Capítulo 5

Implementación

En este capítulo se presenta la implementación del sistema, un sistema que permite, primero que nada, analizar córpora de dominios formulaicos y construir listas de fórmulas que los caracterizan (entrenamiento), después nos permitirá clasificar textos basándose en las fórmulas obtenidas en el entrenamiento, para posteriormente ser capaz de evaluar el funcionamiento (eficacia) del algoritmo de clasificación. La estructura del capítulo será la siguiente: en la sección 5.1 se describirán las consideraciones para la implementación, en la sección 5.2 la implementación del modelo de datos, en la sección 5.3 el sistema por módulos.

Estos módulos son: de **limpieza**, de **reconocimiento de patrones**, de **clasificación**, de **análisis**, de **interfaces**, y por último en la sección 5.4 algunas características de la implementación y conclusiones.

5.1. Consideraciones

La implementación de este sistema está hecha en lenguaje Java, se eligió este lenguaje porque es orientado a objetos y es mucho más sencillo el manejo de cadenas. El sistema está diseñado por módulos para que se puedan realizar los experimentos con

mayor comodidad y poder agregar o quitar módulos para posteriormente comparar los resultados obtenidos con los distintos experimentos, aunque existe cierta dependencia entre los módulos, todos pueden correr de manera independiente.

5.1.1. Algoritmo de Porter

Como ya se había mencionado, este algoritmo se tomó de una paquetería del ICT, un grupo de investigación de la UDLA y, aunque no es una implementación perfecta, funciona bien en general. El código no es público así que no se pudieron hacer mejoras; aunque se hicieron algunas validaciones externas en algunos casos en donde no funcionaba bien. Detalles de los problemas registrados con esta implementación se encuentran en el capítulo de trabajo a futuro.

Se decidió que los córpora que serán utilizados para hacer pruebas estarán en español, sólo se implementará la versión es español de éste algoritmo, pero se podría agregar con mucha facilidad otras versiones de éste para otros idiomas.

5.2. Modelo de datos

En algunos de los módulos existen funcionalidades que requieren hacer uso de las bases de conocimiento definidas en el capítulo anterior; éstas son: la base de abreviaturas y la de palabras vacías. Pero éstas necesitan ser creadas antes de realizar cualquier implementación. Es por esto que se definieron algunas clases que nos permiten administrarlas. Tenemos la clase *Administrador de abreviaturas*, y la clase *Administrador de palabras vacías*, con estas clases se pueden generar los repositorios. En nuestra implementación se eligieron como estructura de datos para representar los repositorios las tablas *Hash* porque están implementadas en Java en el paquete java.util; además son muy útiles para realizar búsquedas dentro de ellas porque son muy eficientes. El

proceso para crear las bases de conocimiento es un proceso independiente de las funcionalidades del sistema; se creó la clase **Llenar tablas**, con ésta se crearán las bases de conocimiento. En esta clase se crean instancias de abreviaturas y de palabras vacías, para posteriormente guardarlas en la tabla *Hash* correspondiente.

5.3. Módulos del sistema

5.3.1. Módulo de limpieza

Debido a que en los tres módulos lógicos presentados en el capítulo anterior (entrenamiento, evaluación y pruebas), utilizan los mismos algoritmos de preprocesamiento, todos ellos se agruparon en el mismo módulo, al cual se le llamó módulo de limpieza. Una vez que es invocado, este módulo inicializa la clase *Limpieza*, ésta es la clase principal de este módulo, recibe como parámetro de entrada una lista de oraciones para procesar. Contiene un método para cada algoritmo; dependiendo de las opciones elegidas por el usuario se llamará a estos procedimientos. Estos métodos son el algoritmo de Porter, sustitución de abreviaturas y eliminación de palabras vacías. Arroja como resultado los oraciones libres de palabras vacías, con palabras reducidas a raíces léxicas o sin abreviaturas (Fig. 5.1).

5.3.2. Módulo de reconocimiento de patrones

Una vez que es invocado este módulo, se inicializa la clase Parser; ésta recibe como parámetro de entrada la salida del módulo de limpieza y es la que le da el formato deseado a dicho parámetro, es decir, separa el córpora en varios dominios: un corpus por dominio. Para cada corpus obtiene las relaciones entre las oraciones que componen a los textos. Una vez que se tienen las relaciones para cada corpus, se inicializa la clase

