ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

METODOLOGÍA RECURSIVA PARA CREACIÓN DE ÁRBOLES DE DECISIÓN EN JAVA

AUTOR: Sonia Sánchez Morga
DIRECTOR: M" Francisca Rosique Contreras
Noviembre / 2013

TABLA DE CONTENIDO

1. FICHA DE PROPUESTA DE PFC

Titulación	Ingeniería Técnica de Telecomunicaciones, esp. Telemática ERASMUS (SI/NO): SI
Título del Proyecto: **METODOLOGÍA RECURSIVA PARA CREACIÓN DE ÁRBOLES DE DECISIÓN EN JAVA**	
Traducción del Título del Proyecto a lengua inglesa: **Recursive method for creation of decision trees in Java**	

Propuesta de Proyecto Final de Carrera/ Trabajo Fin de Grado

1. Planteamiento inicial del proyecto

El proyecto surge a partir de la complejidad adherida a la representación y ordenación de datos, especialmente en aquellos en los que aparece un factor de decisión que nos hace movernos de un dato a otro sin seguir una estructura básica.

Con la técnica de los árboles de decisión es posible analizar decisiones secuenciales basadas en el uso de resultados y probabilidades asociadas. Los árboles de decisión presentan las siguientes ventajas respecto a otros métodos de ordenación y representación:

- Presenta de manera resumida los ejemplos de partida, permitiendo la clasificación de nuevos casos siempre y cuando no existan modificaciones sustanciales en las condiciones bajo las cuales se generan los nodos base que sirvieron para su construcción. En caso de tener modificaciones sustanciales se procedería a la creación de un nuevo ramal que contemple estas modificaciones.
- Facilita la interpretación de la decisión adoptada, explicando el comportamiento respecto a una determinada tarea de decisión.
- Reduce el número de variables independientes.
- Es una gran herramienta en el campo de las telecomunicaciones, para el control de la gestión empresarial, juegos, etc.

Es una manera de simplificar un problema complejo de representar y comprender.

2. Objetivos del proyecto

El proyecto simplifica el uso de árboles de datos irregulares. Para ello se ha usado la programación recursiva en java ampliándose de una manera muy significativa los conocimientos adquiridos a lo largo de la carrera, tanto en programación en lenguaje java como en el uso de métodos recursivos de mayor complejidad.

2. ALCANCE Y CONTENIDO

2.1 Introducción

Es complicado comparar varios cursos de acción, y finalmente seleccionar la tarea a realizar en aquellos casos en los que las tomas de decisiones se bifurcan según las decisiones que se tomen.

La manera de procesar la información es limitada. Para tomar muchas de las decisiones y/o comparar diferentes opciones es necesaria una visión global para así poder llegar a un punto determinado. Se crea por tanto una incertidumbre que no es posible resolver sin una vista general de las distintas decisiones y objetivos.

El estudio sistemático de árboles de decisión proporciona el marco para elegir diferentes cursos de acciones en situaciones complejas. La visión de las posibles acciones y los resultados finales nos permite la predicción del camino para asegurarnos el éxito de la obtención de nuestro fin deseado.

2.2 Estructura del documento

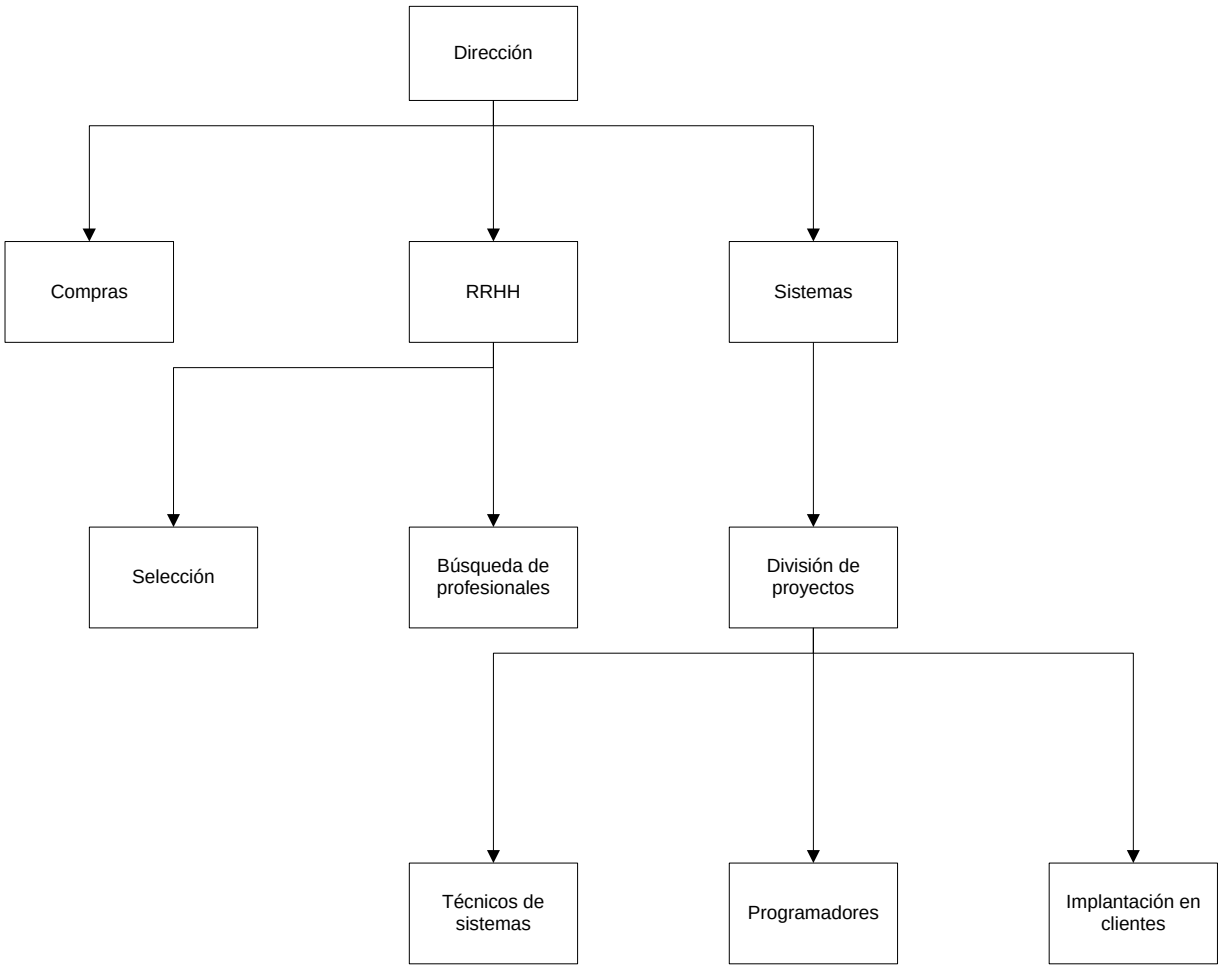
Este documento se encuentra dividido en diferentes secciones, siendo la información más relevante:

- Alcance y contenido: aporta una visión general del proyecto.
- Árboles de decisión: se da una visión global sobre los árboles de decisión, características, ventajas…
- Orden de complejidad de los algoritmos: se hace una pequeña revisión algorítmica de los árboles.
- Caso de estudio: Esta sección presenta uno de los posibles escenarios y muestra cómo interactuar con el programa. Se muestra también el diseño textual y gráfico
- Especificación: se define el diagrama UML así como una API con las clases y métodos utilizados
- Herramientas: Se muestran las herramientas utilizadas en la realización del proyecto (Eclipse y java).
- Conclusiones y líneas futuras: Describe los aspectos más importantes del proyecto.
- Anexos: Incluye código java con la explicación detallada.

3. ÁRBOLES DE DECISIÓN

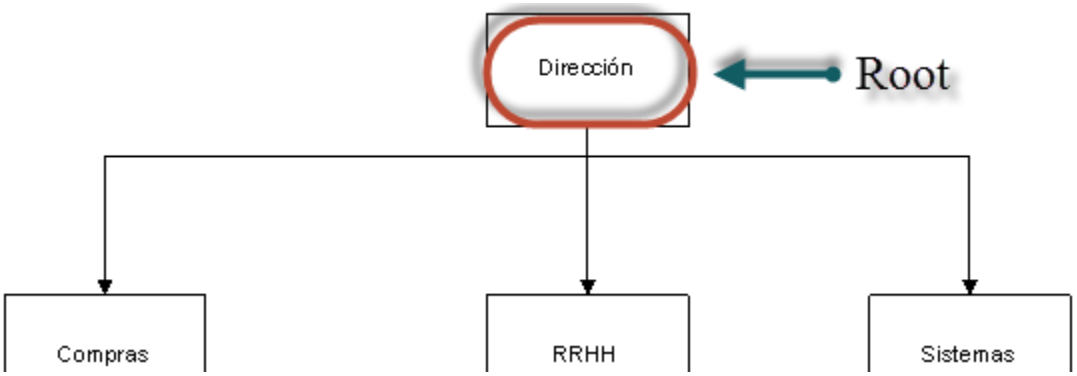
Se define cómo árbol de decisión a la técnica que permite analizar diferentes decisiones secuenciales basadas en el uso de resultados y probabilidades asociadas.

Un árbol se considera un número finito de nodos, donde uno de esos nodos es el nodo principal (root) desde el que el resto de nodos están organizados de manera jerárquica. Un nodo puede constar de un padre y uno o más hijos. Un nodo padre será considerado aquel que se sitúa jerárquicamente en un nivel superior. Un nodo hijo será considerado aquel que se sitúe por debajo jerárquicamente.



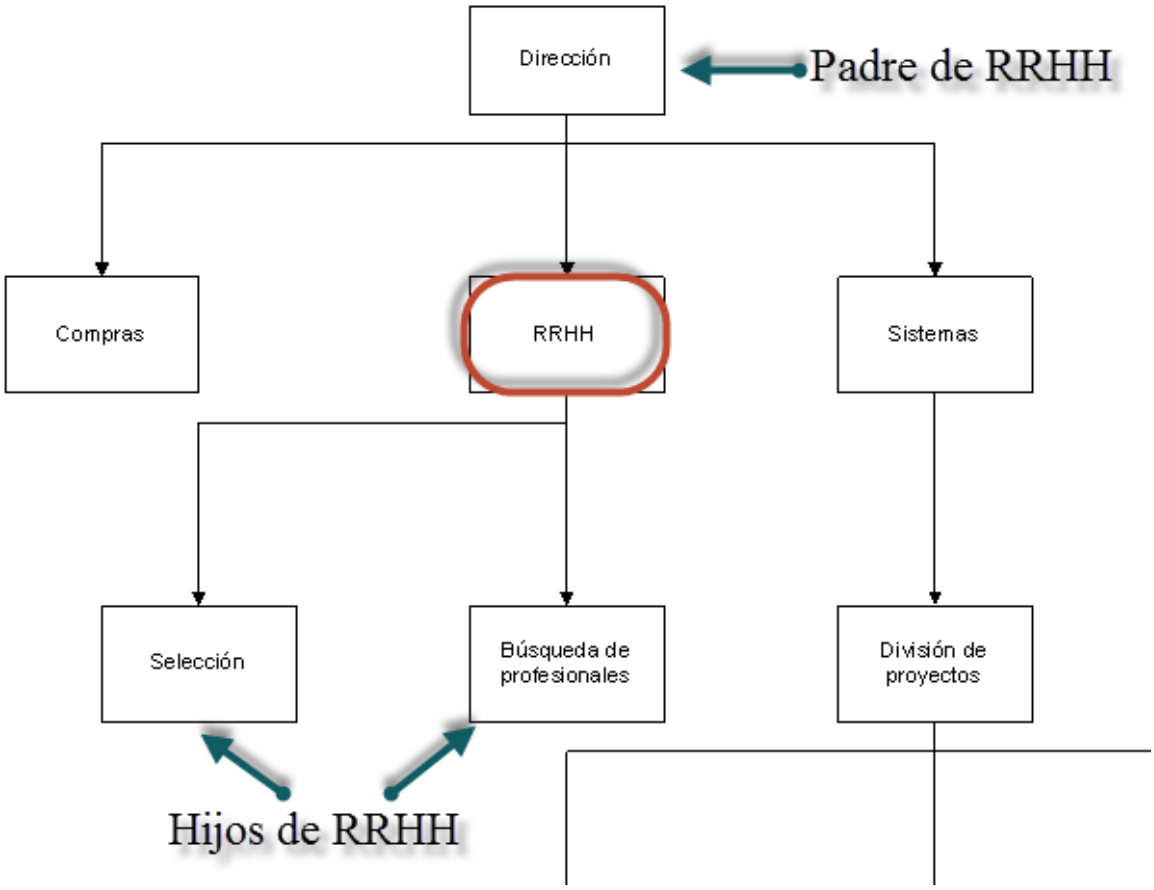
1. Ejemplo árbol de decisión

Basándonos en el ejemplo ilustrado anteriormente, identificaremos los nodos principales. El nodo root en el ejemplo se identificaría como el nodo “Dirección”.



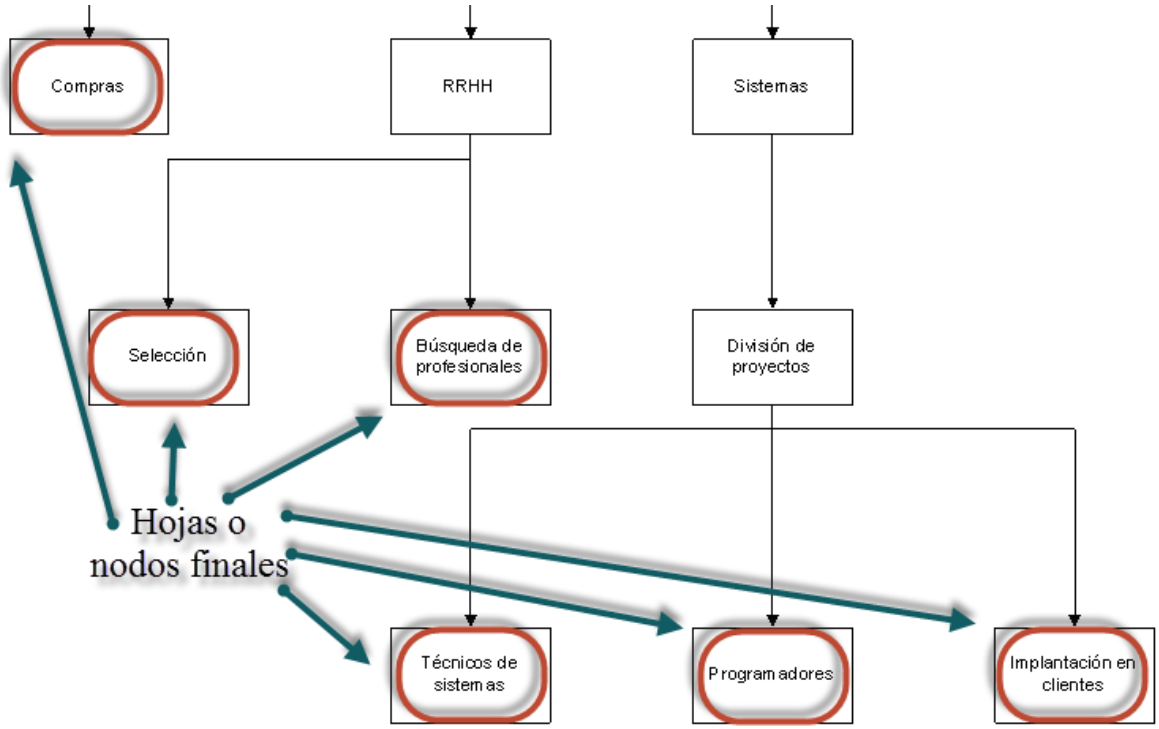
2. Root

Si escogemos como nodo ejemplo “RRHH”, podríamos decir que el padre de “RRHH” es “Dirección” mientras que sus hijos son “Selección” y “Búsqueda de profesionales”.



3. Ejemplos padre e hijos

Como nodos finales o hojas tendríamos a “Compras” de la primera rama, a “Selección” y “Búsqueda de profesionales” de la segunda rama y a “Técnicos de sistemas”, “Programadores” e “Implantación en clientes” de la tercera rama.



