Sistemas Informáticos 2010/2011

Depuración algorítmica de vistas SQL

Facultad de Informática. Universidad Complutense de Madrid

Autores:

María Fernández Contreras
Laura Fernández Gómez
Aleixa Helguera Gordo

Director:

Rafael Caballero Roldán

Índice:

Índice de figuras:

Resumen del Proyecto

La falta de herramientas de depuración para lenguajes de acceso a bases de datos supone una limitación importante para el desarrollo de aplicaciones en términos de tiempo, obligando al programador a realizar la depuración mediante prueba-error. La principal razón de esta falta de herramientas es que los depuradores habituales utilizados en otros paradigmas de programación, no son válidos en el caso de los lenguajes de las bases de datos relacionales, debido al alto nivel de abstracción de los mismos.

La herramienta implementada para la consecución de este proyecto, trata de utilizar técnicas de depuración algorítmica para SQL, con el objetivo de detectar vistas erróneas en un sistema de múltiples vistas correlacionadas entre si.

El sistema parte de un conjunto de vistas, que tras la ejecución de la vista principal han devuelto un resultado inesperado y construye internamente un árbol representando la dependencia jerárquica entre las diferentes vistas. Cada nodo del árbol representa una vista o una tabla y la relación entre ellos se representa mediante las aristas que las unen. Este árbol se recorre con el fin de localizar la causa del error. El usuario dispone de varias estrategias para llevar a cabo la depuración de las vistas. Él mismo clasifica los nodos como válidos o no válidos, repitiendo el proceso hasta encontrar el nodo crítico responsable del resultado erróneo del que partimos.

Abstract

The lack of debugging tools for database access languages is an important limitation for the development or general systems in terms of time. This implies the programmer must perform the debugging process by trial and error. The main reason for this lack of tools is that the usual trace debuggers used in other paradigms are not available here due to the high abstraction level of the language.

The implemented tool proposed in this project, uses algorithmic debugging techniques for SQL, with the goal of detecting erroneous views in a multiply correlated view system.

Therefore, the system starts with a set of views, such that the main view has returned an unexpected result. Then the debugger internally builds a tree that represents the hierarchical dependency between the different views. Each tree node represents a view or a table and the relation among the nodes is represented by the edges which join them. The system runs through the tree with the goal of locating the cause of the error. The user has several strategies to carry out the debugging of views. The nodes can be classified as valid or invalid, repeating the process until the critical node that is the main cause of the initial error is found.

Autorización

Los ponentes María Fernández Contreras, con DNI 03911386-Y, Laura Fernández Gómez, con DNI 50875117-Z y Aleixa Helguera Gordo, con DNI 47521024-B autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos tanto la propia memoria, como el código, la documentación y el prototipo desarrollado.

María Fernández Contreras

Laura Fernández Gómez

Aleixa Helguera Gordo

En Madrid, a 21 de mayo de 2011.

1. Introducción

En la primera parte de este apartado explicaremos de forma algo más detallada los problemas existentes con la depuración de lenguajes de acceso a bases de datos. La segunda parte habla del llamado *Estado del Arte*, y éste a su vez contendrá otras dos subsecciones, la depuración declarativa y la depuración en SQL. Como última parte de este apartado de Introducción, encontraremos el objetivo de este proyecto, el depurador.

1.1. Motivación

Como se ha explicado anteriormente, existen serias limitaciones a la hora de depurar lenguajes de acceso a bases de datos. Estas limitaciones no tienen mucha importancia en pequeños y sencillos programas, ya que el usuario tiene bastantes posibilidades de encontrar fácilmente el error. El problema viene cuando se trabaja con un gran conjunto de datos y vistas, donde esta tarea se vuelve bastante más complicada.

Nuestro proyecto ha sido creado para tratar la depuración del lenguaje SQL (Structured Query Language), dado que en la actualidad es el estándar de la inmensa mayoría de los SGBD comerciales. Además, es sencillo de usar y fácil de entender, y es ampliamente aceptado por multitud de servidores y programas. Sin embargo, la falta de herramientas auxiliares tales como depuradores o entornos gráficos ha sido señalada como una posible limitación.

Los depuradores que se encuentran disponibles usualmente en sistemas SQL ofrecen muchas, pero no todas las características de depuración habituales. La depuración de SQL admite la mayoría de los comandos de depuración, como establecer puntos de interrupción y ejecutar instrucciones paso a paso. Sin embargo, SQL es un lenguaje declarativo, lo cual quiere decir que especifica qué es lo que se quiere conseguir, sin explicar cómo se ha conseguido. Las sentencias que se utilizan describen el problema que se quiere solucionar, pero no las instrucciones necesarias para solucionarlo. Esto último se realizará mediante mecanismos internos de inferencia de información a partir de la descripción realizada. Debido a esto, no tiene sentido aplicar los métodos convencionales de depuración, como por ejemplo la ejecución de instrucciones paso a paso, dado que cualquier sentencia de este lenguaje no establece explícitamente un orden de ejecución, con lo que dichos métodos de depuración se convertirían en una tarea muy compleja de resolver.

1.2 Estado del arte

La depuración declarativa surge por la necesidad de depurar lenguajes en los cuales no es posible la depuración de traza o paso a paso, como son el paradigma lógico, funcional y lenguajes de acceso a bases de datos. Surge en el contexto de la programación lógica [*E.Y.Shapiro.1982*, *G. Ferrand. 1987*, *J.W. Lloyd 1987*] y adaptada después a la programación funcional [*H. Nilsson and J. Sparud. 1997, H. Nilsson, B. Pope and L. Naish, 2003*] y la programación lógica con restricciones *[A. Tessier and G. Ferrand.1870, 2000*]. Lee Naish indica en [*L. Naish, 1997*-3] que esta técnica puede aplicarse a cualquier paradigma.

La idea fundamental de la depuración declarativa es partir de un cómputo erróneo, llamado síntoma inicial, para posteriormente generar un árbol de cómputo adecuado para dicho síntoma inicial. Finalmente se recorre dicho árbol con la ayuda de un agente externo, normalmente el usuario, al que se le realizan preguntas acerca de la completitud y corrección del resultado tras la ejecución del código contenido en los nodos del árbol. El resultado obtenido en la raíz del árbol está determinado por los resultados obtenidos en la ejecución de los nodos hijos.

Por tanto, cada nodo del árbol debe contener:

- El resultado de algún subcómputo realizado durante el cómputo principal.
- El fragmento de código del programa depurado, responsable de dicho subcómputo.

Los nodos hijos se corresponden con los cómputos auxiliares que han sido necesarios para llegar al resultado final almacenado en el nodo padre, conteniendo cada uno de ellos su resultado y su fragmento de código correspondientes.

Nótese sin embargo, que un nodo puede ser no válido a pesar de que su fragmento de código no sea erróneo, debido a que su resultado puede haberse obtenido a partir de resultados ya incorrectos. Por eso, el depurador sólo señala como causa del error aquellos fragmentos de código correspondientes a nodos erróneos sin ningún hijo erróneo. Si el agente externo identifica un nodo con un resultado asociado incorrecto, siendo correcto el resultado de sus hijos, tendremos que el código asociado a dicho nodo ha producido un resultado erróneo a partir de resultados parciales que son correctos, y por tanto éste será la causa de error señalada por el depurador. A estos nodos se les denomina nodos críticos.

Como se observa en [*Rafael Caballero, 2004*] el teorema de completitud débil del esquema general enuncia que todo árbol de cómputo con raíz errónea tiene al menos un nodo crítico. La demostración de este resultado es sencilla utilizando inducción sobre la profundidad del árbol: en el caso base tenemos un solo nodo, la raíz, que es un nodo crítico. En el caso inductivo examinamos de nuevo la raíz y si no tiene hijos

cuelga de dicho nodo y por hipótesis de inducción sabemos que contiene un nodo crítico.

Aunque el teorema sólo nos asegura la existencia de un nodo crítico, una vez corregida la causa del error, se puede repetir el proceso de forma reiterada. Si en el árbol de cómputo dejan de existir nodos críticos, ya no tendremos síntomas de error para el cómputo, lo que por supuesto no garantiza la corrección del programa. Diferentes tipos de errores requerirán diferentes tipos de árboles y también diferentes nociones del concepto de nodo erróneo.

En el caso de la programación lógica se consideran dos tipos de errores:

- Respuestas perdidas: En el conjunto de todas las respuestas obtenidas para un objetivo dado falta alguna respuesta esperada.
- Respuestas incorrectas: Se obtiene una respuesta inesperada para un objetivo determinado.

En programación funcional, sin embargo, sólo se consideran las respuestas incorrectas.

Cada tipo de árbol, como decíamos, constituye una instancia del esquema general, pero en todas ellas serán aplicables las mismas ideas (nodo erróneo, etc.) así como el teorema de completitud débil enunciado anteriormente. En cambio, los resultados de corrección dependen de la instancia en particular y deben probarse para cada caso concreto.

Los depuradores actuales llevan esta diferencia conceptual al nivel práctico mediante la distinción de dos fases en el proceso de depuración:

- La fase de generación del árbol de cómputo.
- El recorrido o navegación de dicho árbol.

En el ámbito de la navegación se han propuesto distintas estrategias:

- Top Down.
- Divide & Query.

Al comienzo de la ejecución la estrategia seleccionada es la manual, es decir, "*No strategy*". Es el usuario el que va modificando los estados de los nodos a su elección, por medio de los botones de radio, pudiendo seguir o no un orden lógico.

Cuando se selecciona una estrategia se crea su tabla correspondiente. Si no hay ningún nodo seleccionado, la depuración se inicia en la cima del árbol.

Si se selecciona Top Down, la estrategia consiste en el recorrido de los nodos hijos del árbol siguiendo un orden concreto, de arriba a abajo. Se muestra en la tabla de resultados el contenido de los hijos del nodo que esté seleccionado en ese momento y el usuario, después de hacer las comprobaciones pertinentes, marca los estados de los hijos. Cada vez que se cambia de nodo se reconstruye la tabla borrando todas las filas e insertándolas de nuevo con el nombre de cada hijo y el estado seleccionado correspondiente. Si todos los hijos son válidos, el nodo que estaba seleccionado es el nodo objeto de la búsqueda, el nodo crítico o "*buggy*" En cambio, si existe un hijo no válido, se mostrarán los hijos de ese nuevo nodo seleccionado, y se repetirá el mismo proceso hasta que se encuentre un nodo crítico en alguno de los hijos. Si existieran varios hijos no válidos, se muestran todos al usuario y éste es quien elige por cuál quiere continuar.

Si se selecciona "*Divide & Query*" la estrategia a seguir consiste en que a partir del nodo seleccionado en ese momento, el sistema calcula el nodo *mitad* del árbol. El nodo mitad es el nodo que cumple la siguiente propiedad: la diferencia en valor absoluto entre el número de nodos exteriores a él (fuera de su subárbol) y el número de nodos dentro del subárbol que cuelga de él, incluyendo a él mismo, es la más aproximada a 0. Para ello hay que calcular esta diferencia para cada nodo del árbol, y el nodo que será tratado es el que más aproxima esta resta a cero. Una vez hecho esto, el usuario marca el estado que le corresponda.

- Si el estado es válido, el sistema borrará el subárbol que cuelga de él, puesto que no va a ser posible que esta operación elimine todos los nodos críticos; es fácil comprobar que existe un nodo crítico conectado con la raíz por un camino de nodos no válidos; por tanto este nodo crítico no está en el subárbol eliminado.
- Si el estado que se ha marcado es no válido, se repiten los cálculos anteriores pero partiendo de este nuevo nodo seleccionado; tenemos un nuevo de árbol de cómputo con raíz no válida.

1.2.1 Depuración de SQL

Una línea de investigación diferente basada en la depuración declarativa sería la aplicación de esta técnica a otros paradigmas, en particular para la depuración de consultas dentro del lenguaje de consulta de bases de datos relacionales SQL.

Hasta ahora no se ha aplicado la depuración declarativa a SQL. Existen depuradores de traza para PL/SQL, pero no para vistas SQL.

El resultado de una consulta SQL es un conjunto de tuplas con los datos de la base de datos que satisfacen las condiciones impuestas en la consulta. Son varias las características de este lenguaje que llevan a pensar en la adecuación de la depuración declarativa como método de depuración para éste:

- Se trata de un lenguaje declarativo, basado en un cálculo lógico (Álgebra Relacional Extendida).

- La depuración de una sentencia SQL no puede hacerse por mecanismo de traza, ya que cada sentencia se descompone internamente como la composición de muchas operaciones.

- Se puede hablar de respuestas erróneas, en el caso de que la repuesta de la consulta contenga un tupla inesperada, y de respuestas perdidas, cuando falta una tupla en las respuesta final.

1.3. Objetivos

El objetivo de este proyecto es implementar un depurador declarativo para un conjunto de vistas SQL dependientes.

Al ejecutar el programa se deberá seleccionar un archivo fuente .sql que contenga el conjunto de vistas que se desean depurar. Se sobreentiende que la vista principal es por defecto la última en el orden textual del archivo. A partir de éste, se generará un árbol de vistas de manera que una vista tendrá como hijos las vistas o tablas de las que depende (las que se encuentran en las secciones *from*, *where* y *having* de la vista). Sabemos que el resultado incorrecto podría ser debido a un error en los datos de las tablas o al código de las vistas. Si el error está en los datos introducidos en las tablas la solución es clara. El problema está cuando el error reside en el código de las vistas. Para ello podremos, a partir del árbol generado, ir navegando a través de las distintas vistas, los nodos de nuestro árbol, y decidir gracias a las distintas estrategias de depuración qué vistas son correctas y cuáles no. Para ello se dispone de una interfaz de usuario que nos ayudará en esta tarea:

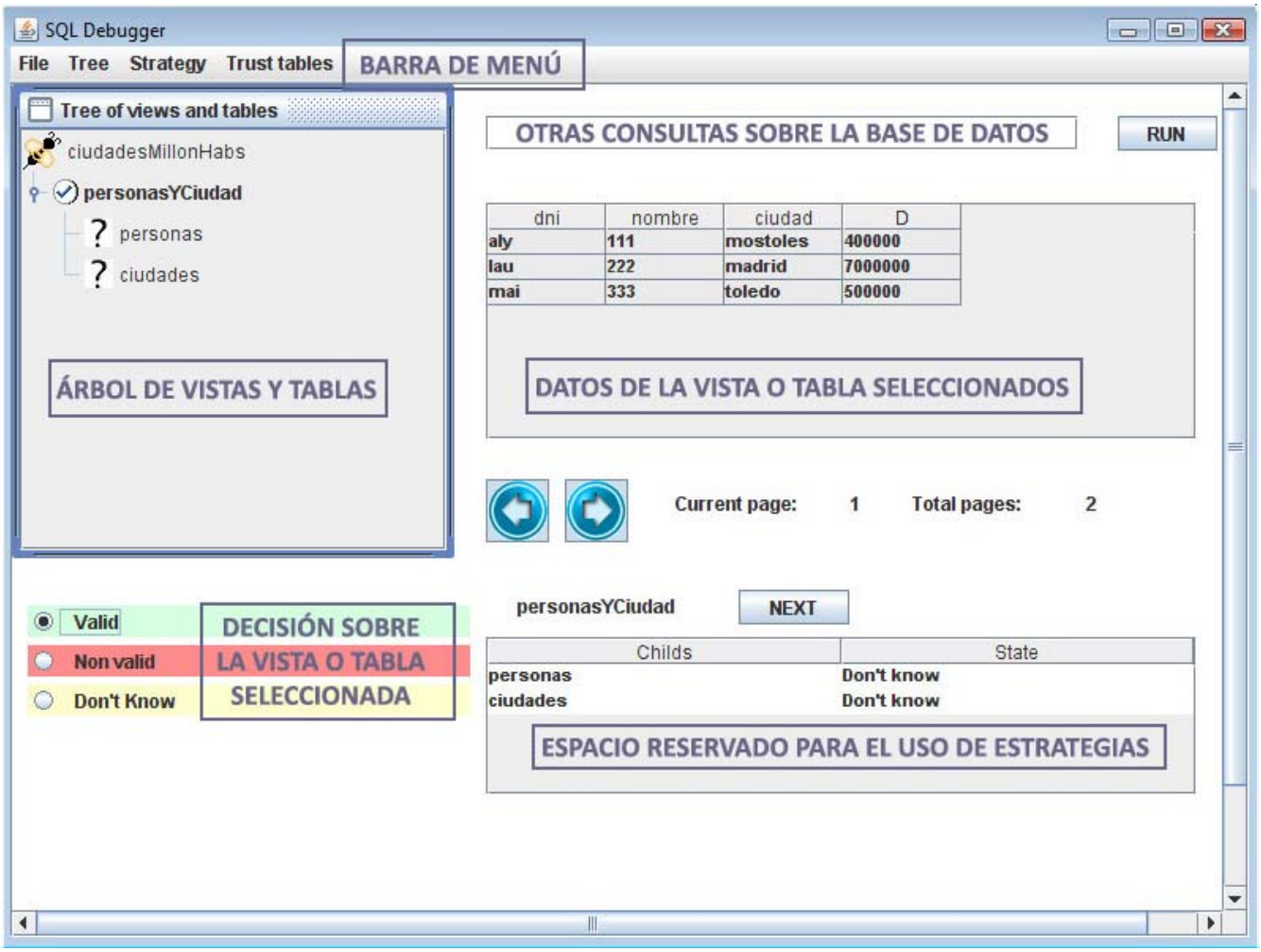

Figura 1 Interfaz gráfica

- **Barra de Menú:** Dentro del menú "*File*" podremos escoger el archivo fuente .sql que contiene el código de las vistas. Una vez cargado el archivo, podemos introducir una especificación fiable, es decir, seleccionar otro archivo .sql en el submenú "*Open trusted..."* en el que todas las vistas son válidas. Este archivo nos servirá para comparar la definición de vistas coincidentes en ambos archivos, pudiendo así marcar el nodo correspondiente como válido en nuestro árbol.
- **Árbol de vistas y tablas:** Muestra el árbol generado a partir del fichero fuente que hemos seleccionado.
- **Decisión sobre la vista o tabla seleccionada:** Se usa para decidir el estado del nodo seleccionado, *Valid* (resultado *Válido*) si es correcto el resultado, *Non valid* (resultado *No válido)* si no lo es o *Don't Know* (resultado *Desconocido*) si no sabemos si es correcto o no.
- **Otras consultas sobre la base de datos:** Esta opción nos permite realizar una consulta cualquiera sobre nuestra base de datos.
- **Datos de la vista o tabla seleccionados:** Muestra los datos de la tabla o vista que tenemos seleccionada.
- **Espacio reservado para el uso de estrategias:** Aquí se mostrarán las preguntas correspondientes a la estrategia escogida (*Top Down* o *Divide & Query*). Una tabla con los hijos del nodo actual en el caso de la estrategia *Top Down*, y unos botones de radio para elegir si la vista o tabla elegida por la estrategia tiene asociada un resultado válido o no válido en el caso de la estrategia *Divide & Query*.

2. Especificación

Este apartado estará dividido en dos partes: la primera hablará de nuestra idea original para la creación de este proyecto, y la segunda explicará los cambios o adaptaciones realizados durante el desarrollo hasta obtener el proyecto final.

Propuesta de proyecto inicial:

La propuesta inicial de este proyecto es ofrecer al usuario un depurador para facilitar la detección de vistas erróneas en programas escritos en lenguaje SQL. Las funcionalidades del proyecto son las siguientes:

- El usuario debe poder escoger el fichero cuyo contenido es un conjunto de vistas relacionadas entre sí.
- También deseamos poder incorporar archivos con especificaciones fiables, que ayuden a reducir el número de preguntas necesarias para detectar el error.
- Se debe poder asignar distintos estados a los nodos del árbol, como pueden ser Válido, No válido o Desconocido, sin seguir ninguna estrategia o seleccionando una de las dos estrategias siguientes: *Top Down* o *Divide & Query.*
- Dar la posibilidad de guardar un árbol con sus estados correspondientes para proseguir con la depuración en otro momento.
- Poder marcar todas las tablas con estado Válido (*Trusted tables*) a priori.

Aparte de las funcionalidades, el proyecto ha pasado por la elección de distintas decisiones como la utilización de distintos sistemas gestores de bases de datos, la conexión entre dicho gestor y la aplicación, la forma de mostrar el árbol, las opciones relacionadas con él y los resultados de las consultas, y algunos rasgos más en los que se profundizará a continuación.

- Como sistemas gestores de bases de datos pensamos en los tres a priori más usados actualmente, como son MySQL, Access y Oracle. **MySQL** es un sistema de gestión de base de datos relacional muy rápido en la lectura, multihilo y multiusuario con más de seis millones de instalaciones. **Microsoft Access** es un programa, utilizado en los sistemas operativos Microsoft Windows, para la gestión de bases de datos creado y modificado por Microsoft y orientado a ser usado en entornos personales o en pequeñas organizaciones. Es un componente de la suite Microsoft Office. Permite crear ficheros de bases de datos relacionales que pueden ser fácilmente gestionadas por una interfaz gráfica sencilla. **Oracle** es un sistema de gestión de base de datos objeto-relacional (o ORDBMS por el acrónimo en inglés de *Object-Relational Data Base Management System*), desarrollado por Oracle Corporation. Se considera a Oracle como uno de los sistemas de bases de datos más completos, destacando:
  - Soporte de transacciones
  - Estabilidad
  - Escalabilidad
  - Soporte multiplataforma

- El protocolo que Eclipse nos ofrecía y por tanto usamos para utilizar bases de datos relacionales desde Java fue JDBC (*Java Database Connectivity*). JDBC es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice.

- Para mostrar el árbol, optamos por la forma que se ve en la imagen siguiente:

Figura 2 Ejemplo de árbol de vistas

Esta visión del árbol, basada en la adaptación del componente *JTree* ofrecido en Java, nos resulta fácil de usar y sencilla de entender, pudiendo desplegar u ocultar los hijos de cada nodo con total facilidad.

- Para guardar el árbol, pensamos en la herramienta *Hibernate*, que realiza el *mapping* entre el mundo orientado a objetos de las aplicaciones y el mundo entidad-relación de las bases de datos en entornos Java. El término utilizado es ORM (**object/relational mapping**) y consiste en la técnica de realizar la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa. Hibernate no sólo realiza esta transformación, sino que nos proporciona capacidades para la obtención y almacenamiento de datos de la base de datos que nos reducen el tiempo de desarrollo.

- Hemos usado la herramienta *JSQLParser*, que analiza una sentencia SQL y lo traduce en una jerarquía de clases de Java. Tuvimos que añadir ciertas modificaciones para que el analizador sintáctico aceptara vistas con distintas opciones, como los

operadores *minus* o *intersection*, entre otras ampliaciones, pues el sistema de partida está implementado sólo para el reconocimiento y análisis de tablas, en el cual nos basamos para llevar a cabo la ampliación del JSQLParser para vistas. Los detalles de la implementación del analizador léxico y sintáctico se detallan en la sección 4.1.3 [JSQLParser].

Características del proyecto final:

La propuesta final del proyecto mantiene las funcionalidades principales propuestas inicialmente.

En cuanto al uso de varios sistemas gestores de bases de datos, hemos optado por implementar sólo el caso de MySQL, por ser éste el SGBD más usado en la actualidad, y porque su continuo desarrollo y su creciente popularidad está haciendo de MySQL un competidor cada vez más directo de gigantes en la materia de las bases de datos como Oracle. También es muy destacable su sencillez, su rapidez y la condición de *open source* de MySQL, que hace que su utilización sea gratuita e incluso se pueda modificar con total libertad, pudiendo descargar su código fuente. Esto ha favorecido muy positivamente en su desarrollo y continuas actualizaciones, para hacer de MySQL una de las herramientas más utilizadas por los programadores orientados a Internet.