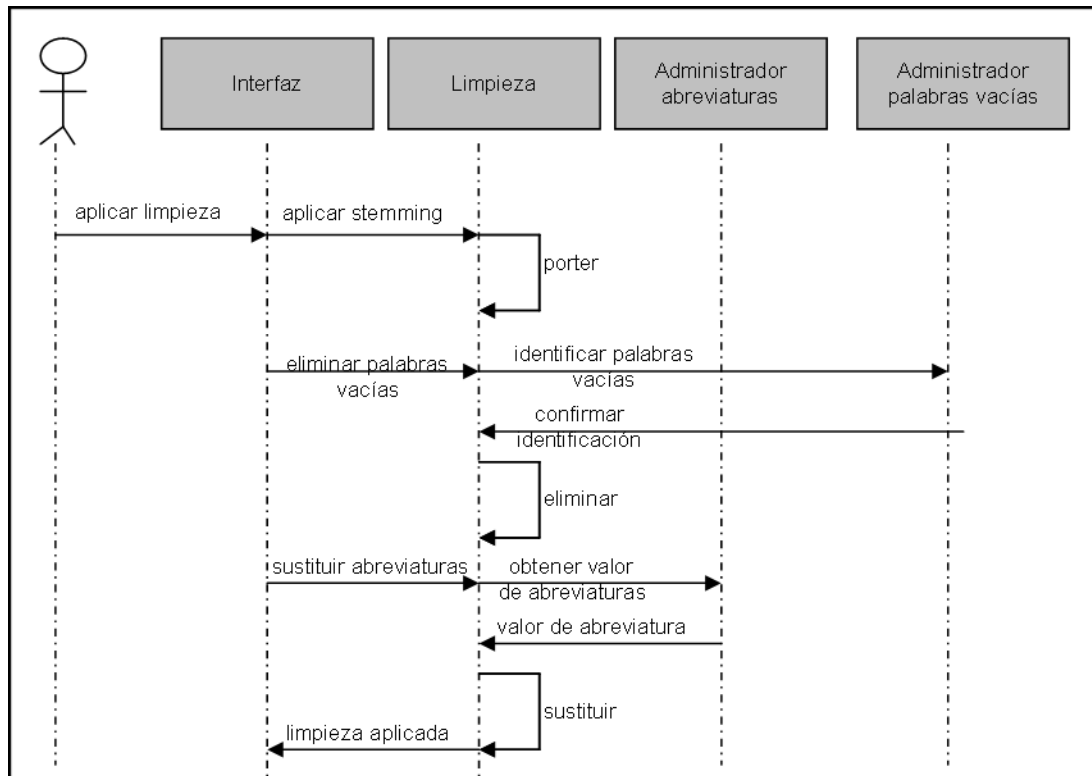

Figura 5.1: Módulo de limpieza

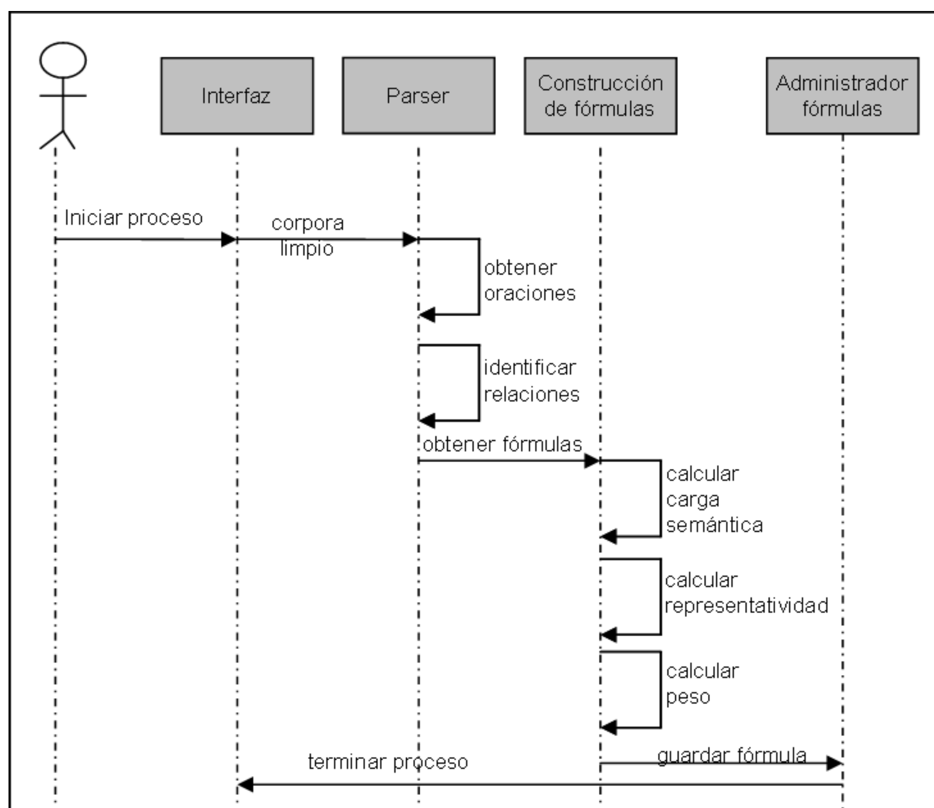

Figura 5.2: Módulo de reconocimiento de patrones

Construcción de fórmulas, y con cada una de ellas se crea una fórmula y se calculan los valores de sus atributos. Para guardar estas fórmulas se inicializa una instancia de la clase *Administrador de fórmulas* para cada corpus, ésta nos ayudará a guardarlas en el archivo correspondiente (un archivo independiente para cada corpus)(Fig. 5.2).