4. Hojas o nodos finales

3.1 Ventajas de los árboles de decisión

Esta técnica para el análisis de decisiones secuenciales tiene las siguientes ventajas:

- Resume los ejemplos de partida, permitiendo la clasificación de nuevos casos.
- Facilita la interpretación de la decisión adoptada.
- Proporciona un alto grado de comprensión del conocimiento utilizado en la toma de decisiones.
- Explica el comportamiento respecto a una determinada tarea de decisión.
- Reduce el número de variables independientes.
- Es una magnífica herramienta para el control de la gestión empresarial.
- Las condiciones y las acciones del árbol de decisión se encuentran en ciertas ramas pero no en otras, a diferencia de las tablas de decisión, donde todas forman parte de la misma tabla.
- Plantea el problema para que todas las opciones sean analizadas.

3.2 Desventajas

- Las reglas de asignación son bastante sencillas a pequeñas perturbaciones en los datos.
- Dificultad para elegir un árbol óptimo.
- Los árboles de decisión requieren un gran número de datos de los que muchas veces no disponemos.

3.3 Resumen

En resumen, los árboles de decisión son un método efectivo para la toma de decisiones por varias razones:

- En el problema que plantea, todas las posibilidades son analizadas.
- Permite analizar totalmente el resultado para cada una de las posibles opciones.
- En el esquema que plantea se puede ver claramente el coste de cada resultado así como la probabilidad de que se de esa opción.
- Nos ayuda a visualizar los mejores resultados sobre la base de información existente.

4. ORDEN DE COMPLEJIDAD DE LOS ALGORITMOS

4.1 Introducción

La necesidad de una mejora en la eficacia de los algoritmos ha sido la motivación de la universidad Fachhochschule St. Pölten, bajo la que se hace la propuesta de este proyecto final de carrera. El diseño de árboles presenta una gran compatibilidad con algoritmos eficaces.

El análisis de algoritmos nos permite medir la dificultad inherente de un problema y evaluar la eficiencia de un algoritmo.

4.2 Tiempos de ejecución

La manera de medir la complejidad de un algoritmo es usando variables como el tiempo de ejecución en función de N, lo que se denomina T(N). Esta función permite la medición real del tiempo de ejecución.

Este tipo de medición es calculada de diferentes maneras. La manera más simple se basaría en el control del tiempo que tarda en ejecutarse el código. Para ello simplemente ejecutaríamos el código controlando con un cronometro la duración de este.

Una manera más efectiva de medir T(N) se basaría en la medición de tiempo necesario para ejecutar una instrucción y multiplicando por el número de instrucciones que están definidas en el código.

De manera simple vemos un ejemplo. Calculamos el tiempo de ejecución en el siguiente código:

```
S1; FOR i:= 1 TO N DO S2 END;
```

Necesita:

```
T(N):= t1 + t2*N
```

Siendo *t*1 el tiempo que lleve ejecutar la serie "*S*1" de sentencias, y *t*2 el que lleve la serie "*S*2".

Cualquier programa que no sea extremadamente básico contiene sentencias condicionales. Este hecho provoca que el número de instrucciones sea desconocido o varíe dependiendo de los datos concretos que se presenten al programa. Por lo tanto no es correcto hablar un valor T(N) si no de un rango de valores, permitiéndonos el estudio en el mejor y/o peor caso.

Rango de valores: *T*min(*N*) <= *T*(*N*) <= *T*max(*N*)

El caso promedio (el más frecuente) se hallara entre Tmin(N) y Tmax(N). Cualquier fórmula *T*(*N*) incluye referencias al parámetro *N* y a una serie de constantes "*Ti*" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del ordenador que lo ejecuta.

Dado que es fácil cambiar de compilador y que la potencia de los ordenadores es mayor cada día, se intentan analizar los algoritmos con algún nivel de independencia de estos factores, es decir, se buscan estimaciones generales ampliamente válidas.

No se puede medir el tiempo en segundos porque no existe un ordenador estándar de referencia, en su lugar se mide el número de operaciones básicas o elementales.

Las operaciones básicas son las que realiza el ordenador en tiempo acotado por una constante, por ejemplo:

- Operaciones aritméticas básicas
- Asignaciones de tipos predefinidos
- Saltos (llamadas a funciones, procedimientos y retorno)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores y matrices)

Es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, por ejemplo, las que más influyen en el tiempo de ejecución.
Para medir el tiempo de ejecución de un algoritmo existen varios métodos.
Veamos algunos de ellos:

a) Benchmarking

La técnica de *Benchmark* considera una colección de entradas típicas representativas de una carga de trabajo para un programa.

b) Profiling

Consiste en asociar a cada instrucción de un programa un número que representa la fracción del tiempo total tomada para ejecutar esa instrucción particular. Una de las técnicas más conocidas es la *Regla 90-10*, que afirma que el 90% del tiempo de ejecución se invierte en el 10% del código.

c) Análisis

Consiste en agrupar las entradas de acuerdo a su tamaño, y estimar el tiempo de ejecución del programa en entradas de ese tamaño, *T(n)*. De este modo, el tiempo de ejecución puede ser definido como una función de la entrada. Se denotará *T(n)* como el tiempo de ejecución de un algoritmo para una entrada de tamaño *n*.

4.3 Concepto de Complejidad

La complejidad o costo de un algoritmo es una medida de la cantidad de recursos (tiempo, memoria) que el algoritmo necesita. La complejidad de un algoritmo se expresa en función del tamaño del problema. La función de complejidad tiene como variable independiente el tamaño del problema y sirve para medir la complejidad. Mide el tiempo/espacio relativo en función del tamaño del problema. El comportamiento de la función determina la eficiencia. No es única para un algoritmo: depende de los datos. Para un mismo tamaño del problema, las distintas presentaciones iniciales de los datos dan lugar a distintas funciones de complejidad. Es el caso de una ordenación, si los datos están todos inicialmente desordenados, parcialmente ordenados o en orden inverso.

4.4 Órdenes de Complejidad

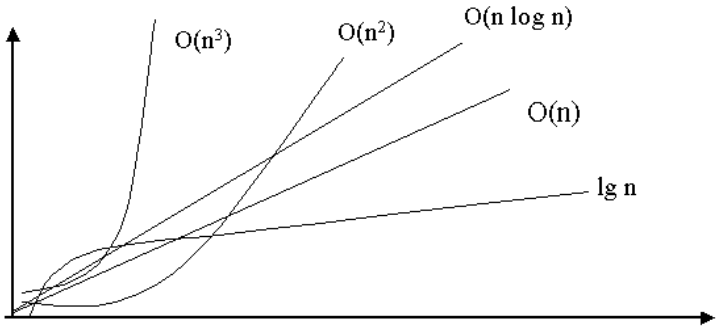
Se dice que *O* (*f*(*n*)) define un "orden de complejidad". Se escogerá como representante de este orden a la función *f*(*n*) más sencilla del mismo. Así se tendrá:

O (1)	orden constante
O (*log n*)	orden logarítmico
O(*n*)	orden lineal
O (*n*2)	orden cuadrático
O (*na*)	orden polinomial (*a* > 2)
O (*an*)	orden exponencial (*a* > 2)
O (*n*!)	orden factorial

Se puede identificar una *jerarquía de órdenes de complejidad* que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo *A* se puede demostrar de un cierto orden *O*1, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir, el menor orden de complejidad que lo cubra.

Antes de realizar un programa es conveniente elegir un buen algoritmo, donde por bueno se entiende que utilice pocos recursos, siendo usualmente los más importantes el tiempo que lleve ejecutarse y la cantidad de espacio en memoria que requiera.

En la siguiente figura se muestra un ejemplo de algunas de las funciones más comunes en análisis de algoritmos:



5. Análisis de algoritmos

La mejor técnica para diferenciar la eficiencia de los algoritmos es el estudio de los órdenes de complejidad. El orden de complejidad se expresa generalmente en términos de la cantidad de datos procesados por el programa, denominada *n*, que puede ser el tamaño dado o estimado.

En el análisis de algoritmos se considera usualmente el peor caso, si bien a veces conviene analizar igualmente el mejor caso y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales, tales como el lenguaje de programación, la habilidad del codificador, la máquina de soporte, etc. se suele trabajar con un cálculo asintótico que indica cómo se comporta el algoritmo para datos muy grandes y salvo algún coeficiente multiplicativo. Para problemas pequeños es cierto que casi todos los algoritmos son "más o menos iguales", primando otros aspectos como esfuerzo de codificación, legibilidad, etc. Los órdenes de complejidad sólo son importantes para grandes problemas.

4.5 Complejidad en árboles

Uno de los muchos motivos por los que se estudian los árboles reside en que, aunque las listas enlazadas ofrecen tiempos de búsqueda de *O(n)*, la mayoría de las implementaciones de árboles admiten tiempos de búsqueda de *O (log n)*.

Para entender cómo se calcula el orden de complejidad en un árbol, primero se va exponer un ejemplo con una lista simplemente enlazada. Si se quiere insertar un elemento en ella se puede hacer al principio que sería directo (a esto se le llama O (1)), al final, que sería el peor de los casos al tener que recorrer todos los elementos (a esto se le conoce como O(n)), o un caso intermedio, que sería O (log n). Este último es un promedio ya que a veces estará más cerca del primer nivel, otras más cerca del último nivel y otras en nivel más centrado.

Ejemplo:

Si tiene 16 elementos (n=16), el peor de los casos será la inserción al final, ya que será necesario recorrer los n elementos, es decir, 16. El mejor caso sería insertar en el nivel inmediato. El promedio daría $\log_2 (16) = 4$.

Pues en un árbol si se tienen 16 elementos, estos se pueden repartir de una forma uniforme en ramas de forma qué por ejemplo cada una de ellas tenga 4 elementos, de esta forma al insertar un elemento en una rama como mucho se tendrá que recorrer 4 elementos, de aquí sale él O (Log n) ya qué $O (\log_2 (16)) = 4$.

5. CASO DE ESTUDIO

5.1 Título

Creación de una interfaz gráfica para la gestión de un organigrama.

5.2 Introducción

Un organigrama es una representación en forma de árbol de los distintos niveles de una organización. Opcionalmente, en organigramas más detallados, se ofrece también el nombre de aquellos que los representan. El comienzo ha de ser el nivel más alto, con más poder, dentro de la organización.

En este caso de estudio, se crea una interfaz gráfica para la gestión del organigrama de una empresa. A su vez podrá también eliminar o añadir cualquier nodo. El programa también permite guardar los datos introducidos en un documento de texto, imprimir el árbol guardado y salir del sistema.

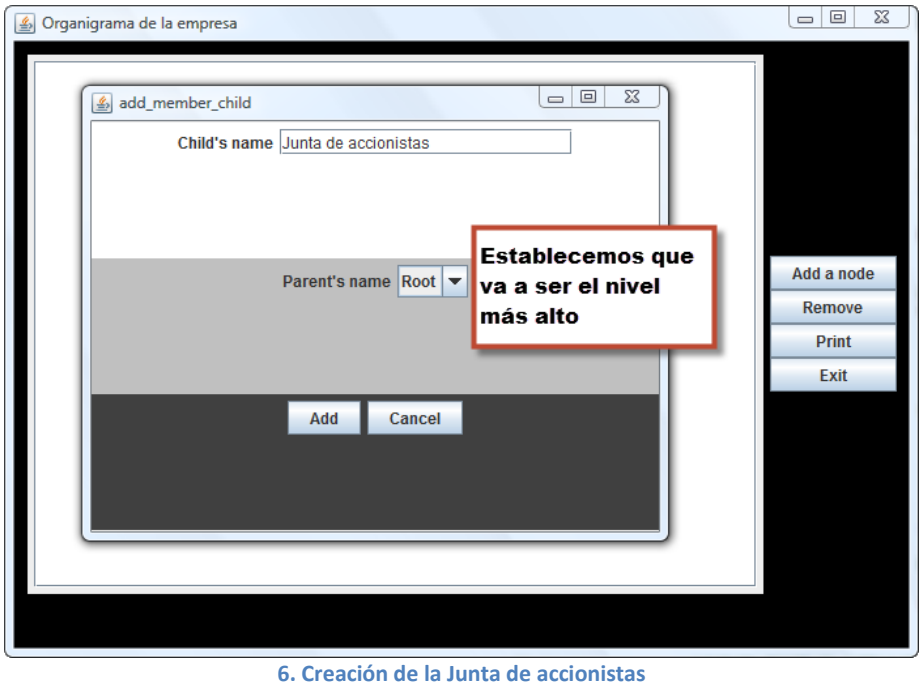
5.3 Casos de uso

5.3.1. Escenario: Creación de un organigrama de una empresa

Precondiciones: La empresa no tiene anteriormente ningún organigrama creado. Partimos del documento en blanco.

Descripción: La secretaría de una empresa decide poner orden y clasificar al personal de la organización de una manera clara y sencilla. Para ello decide ir organizando por líneas de profundidad, desde el nivel más alto al más bajo.

El nivel más alto es el constituido por la junta de accionistas:

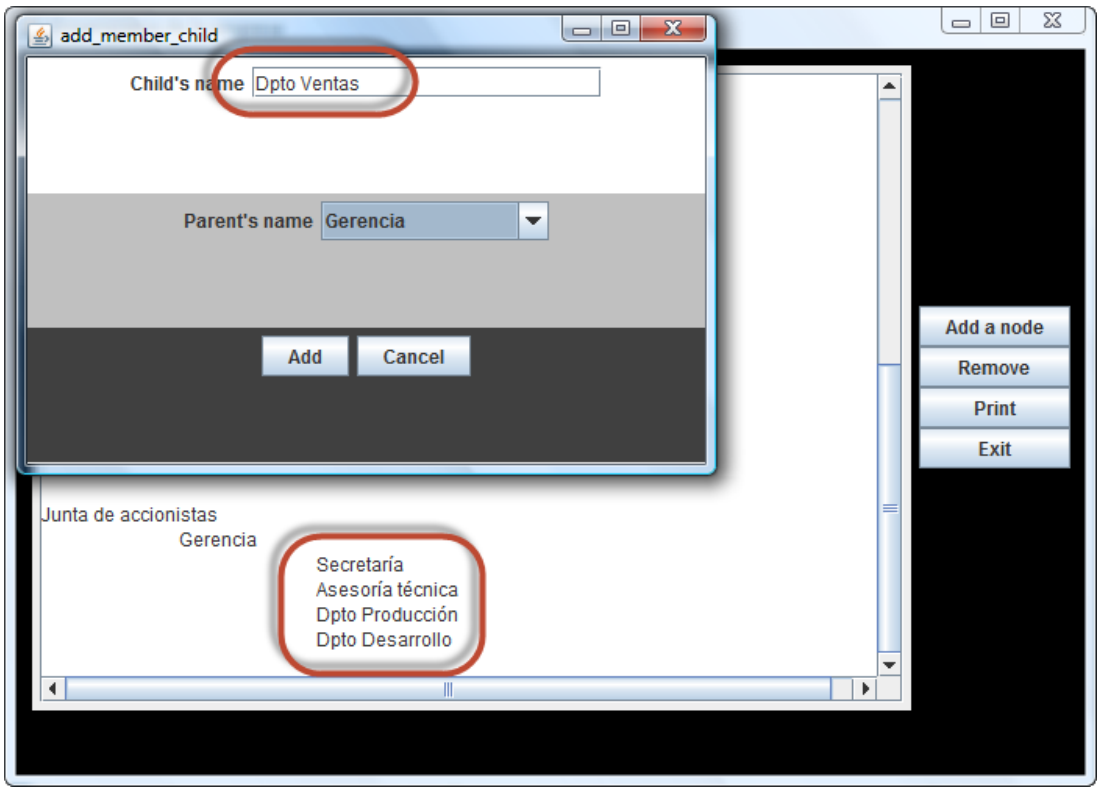


6. Creación de la Junta de accionistas

A continuación establecemos como nivel inferior a éste la gerencia:

7. Creación de Gerencia

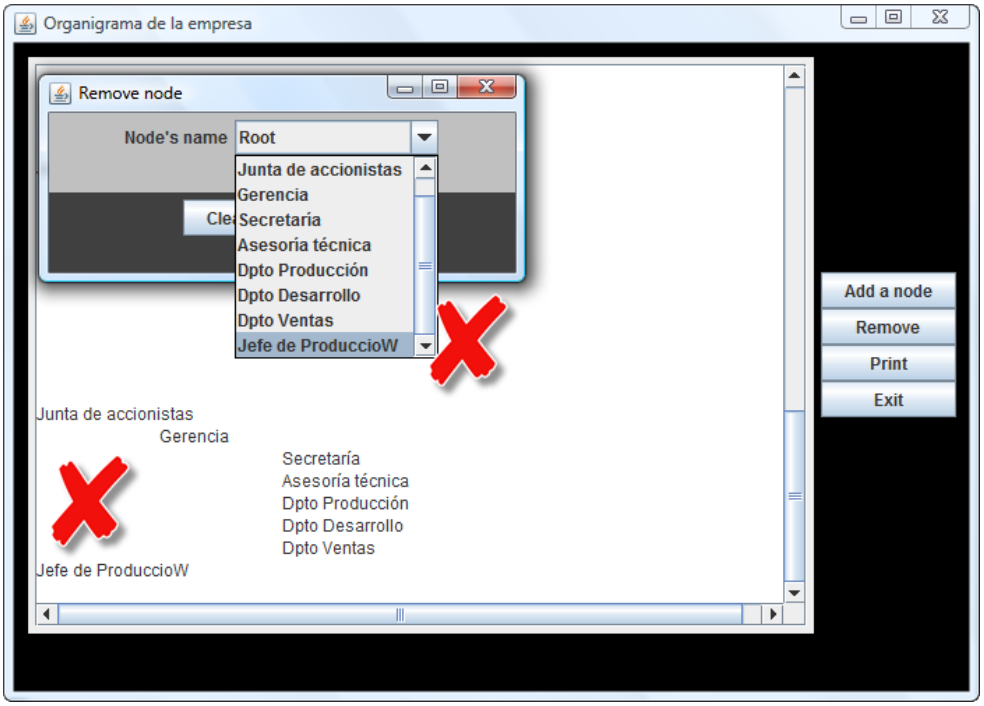
El siguiente subnivel está constituido por secretaría, asesoría técnica, el departamento de producción, el de desarrollo y el de ventas. Añadimos cada uno de los miembros:



8. Creación tercer nivel

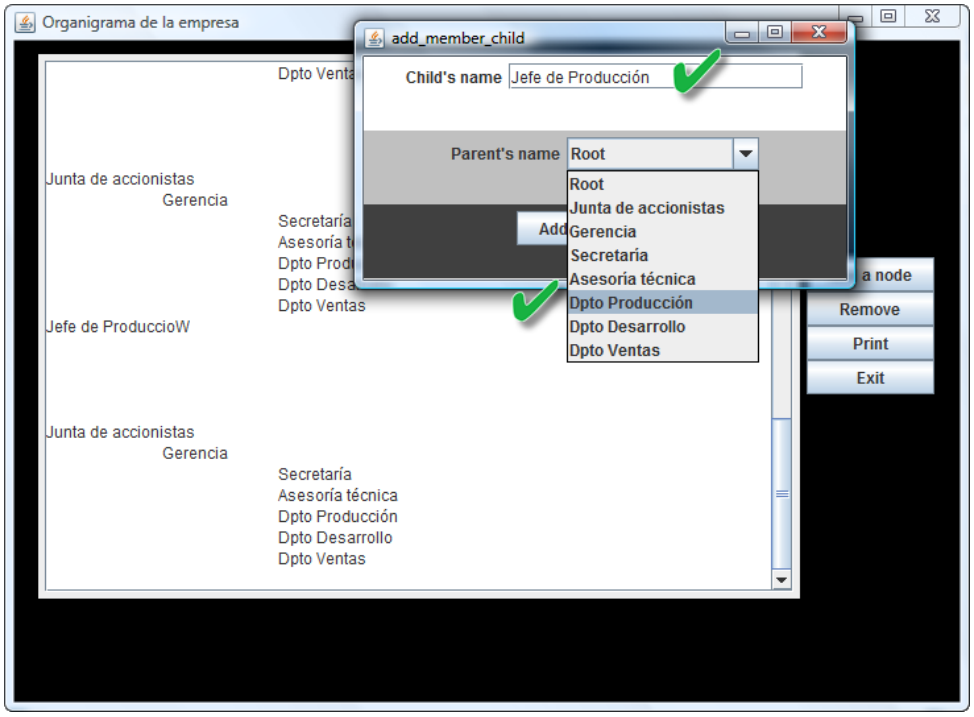
Secretaría y asesoría técnica no tienen niveles inferiores. La atención se centra en los departamentos. Dentro de los departamentos hay dos niveles, el superior serán los jefes de departamento.

Comenzamos a incluir a un jefe de departamento y nos damos cuenta de que hemos cometido un error ortográfico y no lo hemos colocado en el nivel correcto. Eliminamos ese nodo.



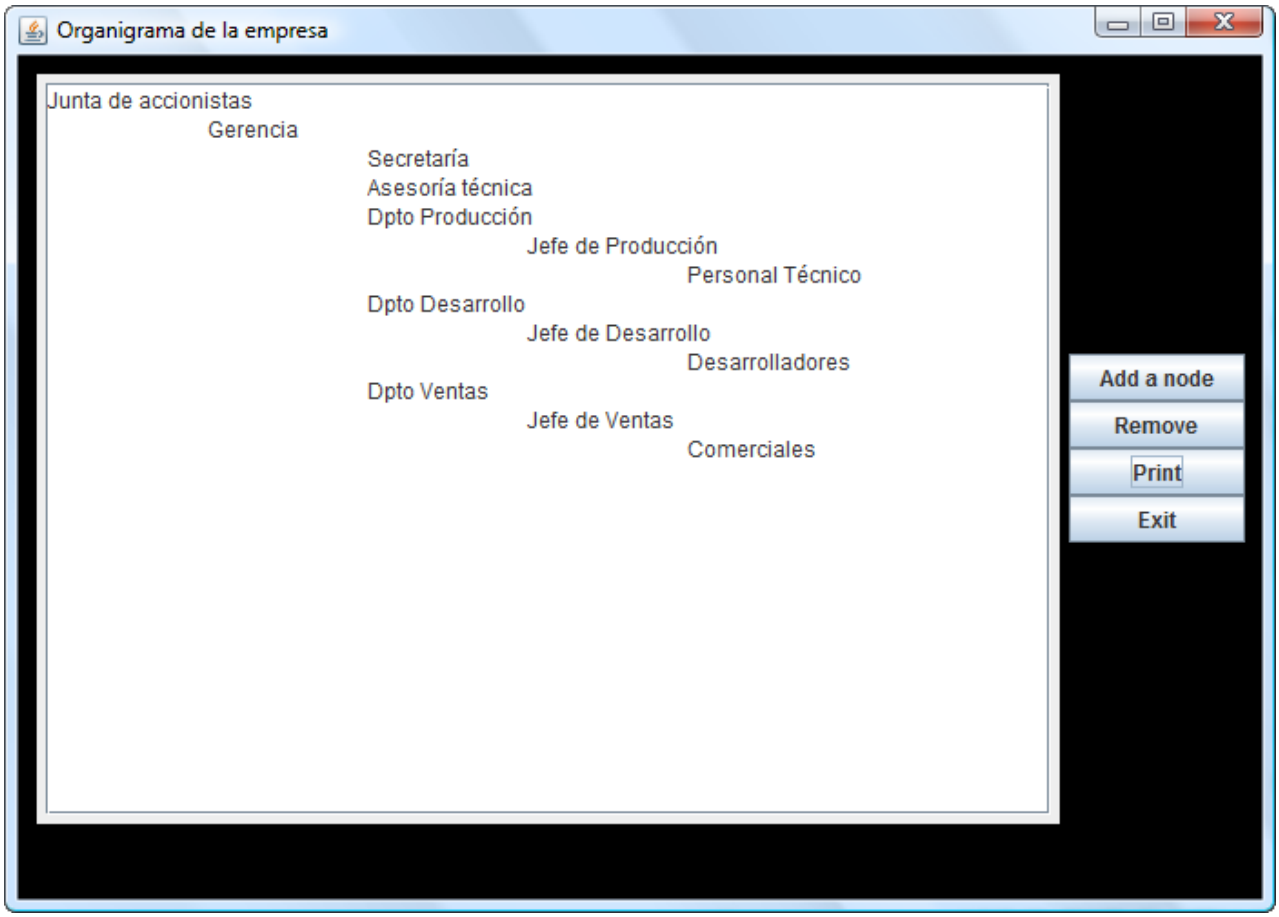
9. Ejemplo eliminación de nodo

Una vez eliminado, lo creamos de nuevo:



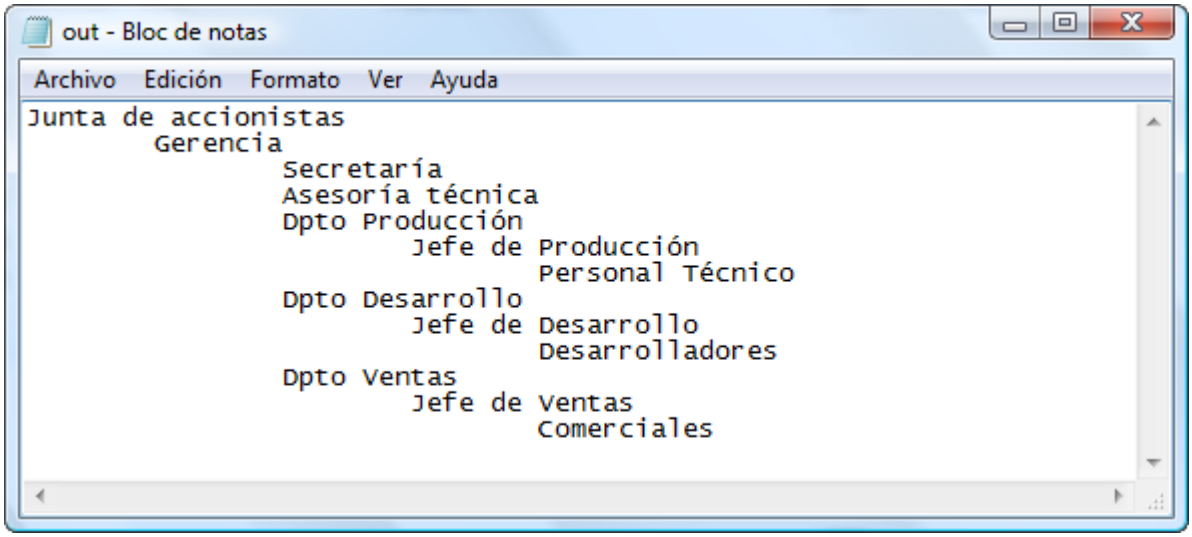
10. Nodo corregido

Añadimos a los demás jefes de departamento, al personal técnico y a los comerciales, quedando el organigrama de la siguiente manera:



11. Organigrama final

Además de esto, tenemos la facilidad de exportar el organigrama, simplemente cogiendo el archivo de texto out.txt que se ha generado. Desde secretaría es posible mandarlo, por ejemplo, como documento adjunto en un email a Gerencia y que nos den el visto bueno.



12. Out.txt

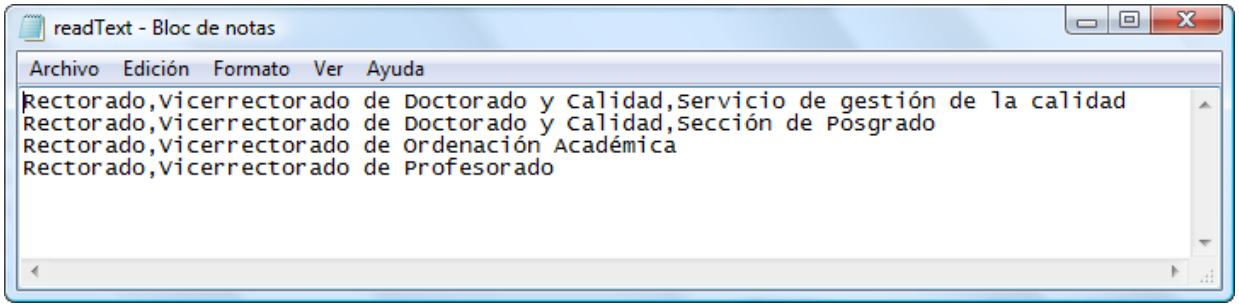
5.3.2. Diseño textual

El programa está formado por cuatro componentes: la clase árbol, la clase hoja, la clase test y la interfaz gráfica de usuario (tree.java, leaf.java, test.java y GUI.java respectivamente).

En una primera versión el programa se ejecutaba a través de la clase test. Por consola, se nos da la bienvenida al programa y se muestra un pequeño menú que nos da la posibilidad de añadir nuevos nodos, eliminarlos, guardarlos, imprimir el árbol y/o salir.

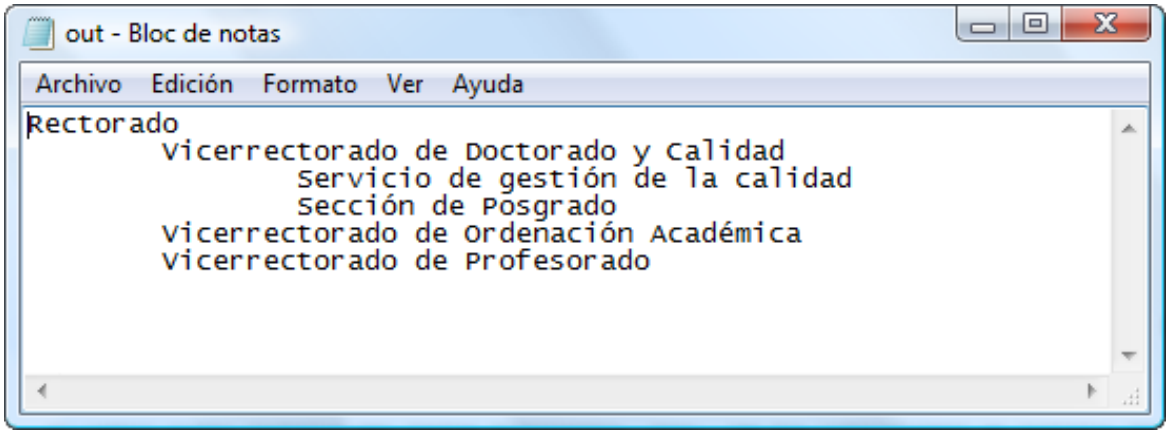
El programa lee los datos de un archivo readText.txt y una vez ejecutado el comando "save" los guarda en el archivo out.txt con formato tabulado.

El archivo readText.txt guarda los datos con un formato lineal, y desde él se cogen los datos cuando se inicia el programa.



13. ReadText.txt

El archivo out.txt guarda los datos con un formato tabulado, y con él tenemos la posibilidad de exportar nuestro árbol, imprimirlo, etc.



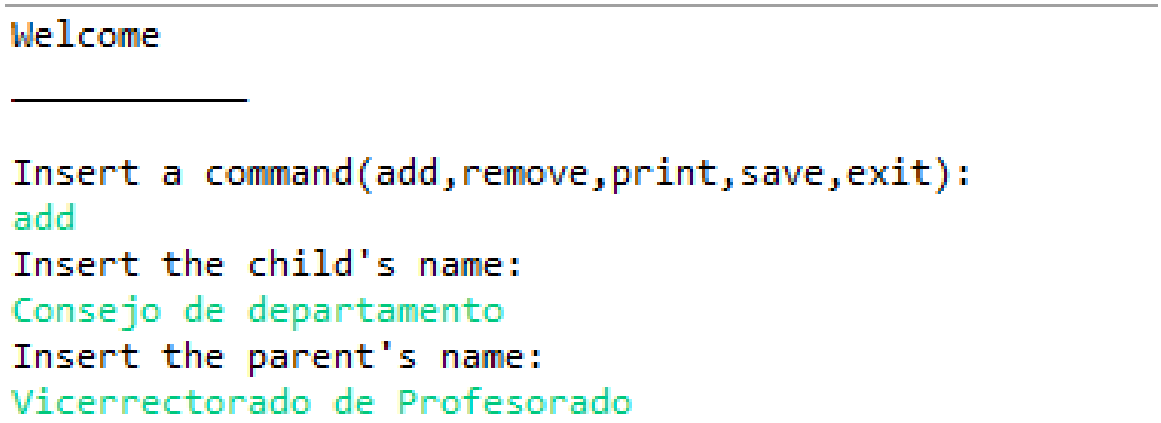
14. Out.txt

La decisión de estructurar la información de esta manera se basa en la facilidad con la que rápidamente nos podemos hacer una idea de cómo está estructurado el árbol. Al grabar el árbol en un archivo que tabula de igual manera la información, es posible exportarlo y ver en cualquier equipo el esquema realizado. Además, creo que es importante haber escogido un archivo .txt por el poco espacio en disco que ocupa.

