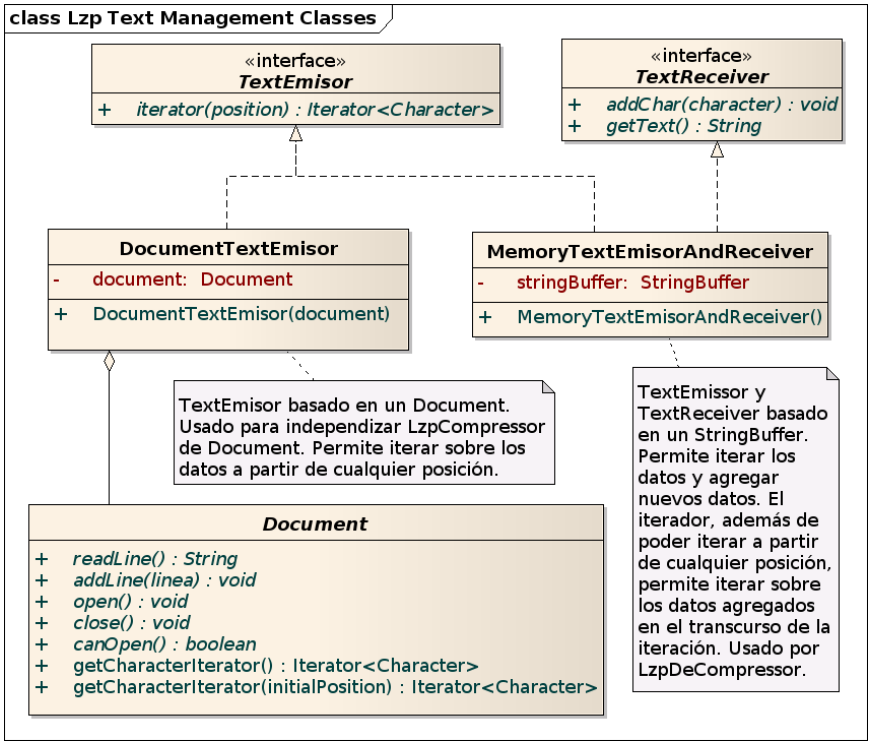

Figura 6: LzpTextManagementClasses

Un *TextEmisor* es un emisor de caracteres. Debe poder ser iterador a partir de cualquier posición.

Un *TextReceiver* es un receptor de caracteres. Debe poderse agregar caracteres al final de él.

DocumentTextEmisor es un *TextEmisor* basado en un *Document*. Permite la interacción de la compresión LZP con un Document sin acoplarlos. El *LzpCompressor* recibe esto.

MemoryTextEmisorAndReceiver es un *TextEmisor* y *TextReceiver* basado en un *StringBuffer*. El iterador, además de poder iterar a partir de cualquier posición del texto, permite iterar sobre los datos agregados en el transcurso de la iteración. Esto es algo necesario para que pueda llevar a cabo su función el *LzpDeCompressor*.

5.2. LzpContextWorkingTable

Por disponer solo de 4KB de memoria debía utilizarse una estructura que podamos medir su tamaño ocupado en memoria. Además se debía intentar economizar el espacio de memoria utilizado por ser muy poca la disponible.

Se eligió entonces el uso de un array primitivo, cuyos elementos son la unión (como si fuera un struct de c) de un char, otro char y un int (lo que llamaremos *LzpContextPosition*). Los chars se corresponden con los caracteres del contexto o digrama y el int con la longitud. El int se maneja como un *UnsignedInt* (una clase especial creada para la ocasión) para dar la posibilidad de disponer de una longitud máxima mayor (el doble) que si fuera signed. Todo esto nos da un total de 8 bytes por posición del array.