5.3.3. Módulo de clasificación

Una vez que es invocado este módulo, se inicializa la clase *Clasificación*, ésta es la clase principal de este módulo y recibe como parámetro de entrada la salida del módulo de limpieza. En esta clase se invoca a la clase Parser que es la que le da el formato deseado al parámetro de entrada, es decir, obtiene las relaciones de las oraciones de los textos. Una vez que se tienen las relaciones, se obtiene la lista de

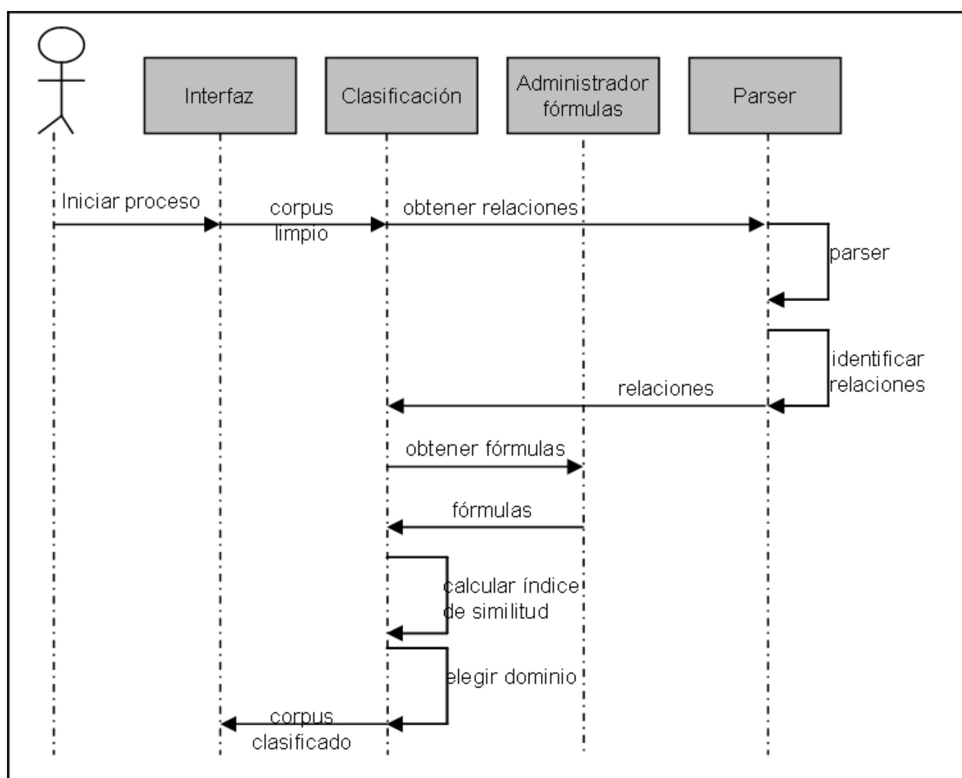

Figura 5.3: Módulo de clasificación

fórmulas para cada dominio; para cada una de estas listas se crea una instancia de la clase *Administrador de fórmulas*, esta instancia tendrá la tarea de administrarla. Posteriormente se comparará la lista de relaciones con cada lista de fórmulas y se calculará el índice de similitud entre ellas. En este momento tendremos asociado a cada texto con todas las listas de fórmulas a través de estos índices de similitud; entonces llegó el momento de elegir el dominio al que pertenece cada uno de los textos, en este procedimiento se elige la lista de fórmulas que mejor clasifique a cada texto o, en su defecto, si no hay una lista de fórmulas que lo clasifique (Fig. 5.3).