5.3.2.1 Añadir

Si se escribe la opción “add” podremos añadir a un nuevo miembro al árbol. Para ello es necesario saber el nombre del nodo a añadir (Child’s name) y el nombre del padre. Vamos a empezar añadiendo como primer nivel “Rectorado”. Al ser el nivel más alto, no tiene padre, por lo tanto como nombre del padre pondremos “null”. Se irá comparando el nombre del padre introducido por teclado con todos los nombres de los nodos que haya en el documento de texto guardado “read.txt” y cuando lo encuentre se insertará el nuevo hijo a ese padre.

Añadimos unos ejemplos más para ver el proceso con más nodos.



15. add

El código para realizar esta acción es el siguiente:

En test.java:

```
//las opciones son añadir, eliminar, guardar, imprimir y salir
System.out.println("Insert a command(add,remove,print,save,exit): ");
command = Teclado.readString();

if(command.equals("add")){
      System.out.println( "Insert the child's name: " );
      child = Teclado.readString();
      System.out.println( "Insert the parent's name: " );
      parent = Teclado.readString();
      if (parent.equals("null")){
            //si el archivo readText.txt esta vacío, entonces
            //se inserta un hijo
            root.insertChild(new Leaf(child, root));
            tree.setControl(true);
      }else{
            //si el archivo readText.txt no está vacío, entonces
            //el programa ha de buscar al padre e insertar al hijo
            root = tree.addNewChild(child, parent,root);
      }
```

```
      if ((tree.isControl()) & (tree.metodoyaexiste()==false)){
            System.out.println("The new member was sucessfully added");
            tree.setControl(false);
      }else if ((tree.isControl()==false) & (tree.metodoyaexiste() ==false)){
            System.out.println("\n”);
      }else if (tree.metodoyaexiste()==true){
            tree.setmetodoyaexiste(false);
            tree.setControl(false);
            System.out.println("This node already exist");
      }
}
```

Vemos que desde este código realiza una llamada a `insertChild(new Leaf(child, root))` en el caso en el que el nodo no tenga padre y a `tree.addNewChild(child, parent,root)` en el caso de que sea un nodo que insertamos bajo otro.

insertChild(**new** Leaf(child, root)) en Leaf.java:

```
public void insertChild(Leaf node) {//se inserta un hijo
      if (ContainsNode(node)==-1){
            // Si este hijo no contiene el nodo que se le está pasando se le
            //añade en la siguiente línea
            this.children.add(node);
      }
}
```

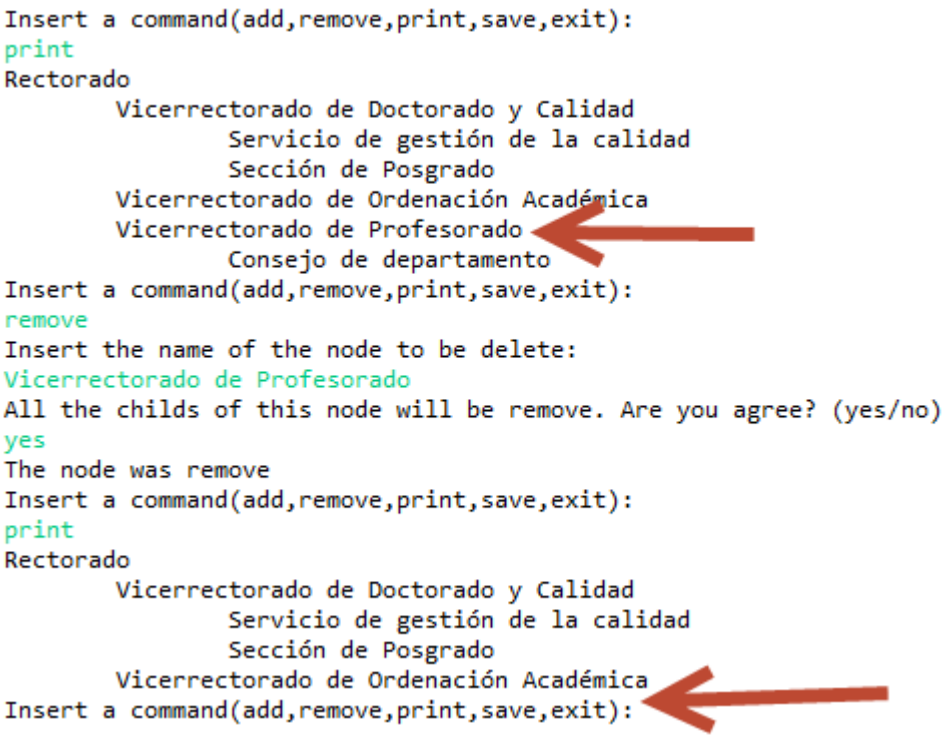
tree.addNewChild(child, parent,root) en Tree.java:

```
public Leaf addNewChild(String childName, String parentName, Leaf rootNode){
      //se da el número de hijos de ese nodo
      for(int i=0; i<rootNode.getChildren().size(); i++){
            //se recorren todos los hijos de esa hoja y el nodo en el que este
            //en ese momento será el padre
            Leaf parentNode = rootNode.getChildren().get(i);
            for(int j= 0;j<parentNode.getChildren().size();j++){
                  Leaf hijo=parentNode.getChildren().get(j);
                  if(hijo.getName().equals(childName)){
                        control=true;
                        yaexiste=true;

                        j=parentNode.getChildren().size();
                        i=rootNode.getChildren().size();
                  }
            }

            if(parentNode.getName().equals(parentName)&& control ==false){

                  // se comparan los nombres
                  if(!control){ //if(control=false){
                        parentNode.insertChild(new Leaf(childName,
                        parentNode));
                        // si es el nombre que el programa está buscando,
```

```
                        //inserta el nodo
                        control = false;
                  }
            }
            if (parentNode.getChildren().size() != 0){
                  //si el nodo tiene hijos, el programa llama al método
                  //recursivo
                  addNewChild(childName, parentName, parentNode);
            }
      }
      return rootNode;
}
```

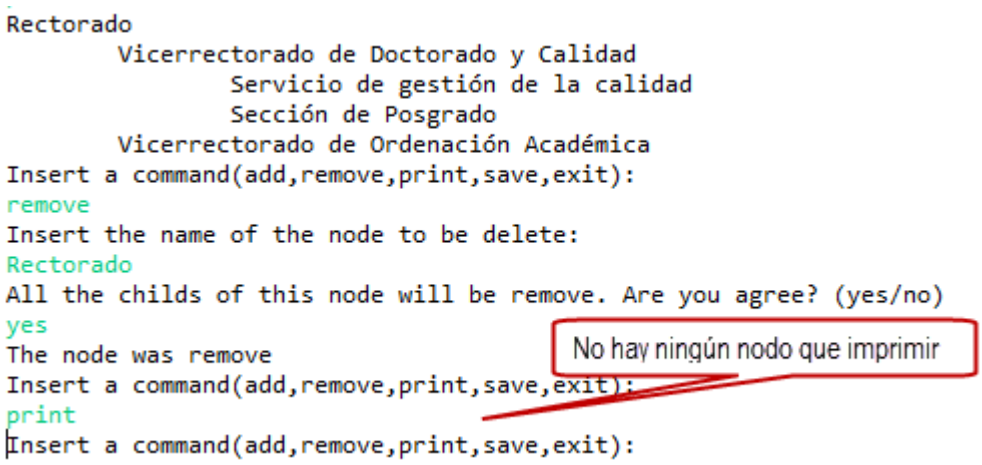
5.3.2.2 Eliminar

Si se escribe la opción “remove” podremos eliminar a un miembro al árbol. Para ello es necesario saber el nombre del nodo a eliminar (Child’s name). Hay que tener en cuenta que una vez que se elimine ese nodo, también se eliminarán los descendientes de él. Para que el usuario tenga conocimiento de esto, se le pregunta se realmente está seguro de la eliminación, ya que se eliminarán los nodos descendientes. En el caso de eliminar el primer nodo, eliminaríamos todo el árbol. Vamos a empezar eliminando el nodo Vicerrectorado de Profesorado.



16. Eliminación Vicerrectorado de Profesorado

Como se ve en el ejemplo, en el caso de una eliminación correcta, el programa nos confirma la acción. Si el nodo no existiera, nos mostraría un mensaje por pantalla advirtiéndonos. Si cuando nos pregunta si estamos seguros de eliminar dijéramos que no, también nos mostraría un mensaje por pantalla confirmándonoslo. Eliminamos ahora Rectorado para ver la eliminación del árbol completo.



17. Eliminación Rectorado

El código para realizar esta acción es el siguiente:

En test.java:

```
}else if(command.equals("remove")){
      System.out.println( "Insert the name of the node to be delete: " );
      node = Teclado.readString();
      System.out.println( "All the childs of this node will be remove. Are you
      agree? (yes/no)" );
      agree = Teclado.readString();
      if (agree.equals("no")){
            System.out.println ("The node was no removed");
      }if (agree.equals("yes")){
            tree.removeChild(node,root);
            if (!tree.isControl()){
                  System.out.println("The node was remove");
                  tree.setControl(false);
            }else{
                  System.out.println("The node was not found");
            }
      }
}
```

Vemos que desde este código se realiza una llamada a `tree.removeChild(node,root)`. Este método de Tree.java es el siguiente:

```
public void removeChild(String Node, Leaf rootNode){
	if (Node.equals("Root")){
	//este if borra la primera rama que cuelga de root si elijo borrar root
		while(rootNode.getChildren().size()!=0){
			int n=0;
			rootNode.getChildren().remove(n);
		}
	}else{
		for(int i=0; i<rootNode.getChildren().size(); i++){
			Leaf parentNode = rootNode.getChildren().get(i);
			//se recorren uno a uno los nodos del arbol
			if(parentNode.getName().equals(Node)){
				// se comparan los nombres
				rootNode.getChildren().remove(i);
				// si es el nombre que el programa está buscando,
				// elimina el nodo
				control = true;
				// se encuentra un nodo con el mismo nombre que el
				//usuario ha introducido
			}
			removeChild(Node,parentNode);//llamada metodo recursivo
		}

	}
	control=false;
}
```

5.3.2.3 Imprimir

Si se escribe la opción “print” podremos imprimir por consola nuestro árbol. En este caso no necesitamos saber ningún parámetro, simplemente nos imprimirá por pantalla el árbol. El árbol se muestra tabulado para ver rápidamente los distintos niveles de profundidad, de que nodo son hijos, etc. El archivo que nos muestra así el árbol es “out.txt”.

```
Insert a command(add,remove,print,save,exit):
print
Rectorado
	Vicerrectorado de Doctorado y Calidad
		Servicio de gestión de la calidad
		Sección de Posgrado
	Vicerrectorado de Ordenación Académica
	Vicerrectorado de Profesorado
		Consejo de departamento
```

18. Print

El código para realizar esta acción es el siguiente:

En test.java:

```
else if (command.equals("print")){
      tree.getTextFile().setLength(0);// va a la primera linea
      tree.getFirstFile().setLength(0);
      tree.printChildren(root, false);
}
```

Vemos que desde este código se realiza una llamada a tree.printChildren(root, false). Este método de Tree.java es el siguiente:

```
public void printChildren(Leaf rootNode, Boolean command){
      String tab = "";
      for(int i=0;i<rootNode.getChildren().size(); i++){
            tab = "";
            for(int j=0; j<rootNode.rowofChild(rootNode); j++){
                  //con este bucle se puede ver el nivel de profundidad que
                  //hay. El programa añade una nueva tabulación para cada nivel
                  tab += '\t';
            }
            String str = tab + rootNode.getChildren().get(i).getName();
            if (command){
                  //true = actualiza el búfer que luego se utiliza para
                  //imprimir el archivo out.txt
                  textFile.append(str + " " +
                  System.getProperty("line.separator"));
                  //el programa añade las lineas al bufer que escribe el arbol
            }else{
                  //false = el programa actualiza el buffer que mas tarde
                  muestra el //árbol
                  System.out.println(str);
            }
            if (rootNode.getChildren().get(i).getChildren().size() != 0){
                  //si el nodo tiene hijos, el programa llama de nuevo al
                  //metodo recursivo
                  printChildren(rootNode.getChildren().get(i),command);
            }else{
                  firstFile.append(setLineOfFirstFile(rootNode.getChildren().ge
                  t(i)) + System.getProperty("line.separator"));
                  // el programa obtiene la línea para añadir en el búfer del
                  //archivo readText.txt
            }
      }
}
```

5.3.2.4 Guardar

Si se escribe la opción “save” guardaremos los cambios realizados en nuestro árbol. En este caso no necesitamos saber ningún parámetro, simplemente nos guardará el árbol. Se guardarán los cambios en el archivo readText.txt y en el archivo out.txt.

```
Insert a command(add,remove,print,save,exit):
save
Insert a command(add,remove,print,save,exit):
```

19. Save

El código para realizar esta acción es el siguiente:

```
}else if(command.equals("save")){
      tree.getTextFile().setLength(0);
      tree.getFirstFile().setLength(0);
      tree.printChildren(root, true);
      tree.saveTreeStructureToFile(root, "out.txt", tree.getTextFile());
      //escribe el árbol tabulado en el archivo out.txt
      tree.saveTreeStructureToFile(root, "readText.txt", tree.getFirstFile());
      //escribe el árbol estructura en el archivo readText.txt
}
```

Vemos que desde este código se realiza una llamada a tree.saveTreeStructureToFile (), donde se vuelcan los datos guardados en el buffer a los archivos. Este método de Tree.java es el siguiente:

```
public void saveTreeStructureToFile(Leaf rootNode, String filename,
StringBuffer strBuffer){
      try{
          // se crea el archivo
          FileWriter fstream = new FileWriter(filename);
          BufferedWriter out = new BufferedWriter(fstream);
          out.write(strBuffer.toString());
          out.close();
      }catch (Exception e){
          System.err.println("Error: " + e.getMessage());
      }
}
```

5.3.2.5 Salir

Si se escribe la opción “exit” salimos del programa y veremos un mensaje de salida.

```
Insert a command(add,remove,print,save,exit):
exit
The program was closed. Thank you for use this program
```