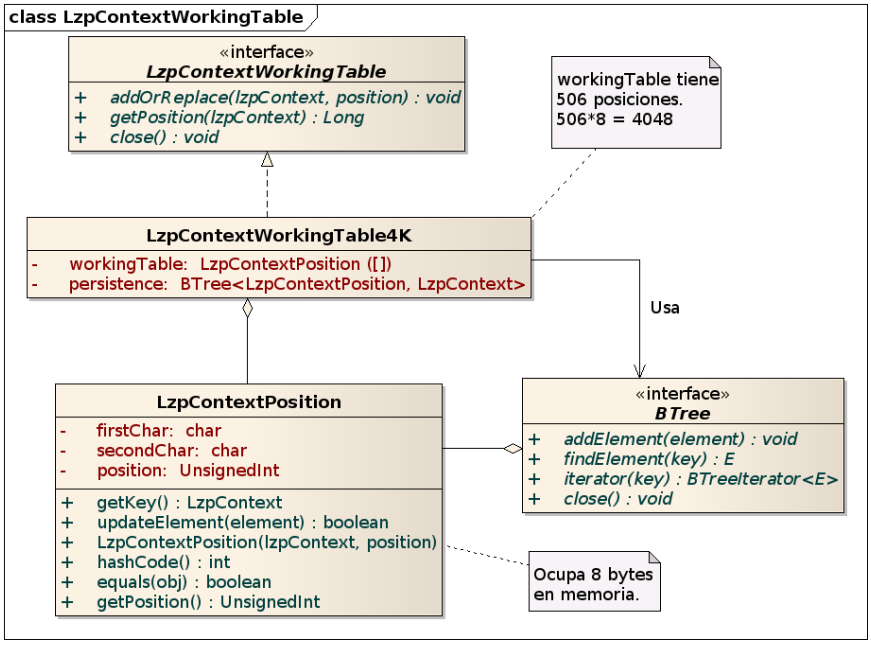

Figura 7: LzpContextWorkingTable

Para el manejo del array se eligió hacer una especie de hashtable basada en el vector. Es decir elegimos un módulo, en este caso 22. Se divide al array en 22 partes iguales, obteniendo 22 partes de 23 posiciones (22*23*8 = 4048 [los bytes restantes son usados en otras cosas o reservados]). Para ver en que parte debe ponerse o buscarse un digrama se obtiene el hashCode del digrama y se le aplica el módulo de 22. Si se buscara un contexto/digrama y no se lo encontrase en la porción elegida por el hashCode, se lo buscará en disco (sin modificar para nada la estructura en memoria). Si se estuviera intentando agregar un digrama y no hubiese lugar en esa porción del array se baja a disco el contexto/digrama más antiguo de los contenidos en esa porción del array (que puede o no ser el más antiguo de todo el array) y se deja el digrama en la posición liberada. Notar que las operaciones de búsqueda/agregado son independientes; cuando se busca algo, sea encontrado en memoria o en disco, por como funciona lzp terminará siendo agregado a la tabla con la posición actualizada, pero esta será una operación independiente de la búsqueda. En el caso de necesitar liberar una posición en la hashtable por agregarse un digrama que no se encuentra en memoria en ese momento, solo será bajado a disco UN solo LzpContextPosition.

Para el guardado en disco se utiliza un Arbol B#. Se utilizó el arbol B# genérico que fue documentado en la segunda entrega. Se crearon para ello los serializadores correspondientes a la clave (el digrama) y el elemento (LzpContextPosition).

La estructura simil hashtable elegida es mucho más rápida que si se hubiese usado una tabla secuencial. Como contrapartida la estructura secuencial mantendría en memoria siempre los digramas más recientes. Notar que, en el caso de hashtable, el extremo de usar un mod 506 (22*23 = 506) sería lo mismo que usar una tabla secuencial. En el otro extremo, usar un mod 1 se tendría acceso inmediato al LzpContextPosition buscado pero no se utilizaría en absoluto el criterio de mantener en memoria los registros más recientes. El mod elegido (22) es el que promedia ambas situaciones.

5.3. Serializer

Se creó además un *LzpSerializer* que utiliza para la deshidratación un *LzpCompressor* y para la hidratación un *LzpDeCompressor*. Ver diagrama LzpMainPicture.

6. Otras observaciones y variantes

Finalmente tenemos 2 serializadores distintos para comprimir los documentos (LZP y PPMC), y uno extra para guardarlo sin compresión (el que ya existía en la 2da entrega), sin embargo, los documentos se almacenan en un único archivo que posee un único serializador. Por lo cual, el serializador de *DocumentLibrary* es el encargado de lidiar con esos tres serializadores dependiendo de que modo se seleccione para la compresión (o no) del documento a almacenar. Para poder realizar luego la recuperación de los mismos, debe poderse identificar el modo en que se almacenó, para lo cual el serializador agrega información de control que es lo primero que lee antes de hidratar el documento (1 byte indicando el 'formato' del documento almacenado).

Algo para destacar, común a todos los compresores, es que por consigna/requisito del TP debe utilizarse un alfabeto de 16 bits (y en lzp longitudes de 16 bits). Pero los textos que utiliza como entrada el programa utilizan una codificación de 8 bits (ISO8859-1). Y la codificación más utilizada (UTF-8) también lo hace. Esto provoca que en textos cortos o medianos la compresión genere una salida de mayor tamaño que el texto original. Nos planteamos tomar desde disco los caracteres de a dos, manejándolos como "raw", generando un caracter de fantasia de 16 bits, pero haciendo esto hubiesemos perdido la ventaja de usar PPMC y LZP que son compresores pensados para trabajar con texto de lenguaje natural.

Los documentos, por facilidad, se trataron en memoria, pero también estuvo el planteo de volcarlos a archivos de trabajo y hacer lecturas de dicha información cuando fuera requerida. Si bien tiene la ventaja de reducir la memoria consumida incrementan las lecturas a disco que se verían muy perjudicados los algoritmos que acceden a muchas posiciones distintas como es el caso del LZP que requiere la relectura de información pasada podría. Para solucionar esto habría que implementar algún caché de lectura para evitar ralentizar el proceso.