Por otra parte, decidimos dejar de lado la opción de Hibernate, ya que nos dio varios problemas al intentar guardar los datos relacionados con el árbol. En su lugar, optamos por guardar la estructura de los árboles en archivos XML (e***X****tensible* ***M****arkup* ***L****anguage.* La tecnología XML busca dar solución al problema de expresar información estructurada de la manera más abstracta y reutilizable posible. Que la información sea estructurada quiere decir que se compone de partes bien definidas, y que esas partes se componen a su vez de otras partes. Entonces se tiene un árbol de trozos de información. Un ejemplo de un árbol guardado como XML es el siguiente:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <java version="1.6.0_24" class="java.beans.XMLDecoder">
 - <object class="parser.Node">
  - <void property="codigoSql">
      <string>CREATE VIEW guest(ID,name)AS (SELECT O.Id, O.name FROM Owner AS O, NoCommonName AS N, LessThan6
        AS L WHERE O.ID = N.Id AND N.ID = L.ID) Lista_Relaciones_From: [Owner, NoCommonName, LessThan6]
        Lista_Expresiones_Select: [O.Id, O.name] Tipo:BASICO HASN´T GROUP BY HAS WHERE HASN´T HAVING</string>
    </void>
  - <void property="hijos">
    - <void method="add">
      - <object class="parser.Node">
        - <void property="estado">
          - <object id="Estado0" class="Vista.Estado" method="valueOf">
              <string>Valid</string>
            </object>
          </void>
        - <void property="nombre">
            <string>Owner</string>
          </void>
        </object>
      </void>
    + <void method="add">
    + <void method="add">
    </void>
  + <void property="nombre">
   </object>
</java>
```

Figura 3 Ejemplo de árbol guardado en XML

2.1. Casos de uso

Estos son los casos de uso para nuestro proyecto:

- **Cargar script de vistas SQL:** Escoger un fichero fuente .sql y cargarlo para generar un árbol de vistas.
- **Cargar especificación fiable:** Una vez cargado el árbol, escoger un fichero de vistas .sql, que compararemos con las del árbol cargado, marcando como Válidas, aquellas que sean coincidentes en ambos ficheros.
- **Cargar árbol:** Cargar un árbol que previamente habíamos guardado.
- **Salvar árbol:** Grabar en disco el árbol que llevamos evaluado.
- **Desplegar un nodo del árbol:** Desplegar o contraer los hijos de un nodo concreto.
- **Mostrar tabla:** Mostrar los datos de una tabla o vista de la base de datos.
- **Ejecutar consulta:** Ejecutar una consulta cualquiera sobre la base de datos y mostrar su resultado.
- **Depuración sin estrategia:** Búsqueda de los errores en las vistas sin utilizar ninguna estrategia, es decir, de manera manual.
- **Estrategia Top Down:** Búsqueda de los errores en las vistas utilizando la estrategia Top Down.
- **Estrategia Divide & Query:** Búsqueda de los errores en la vista utilizando la estrategia Divide & Query.
- **Trust tables:** Confiar en que las tablas son correctas.

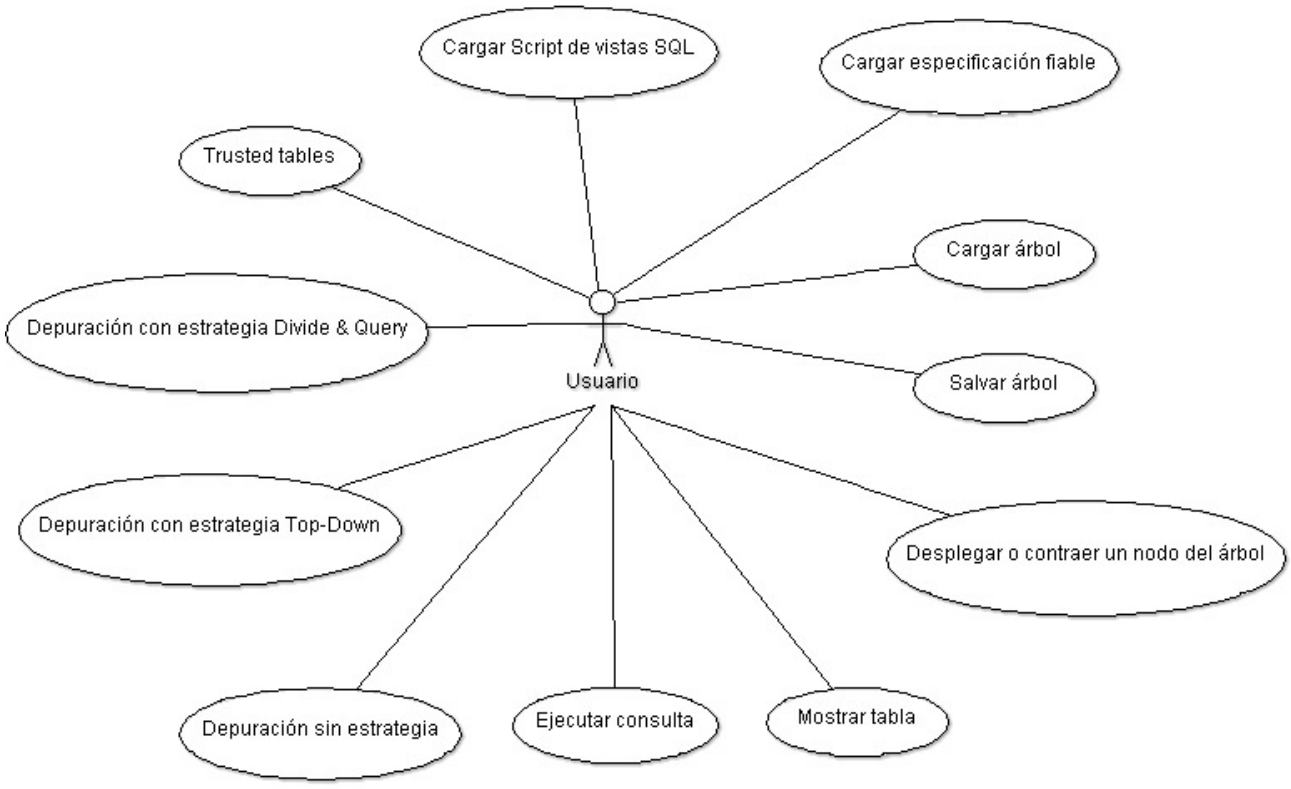

Figura 4 Diagrama de casos de uso

2.2. Diagrama de secuencias y actividades

2.2.1. Sin estrategia:

DIAGRAMA DE SECUENCIAS

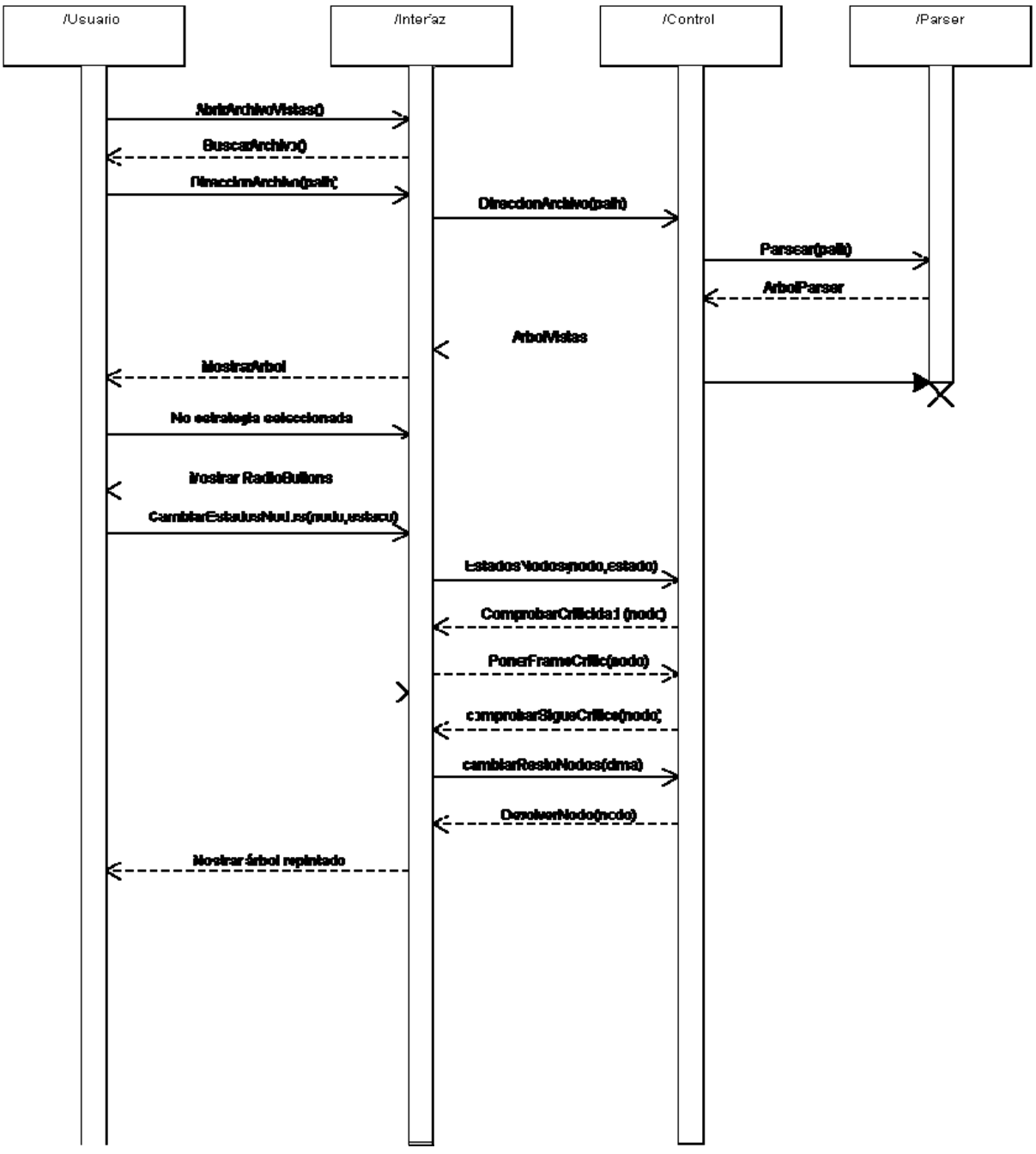

Figura 5 Diagrama de secuencias de búsqueda sin estrategia

DIAGRAMA DE ACTIVIDADES

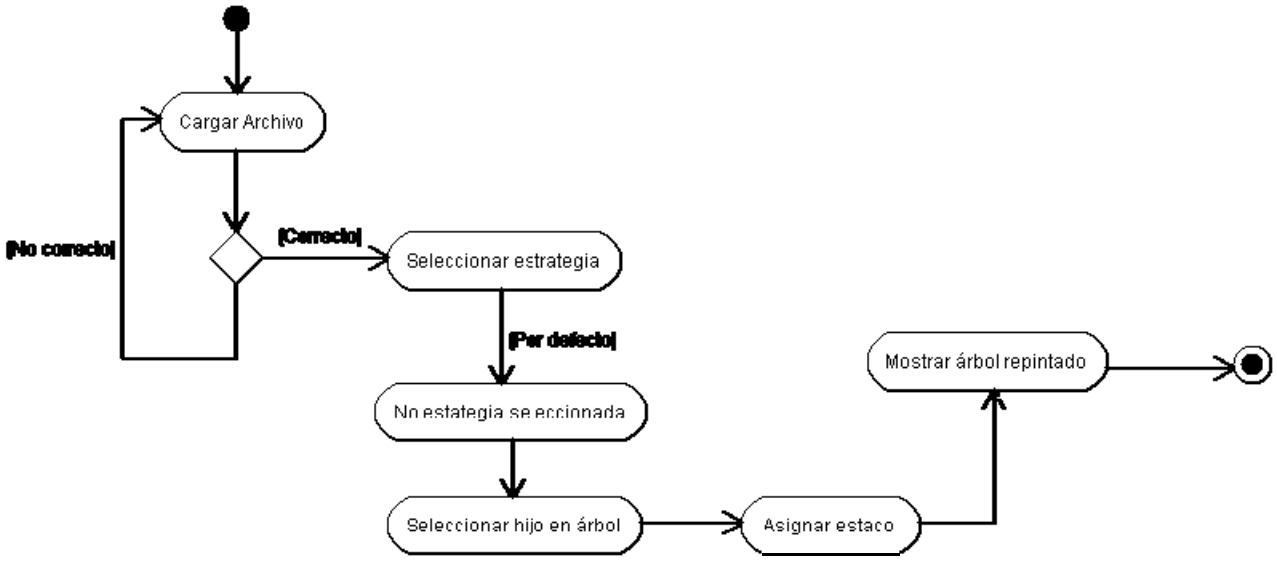

Figura 6 Diagrama de actividades de búsqueda sin estrategia

2.2.2. Top Down:

DIAGRAMA DE SECUENCIAS

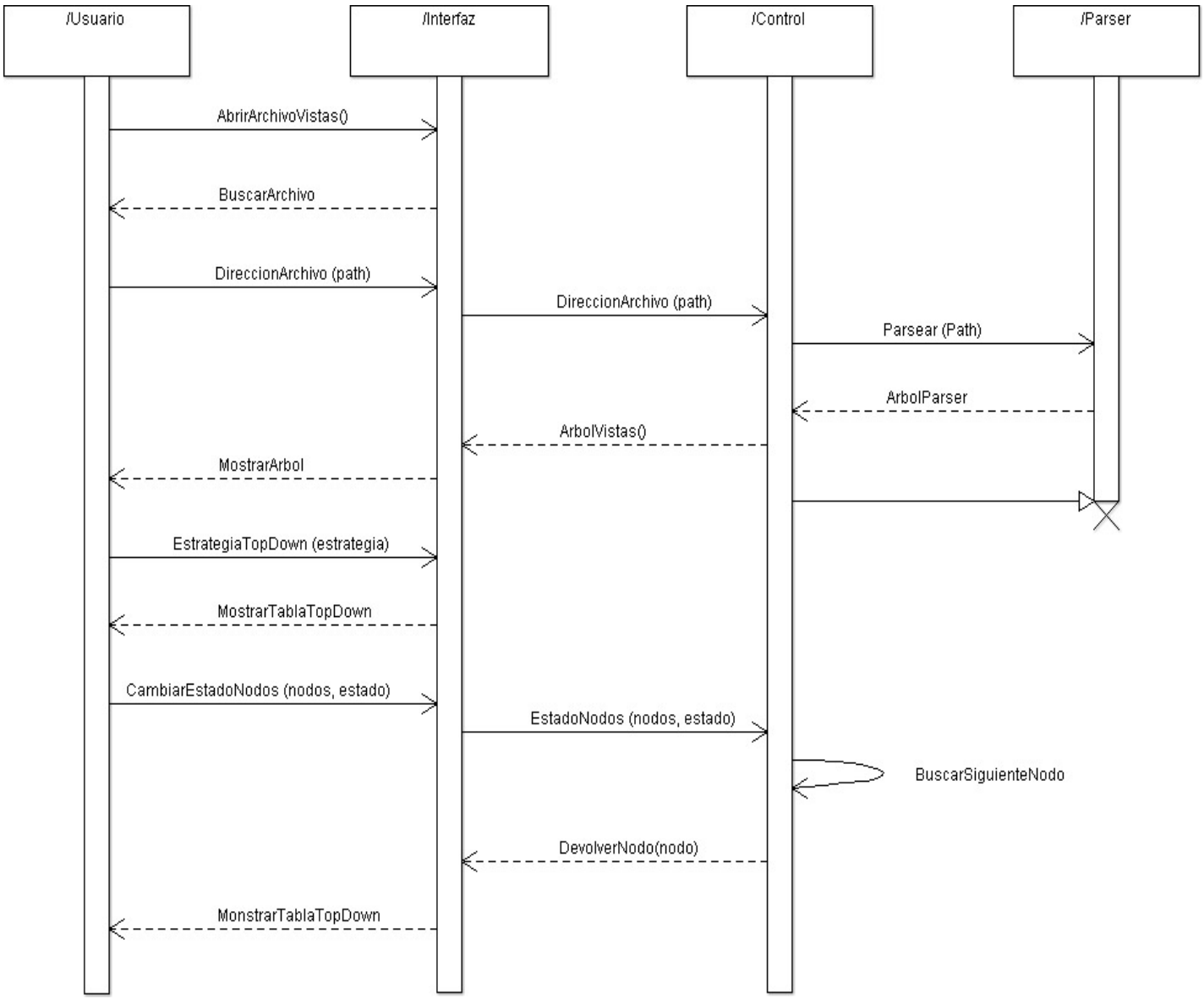

Figura 7 Diagrama de secuencias de estrategia Top Down

DIAGRAMA DE ACTIVIDADES

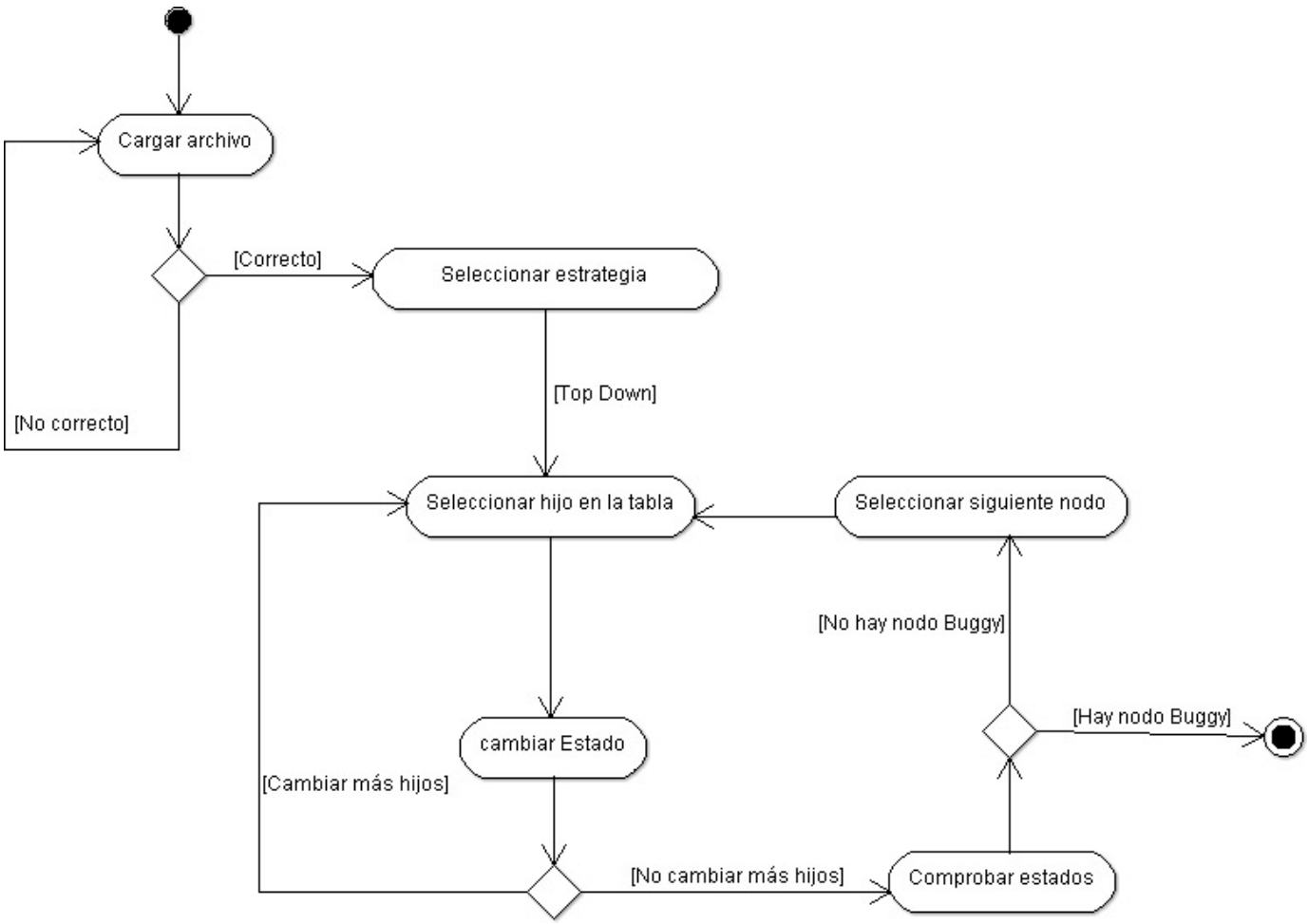

Figura 8 Diagrama de actividades de estrategia Top Down

2.2.3. Divide & Query:

DIAGRAMA DE SECUENCIAS

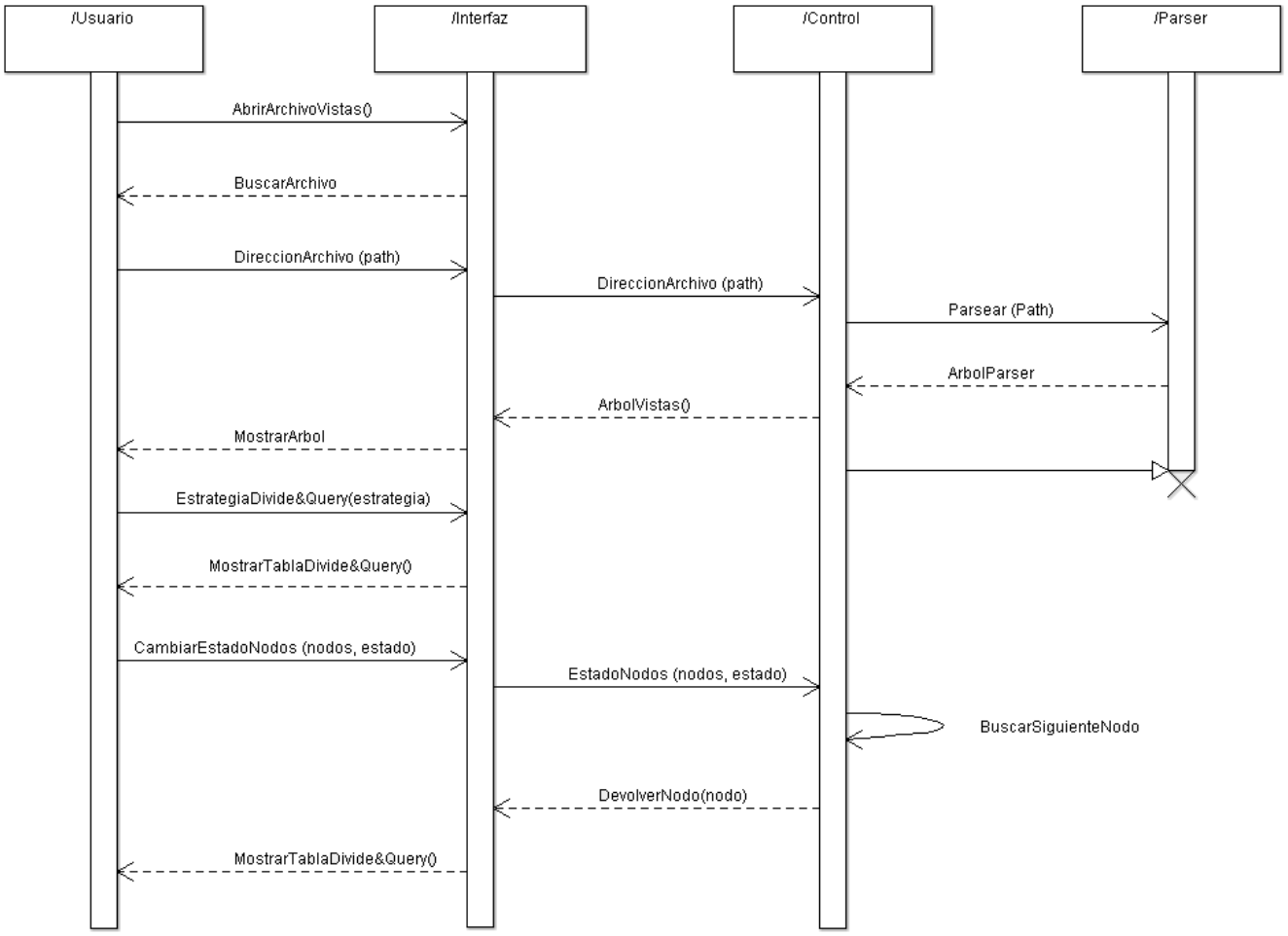

Figura 9 Diagrama de secuencias de estrategia Divide&Query

DIAGRAMA DE ACTIVIDADES

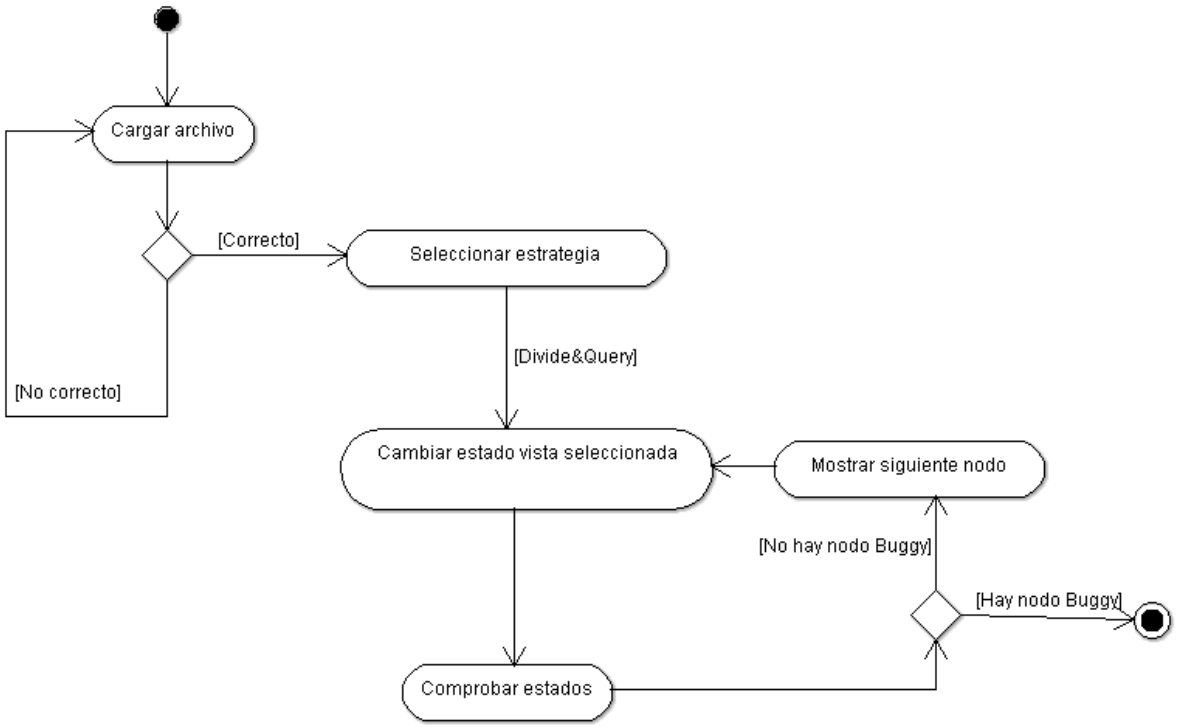

Figura 10 Diagrama de actividades de estrategia Divide & Query

2.2.4. Especificación fiable:

DIAGRAMA DE SECUENCIAS

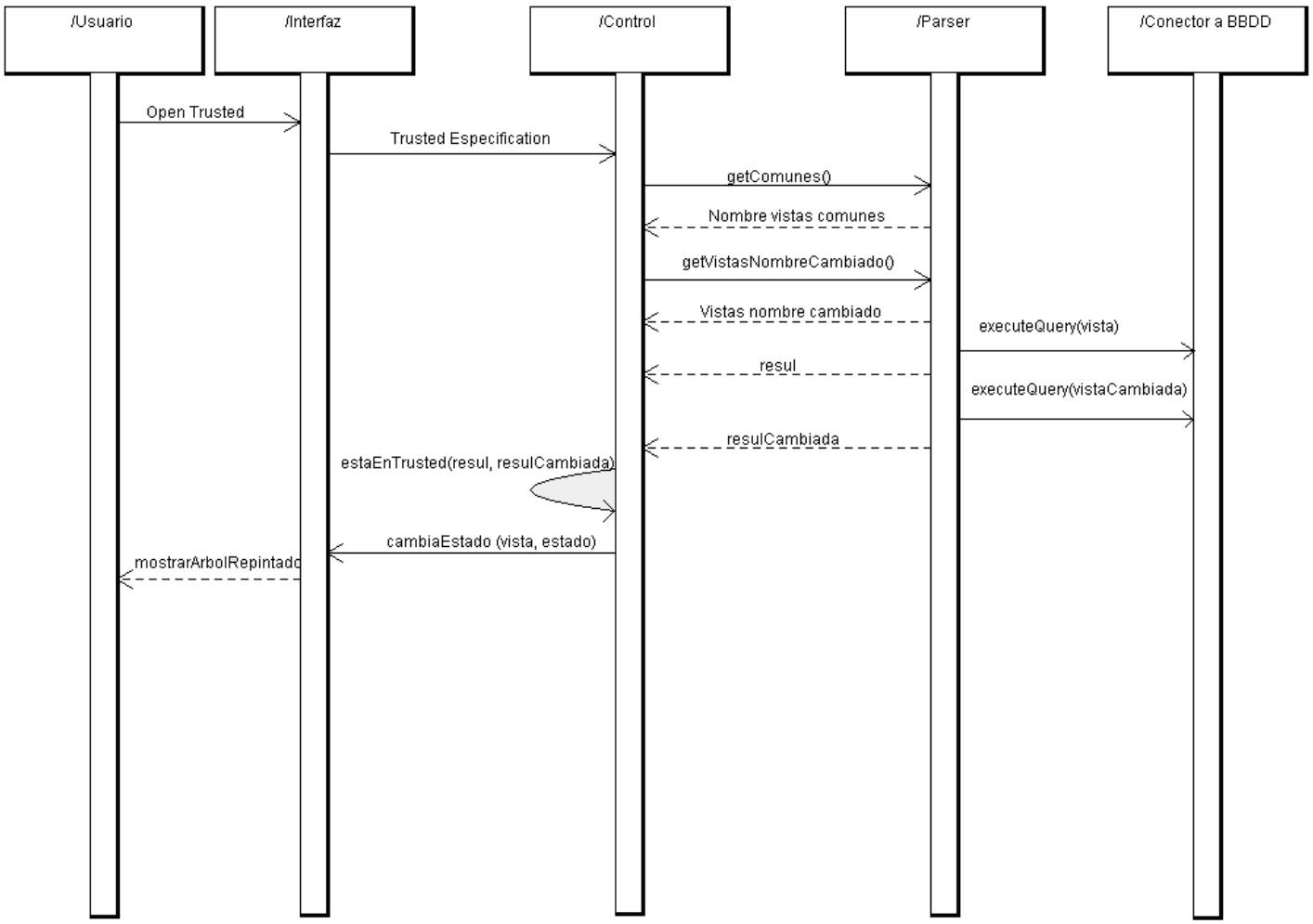

Figura 11 Diagrama de secuencias de abrir especificación fiable

DIAGRAMA DE ACTIVIDADES

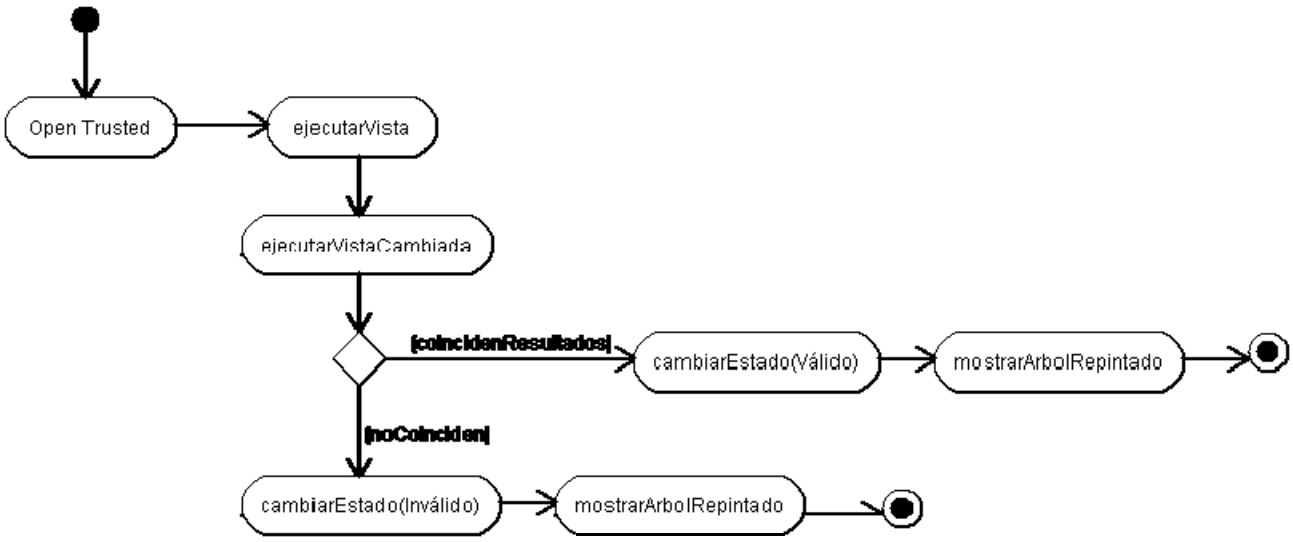

Figura 12 Diagrama de actividades de estrategia especificación fiable

2.2.5. Trusted tables:

DIAGRAMA DE SECUENCIAS

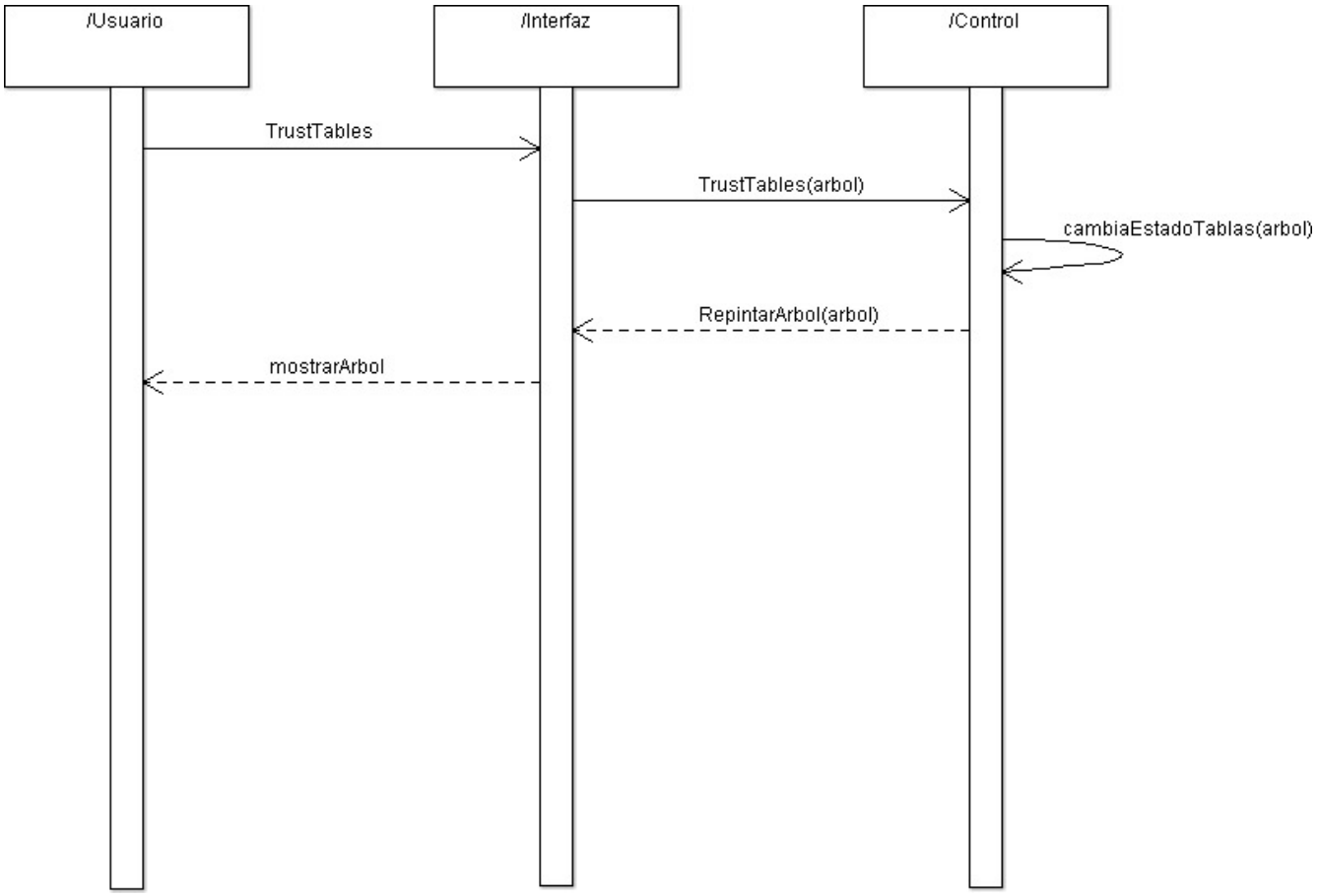

Figura 13 Diagrama de secuencias de "Trust tables"

2.2.6. Consulta a Base de Datos:

DIAGRAMA DE SECUENCIAS

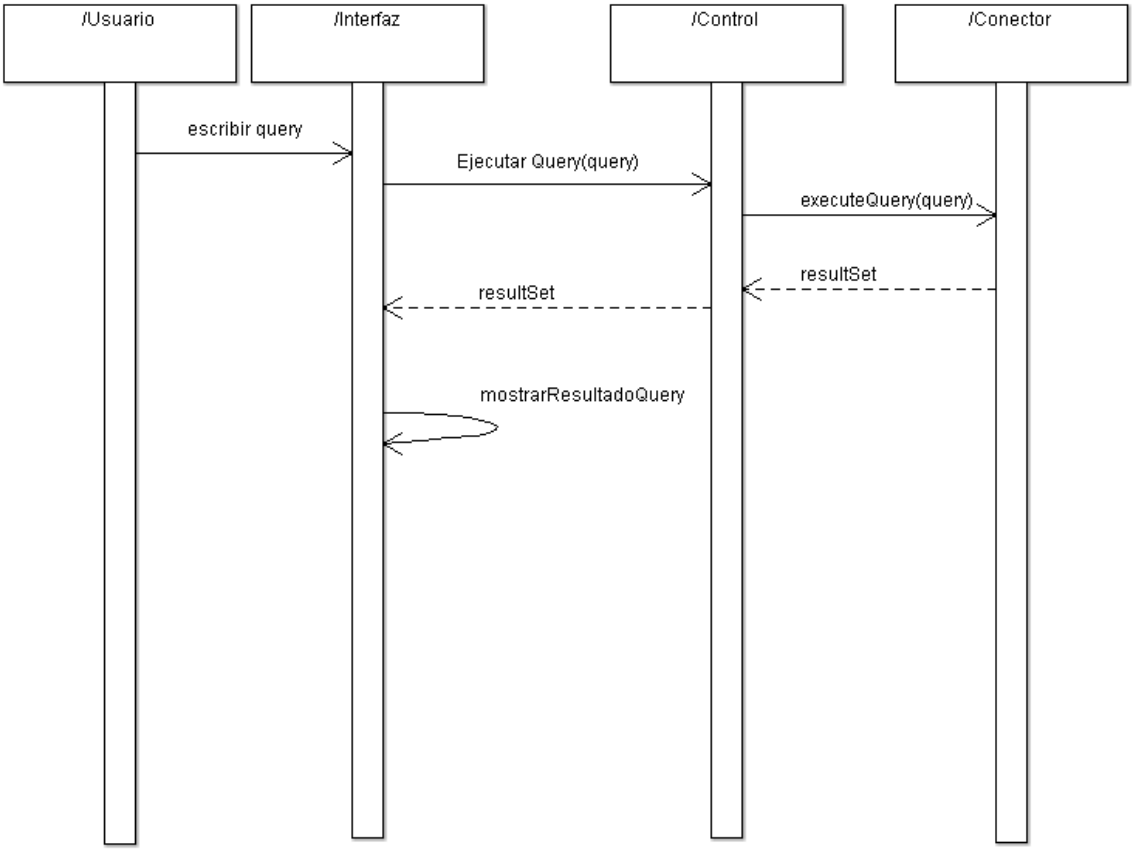

Figura 14 Diagrama de secuencias de consulta a la base de datos

DIAGRAMA DE ACTIVIDADES

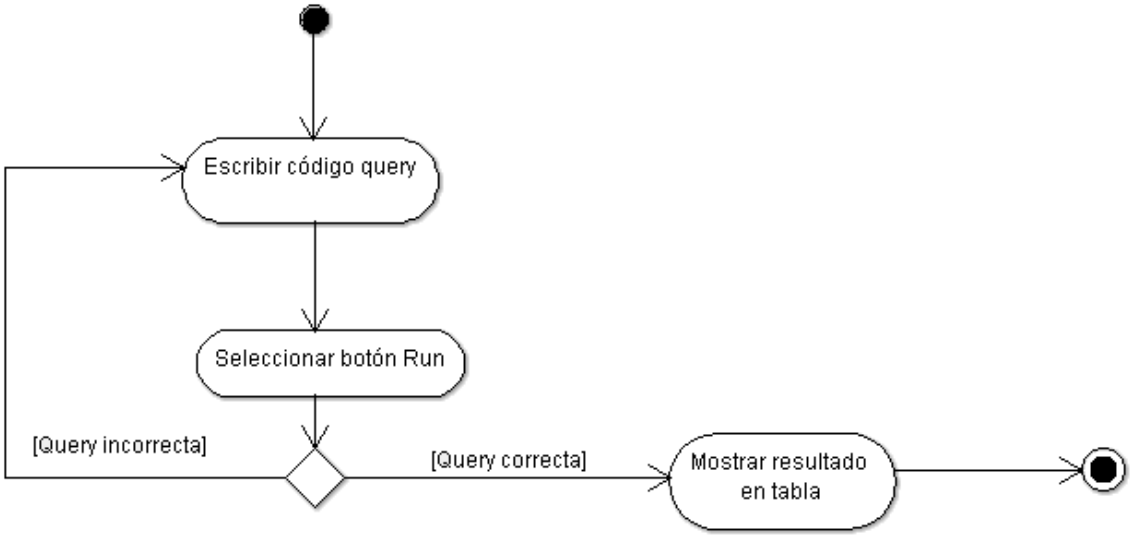

Figura 15 Diagrama de actividades de consulta a la base de datos

2.2.7. Cargar árbol:

DIAGRAMA DE SECUENCIAS

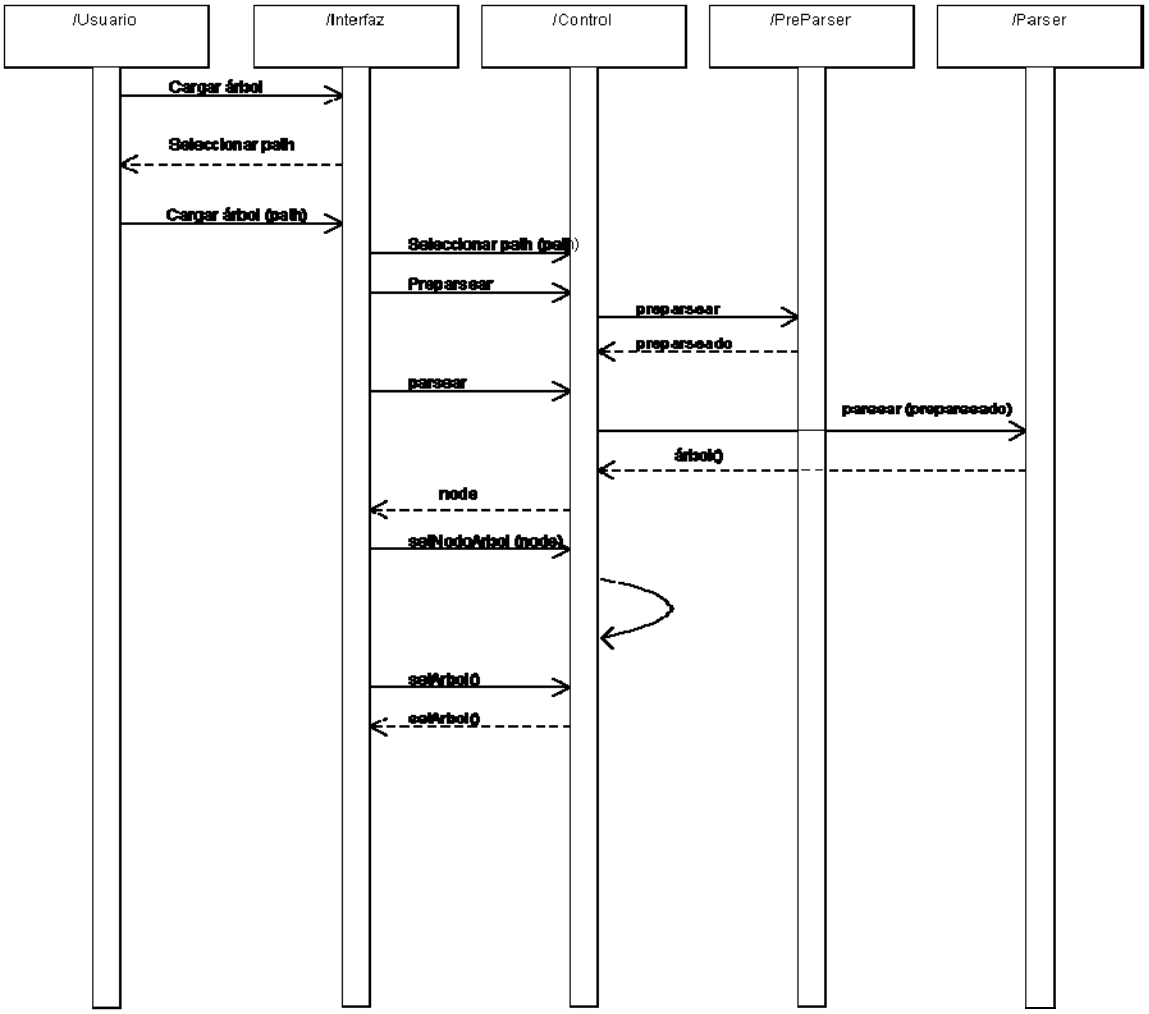

Figura 16 Diagrama de secuencias de cargar árbol

DIAGRAMA DE ACTIVIDADES

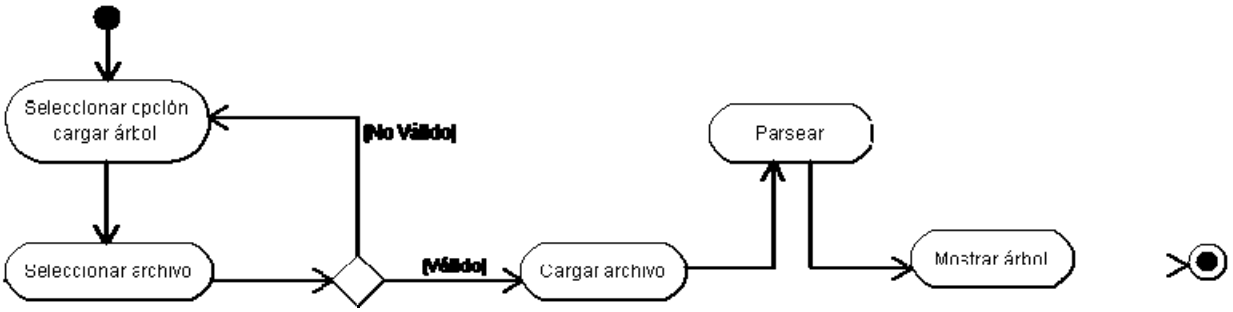

Figura 17 Diagrama de actividades de cargar árbol

2.2.8. Salvar árbol:

DIAGRAMA DE SECUENCIAS

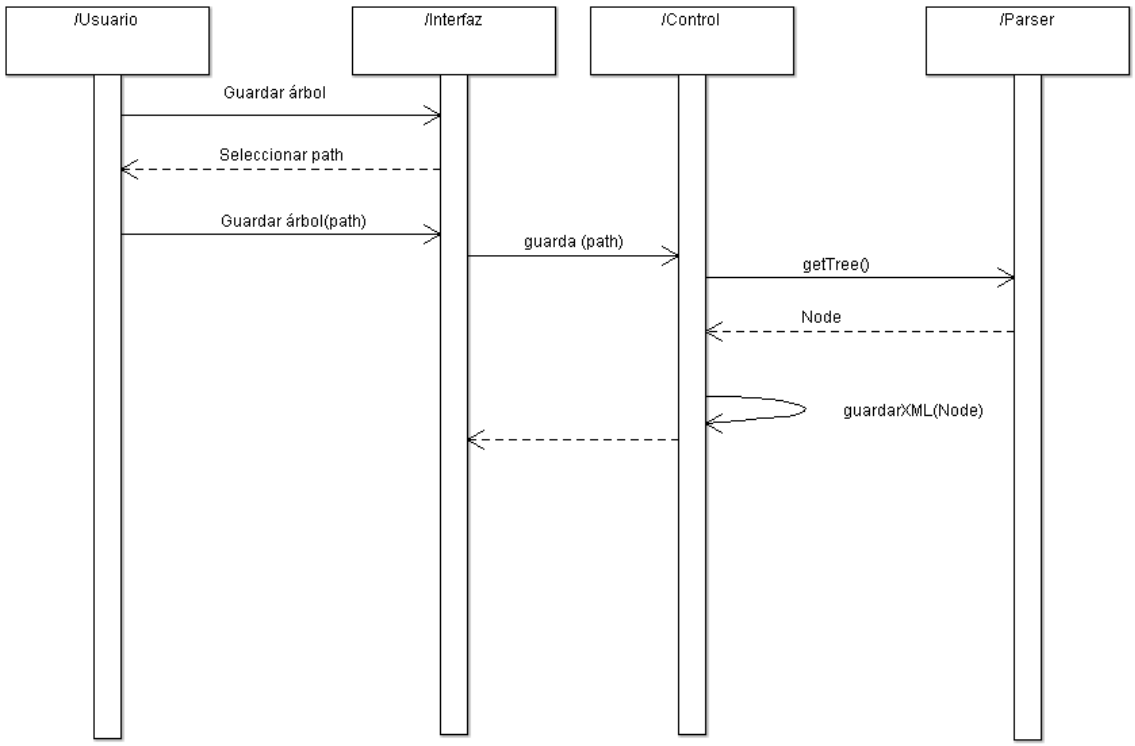

Figura 18 Diagrama de secuencias de salvar árbol

DIAGRAMA DE ACTIVIDADES

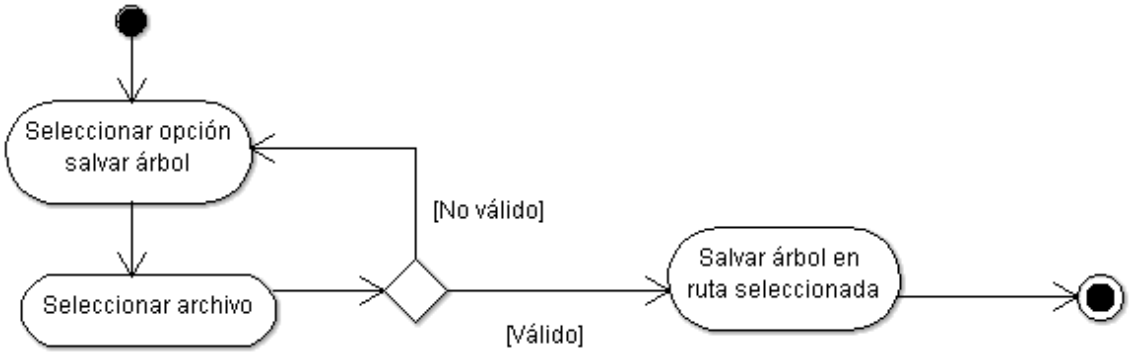

Figura 19 Diagrama de actividades de salvar árbol

3. Diseño

El desarrollo de software que este proyecto propone, al ser una herramienta que pretende tener aplicación dentro del contexto de un problema real, tiene que seguir un proceso de análisis y diseño que proporcione los cimientos sobre los cuales se va a desarrollar la aplicación. El capítulo en sí proporciona una pequeña introducción a lo que es la disciplina de la Ingeniería del Software, y posteriormente detallará los procesos y principios de análisis y diseño del software que sustentan este proyecto.