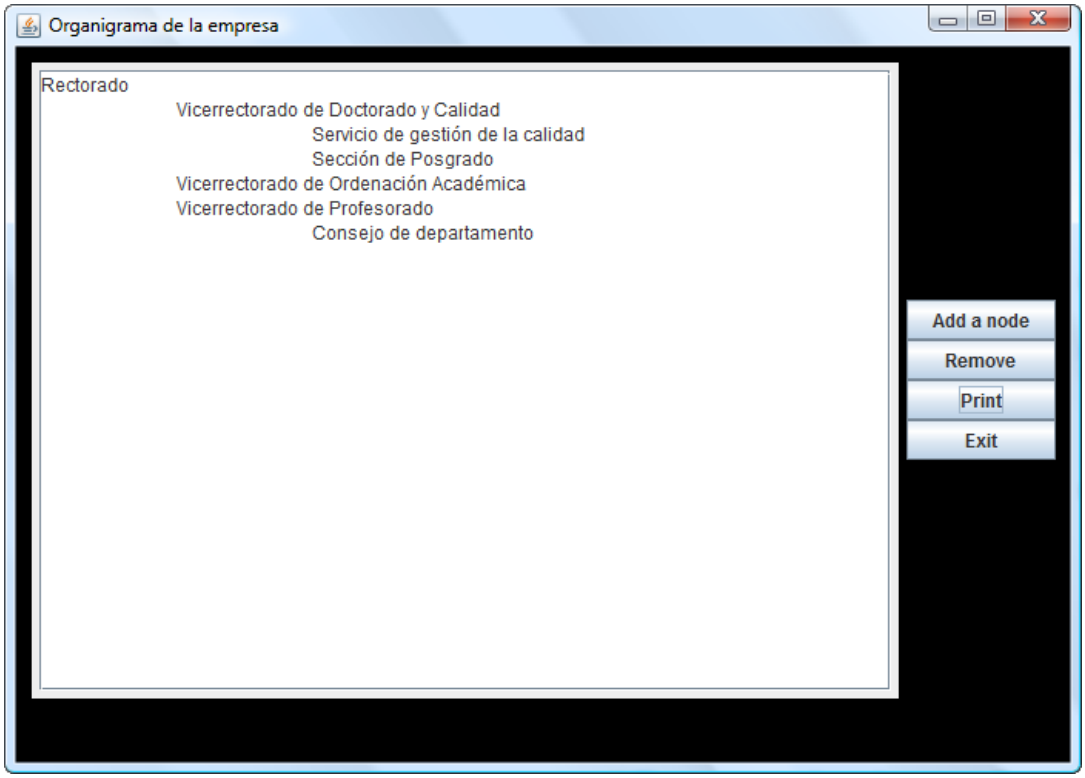
20. Exit

El código para realizar esta acción es el siguiente:

```
do{
…
}while((!command.equals("exit")) && (!command.equals("" +"")));
System.out.println("The program was closed. Thank you for use this program");
```

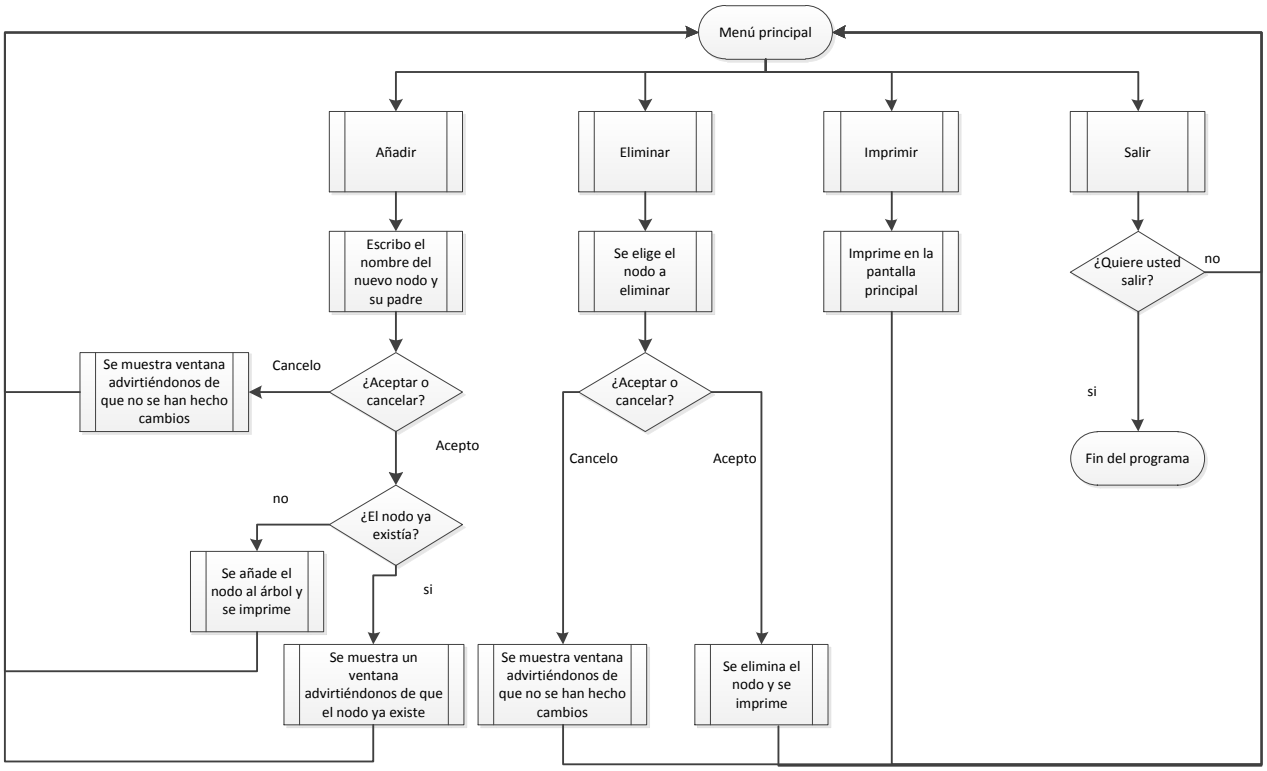
5.3.3. Diseño gráfico

Para diseñar la interfaz gráfica que visualice de la forma más clara y sencilla los árboles, en nuestro caso, el organigrama, se crea una ventana principal con un área de texto, donde se muestra el árbol. A la derecha de esta área se han colocado los diferentes botones de funciones.



21. Ejemplo GUI

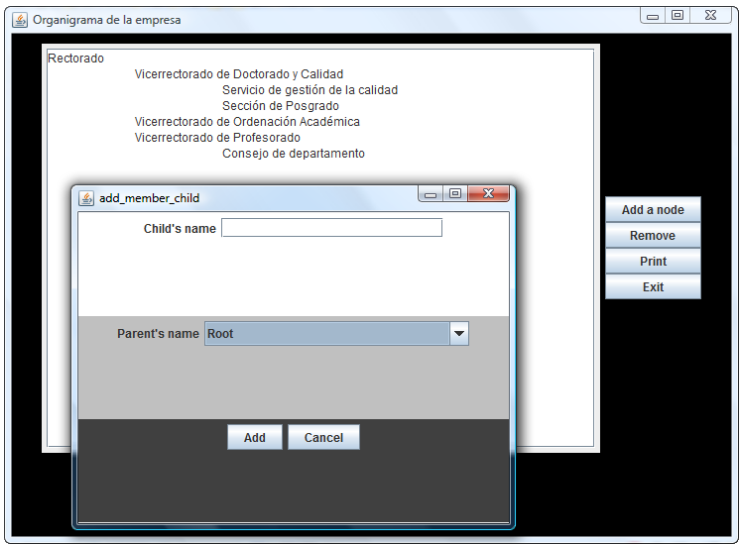
Se define el funcionamiento del programa con el siguiente flujograma:



La decisión de este tipo de diseño se basa, además de en lo descrito en punto 5.3.2, en la claridad de la interfaz gráfica. Se pretende que todas las opciones estén claramente definidas. Se facilita la visión en todo momento del árbol, ya que tras añadir o eliminar un nodo, automáticamente se imprime el árbol en el área de texto.

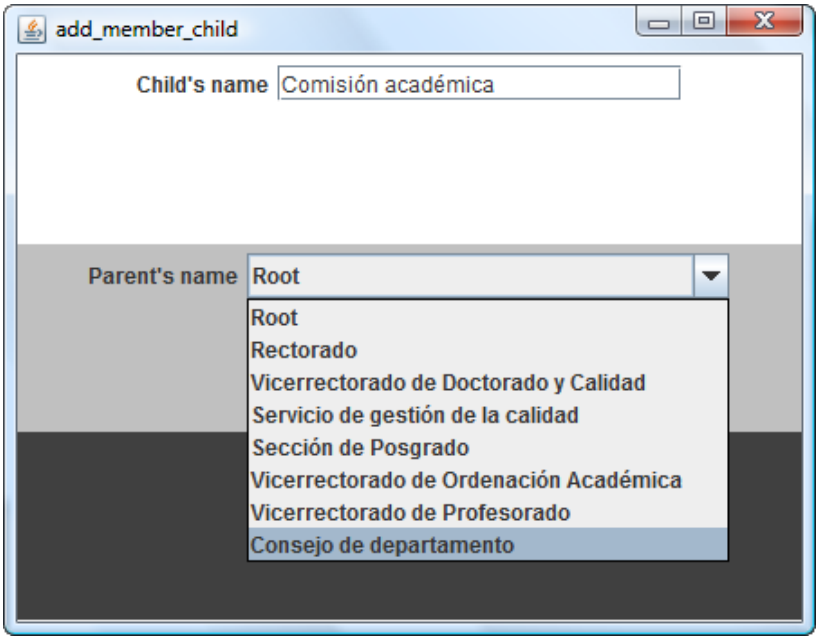
5.3.3.1 Añadir

Si se pulsa sobre el botón “Add a node” se abre una ventana donde nos pide el nombre del nuevo nodo y el nombre del padre. EL nombre del hijo le escribimos en el cuadro de texto mientras el nombre del padre lo escogemos del desplegable.



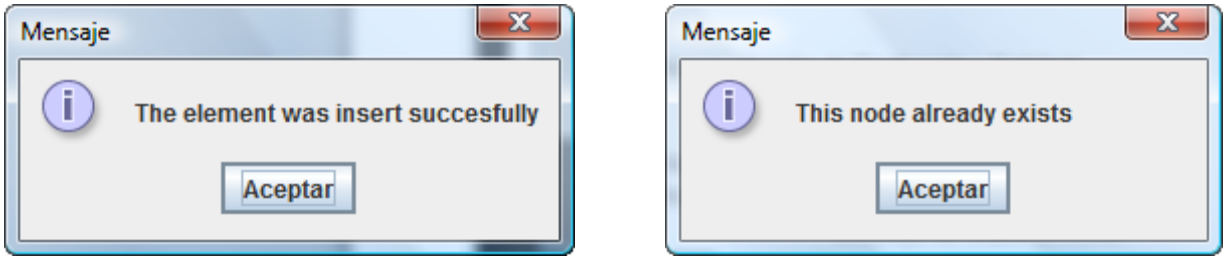
22. Add a node

Como ejemplo vamos a añadir el hijo “Comisión académica” al padre “Consejo de departamento”.



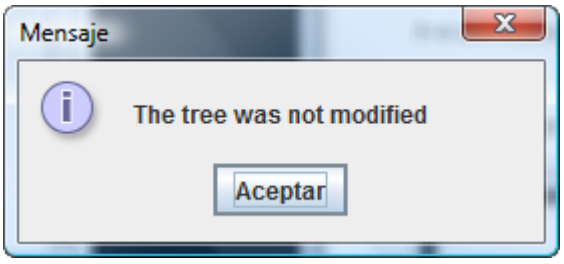
23. Ejemplo Add a node

Cuando pulsamos “Add”, se añade el nodo, y para tener una confirmación, se programa la aparición de una ventana emergente que nos confirme el éxito de la acción. Así mismo, en caso de introducir un nodo ya existente, también emerge una ventana advirtiéndonos de ello. Además de añadirse el nodo, se guardan los cambios y se imprime el árbol en la ventana principal.



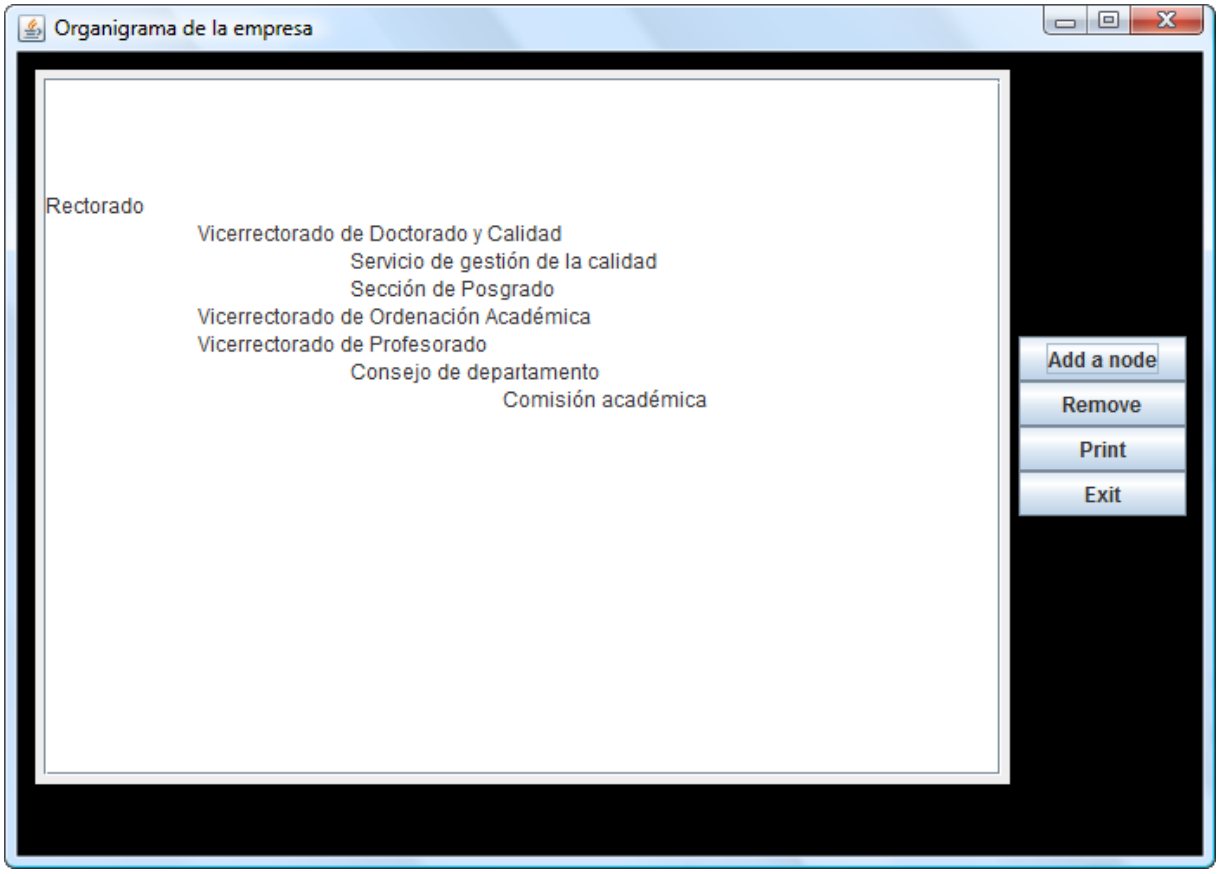
24. Ventanas emergentes

En el caso de pulsar “Cancel”, la ventana emergente será:



25. Cancel

Quedando el organigrama de la siguiente manera:

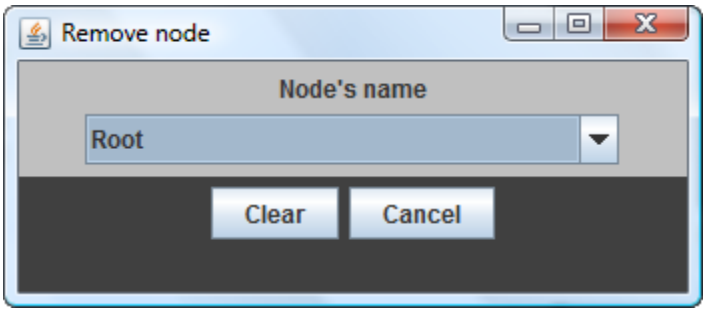


26. Nodo añadido

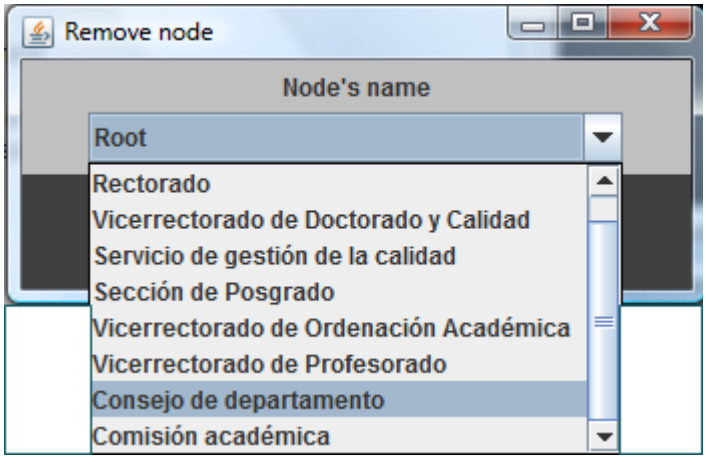
Se ha querido diseñar la ventana en diferentes colores para mostrar la ampliación de conocimientos en interfaces gráficas, desde pequeños detalles como la selección de colores, hasta la colocación de botones en diferentes paneles o áreas del panel, cuadros de texto, desplegables, ventanas emergentes, etc.