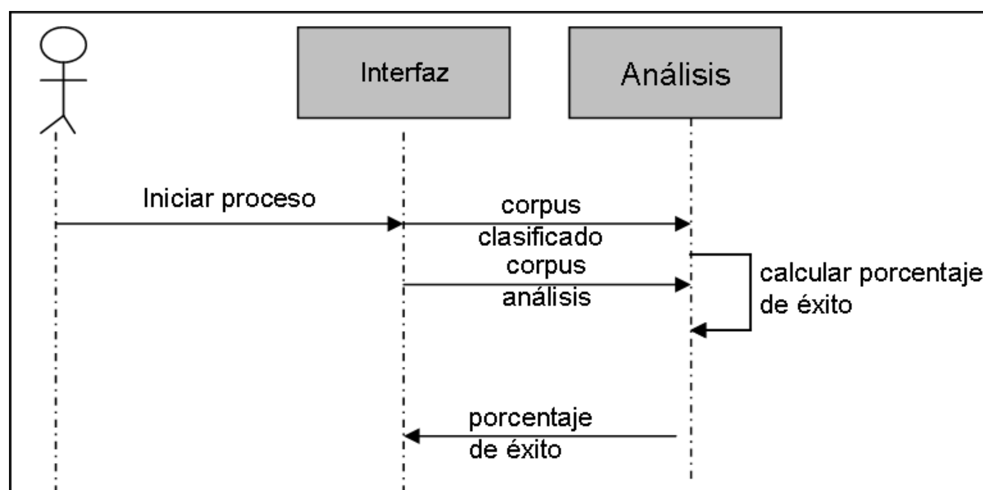

Figura 5.4: Módulo de análisis

5.3.4. Módulo de análisis

Una vez que es invocado este módulo, se inicializa la clase *Análisis*; ésta es la clase principal de este módulo. Esta clase recibe como parámetros de entrada la salida del módulo de clasificación y el archivo del corpus de análisis (brindado por el usuario); con estos dos parámetros se llama al procedimiento que los compara y calcula el porcentaje de éxito, esto es el porcentaje de los textos que fueron clasificados correctamente (Fig. 5.4).

5.3.5. Módulo de interfaces

Permite administrar y controlar la interacción del usuario con todos los módulos del sistema. Se diseñaron algunas interfaces que orientarán al usuario acerca de cuáles son las funcionalidades que el sistema le ofrece, además permitirán que éste ingrese los valores requeridos para que los módulos funcionen (Fig. 5.5).

Este módulo es la parte central del sistema. Cuando el sistema inicia, el modulo es iniciado con la clase *Interfaz Principal*; ésta es la clase principal que despliega el

menú principal. Esta clase maneja los eventos relacionados con las elecciones del usuario. En el menú desplegado el usuario tiene tres opciones, sólo se puede elegir una vez; estas opciones son:

- iniciar el módulo de entrenamiento,
- iniciar el módulo de evaluación o
- iniciar el módulo de pruebas.

Además tiene un botón con la leyenda “Aceptar” con la que usuario indicará al sistema que está listo para proseguir. Una vez que el usuario oprima el botón “Aceptar”, el sistema iniciará la interfaz del módulo que el usuario haya elegido.

Si se trata del módulo de entrenamiento, se inicializa la clase *Interfaz Entrenamiento*. Esta clase despliega una pantalla que consta de tres partes:

1. un menú de opciones para elegir los algoritmos de preprocesamiento que serán aplicados a córpora de entrada;
2. un navegador de archivos para que el usuario brinde el archivo donde se encuentra descrito el córpora;
3. un botón con la leyenda “Aceptar” para indicar al sistema que se está listo para proseguir, y un botón con la leyenda “Cancelar” que cierra esta ventana y despliega de nuevo el menú principal.

Cuando el usuario oprime el botón “Aceptar”, la clase *Interfaz Entrenamiento* inicializa la clase *Parser* que separa los córpora y obtiene los textos asociados a cada dominio. Al finalizar el parser, se inicia el módulo de limpieza, y dependiendo de las opciones de

preprocesamiento que el usuario haya elegido, utilizarán los procedimientos correspondientes. Posteriormente, se iniciará el módulo de reconocimiento de patrones, al cuyo término se da por concluido este proceso.

Por otro lado, si se trata del módulo de evaluación, se inicializa la clase *Interfaz Evaluación*. Esta clase despliega una pantalla que consta de tres partes:

1. un menú de opciones para elegir los algoritmos de preprocesamiento que serán aplicados al corpus de entrada;

2. dos navegadores de archivos para que el usuario brinde el archivo donde se encuentra el corpus para la etapa de clasificación y el archivo donde se encuentra el corpus para la etapa de análisis;

3. un botón con la leyenda “Aceptar” para indicar al sistema que se está listo para proseguir y un botón con la leyenda “Cancelar” que cierra está ventana y despliega de nuevo el menú principal.

Cuando el usuario oprime el botón “Aceptar”, la clase *Interfaz Evaluación* inicializa la clase *Parser* y obtiene los textos que conforman el corpus para clasificar; al terminar el parser, se inicializa el módulo de limpieza, y dependiendo de las opciones de preprocesamiento que el usuario haya elegido, se utilizarán los procedimientos correspondientes. Posteriormente, se iniciará el módulo de clasificación, tomando como parámetro de entrada el archivo del corpus para clasificar; con el resultado de éste se inicializa el módulo de análisis tomando como parámetros de entrada el resultado del módulo de clasificación y el archivo del corpus de análisis. Una vez que se haya terminado el módulo de análisis, se inicializa la clase *Mostrar Resultado*, ésta tiene el objetivo de crear una ventana, en la cual se desplegará el resultado de éste módulo. Esta ventana tendrá un botón con la leyenda “Aceptar” con el que ésta se cerrará y se regresará al menú principal.