El diseño es el primer paso en la fase de desarrollo de cualquier producto o sistema de ingeniería. Generalmente, la fase de diseño produce un diseño de datos, arquitectónico, de interfaz y procedimental:

- El **diseño de datos** se encarga esencialmente de transformar el modelo de dominio de la información durante el análisis/la especificación. En el caso particular de este proyecto el diseño de datos está caracterizado por un árbol donde se guardan los distintos datos relacionados con las vistas cargadas desde el archivo SQL. Cada nodo del árbol contendrá el nombre de la vista, una lista (array) de nodos hijos que se corresponden con las vistas relacionadas a dicho nodo, así como otros atributos necesarios para el funcionamiento del proyecto.

- En el **diseño arquitectónico** se definen las relaciones entre los principales elementos estructurales del programa para una herramienta de software basada en el desarrollo. El objetivo es desarrollar una estructura de programa modular y representar las relaciones de control entre módulos. Un diseño arquitectónico describe en general cómo se construirá una aplicación de software. Para ello se documenta utilizando ciertos diagramas, dentro de los cuales destaca el de clases. Los diagramas de clases son utilizados durante el proceso de análisis y diseño de los sistemas, donde se crea el diseño conceptual de la información que se manejará en el sistema, y los componentes que se encargaran del funcionamiento y la relación entre uno y otro. El diseño arquitectónico mezcla la estructura de programas y la de datos definiendo las interfaces que facilitan el flujo de los datos a lo largo del programa. El diagrama de clases correspondiente a nuestro proyecto sería el siguiente:

DIAGRAMA DE CLASES

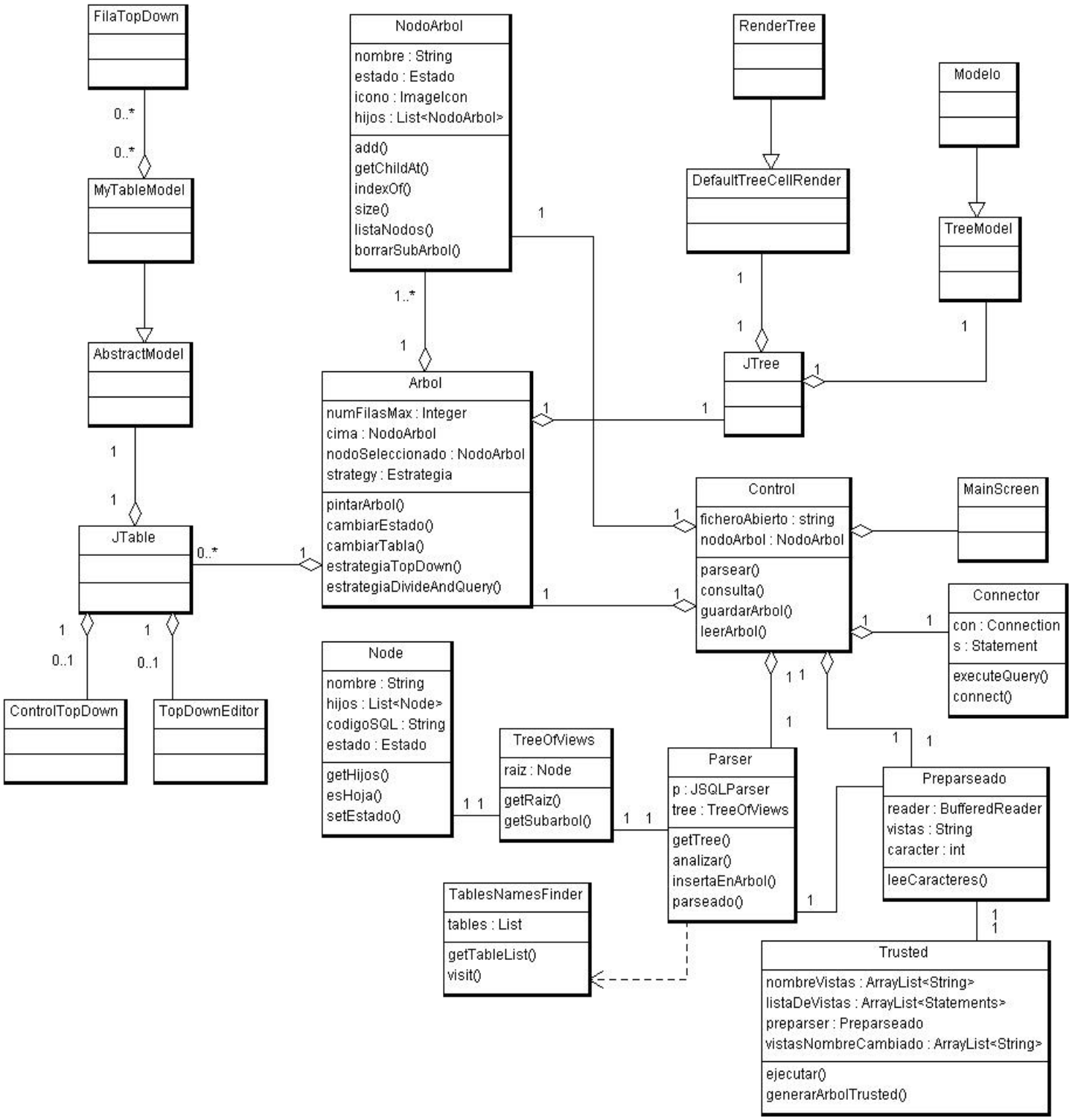

Figura 20 Diagrama de Clases

- El **diseño de interfaz** describe cómo se comunica el software consigo mismo y con su entorno.

- La **interfaz gráfica de usuario**, conocida también como **GUI** (del inglés *Graphical User Interface*) es un programa informático que actúa de interfaz de usuario, utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. En nuestro caso, consta de un componente gráfico donde se muestra la estructura del árbol creado a partir de las vistas seleccionadas. Además, en la barra de menú existen varios botones con las acciones que el usuario puede seleccionar para llevar a cabo la depuración de dichas vistas. El principal uso de la interfaz consiste en proporcionar un entorno visual sencillo e intuitivo para permitir al usuario la comunicación con el sistema operativo de una máquina o computador.

  Algunas características importantes del diseño de interfaz son:

  - Los sistemas interactivos se caracterizan por la importancia del diálogo con el usuario.
  - La interfaz es por tanto una parte fundamental en el proceso de desarrollo y debe tenerse en cuenta desde el principio.
  - Además, la interfaz determina en gran medida la percepción e impresión que el usuario posee de una aplicación.
  - El usuario no está interesado en la estructura interna de una aplicación, sino en cómo usarla.

- El **diseño procedimental** transforma los elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del software.

  En el diseño procedimental se utiliza una técnica conocida como programación estructurada, cuya filosofía es la construcción de algoritmos y programas modulares, descendentes (top-down) y de una entrada- una salida, lo cual facilita la legibilidad, prueba y mantenimiento.

  El diseño procedimental se realiza después de haber establecido la estructura del programa y de datos, y especifica los detalles algorítmicos del software. En el caso de nuestro proyecto, se pueden deducir los mencionados detalles algorítmicos de las distintas operaciones consultando los diagramas de actividades (sección 2.2), los cuales tienen gran parecido con los diagramas de flujo usados normalmente en el diseño procedimental.

4. Implementación

Para describir la arquitectura de nuestra herramienta vamos a ver un ejemplo de ejecución en el que se hablará de las distintas clases que intervienen en cada caso.

Para poder depurar un árbol de vistas, es necesario estar conectado a la base de datos donde se encuentra la instancia que deseamos depurar, por ello, nada más iniciar la aplicación, se mostrará una ventana donde podremos rellenar los parámetros necesarios para la conexión.

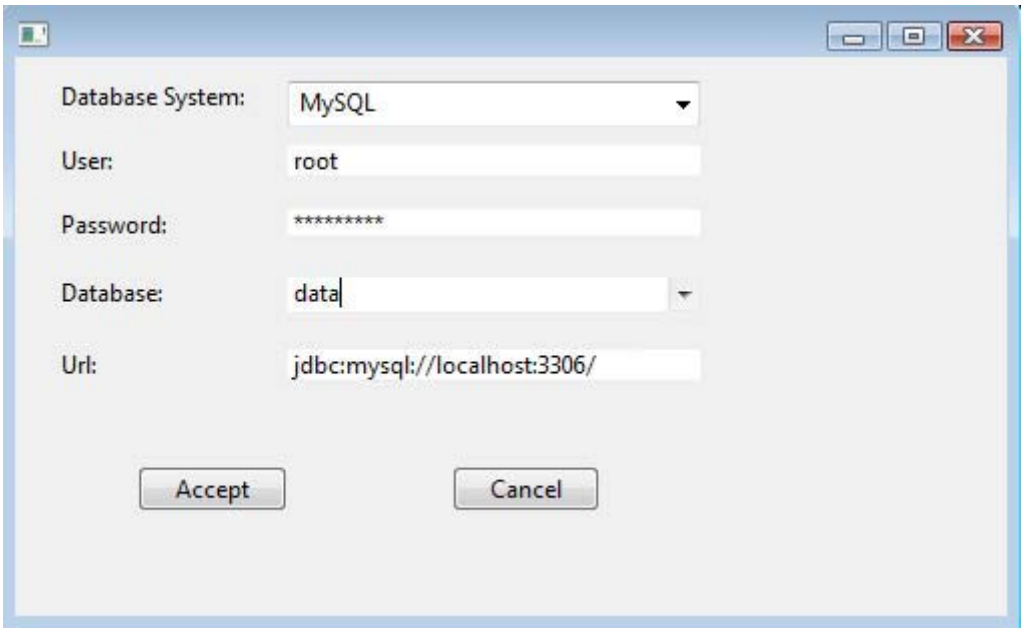

Figura 21 Interfaz para la conexión a la base de datos

La clase *MainScreen.java* recoge los parámetros introducidos por el usuario y los pasa a la clase *Connector.java* que haciendo uso de la API para la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, JDBC (*Java Database Connectivity),* conectará a la base de datos indicada. Esta API se usará siempre que se necesite realizar cualquier operación sobre nuestra base de datos, como por ejemplo, en las consultas.

Una vez hemos conectado a la base de datos nos aparecerá la pantalla principal de la aplicación. Esta pantalla se muestra porque desde la clase *Control.java* se llamará a la clase *Arbol.java* que es la encargada de la mayoría de la parte gráfica.

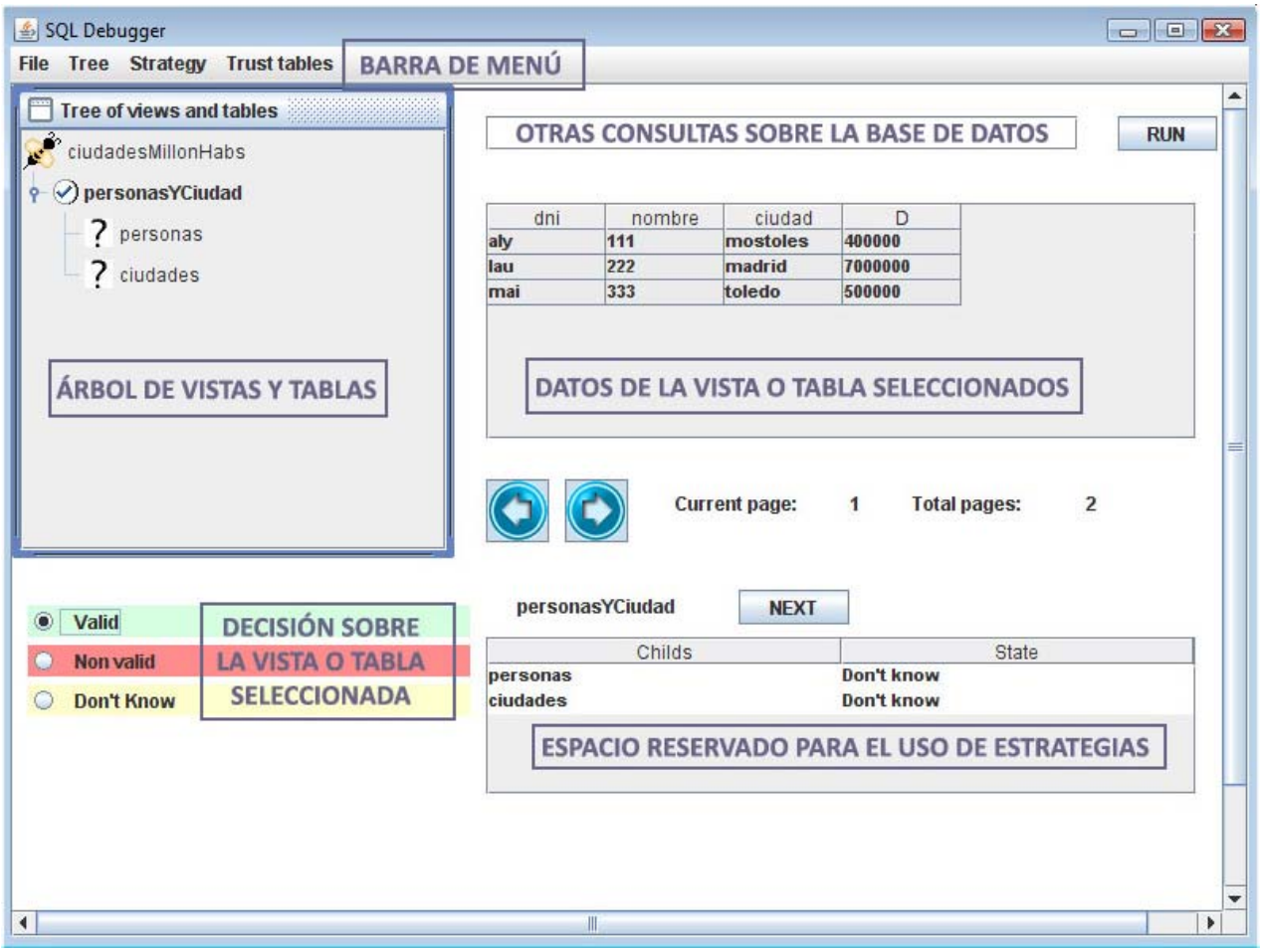