5.3.3.2 Eliminar

Si se pulsa sobre el botón “Remove” se abre una ventana donde nos pide el nombre del nodo a eliminar. El nombre del nodo lo escogemos del desplegable.

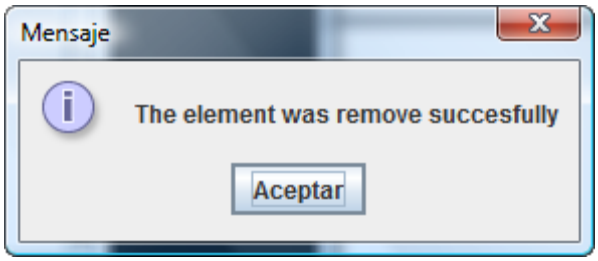


27. Remove node



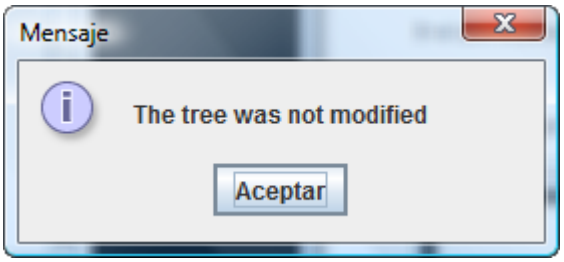
28. Desplegable submenú

Una vez pulsemos "Clear" el nodo se eliminará, se guardarán los cambios y se imprimirá el árbol en la ventana principal saliendo el siguiente mensaje:



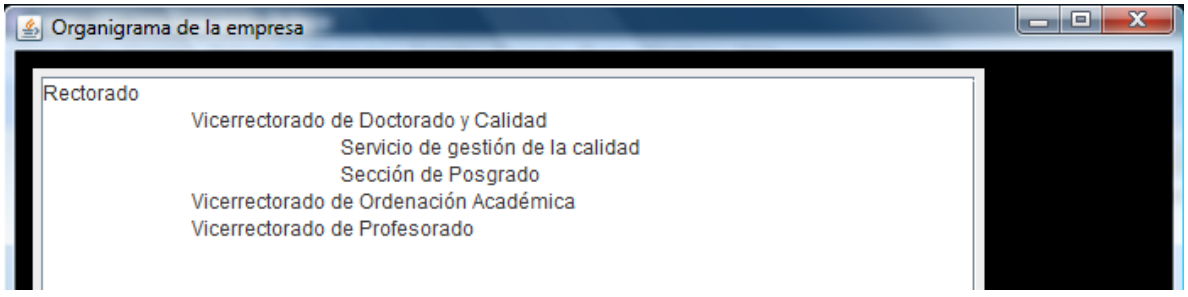
29. Nodo eliminado

En el caso de pulsar "Cancel", la ventana emergente será:



30. Cancel

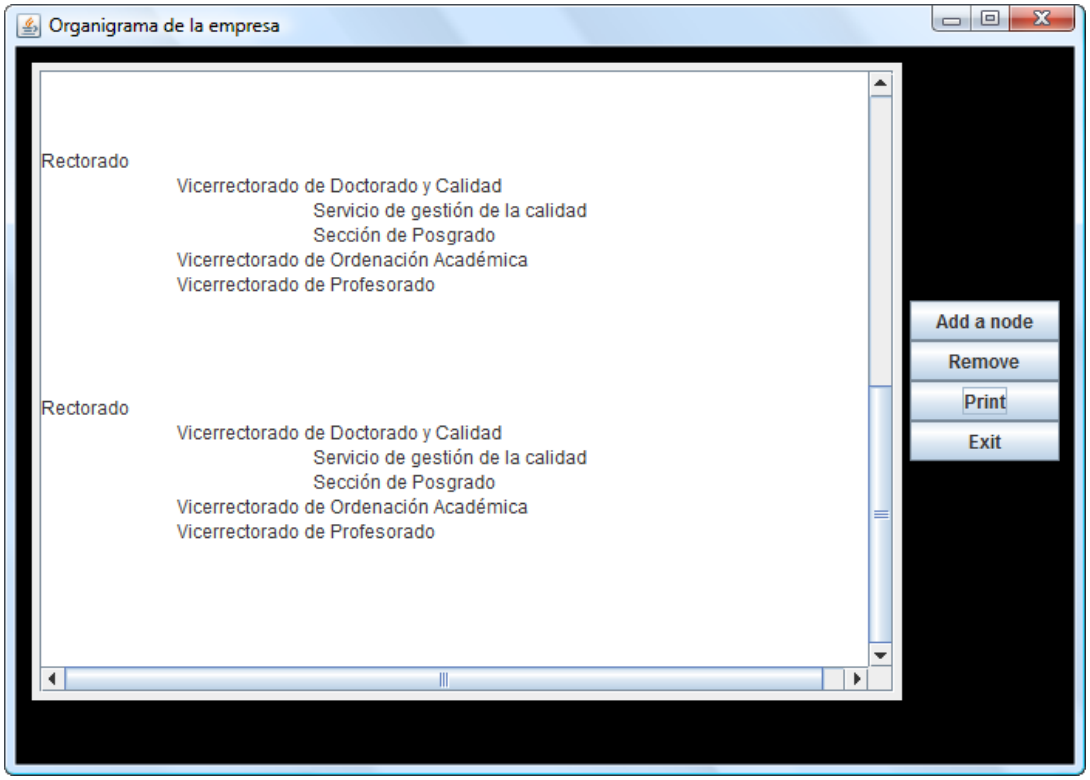
Quedando el organigrama de la siguiente manera:



31. Nodo eliminado

5.3.3.3 Imprimir

Si se pulsa sobre el botón “Print” se imprime en el cuadro de texto de la ventana principal el árbol creado. Se imprimirá tantas veces se desee, imprimiéndose al final del cuadro de texto.



32. Imprimir

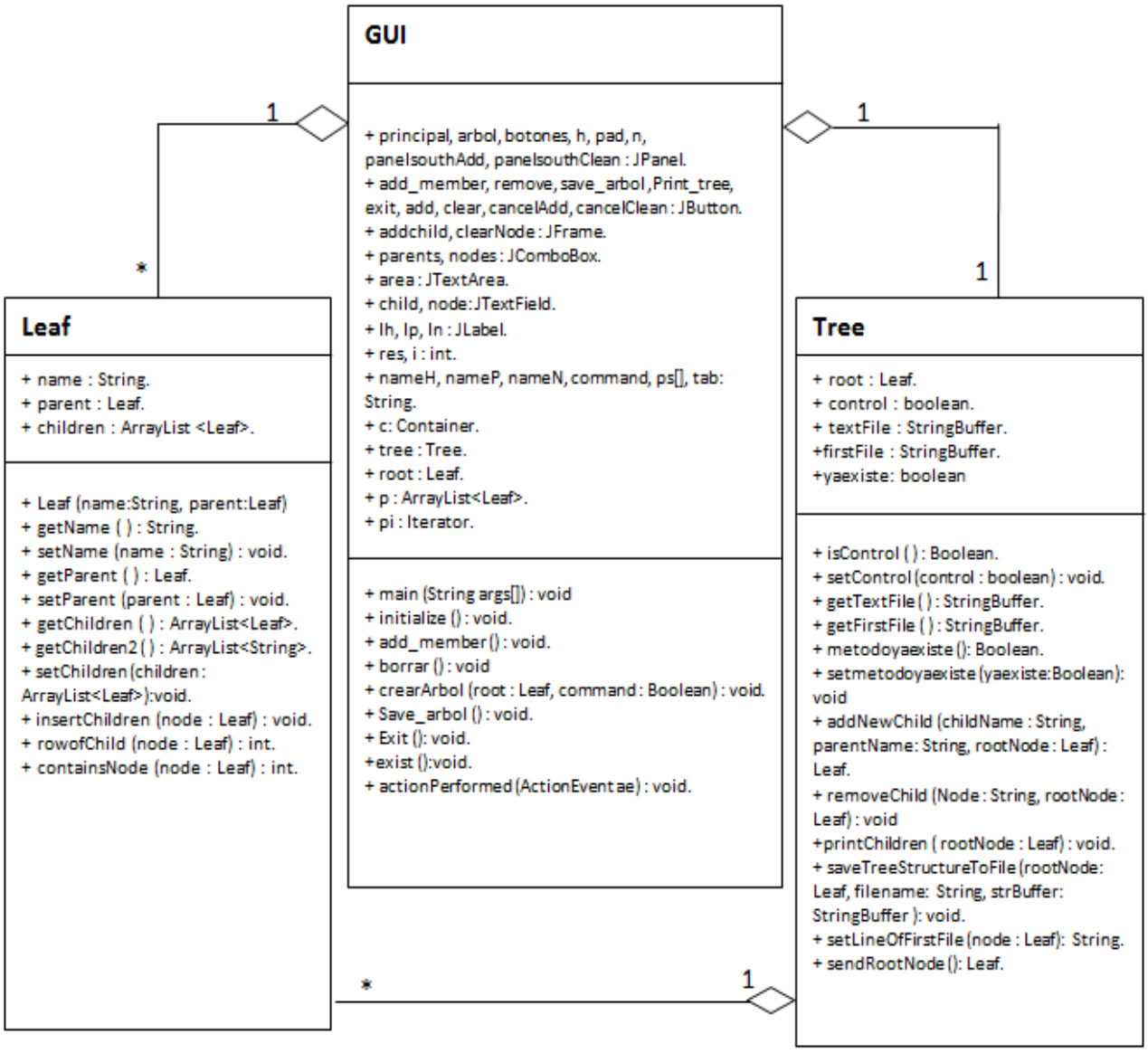
5.3.3.4 Salir

Si se pulsa sobre el botón “Exit” emerge una venta preguntándonos si queremos salir del programa. Siempre es recomendable este tipo de ventanas emergentes para evitar errores del usuario. En caso de pulsar si, el programa se cerrará. En caso de pulsar no, se cerrará la ventana emergente y volveremos al programa principal.

33. Salir

5.4 Especificación

5.4.1. Diagrama UML



Las clases de este programa tienen una relación de asociación, es decir, una clase puede estar formada por objetos de otras clases.

A veces, en una relación de asociación se entiende expresamente que se cuenta con una composición total de partes. En este caso existe una relación de agregación y en el diagrama de clases un diamante vacío se utiliza en la parte que corresponde a todos. Un árbol puede tener infinitas hojas. A su vez la interfaz gráfica de usuario (GUI) tiene un árbol e infinitas hojas.

5.4.2. API

De una manera esquemática se muestran las clases más importantes con una breve descripción de su finalidad:

Clases	Métodos	Explicación
GUI	main	Se crea la ventana principal del programa.
	initialize	Se añaden a la ventana principal todos los botones, el área de texto, etc. Se enlaza cada botón con el método actionPerformed.
	add_member	Se crea la ventana secundaria para añadir a un miembro, se añaden los elementos de la ventana (botones, paneles, etc.)
	borrar	Se crea la ventana secundaria para borrar a un miembro, se añaden los elementos de la ventana (botones, paneles, etc.)
	crearArbol	Se crea el árbol con una estructura tabulada.
	save_arbol	Se guarda el árbol en los archivos readText.txt y out.txt
	exit	Se crea la ventana de salida del programa
	exist	Si un nodo que queremos insertar ya existe, este metodo nos muestra una ventana advirtiéndonoslo.
	actionPerformed	Es el método que recoge la ejecución para cada botón de la interfaz gráfica.
Leaf	leaf	Constructor
	getName	Devuelve el nombre
	setName	Cambia el nombre
	getParent	Devuelve el padre
	setParent	Cambia el nombre del padre
	getChildren	Devuelve una lista de los hijos (Leaf)
	setChildren	Cambia el hijo
	getChildren2	Devuelve un array con los nombres de los hijos (String)
	insertChildren	Inserta un hijo
	rowofChild	Devuelve el número de hijos
	containsNode	Compara el nombre que coge con los que ya están en el árbol
Tree	isControl	Devuelve el valor de la variable "control" (booleana)
	setControl	Cambia el valor de la variable control
	getTextFile	Devuelve el archivo "out.txt"
	GetFirstFile	Devuelve el archivo "readText.txt"
	metodoyaexiste	Devuelve el valor de la variable "yaexiste"
	setmetodoyaexiste	Cambia el valor de la variable "yaexiste"
	addNewChild	Añade a un miembro al árbol
	removeChild	Elimina a un miembro (con sus descendientes) del árbol
	printChildren	Escribe en el búfer el árbol
	saveTreeStructureToFile	Vuelca la información del búfer al archivo "out.txt"
	setLineOfFirstFile	Saca la línea en el árbol de un nodo (hijo, padre, abuelo...)
	sendRootNode	Vuelca la información del búfer al archivo "readText.txt" tabulándolo.

5.5 Herramientas

5.5.1. Java

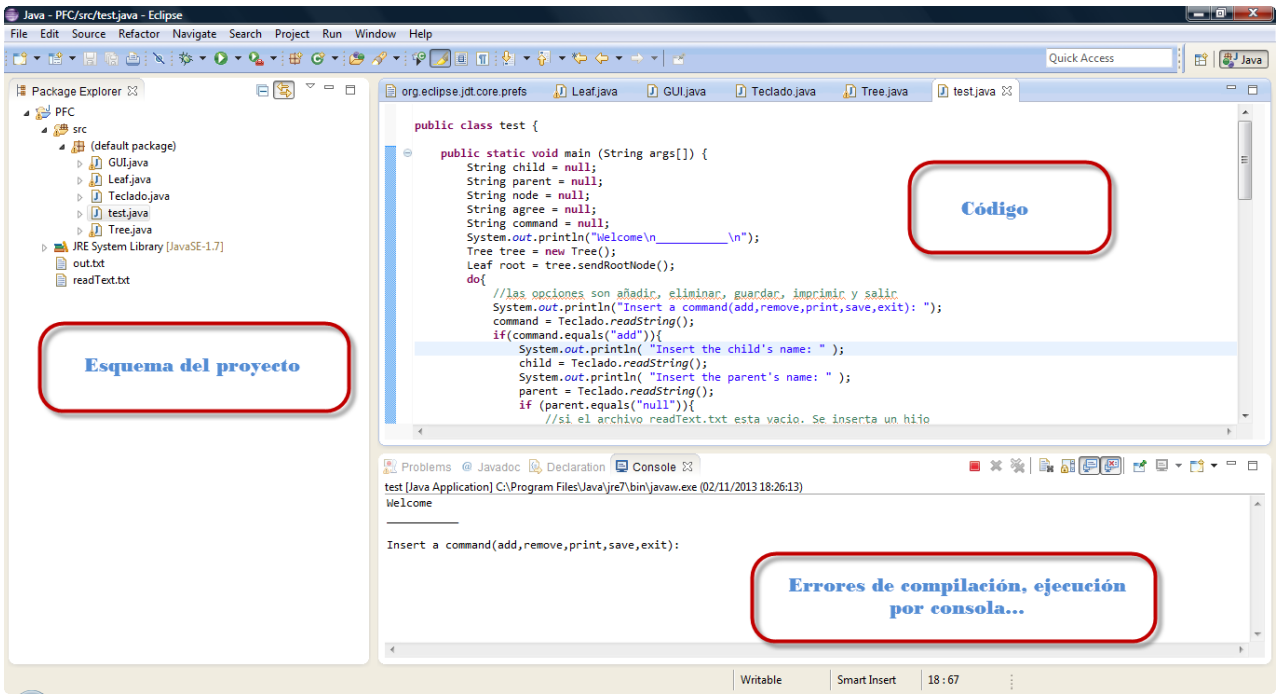
Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems a principios de los años 90. El lenguaje toma mucha de su sintaxis de C y C++, pero tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria. Con respecto a la memoria, su gestión no es un problema ya que ésta es gestionada por el propio lenguaje y no por el programador.