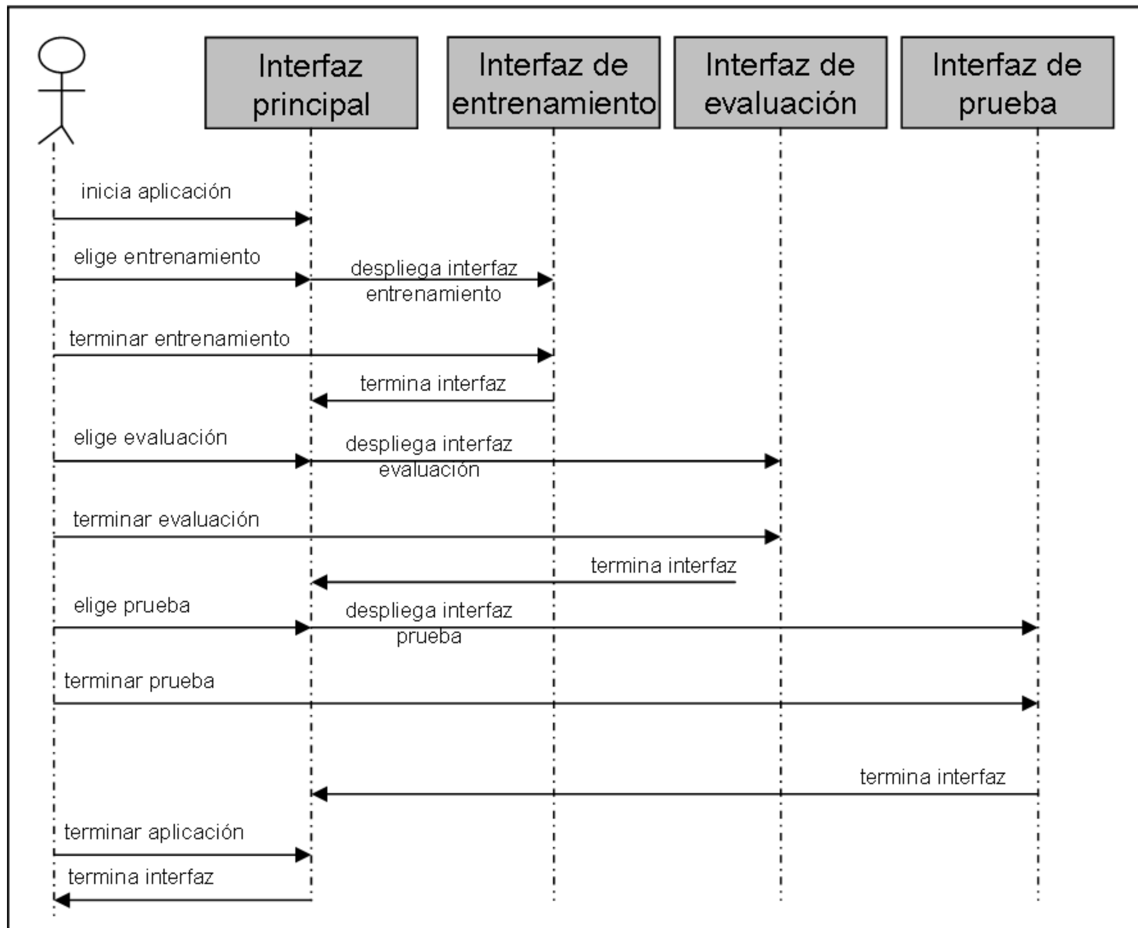

Figura 5.5: Módulo de interfaces

Por último, si se trata del módulo de prueba, se inicializa la clase *Interfaz Prueba*. Esta clase despliega una pantalla que consta de los mismos elementos que el menú del módulo de evaluación. Cuando el usuario oprime el botón "Aceptar", la clase *Interfaz Prueba* lleva a cabo el mismo procedimiento que la clase *Interfaz Evaluación*.

5.4. Funciones principales del sistema

A continuación se describen las funciones más importantes del sistema: en algunos casos se incluyó código, pero en los casos en que el código era demasiado grande, se

```
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio1.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio2.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio3.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio4.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio5.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\dominio6.txt
```

Figura 5.6: Córpora

```
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta1.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta2.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta3.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta4.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta5.txt
C:\Documents and Settings\@le\Mis documentos\Udla\Tesis\corpus\Carta6.txt
```

Figura 5.7: Corpus

limitó a describir el funcionamiento o se incluyó pseudo código. De todos modos en el Apéndice C se incluyeron los diagramas de clases, que sirven para definirlas, y en el Apéndice D se incluyó el código fuente de todas las clases.

5.4.1. Parser

En varios de los módulos del sistema se requiere utilizar un parser; cabe señalar que no siempre se requiere utilizar todas las funcionalidades del parser. Para el corpus de entrenamiento se implementó una funcionalidad particular, debido a la estructura del archivo que describe el corpus (Fig. 5.6). Este archivo tiene una lista de córpora, es decir, la localización de los archivos que describen a cada dominio.

Cada corpus consiste de una lista de textos descritos por la localización de éstos (Fig. 5.7).