Figura 22 Interfaz gráfica del depurador

Recordamos su uso:

- **Barra de Menú:** Dentro del menú "*File*" podremos escoger el archivo fuente .sql de vistas. Una vez cargado el archivo, podemos introducir una especificación fiable, es decir, seleccionar un archivo .sql en el submenú "*Open trusted...*" en el que todas las vistas son válidas. Este archivo nos servirá para comparar la definición de vistas coincidentes en ambos archivos, pudiendo así marcar éstas como válidas.
- **Árbol de vistas y tablas:** Muestra el árbol generado a raíz del fichero que hemos seleccionado.
- **Decisión sobre la vista o tabla seleccionada:** Se usa para decidir el estado del nodo seleccionado, *Valid* (resultado Válido) si es correcto el resultado, *Non valid* (resultado No válido) si no lo es o *Don't Know* (resultado Desconocido) si no sabemos si es correcto o no.
- **Otras consultas sobre la base de datos:** Esta opción nos permite realizar una consulta cualquiera sobre nuestra base de datos.
- **Datos de la vista o tabla seleccionados:** Muestra los datos de la tabla o vista que tenemos seleccionada.
- **Espacio reservado para el uso de estrategias:** Aquí se mostrará la navegación pertinente según la estrategia escogida. Una tabla con los hijos del nodo actual en el caso de la estrategia *Top Down* y unos botones de radio para elegir si la vista o tabla elegida por la estrategia es correcta o no en el caso de la estrategia *Divide & Query*.

Nada más empezar lo primero que debemos hacer es escoger un fichero fuente .sql para depurar. Para ello nos iremos a "*File*" y allí seleccionaremos el submenú "*Open…".*

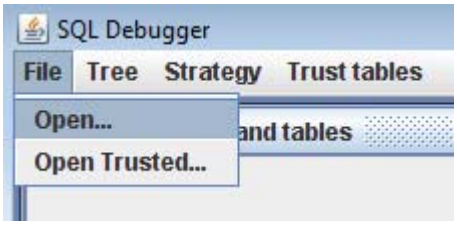

Figura 23 Submenú "Open…"

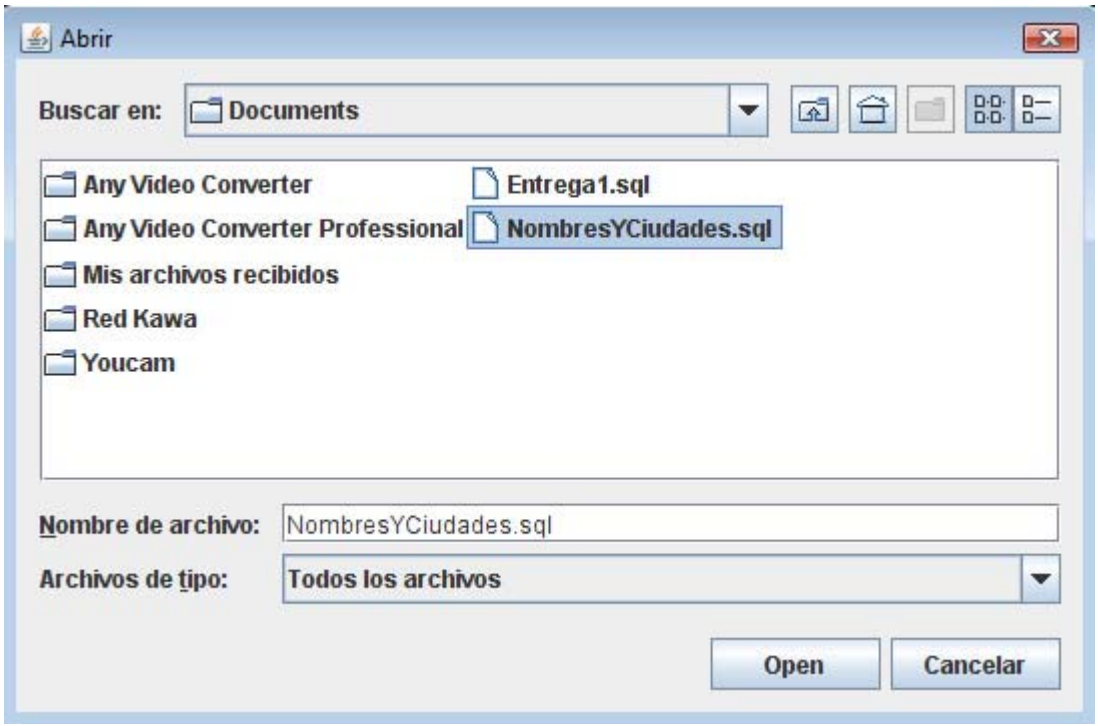

Figura 24 Selección de archivo

Con el fin de tratar este archivo necesitamos conocer la ruta del fichero dentro de la máquina. Para ello, los *JMenuItem* ("*Open…"*) disponen de acciones asociadas a ellos. En nuestro caso hemos extendido la clase *AbstractAction.java* que es la que utilizan los *JMenuItem* y hemos creado *AccionAbrir.java* que abre el fichero y le pasa su ruta a la clase *Control.java.*

Después de seleccionar el archivo, *Control.java* necesita obtener un árbol de dependencias de vistas y tablas. De ello se encarga la clase *Parser.java*. Previamente, *Control.java* pedirá a la clase *Preparseado.java* que devuelva la lista de sentencias de creación o modificación de vistas sin comentarios ni espacios innecesarios y pasará esta lista a la clase *Parser.java* que se encargará de generar un árbol de la clase *TreeOfViews.java* que contendrá nodos de la clase *Node.java.* Este árbol contiene la raíz de las vistas y la clase *Node.java* contiene el nombre y los hijos de cada vista. Así

tenemos ya un árbol de las vistas que queremos depurar, pero este árbol no es el que mostraremos, pues necesitamos además de todo esto, una parte gráfica para mostrarlo.

Control.java se encargará de transformar el árbol con nodos *Node.java* a un árbol con nodos *NodoArbol.java* que además de contener la lista de hijos y el nombre, contendrá el estado y la imagen asociada al estado en el que se encuentra cada nodo. Después *Control.java* se encargará de cambiar en su instancia la raíz del atributo árbol de *Arbol.java* que es de la clase *JTree.java*, que permite ser visualizado en un panel. Se mostrará de la siguiente manera:

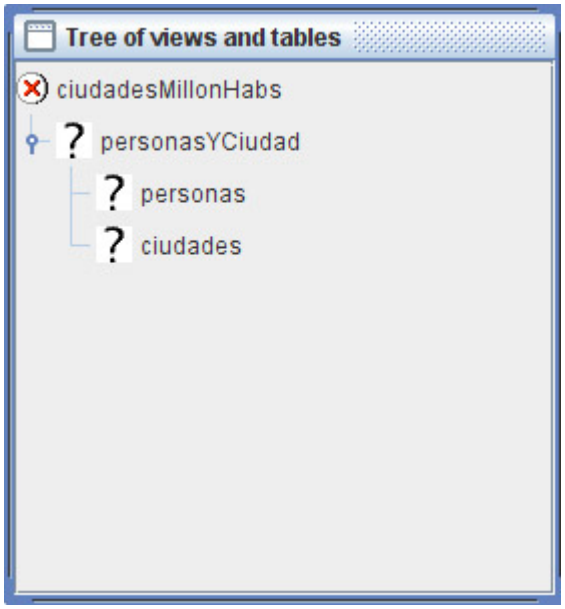

Figura 25 Árbol de vistas

Obtenido a partir de las siguientes vistas:

- CREATE VIEW **personasYCiudad**(nombre,dni,ciudad,poblacion) AS SELECT nombre, dni, nombreCiudad, poblacion FROM **personas**, **ciudades** WHERE ciudad = nombreCiudad;

- CREATE VIEW **ciudadesMillonHabs**(nombreCiudad,habitantes) AS SELECT ciudad, poblacion FROM **personasYCiudad** WHERE poblacion > 1000000;

Que como se ve, queda la raíz CiudadesMillonHabs como No válida, pues seguro sabemos que en el grupo de vistas introducidas alguna será errónea, y el resto todavía como interrogación, es decir, no sabemos si están bien o mal, estado Desconocido.

Si escogemos abrir un fichero con una especificación fiable, tendremos que ir al menú "*File*", submenú "*Open Trusted...*"

Figura 26 Ssubmenú "Open Trusted"

Una vez escogido el fichero, se le pasará, de la misma manera explicada arriba para el fichero que queremos depurar, a la clase *Control.java* la ruta del fichero con la especificación fiable. *Control.java* le pedirá a la clase *Parser.java* una lista con las vistas coincidentes en ambos ficheros, el fiable y el del árbol de vistas que tenemos actualmente cargado. Para saber si su definición es la misma, *Control.java* ejecutará una consulta sobre cada vista coincidente de cada fichero y comparará los resultados. Si ambas coinciden *Control.java* pedirá a *Arbol.java* que cambie el nodo coincidente a estado Válido.

Para comenzar la depuración podremos pinchar sobre el nombre de la vista sobre la que deseemos tomar una decisión y se mostrarán sus datos en forma de tabla. Cada vez que se pincha un nodo del árbol *JTree* se le pide a *Control.java* las siguientes filas que se deben mostrar de este nodo, esto es, no mostramos todas las filas a la vez, si no que vamos mostrándolas poco, para ello *Control.java* mantiene un índice que va cambiando según se pidan más o menos filas o si se cambia de nodo, vuelve al principio. *Control.java* hace una llamada a la JDBC con una consulta de selección de todas las filas de esa vista y devuelve a *Arbol.java* esas filas junto con el nombre de las columnas de la vista, que *Arbol.java* mostrará en una tabla del tipo *JTable* de la siguiente manera:

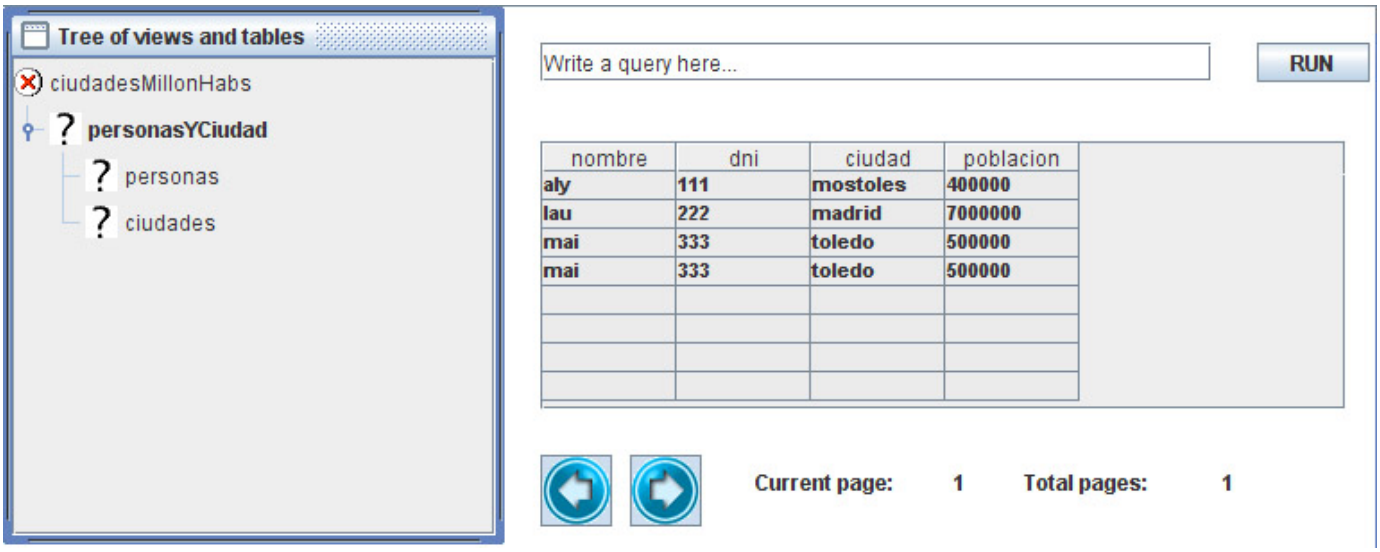

Figura 27 Tabla con los resultados de la vista seleccionada

Viendo los resultados obtenidos el usuario decide si la vista es Válida o no usando los *JRadioButton* que hay debajo del árbol:

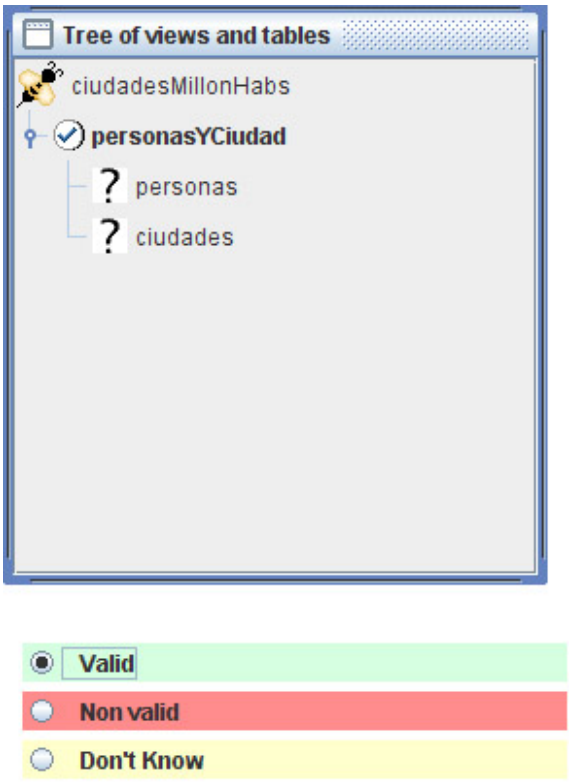

Figura 28 Cambio de estado del nodo seleccionado

Cuando un nodo se cambia de estado a parte de cambiar del árbol el estado del nodo de la clase *NodoArbol.java* seleccionado, se cambian todas las repeticiones en el árbol de este nodo, pues corresponden a una misma vista y se comprueba la existencia de nodos Críticos o de tipo *Buggy*, esto es un nodo que tenga todos sus hijos en estado Válido, como es el caso de la vista **CiudadesMillonHabs**, que como hemos visto es Crítico. Ya hemos encontrado la vista que está mal en nuestro ejemplo. Para avisarnos aparecerá una ventana emergente diciéndonos que una vista es crítica.

Figura 29 Anuncio de nodo crítico

Esta manera de depuración que hemos visto corresponde a una depuración sin estrategia, depuración manual, a continuación vamos a ver cómo se haría la depuración usando las diferentes estrategias:

- **TOP DOWN:**

Una vez cargado el archivo fuente de la manera en la que explicamos arriba y generado el árbol, seleccionamos en el menú "*Strategy*" la opción "*Top down":*

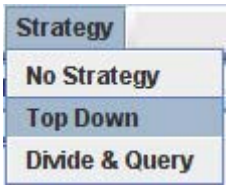

Figura 30 Submenú "Top Down"

La clase *PonerEstrategia.java* que extiende el *AbstractAction* de este menú se encargará de cambiar el atributo global *strategy* para la clase *Arbol.java* que indica que estrategia se está usando. Al seleccionar esta estrategia la clase *Arbol.java* se encargará de generar una tabla con tantas filas como hijos tenga el nodo seleccionado en ese momento, en caso de no tener ningún nodo seleccionado, se hará con los hijos de la raíz:

Figura 31 Tabla para la navegación de estrategia Top Down

Como se observa, para cada hijo, habrá una columna que se refiere a su estado. Al pinchar sobre cada fila, aparecerá en la tabla de "Datos de la vista o tablas seleccionadas" el contenido de la vista que se corresponde con el nombre del hijo de la fila seleccionada. De esta manera, usando el combo de la columna *State*, podremos elegir su estado:

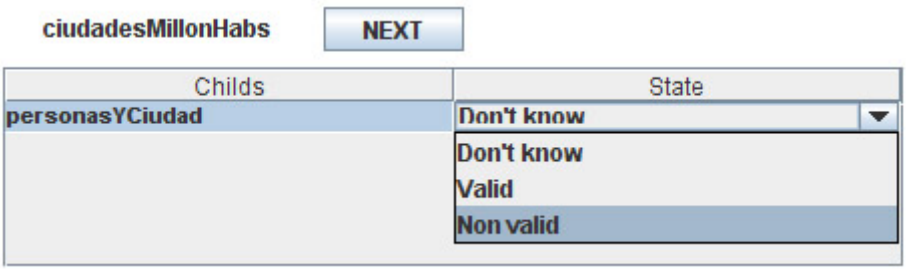

Figura 32 Selección de estado de los hijos en la tabla Top Down

Al pulsar sobre el botón *Next*, *Arbol.java* enviará a *Control.java* una lista con los hijos y su estado seleccionado y *Control.java* seleccionará uno de los hijos que se han cambiado a estado No válido y repetiremos estrategia hasta que lleguemos a un nodo Crítico.

- **DIVIDE & QUERY:**

Una vez cargado el archivo de vistas SQL de la manera en la que explicamos arriba y generado el árbol, seleccionamos en el menú "*Strategy"* la opción "*Divide & Query":*

Figura 33 Submenú Divide & Query

La clase *PonerEstrategia.java* que extiende el *AbstractAction* de este menú se encargará de cambiar el atributo global *strategy* para la clase Arbol.java que indica que estrategia se está usando. Al seleccionar esta estrategia la clase *Arbol.java* pedirá a *Control.java* el nodo mitad del árbol y se mostrará en la tabla de "Datos de la vista o tablas seleccionadas" los datos de este nodo mitad y en el "espacio reservado para el uso de estrategias" podremos seleccionar el estado de este mismo:

Figura 34 Navegación de estrategia Divide & Query

Si el estado del nodo es No válido (*Non Valid)* y damos a *Next*, se volverá a calcular el nodo mitad, pero entre los hijos del nodo seleccionado anteriormente como nodo mitad, si el nodo es Válido (*Valid)* y pulsamos *Next*, se calculará entre los demás hijos y descendientes del padre del nodo seleccionado anteriormente como nodo mitad, excluyendo a este. La estrategia se repite hasta encontrar un nodo crítico.

4.1. Herramientas utilizadas

4.1.1 Bases de Datos: MySQL

El sistema de gestión de bases de datos de nuestro proyecto es MySQL, licenciado bajo la GPL (*GNU General Public License*) de la GNU. La licencia GNU GPL de MySQL obliga a que la distribución de cualquier producto derivado se haga bajo esa misma licencia. Si un desarrollador desea incorporar MySQL en su producto pero desea distribuirlo bajo otra licencia que no sea ésta, puede adquirir una licencia comercial de MySQL que le permita llevar esto a cabo.

MySQL es un sistema de administración relacional de bases de datos. Una base de datos relacional archiva datos en tablas separadas en vez de colocar todos los datos en un gran archivo. Esto permite obtener más velocidad y flexibilidad. Las tablas están conectadas por relaciones definidas que hacen posible combinar datos de diferentes tablas.

La base de datos MySQL se ha convertido en la base de datos de código abierto más popular debido a su alto rendimiento, alta fiabilidad y facilidad de uso. Muchas de las más grandes organizaciones con mayor crecimiento del mundo, tales como Facebook, Google y Adobe, se basan en MySQL para ahorrar tiempo y dinero en sus grandes volúmenes de sitios Web, los sistemas críticos de negocio y paquetes de software.

MySQL se ejecuta en más de 20 plataformas, incluyendo Linux, Windows, Mac OS, Solaris, HP-UX, IBM AIX, lo que le dota de una gran flexibilidad. Así mismo existen varias APIs que permiten, a aplicaciones escritas en diversos lenguajes de programación, acceder a las bases de datos MySQL. En nuestro caso Java, para el cual existe una implementación nativa el driver de Java.