Las aplicaciones Java están típicamente compiladas en un bytecode, aunque la compilación en código máquina nativo también es posible. En el tiempo de ejecución, el bytecode es normalmente interpretado o compilado a código nativo para la ejecución, aunque la ejecución directa por hardware del bytecode por un procesador Java también es posible.

5.5.2. Eclipse

Se ha empleado como entorno de desarrollo Eclipse, por su recomendación por parte de la Fachhochschule St. Pölten, su sencillez de utilización y por las facilidades que ofrece a la hora de programar. Como lenguaje de programación se ha elegido Java, debido al previo conocimiento que se tenía de ese lenguaje y al reto de ampliar conocimientos en él.

Eclipse es una plataforma de desarrollo software genérica desarrollada por una comunidad "open source" que permite crear entornos de desarrollo integrados para distintos lenguajes programación.



34. Eclipse

Aquí se muestra una imagen del entorno de programación. Como se puede observar, en la parte central es dónde aparecen las clases e interfaces creadas con sus respectivos códigos. En la parte de la izquierda de la imagen es donde nos aparece un esquema del proyecto creado incluidas las librerías importadas necesarias a lo largo del desarrollo del proyecto. Y en la parte inferior del entorno nos aparecen los errores que se producen tanto en compilación como durante la ejecución del código creado, además de los resultados obtenidos en un proceso de búsqueda, entre otras cosas.

5.5.3. Librerías utilizadas

En la siguiente tabla se muestra de manera general las librerías usadas para el desarrollo del proyecto:

Librería usada	Uso
`java.awt.*;`	Para la creación de interfaces gráficas
`javax.swing.*;`	Para la creación de interfaces gráficas
`javax.swing.*;`	Para la creación de interfaces gráficas
`java.util.*;`	Es usada para arrays, iteradores, frameworks,etc
`java.io.IOException;`	Para tratar las excepciones
`java.util.ArrayList;`	Para el uso de arraylist
`java.util.Iterator;`	Para el uso de iteradores
`java.io.BufferedReader;`	Lectura en buffer
`java.io.BufferedWriter;`	Almacenamiento en buffer
`java.io.File;`	Tratamiento de archivos

6. CONCLUSIONES Y LÍNEAS FUTURAS

Desde el comienzo de este proyecto se han intentado alcanzar los objetivos exigidos a fin de que la ejecución sea clara y viable.

Este proyecto puede ser útil en múltiples áreas ya que los árboles son estructuras compuestas por nodos, usados para representar datos de forma jerárquica entre los elementos que la conforman y que son utilizados para la solución de una gran variedad de problemas, ya que pueden implementarse fácilmente en un ordenador.

7. ANEXOS

7.1 GUI.java

```
public class GUI extends JFrame implements ActionListener{
	private JPanel principal,arbol,botones,h,pad,n, panelsouthAdd, panelsouthClean;
	//las opciones del menu principal son add_member, remove, print y exit
	private JButton add_member,remove,Save_arbol,Print_tree,Exit;
	private JFrame addchild,clearNode;
	private JButton add,clear,cancelAdd,cancelClean;
	private JComboBox parents,nodes;  //desplegable con todos los nodos del arbol
	private JTextArea area;
	private JTextField child,node;
	private JLabel lh,lp,ln;
	private int res;
	private String nameH, nameP, nameN;
	private String command = null;
	private Tree tree;
	private Leaf root;

	public GUI(){
			super ("Organigrama de la empresa");
			initialize();
	}

	public static void main (String args[]){
		GUI frame = new GUI();
		frame.setSize(700,500);
		frame.setVisible(true);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}


	private void initialize(){
		//se obtiene el contenedor asociado a la ventana
		Container c = getContentPane();
		c.setLayout(new FlowLayout());
		c.setBackground(Color.black);

		//se crean los diferentes paneles
		principal = new JPanel();
		arbol = new JPanel();

		//se crea el bordeado del área de texto,
		//el área central donde se imprimen los arboles
		Box cuadro = Box.createHorizontalBox();
		area = new JTextArea(25,50);
		area.setEditable(false);
		cuadro.add(new JScrollPane(area));

		arbol.add(cuadro);

		//se crean los botones y se añaden los botones a los escuchadores
		botones = new JPanel(new GridLayout(4,1));

		add_member = new JButton("add_member");
		add_member.addActionListener(this);
		botones.add(add_member);

		remove = new JButton("remove");
		remove.addActionListener(this);
		botones.add(remove);
```

```
        Print_tree = new JButton("Print");
        Print_tree.addActionListener(this);
        botones.add(Print_tree);

        Exit = new JButton("Exit");
        Exit.addActionListener(this);
        botones.add(Exit);

        principal.add(arbol);
        principal.add(botones);
        c.add(principal);

        principal.setBackground(Color.black);

        tree = new Tree();
        root = tree.sendRootNode();
}



public void add_member(){
        addchild = new JFrame("add_member_child");
        addchild.setSize(450,350);
        addchild.setDefaultCloseOperation(addchild.DISPOSE_ON_CLOSE);
        addchild.setVisible(true);
        addchild.setLayout(new GridLayout(3,1));
        addchild.setBackground(Color.black);

        h = new JPanel();
        lh = new JLabel ("Child's name");
        child = new JTextField(20);
        h.add(lh);
        h.add(child);
        h.setBackground(Color.white);


        //SE OBTIENEN LOS NODOS EXISTENTES
        //se crea un array con todos los nodos que cuelgan de la raiz
        ArrayList<String> p = root.getChildren2();
        //se crea un iterador para poder ir recorriendo todos los elementos
        //del array
        Iterator pi = p.iterator();
        String []ps = new String[p.size()+1];
        ps[0] = "Root";
        int i = 1;
        while(pi.hasNext()){
                ps[i] = ((String)pi.next());
                i++;
        }

        pad = new JPanel();
        parents = new JComboBox(ps);
        lp = new JLabel("Parent's name");
        pad.add(lp);
        pad.add(parents);
        addchild.add(h);
        addchild.add(pad);
        pad.setBackground(Color.LIGHT_GRAY);

        //CREAR BOTON DE AÑADIR EN EL SUBMENU DE AÑADIR A UN HIJO
```

```
        add = new JButton("Add");
        add.addActionListener(this);
        addchild.add(add);

        //CREAR BOTON DE CANCELAR EN EL SUBMENU DE AÑADIR A UN HIJO
        cancelAdd = new JButton("Cancel");
        cancelAdd.addActionListener(this);
        addchild.add(cancelAdd);

        panelsouthAdd = new JPanel();
        panelsouthAdd.setLayout(new FlowLayout());
        panelsouthAdd.add(add);
        panelsouthAdd.add(cancelAdd);

        addchild.add(panelsouthAdd, BorderLayout.SOUTH);
        panelsouthAdd.setBackground(Color.darkGray);
}

public void borrar(){
        clearNode = new JFrame("Remove node");
        clearNode.setSize(350,150);
        clearNode.setDefaultCloseOperation(clearNode.DISPOSE_ON_CLOSE);
        clearNode.setVisible(true);
        clearNode.setVisible(true);
        clearNode.setLayout(new GridLayout(2,1));
        n = new JPanel();


        //SE OBTIENEN LOS NODOS EXISTENTES
        //para mostrar las opciones de los nodos que es posible borrar
        ArrayList<String> p = root.getChildren2();
        //se crea un iterador para poder ir recorriendo todos los elementos
        //del array
        Iterator pi = p.iterator();
        String []ps = new String[p.size()+1];
        ps[0] = "Root";
        int i = 1;
        while(pi.hasNext()){//se pasa al siguiente
                ps[i] = ((String)pi.next());
                i++;
        }

        nodes = new JComboBox(ps);
        ln = new JLabel ("Node's name");
        n.add(ln);
        n.add(nodes);
        clearNode.add(n);
        n.setBackground(Color.LIGHT_GRAY);

        clear = new JButton("Clear");
        clear.addActionListener(this);
        clearNode.add(clear);

        panelsouthClean = new JPanel();

        //AGREGAR BOTON DE CANCELAR BORRAR UN NODO
        cancelClean = new JButton("Cancel");
        cancelClean.addActionListener(this);
        clearNode.add(cancelClean);

        panelsouthClean.setLayout(new FlowLayout());
```

```
        panelsouthClean.add(clear);
        panelsouthClean.add(cancelClean);

        clearNode.add(panelsouthClean, BorderLayout.SOUTH);
        panelsouthClean.setBackground(Color.DARK_GRAY);
}



public void crearArbol(Leaf root,Boolean command){
        //metodo Print_tree de la clase Tree.java
         String tab = "";
        for(int i=0;i<root.getChildren().size(); i++){
                tab = "";
                for(int j=0; j<root.rowofChild(root); j++){
                        //con este bucle se puede ver el nivel de profundidad del
                        //nodo.
                        //el programa añade una nueva tabulacion para cada nivel
                        tab += '\t';
                }
                String str = tab + root.getChildren().get(i).getName();
                if (command){
                        //true = actualiza el búfer que luego se utiliza para
                        //Print_arbol el archivo out.txt
                        tree.getTextFile().append(str + " " +
                        System.getProperty("line.separator"));
                        //el programa añade las lineas al bufer que escribe el arbol
                }else{
                        //false = el programa actualiza el bufer que mas tarde
                        //muestra el arbol
                        area.append(str+"\n");
                }
                if (root.getChildren().get(i).getChildren().size() != 0){
                        //si el nodo tiene hijos, el programa llama de nuevo al
                        //metodo recursivo
                        crearArbol(root.getChildren().get(i),command);
                }else{
                        tree.getFirstFile().append(tree.setLineOfFirstFile(root.
                        getChildren().get(i)) +
                        System.getProperty("line.separator"));
                        // el programa obtiene la línea para add_member en el
                        //búfer del archivo readText.txt
                }
        }
}


public void Save_arbol(){
        System.out.println("saving...");
        tree.getTextFile().setLength(0);
        tree.getFirstFile().setLength(0);
        tree.printChildren(root, true);
        area.append("\n\n\n\n");
        tree.saveTreeStructureToFile(root, "out.txt", tree.getTextFile());
        //escribe el arbol estructura en el archivo out.txt
        tree.saveTreeStructureToFile(root, "readText.txt", tree.getFirstFile());
        //escribe el arbol estructura en el archivo readText.txt
}
```

```
public void Exit(){
	res = JOptionPane.showConfirmDialog(null, "¿Would you like Exit?",
	"Exit", JOptionPane.YES_NO_OPTION);
	if (res==JOptionPane.YES_OPTION){
		System.exit(0);
	}
}



public void exist(){
	addchild.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
	addchild.setVisible(false);
	JOptionPane.showMessageDialog(addchild,"This node already exists");
}


public void actionPerformed (ActionEvent ae){
	if (ae.getSource()==Print_tree){
		crearArbol(root,false);
		area.append("\n\n\n\n");
	}

	if (ae.getSource()== add_member){
		add_member();
	}

	if (ae.getSource()== add){
		//este boton es el "ADD" de la ventana secundaria
		nameH = child.getText();
		nameP = parents.getSelectedItem().toString();
		boolean comprobar=false;
		//se obtienen los nodos existentes
		//se crea un array con todos los nodes que cuelgan de la raiz
		ArrayList<String> p = root.getChildren2();
		//se crea un iterador para poder ir recorriendo todos los elementos
		//del array
		Iterator pi = p.iterator();
		String []ps = new String[p.size()+1];
		ps[0] = "Root";
		int i = 1;
		while(pi.hasNext()){//se pasa al siguiente
			ps[i] = ((String)pi.next());
			i++;
		}
		//compruebo si el hijo ya existia
		for(int j=0; j<i; j++){
			if(ps[j].equals(nameH)){
				comprobar=true;
				exist();
			}
		}

		if(nameP == "Root"){
			root.insertChild(new Leaf(nameH, root));
		}else{
			root = tree.addNewChild(nameH,nameP,root);
		}

		Save_arbol();
		crearArbol(root,false);
```

```
                addchild.dispose();
                //cuando se pulsa el boton "ADD" se guarda el hijo en el padre
                //seleccionado
                if(comprobar == false){
                JOptionPane.showMessageDialog(addchild,"The element was insert
                succesfully");
            }
        }

        if (ae.getSource()== cancelAdd ){      //Accion de cerrar ventana secundaria
            addchild.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
            addchild.setVisible(false);
            crearArbol(root,false);
            JOptionPane.showMessageDialog(addchild,"The tree was not modified");
        }

        if (ae.getSource()== cancelClean ){    //Accion de cerrar ventana secundaria
            clearNode.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
            clearNode.setVisible(false);
            crearArbol(root,false);
            JOptionPane.showMessageDialog(addchild,"The tree was not modified");
        }


        if (ae.getSource()== remove){    //pantalla principal
            borrar();
        }

        if (ae.getSource()== clear){//este boton es el "Clear" de la segunda ventana
            nameN = nodes.getSelectedItem().toString();//nombre del nodo seleccionado
            tree.removeChild(nameN, root);
            Save_arbol();
            crearArbol(root,false);
            clearNode.dispose();
            //cuando se pulsa el boton "Clear" se elimina el nodo introducido

            if(nameN.equals("Root")){ //se comprueba si el elemento eliminado es root
                JOptionPane.showMessageDialog(clearNode,"All the elements were
                removed");
            }else{
                JOptionPane.showMessageDialog(clearNode,"The element was remove
                succesfully");
            }
        }
        if (ae.getSource() == Exit){
            Exit();
        }

}//cierra metodo actionPerformed

}//cierra GUI
```

7.2 Test.java

```
public class test {
	public static void main (String args[]) {
		String child = null;
		String parent = null;
		String node = null;
		String agree = null;
		String command = null;
		System.out.println("Welcome\n__________\n");
		Tree tree = new Tree();
		Leaf root = tree.sendRootNode();
		do{
			//las opciones son añadir, eliminar, guardar, imprimir y salir
			System.out.println("Insert a command(add,remove,print,save,exit):
			");
			command = Teclado.readString();
			if(command.equals("add")){
				System.out.println( "Insert the child's name: " );
				child = Teclado.readString();
				System.out.println( "Insert the parent's name: " );
				parent = Teclado.readString();
				if (parent.equals("null")){
					//si el archivo readText.txt esta vacio, entonces
					//se inserta un hijo
					root.insertChild(new Leaf(child, root));
					tree.setControl(true);
				}else{
					//si el archivo readText.txt no esta vacio, entonces
					//el programa ha de buscar al padre e insertar al hijo
					root = tree.addNewChild(child, parent,root);
				}
				if ((tree.isControl()) & (tree.metodoyaexiste()==false)){
					//control=true
					System.out.println("The new member was sucessfully
					added");
					tree.setControl(false);
				}else if ((tree.isControl()==false) & (tree.metodoyaexiste()
				==false)){
					System.out.println("\n”);
				}else if (tree.metodoyaexiste()==true){
					tree.setmetodoyaexiste(false);
					tree.setControl(false);
					System.out.println("This node already exist");
				}
			}

			else if (command.equals("print")){
				// me muevo a la primera linea
				tree.getTextFile().setLength(0);
				tree.getFirstFile().setLength(0);
				tree.printChildren(root, false);

			}else if(command.equals("save")){
				tree.getTextFile().setLength(0);
				tree.getFirstFile().setLength(0);
				tree.printChildren(root, true);
				tree.saveTreeStructureToFile(root,"out.txt",tree.getTextFile
				());
				//escribe el arbol estructura en el archivo out.txt
```

```
                        tree.saveTreeStructureToFile(root,"readText.txt",
                        tree.getFirstFile());
                        //escribe el arbol estructura en el archivo readText.txt

                  }else if(command.equals("remove")){
                        System.out.println( "Insert the name of the node to be
                        delete: " );
                        node = Teclado.readString();
                        System.out.println( "All the childs of this node will be
                        remove. Are you agree? (yes/no)" );
                        agree = Teclado.readString();
                        if (agree.equals("no")){
                              System.out.println ("The node was no removed");
                        }if (agree.equals("yes")){
                              tree.removeChild(node,root);

                              if (!tree.isControl()){
                                    System.out.println("The node was remove");
                                    tree.setControl(false);
                              }else{
                                    System.out.println("The node was not found");
                              }
                        }
                  }
            }while((!command.equals("exit")) && (!command.equals("" +"")));
            System.out.println("The program was closed. Thank you for use this
            program");
      }

}
```