Con esta lista se recupera cada uno de los textos y se le separa en oraciones. Estas

```
import java.util.*;

public class Oracion{
   Vector oracion = null;

/*oracion es un vector de objetos String
 cada String representa el valor de una palabra*/
   public Oracion(Vector v){
        oracion = v;
   }

   public void imprimir(){
        if(oracion!=null)
        {
            for(int i=0; i < oracion.size(); i++)
               System.out.print(oracion.elementAt(i) + " ");
            }
        }
}
```

Figura 5.8: Código 1

listas de oraciones se representan como un vector de objetos *Oración*.

Para el corpus de clasificación, el parser no requiere utilizar la funcionalidad extra utilizada en el corpus de entrenamiento. El corpus tiene la misma estructura que el del archivo descrito en la figura 5.7, por lo tanto se omite la primera parte del parser.

5.4.2. Sustituir abreviaturas

Si es decisión del usuario que se sustituyan las abreviaturas, estas deberán ser identificadas y sustituidas por sus valores. Entonces, cada vez que se identifique una cadena como abreviatura, ésta será sustituida dentro de la *Oración* por su valor; esto se puede ver en el Código 1 (Fig. 5.8). Para identificar si se trata o no de una abreviatura se implementó el método *abreviaturas* (Fig. 5.9), este busca en la tabla de abreviaturas "abre.hash" si se trata de una de ellas y en este caso devuelve su valor. La llamada a

```
public String abreviaturas(String abr){
   AdminAbre ab = new AdminAbre();
   try{
       FileInputStream fis = new FileInputStream("abre.hash");
       ab.load(fis);
       Abreviatura temp = ab.getNode(abr);
       if(temp!=null)
          return temp.valor;
       else
          return null;
   }
   catch(Exception e){
       e.printStackTrace();
       return null;
   }
}
```

Figura 5.9: Código 2

este método se hace vía parser.

Para abrir la tabla *Hash* se hace uso de la clase **AdminAbre** que es la clase que nos ayuda a administrar el uso de esta tabla. Esta tabla contiene objetos **Abreviatura** (Fig. 5.10) que fueron diseñados para guardar la relación entre la abreviatura y la palabra que representa.

Finalmente, una vez que se identificó que se trata de una abreviatura, se sustituye por el valor que representa, esto se implementó dentro del parser de un texto a oraciones. Esta incluido en la parte de código (Fig. 5.11).

5.4.3. Eliminar palabras vacías

Si es decisión del usuario que se eliminen las palabras vacías, entonces estas deben ser identificadas y eliminadas de la *Oración*. Con el fin de identificar si una cadena se trata o no de una palabra vacía se implementó el método *elim_palabrasvacias* (Fig. 5.12), este busca en la tabla de palabras vacías "vacias.hash" si se trata de una de ellas.

```
import java.util.*;
import java.io.Serializable;

public class Abreviatura implements Serializable{
  String abre;
  String valor;

  Abreviatura(String ab, String va){
     valor = va;
     abre = ab;
  }

  public String toString(){
    return ("La abreviatura = " + abre + " significa "+ valor);
  }
}
```

Figura 5.10: Código 3

```
String valor_abre = abreviaturas(abre);
if(valor_abre!=null){
  if(sustituir){
    oracion.setElementAt(valor_abre, oracion.size()-1);
  }
  else{
    oracion.setElementAt(abre + ".", oracion.size()-1);
  }
}
```

Figura 5.11: Código 4

```
public Vector elim_palabrasvacias(Vector oraciones){
   AdminPalVac pv = new AdminPalVac();
   int i, j;
   try{
     FileInputStream fis = new FileInputStream("vacias.hash");
     pv.load(fis);
     for(i=0; i<oraciones.size(); i++){
     Vector v = (Vector)((Oracion)(oraciones.get(i))).oracion;
     for(j=0; j<v.size(); j++){
        String candidata = (String)v.get(j);
        Palabra_Vacia temp = pv.getNode(candidata);
        if(temp!=null)
           v.remove(j);
        }
        Oracion o = new Oracion(v);
        oraciones.removeElementAt(i);
        oraciones.add(i, o);
     }
   }
   catch(Exception e){
      e.printStackTrace();
   }
   return oraciones;
}
```

Figura 5.12: Código 5

La llamada a este método se hace vía parser.

Para abrir la tabla *Hash* se hace uso de la clase *AdminPalVac*, ésta es la clase que nos ayuda a administrar el uso de esta tabla. Esta tabla contiene objetos *Palabra_Vacia* (Fig. 5.13).

5.4.4. Obtener relaciones

Las relaciones obtenidas se almacenan en una tabla hash, una para cada corpus o dominio. Esta tabla hash contiene objetos **Palabra**. La estructura que tienen estos objetos se muestran en el fragmento de código 7 (Fig. 5.14).

Las relaciones se obtienen a través del método *principal*, éste tiene como parámetros de entrada un *Vector* de oraciones. Para cada oración se hace un análisis de cómo se

```
import java.util.*;
import java.io.Serializable;

public class Palabra_Vacia implements Serializable{
   String valor;
   Palabra_Vacia(String va){
     valor = va;
   }
   public String toString(){
      return ("La palabra vacía es = " + valor);
   }
}
```

Figura 5.13: Código 6

```
import java.util.*;
import java.io.Serializable;
public class Palabra implements Serializable{
   public String valor;
   public Hashtable v;
   int r;
   int re;
   Palabra(){
      v= new Hashtable();
   }
   Palabra(String va){
      valor = va;
   }
   public void contar(){
      r++;
   }
   public void cont_R(){
      re++;
   }
   public String toString(){
    return("La palabra "+ valor + " tiene "+ r +"
       repeticion y tiene "+ re + "de representatividad ");
   }
}
```

Figura 5.14: Código 7

```
import java.util.*;
import java.io.Serializable;
public class Formula implements Serializable{
   Vector palabras;
   String[] pal = new String[2];
   int repeticiones;
   float representatividad;
   float carga;
   float peso;
   int relacion;
   String llave;
   public Formula(String[] p, int rel){
    relacion = rel;
    pal = p;
    llave = p[0]+rel+p[1];
   }
   public Formula(Vector p){
    palabras = p;
   }
   public String toString(){
     return ("Palabra = " + pal[0] + "palabra 2 "+pal[1]+
" con relacion "+ relacion + " con repeticiones "+ repeticiones)
   }
   public void contar(){
     repeticiones++;
   }
   public void cont_rep(){    representatividad++;            }}
```

Figura 5.15: Código 8

relacionan las palabras, estas relaciones se guardan en la tabla hash. No hay más de una repetición de la misma relación.

5.4.5. Construir fórmulas

Para la construcción de fórmulas se implementó la clase *Proceso*, que no sólo incluye la creación de la tabla hash donde se guardan las fórmulas, también incluye los métodos necesarios para calcular los valores de los atributos de *Fórmula*. La estructura que tienen estos objetos se muestran en el fragmento de código 8 (Fig. 5.15).

5.4.6. Calcular índice de similitud

Para calcular el índice de similitud, se implementó dentro de la clase *Clasificación* el método *evaluación*; dentro de éste está implementada la ecuación que fue definida en el capítulo de metodología y que describe los pasos de este proceso como sigue:

- representar texto con un conjunto de relaciones;
- buscar en la lista de fórmulas que describe un dominio, cada una de las relaciones;
- se cuenta con una variable dónde se guarda el valor acumulado de sumar los pesos de aquellas fórmulas que fueron encontradas en la lista de relaciones;
- también se cuenta con una variable dónde se guarda el valor acumulado de sumar la representatividad de aquellas fórmulas que fueron encontradas en la lista de relaciones;
- una vez que se terminó de recorrer la lista de relaciones, el índice de similitud se obtiene de la división de la suma de los pesos entre la suma de las representatividades y se multiplica por cien;
- este proceso se repite para cada dominio.

5.4.7. Elegir dominio

Se tiene asociado a cada texto un arreglo de *Indices_similitud* entre éste y cada lista de fórmulas, para elegir el dominio se ordena este arreglo de índices de manera descendente. El índice que encabeza la lista es el candidato más fuerte, sólo queda asegurarse que su valor sea lo suficientemente grande (se tomó como valor arriba del cincuenta por ciento).

```
public float comparacion(String archivo_coprima, Vector res){
  float porcentaje_exito = 0;
  try{
     File file = new File(archivo_coprima);
     FileInputStream f = new FileInputStream(file);
     DataInputStream in = new DataInputStream(f);
     String arch = in.readLine();
     int i = 0, cont = 0;
     while((arch!=null)&&(arch.length()>3))
     {
        if(arch.equals(""+((Indice_similitud)res.elementAt(i)).indice))
            cont++;
         i++;
      }
      porcentaje_exito=cont/res.size();
   }
   catch(Exception e){
      e.printStackTrace();
   }
   return porcentaje_exito;
}
```

Figura 5.16: Código 9

5.4.8. Calcular porcentaje de éxito

Para calcular este porcentaje se implementó el método *comparación*, recibe de entrada la ruta del corpus que está clasificado correctamente, además el vector resultante de la clase **Clasificación** que contiene objetos **Indice_similitud** que contienen el índice del dominio que mejor describe a cada texto. La estructura del archivo de entrada se puede apreciar en la figura 5.16.

5.4.9. Iniciar el sistema

Una vez que se tiene instalado y configurado el sistema (las instrucciones están incluidas en el Apéndice B), está listo para usarse. La clase que contiene la interfaz principal es la clase **Interfaz_Principal**. En la Figura 5.14 se puede ver la pantalla que despliega la clase principal.