7.3 Teclado.java

```
import java.io.*;
public class Teclado {

	public static final byte  BYTE_ERR     =      Byte.MAX_VALUE;
	public static final short       SHORT_ERR    =      Short.MAX_VALUE;
	public static final int   INT_ERR      =      Integer.MAX_VALUE;
	public static final double DOUBLE_ERR  =      Double.MAX_VALUE;
	public static final float  FLOAT_ERR   =      Float.MAX_VALUE;
	public static final char        CHAR_ERR     =      Character.MAX_VALUE;
	public static final String      STRING_ERR   =      null;


	/**
	 *  Método que lee una línea de teclado y devuelve el <code><b>byte</b></code>
	 *  escrito por el usuario.
	 *  @return Devuelve el <code><b>byte</b></code> introducido por el usuario o
	 *  <code><b>Teclado.BYTE_ERR</b></code> si no se introdujo un byte.
	 */

	 public static byte readByte () {
		byte val=Byte.MAX_VALUE;
		try    {
			BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
			val = Byte.parseByte (in.readLine());
		} catch (IOException ioe) {
			System.out.println ("Imposible leer línea");
		} catch (NumberFormatException nfe) {
			System.out.println ("Valor introducido no byte");
		} catch (Exception e) {
			System.out.println ("Ocurrió una excepción");
		}
		return val;
	}

	/**
	 *  Método que lee una línea de teclado y devuelve el <code><b>short</b></code>
	 *  escrito por el usuario.
	 *  @return Devuelve el <code><b>short</b></code> introducido por el usuario o
	 *  <code><b>Teclado.SHORT_ERR</b></code> si no se introdujo un byte.
	 */
	public static short readShort () {
		short val=Short.MAX_VALUE;
		try    {
			BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
			val = Short.parseShort (in.readLine());
		} catch (IOException ioe) {
			System.out.println ("Imposible leer línea");
		} catch (NumberFormatException nfe) {
			System.out.println ("Valor introducido no short");
		} catch (Exception e) {
			System.out.println ("Ocurrió una excepción");
		}
		return val;
	}

	/**
	 *  Método que lee una línea de teclado y devuelve el <code><b>int</b></code>
```

```
     * escrito por el usuario.
     *  @return Devuelve el <code><b>int</b></code> introducido por el usuario o
    * <code><b>Teclado.INT_ERR</b></code> si no se introdujo un byte.
     */
    public static int readInt () {
          int val=Integer.MAX_VALUE;
          try    {
                BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
                val = Integer.parseInt (in.readLine());
          } catch (IOException ioe) {
                System.out.println ("Imposible leer línea");
          } catch (NumberFormatException nfe) {
                System.out.println ("Valor introducido no entero");
          } catch (Exception e) {
                System.out.println ("Ocurrió una excepción");
          }
          return val;
    }

    /**
     * Método que lee una línea de teclado y devuelve el <code><b>double</b></code>
     * escrito por el usuario.
     * @return Devuelve el <code><b>double</b></code> introducido por el usuario o
     * <code><b>Teclado.DOUBLE_ERR</b></code> si no se introdujo un byte.
     */
    public static double readDouble () {
          double val=Double.MAX_VALUE;
          try    {
                BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
                val = Double.parseDouble (in.readLine());
          } catch (IOException ioe) {
                System.out.println ("Imposible leer línea");
          } catch (NumberFormatException nfe) {
                System.out.println ("Valor introducido no double");
          } catch (Exception e) {
                System.out.println ("Ocurrió una excepción");
          }
          return val;
    }

    /**
     *  Método que lee una línea de teclado y devuelve el <code><b>float</b></code>
     *  escrito por el usuario.
     *  @return Devuelve el <code><b>float</b></code> introducido por el usuario o
     *  <code><b>Teclado.FLOAT_ERR</b></code> si no se introdujo un byte.
     */
    public static float readFloat () {
          float val=Float.MAX_VALUE;
          try    {
                BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
                val = Float.parseFloat (in.readLine());
          } catch (IOException ioe) {
                System.out.println ("Imposible leer línea");
          } catch (NumberFormatException nfe) {
                System.out.println ("Valor introducido no float");
          } catch (Exception e) {
                System.out.println ("Ocurrió una excepción");
          }
```

```
		return val;
	}

	/**
	 *  Método que lee una línea de teclado y devuelve el <code><b>char</b></code>
	 *  escrito por el usuario.
	 *  @return Devuelve el <code><b>char</b></code> introducido por el usuario o
	 *  <code><b>Teclado.CHAR_ERR</b></code> si no se introdujo un byte.
	 */
	public static char readChar () {
		char val = Character.MAX_VALUE;
		try	{
			BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
			val = (char) in.read();
		} catch (IOException ioe) {
			System.out.println ("Imposible leer caracter");
		} catch (Exception e) {
			System.out.println ("Ocurrió una excepción");
		}
		return val;
	}

	/**
	 *  Método que lee una línea de teclado y devuelve el <code><b>String</b></code>
	 *  escrito por el usuario.
	 *  @return Devuelve el <code><b>String</b></code> introducido por el usuario o
	 *  <code><b>Teclado.STRING_ERR</b></code> si no se introdujo un byte.
	 */
	public static String readString () {
		String val=null;
		try	{
			BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
			val = in.readLine();
		} catch (IOException ioe) {
			System.out.println ("Imposible leer línea");
		} catch (Exception e) {
			System.out.println ("Ocurrió una excepción");
		}
		return val;
	}

} // Class Teclado.java
```

7.4 Leaf.java

```
import java.util.ArrayList;
import java.util.Iterator;


public class Leaf {
	private String name;
	private Leaf parent;
	public ArrayList<Leaf> children=null;// array de hijos

	//constructor
	public Leaf(String name, Leaf parent){//un hijo tiene un nombre y un padre
		this.name = name;
		this.parent = parent;
		children = new ArrayList<Leaf>();
	}

	public String getName() {//se da el nombre
		return name;
	}

	public void setName(String name) {//se cambia el nombre
		this.name = name;
	}

	public Leaf getParent() {//se da el padre
		return parent;
	}

	public void setParent(Leaf parent) {//se cambia el nombre del padre
		this.parent = parent;
	}

	public ArrayList<String> getChildren2() {
		//se da la lista de los nombres de los hijos
		ArrayList<String> aux= new ArrayList<String>();
		if(!name.equals("RootNode"))aux.add((String)name);
		if(children==null){
			return aux;
		}else{
			Iterator pi = children.iterator();
			//se crea un iterador para poder ir recorriendo todos los elementos
			//del array
			while(pi.hasNext()){//mientras que ese nodo tenga hijos
				ArrayList<String> aux2=((Leaf)pi.next()).getChildren2();
				// recorro los hijos que tenga
				Iterator pi2 = aux2.iterator();//vuelvo a recorrer esa lista
				while(pi2.hasNext()){//paso al siguiente
					aux.add((String)pi2.next());
				}
			}
			return aux;
		}
	}

	public ArrayList<Leaf> getChildren() {//se da la lista de hijos
		return children;
	}

	public void setChildren(ArrayList<Leaf> children) {
```

```
		this.children = children;
	}

	public void insertChild(Leaf node) {//se inserta un hijo
		if (ContainsNode(node)==-1){
			//si este hijo no contiene el nodo que se le esta pasando se le
			//añade en la siguiente linea
			this.children.add(node);
		}
	}

	public int rowofChild(Leaf node){
		int row = 0;
		while (node.name!= "RootNode"){
			row +=1;
			node = node.parent;
		}
		return row;
	}

	public int ContainsNode(Leaf node){
		for(int i=0; i<this.children.size(); i++){
			//children.size() = da la cantidad de hijos
			if(this.children.get(i).getName().equals(node.getName())){
				//se compara si el nombre de ese hijo es igual al nombre del
				//nodo que te pasa
				return i;
			}
		}
		return -1;
	}
}
```

7.5 Tree.java

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import javax.swing.JOptionPane;


public class Tree {

	private boolean control = false;
	// se escribe la estructura arbol en el archivo out.txt
	private StringBuffer textFile = new StringBuffer();
	//escribe la estructura arbol en el archivo readText.txt
	private StringBuffer firstFile = new StringBuffer();
	boolean yaexiste=false;

	public Tree(){}

	public boolean isControl() {
		return control;
	}

	public void setControl(boolean control) {
		this.control = control;
	}

	public StringBuffer getTextFile() {// se da el archivo out.txt
		return textFile;
	}

	public StringBuffer getFirstFile() {// se da el archivo read.txt
		return firstFile;
	}
	public boolean metodoyaexiste() {
		return yaexiste;
	}
	public void setmetodoyaexiste(boolean yaexiste) {
		this.yaexiste = yaexiste;
	}


	public Leaf addNewChild(String childName, String parentName, Leaf rootNode){
		for(int i=0; i<rootNode.getChildren().size(); i++){
			Leaf parentNode = rootNode.getChildren().get(i);

			for(int j= 0;j<parentNode.getChildren().size();j++){
				Leaf hijo=parentNode.getChildren().get(j);
				if(hijo.getName().equals(childName)){
					control=true;
					yaexiste=true;
					j=parentNode.getChildren().size();
					i=rootNode.getChildren().size();
				}
			}
```

```
                //se recorren todos los hijos de esa hoja y el nodo en el que este
                //en ese momento sera el padre
                if(parentNode.getName().equals(parentName)&& control ==false &&
                yaexiste==false){
                        // se comparan los nombres
                    if(!control){ //if(control=false){
                        System.out.println("Adding...");
                        parentNode.insertChild(new Leaf(childName,
                        parentNode));
                        // si es el nombre que el programa está buscando,
                        //inserta el nodo
                        control = false;
                    }
                }
                if (parentNode.getChildren().size() != 0){
                //si el nodo tiene hijos, el programa llama al metodo recursivo
                    addNewChild(childName, parentName, parentNode);
                }
        }
        return rootNode;
}

public void removeChild(String Node, Leaf rootNode){
        if (Node.equals("Root")){
        //este if borra la primera rama que cuelga de root si elijo borrar //root
                while(rootNode.getChildren().size()!=0){
                    int n=0;
                    rootNode.getChildren().remove(n);
                }
        }else{
                for(int i=0; i<rootNode.getChildren().size(); i++){
                    Leaf parentNode = rootNode.getChildren().get(i);
                    //se recorren uno a uno los nodos del arbol
                    if(parentNode.getName().equals(Node)){
                        // se comparan los nombres
                        rootNode.getChildren().remove(i);
                        // si es el nombre que el programa está buscando,
                        // elimina el nodo
                        control = true;
                        // se encuentra un nodo con el mismo nombre que el
                        // usuario ha introducido
                    }
                removeChild(Node,parentNode);//llamada metodo recursivo
                }

        }
control=false;
}


public void printChildren(Leaf rootNode, Boolean command){
        String tab = "";
        for(int i=0;i<rootNode.getChildren().size(); i++){
                tab = "";
                for(int j=0; j<rootNode.rowofChild(rootNode); j++){
                //con este bucle se puede ver la profundidad que hay.
                //el programa añade una nueva tabulacion para cada nivel
                    tab += '\t';
                }
                String str = tab + rootNode.getChildren().get(i).getName();
```

```
                if (command){
                //true = actualiza el búfer que luego se utiliza para imprimir el
                //archivo out.txt
                textFile.append(str + " " + System.getProperty("line.separator"));
                //el programa añade las lineas al bufer que escribe el arbol
            }else{
                //false = el programa actualiza el bufer que mas tarde muestra el
                //arbol
                System.out.println(str);
            }
            if (rootNode.getChildren().get(i).getChildren().size() != 0){
                //si el nodo tiene hijos,el programa llama de nuevo al metodo
                //recursivo
                printChildren(rootNode.getChildren().get(i),command);
            }else{
                firstFile.append(setLineOfFirstFile(rootNode.getChildren().get(i))
                + System.getProperty("line.separator"));
                // el programa obtiene la línea para añadir en el búfer del archivo
                //readText.txt
            }
        }
}

public void saveTreeStructureToFile(Leaf rootNode, String filename, StringBuffer
strBuffer){
        try{
            // se crea el archivo
            FileWriter fstream = new FileWriter(filename);
            BufferedWriter out = new BufferedWriter(fstream);
            out.write(strBuffer.toString());
            out.close();
        }catch (Exception e){
            System.err.println("Error: " + e.getMessage());
        }
}


public String setLineOfFirstFile(Leaf node){
        //se obtiene la linea que se añade en el archivo readText.txt con los
        //predecesores del nodo
        String line = "";
        //se crea un objeto ArrayList
        ArrayList<String> lineList = new ArrayList<String>();
        while (node.getName()!= "RootNode"){
            //se continua mientras el bucle no obtenga todos los predecesores
            lineList.add(node.getName());
            node = node.getParent();
        }
        Collections.reverse(lineList);
        //el bucle obtiene los nombres de los nodos desde el ultimo hasta el primero.
        //Es necesario invertirlo.
        for(int i=0; i< lineList.size();i++){
            line += lineList.get(i) + ",";
        }
        return line.substring(0, line.length()-1);
        //se coge la linea sin la ultima coma
}

public Leaf sendRootNode(){
        File file = new File("readText.txt");
        BufferedReader reader = null;
```

```
        //se crea una nueva hoja, que sera la raiz
         Leaf root = new Leaf("RootNode", null);
         Leaf broot = root; //se hace una copia para ejecutar el arbol
         try {
               reader = new BufferedReader(new FileReader(file));
               String line = null;
               // se repite hasta que todas las lineas son leidas
               while ((line = reader.readLine()) != null) {
                       String []fields = line.split(",");
                       for(int i =0;i<fields.length;i++){
                              Leaf leaf = new Leaf(fields[i], broot);
                              //se crea una nueva hoja
                              int containsIndex = broot.ContainsNode(leaf);
                              //se comprueba si el padre contiene esta hoja
                              if (containsIndex != -1){
                              //si el padre contiene este hijo, a continuación se busca la
                              //fila de ese hijo, y temporalmente hace que sea la nueva
                              //raíz
                                     broot = broot.getChildren().get(containsIndex);
                                     //containsIndex = obtiene el indice de la hoja hijo
                              }else{
                                     broot.insertChild(leaf);
                                     //si el padre no contiene esta hoja, entonces se añade
                                     //esta hoja como hijo de este padre
                                     broot = leaf;
                                     //esta hoja es el nuevo padre
                              }
                       }
                       broot = root;
                       //para la nueva línea del archivo, se comienza desde el principio a
                       //hacer el mismo proceso
               }
          } catch (FileNotFoundException e) {
               System.out.println("El archivo no ha sido encontrado");
          } catch (IOException e) {
               System.out.println("Ha ocurrido un error");
          } finally {
               try {
                   if (reader != null) {
                       reader.close();
                   }
               } catch (IOException e) {
                   e.printStackTrace();
               }
          }
          return root;
        }
}
```

8. BIBLIOGRAFIA Y REFERENCIAS

1. “Introducción a la programación con orientación a objetos”. Camelia Muñoz Caro, Alfonso Niño Ramos y otros. Prentice Hall.
2. La Biblia de Java. Schildt, Herbert. Anaya
3. Java. Cómo programar. Dietel Harvey M., Deitel Paul J. Pearson Addison-Wesley.
4. Estructuras de datos en Java. John Lewis y Joseph Chase. Pearson Addison-Wesley.
5. Estructura de datos y algoritmos en Java. Adam Drozdek. Thomson.
6. An Introduction to Data Structures and Algorithms with Java, Rowe, Glen; Prentice.
7. Estructura de datos, algoritmos y programación orientada a objetos. Gregory L. Heileman. McGraw-Hill.
8. Diferentes webs:
    - 8.1. http://descuadrando.com
    - 8.2. http://jcsites.juniata.edu
    - 8.3. http://www2.uah.es/jcaceres/capsulas/DiagramaCasosDeUso.pdf
    - 8.4. http://www.jorgesanchez.net/programacion/