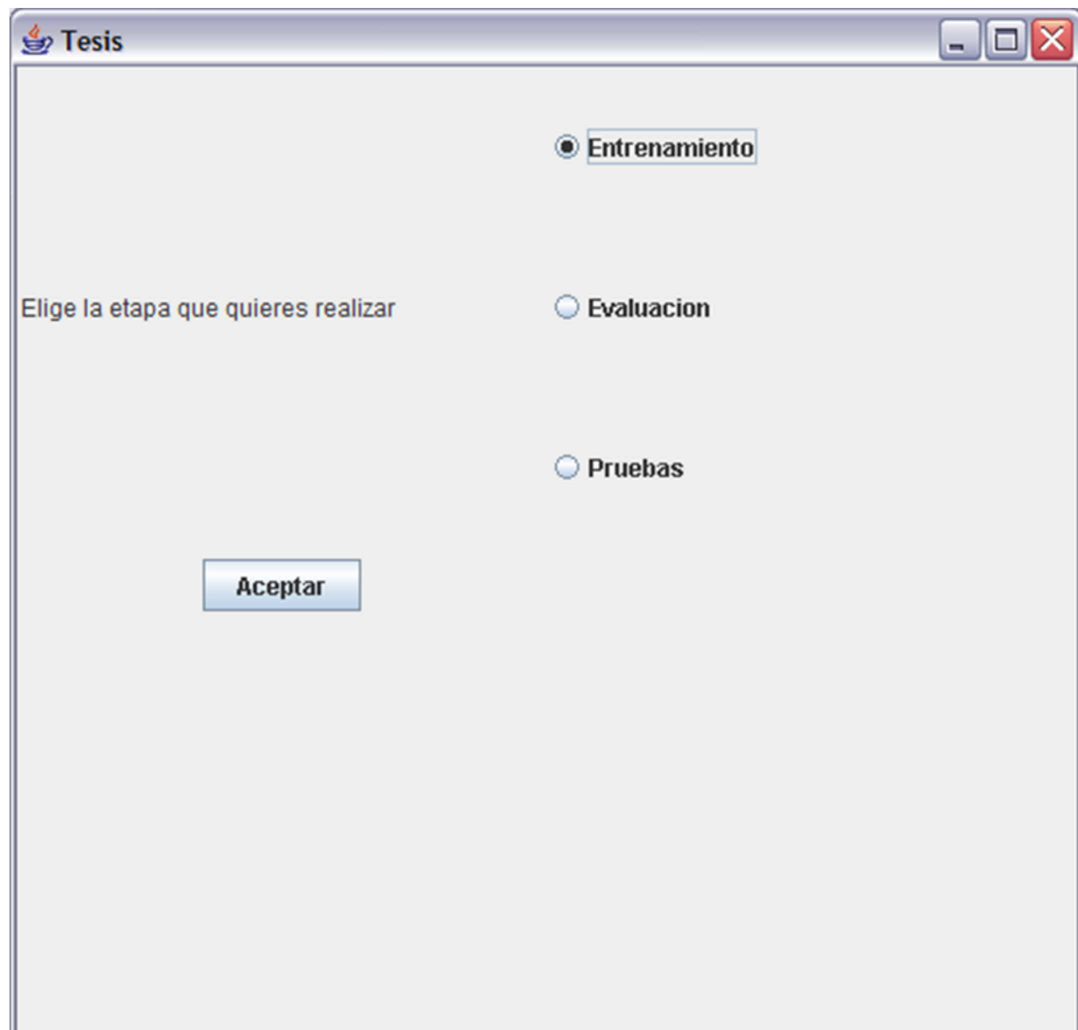

Figura 5.17: Interfaz principal

5.5. Características de la implementación

5.5.1. Cohesión

Existe gran cohesión en la implementación, esto se puede decir porque los métodos de las clases contribuyen a la ejecución de una sola tarea. Lo mismo pasa con los módulos: las clases de un módulo trabajan en común para llevar a cabo las tareas de cada fase.

5.5.2. Acoplamiento

Existe poco acoplamiento entre los módulos, esto se puede ver porque los métodos sólo se comunican a través de parámetros. Además, en la mayoría de las clases no se usan variables globales, aunque varias clases pueden afectar los datos y esto aumenta el acoplamiento; en general, podemos resumir que hay poco acoplamiento.

5.5.3. Modularidad

Existe una dependencia secuencial entre los módulos del sistema, es decir, en todos los casos la entrada de cada módulo es la salida del módulo anterior.

5.5.4. Extensibilidad

Debido a su carácter modular realizar cualquier extensión es relativamente fácil, por ejemplo, si se requiere tratar textos escritos en inglés o mejorar los módulos que están implementados, sólo se tienen que agregar los módulos nuevos. Por ejemplo, se puede agregar el algoritmo de Porter en inglés o modificar la lista de palabras vacías agregándole palabras vacías en inglés para que esté más completa, etc. Todos estos cambios son posibles sin necesidad de modificar ninguno de los otros módulos, en realidad

lo único que tendría que cambiarse es la clase principal del módulo al que pertenezca para que haga la llamada al método correspondiente.