Entre las características disponibles en las últimas versiones se puede destacar:

- Amplio subconjunto del lenguaje SQL.
- Disponibilidad en gran cantidad de plataformas y sistemas.
- Diferentes opciones de almacenamiento según si se desea velocidad en las operaciones o el mayor número de operaciones disponibles.
- Transacciones y claves foráneas...
- Conectividad segura.
- Agrupación de transacciones, reuniendo múltiples transacciones de varias conexiones para incrementar el número de transacciones por segundo.

4.1.2 Lenguaje de programación: Java

Las características principales que ofrece Java frente a otros lenguajes de programación son:

- Lenguaje Simple:

Java posee una curva de aprendizaje muy rápida. Resulta relativamente sencillo escribir *applets* interesantes desde el principio. Todos aquellos familiarizados con C++ encontrarán que Java es más sencillo, ya que se han eliminado ciertas características, como los punteros. Pese a su simpleza se ha conseguido un considerable potencial de expresión e innovación desde el punto de vista del programador.

El paquete de utilidades de Java viene con un conjunto completo de estructuras de datos complejas y sus métodos asociados, que serán de inestimable ayuda para implementar applets y otras aplicaciones más complejas. Se dispone también de estructuras de datos habituales, como *pilas* y *tablas hash*, como clases ya implementadas.

- Orientado a Objetos:

Java fue diseñado como un lenguaje orientado a objetos desde el principio. Los objetos agrupan en estructuras encapsuladas tanto sus datos como los métodos (o funciones) que manipulan esos datos.

- Distribuido:

Java proporciona una colección de clases para su uso en aplicaciones de red, que permiten abrir sockets y establecer y aceptar conexiones con servidores o clientes remotos, facilitando así la creación de aplicaciones distribuidas y proporcionando una colección de clases para aplicaciones en red.

- Robusto:

Java fue diseñado para crear software altamente fiable. Para ello proporciona numerosas comprobaciones en compilación y en tiempo de ejecución. Sus características de memoria liberan a los programadores de los errores derivados de la aritmética de punteros, ya que se ha prescindido por completo de ellos.

Además Java modifica completamente la gestión de la memoria que se hace en C/C++. En C/C++ se utilizan punteros, reservas de memoria (con las órdenes *malloc*, *new*, *free*, *delete*...) y otra serie de elementos que dan lugar a graves errores en tiempo de ejecución difícilmente depurables.

Java tiene operadores nuevos para reservar memoria para los objetos, pero no existe ninguna función explícita para liberarla.

La recolección de basura (objetos ya inservibles) es una parte integral de Java durante la ejecución de sus programas. Una vez que se ha almacenado un objeto en el tiempo de ejecución, el sistema hace un seguimiento del estado del objeto, y en el momento en que se detecta que no se va a volver a utilizar ese objeto, el sistema vacía ese espacio de memoria para un uso futuro. Esta gestión de la memoria dinámica hace que la programación en Java sea más fácil.

Java, como decimos, realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución, lo que hace que se detecten errores lo antes posible, normalmente en el ciclo de desarrollo. Algunas de estas verificaciones que hacen que Java sea un lenguaje robusto son:

- Verificación del *código de byte*.
- Gestión de excepciones y errores.
- Comprobación de punteros y de límites de vectores.

Se aprecia una clara diferencia con C++ quién no realiza ninguna de estas verificaciones.

- Seguro:

Dada la naturaleza distribuida de Java, donde las applets se bajan desde cualquier punto de la Red, la seguridad se impuso como una necesidad de vital importancia. Se han implementado barreras de seguridad en el lenguaje y en el sistema de ejecución de tiempo real.

Se ha conseguido lograr cierta inmunidad en el aspecto de que un programa realizado en Java no puede realizar llamadas a funciones globales ni acceder a recursos arbitrarios del sistema, por lo que el control sobre los programas ejecutables no es equiparable a otros lenguajes.

Los niveles de seguridad que presenta son:

- Fuertes restricciones al acceso a memoria, como son la eliminación de punteros aritméticos y de operadores ilegales de transmisión.
- Rutina de verificación de los *códigos de byte* que asegura que no se viole ninguna construcción del lenguaje.
- Verificación del nombre de clase y de restricciones de acceso durante la carga.
- Sistema de seguridad de la interfaz que refuerza las medidas de seguridad en muchos niveles.

✓ Independiente de la arquitectura:

Java está diseñado para soportar aplicaciones que serán ejecutadas en los más variados entornos de red, desde Unix a Windows Nt, pasando por Mac y estaciones de trabajo, sobre arquitecturas distintas y con sistemas operativos diversos. Para acomodar requisitos de ejecución tan variopintos, el compilador de Java genera bytecodes: un formato intermedio indiferente a la arquitectura diseñado para transportar el código eficientemente y con gran versatilidad, a múltiples plataformas hardware y software. El resto de problemas los soluciona el intérprete de Java.

✓ Portable:

Por ser indiferente a la arquitectura sobre la cual está trabajando, esto hace que su portabilidad sea muy eficiente. Sus programas son iguales en cualquiera de las plataformas, ya que Java especifica tamaños de sus tipos de datos básicos y el comportamiento de sus operadores aritméticos, esto se conoce como la máquina virtual de Java (JVM).

✓ Interpretado y compilado a la vez:

Java puede ser compilado e interpretado en tiempo real, ya que cuando se construye el código fuente éste se transforma en una especie de código de máquina. Esta característica no la posee C++. No obstante, y aunque en teoría se consumen menos recursos siendo los lenguajes interpretados, el actual compilador que existe es bastante lento, unas 20 veces menos rápido que C++. Esto normalmente no es vital para la aplicación ni demasiado apreciable por el usuario.

✓ Multihebra o Multihilos:

Java soporta sincronización de múltiples hilos de ejecución (*multithreading*) a nivel de lenguaje, especialmente útiles en la creación de aplicaciones de red distribuidas. Así,

mientras un hilo se encarga de la comunicación, otro puede interactuar con el usuario mientras otro presenta una animación en pantalla y otro realiza cálculos. Es decir, por cada hilo que el programa tenga se ejecutaran en tiempo real muchas funciones al mismo tiempo.

- Dinámico. Alto rendimiento:

El lenguaje Java y su sistema de ejecución en tiempo real son dinámicos en la fase de enlazado. Las clases sólo se enlazan a medida que son necesitadas. Se pueden enlazar nuevos módulos de código bajo demanda, procedente de fuentes muy variadas, incluso desde la red. Es veloz en el momento de ejecutar los programas y ahorra muchas líneas de código.

4.1.3 JSQLParser

JSQLParser es un analizador sintáctico basado en el lenguaje de acceso a base de datos SQL. El sistema acepta una sentencia SQL y tras el análisis, devuelve una estructura de clases Java, navegable mediante el patrón *visitor*. El patrón *visitor* es una forma de separar el algoritmo de la estructura de un objeto.

La idea básica es que se tiene un conjunto de clases que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método aceptar (*accept()*) que recibe al objeto visitador (*visitor*) como argumento. El visitador es una interfaz que tiene un método *visit* diferente para cada clase. El método *accept* de una clase llama al método *visit* de su clase. *Visitor* permite definir nuevas operaciones sin cambiar las clases de los elementos en los que opera. Este patrón es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general.

Se incluyó en el proyecto la clase *TablesNamesFinder* pero modificando su funcionalidad para que sea posible el análisis de vistas. Como se observa en el código de ejemplo que aparece a continuación de este párrafo, esta clase acepta distintos tipos de "visitadores" dependiendo de su morfología, es decir, realizará una acción u otra si el objeto que la visita es de un tipo u otro. En nuestro proyecto, la mayor parte de las veces es accedida mediante objetos que conforman sentencias tipo SELECT (PlainSelect), pero puede ser extensible al resto de objetos que instancien muchas de las clases de la jerarquía.

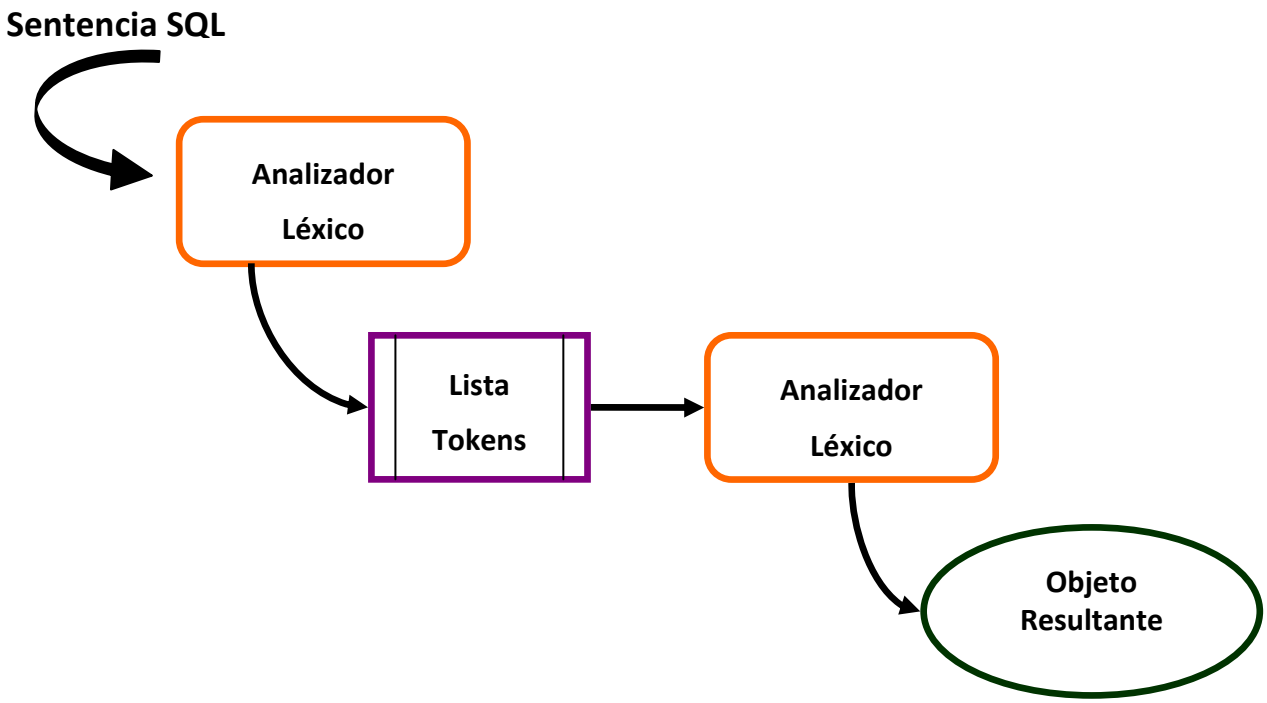

Figura 35 Estructura de JSQLParser

4.1.3.1 Funcionalidades de JSQLParser original

JSQLParser, en su versión inicial, es capaz de analizar tablas y generar una estructura de clases con la información correspondiente para cada uno de los campos de la tabla: campos, nombres de campos, nombre de tabla, etc.

El diccionario léxico de esta versión inicial es limitado ya que no permite la inclusión de palabras normalmente presentes en las sentencias SQL, como pueden ser: *view*, *minus* o *intersection*. Con lo que su funcionalidad se encuentra muy limitada a la hora de ser utilizado para un uso general.

4.1.3.2 Estructura del sistema

JSQLParser es un analizador de código SQL, es decir, es capaz de leer como entrada un fichero de sentencias SQL, analizar el contenido de las mismas y generar una estructura de datos (estructura de clases) con la información estructurada en las tablas analizadas.
Para ello cuenta, como estructura general, con un diccionario de palabras reservadas perteneciente al lenguaje SQL y un analizador sintáctico que recorre las cadenas de

El léxico se encuentra en la clase **net.sf.jsqlparser.parser.CCJSqlParserConstants.java** que consiste en una lista de constantes cuyos nombres son cada una de las palabras reservadas de SQL.

El analizador léxico, es decir, el parser que lee carácter a carácter una palabra y detecta si está en el diccionario (o léxico), está en la clase **net.sf.jsqlparser.parser.TokenManager.java.** Esta clase devuelve una lista de identificadores presentes en la lista de constantes para cada sentencia SQL. El estilo de programación de esta clase es muy anticuado ya que utiliza vectores y productos vectoriales para hacer las comprobaciones correspondientes. Como se observa en la sección de ampliaciones esto se ha subsanado, en parte, por los autores ya que se ha utilizado un sistema de testigos (variables booleanas) más propio del lenguaje utilizado, siendo una forma de programación legible y extensible.

El **analizador sintáctico**, o parser como tal, se encuentra en la clase **net.sf.jsqlparser.parser.CCJSqlParser.java.** Utiliza la lista de tokens que devuelve el analizador léxico y utiliza una serie de *subanalizadores* sintácticos que son capaces de generar las estructuras de datos deseadas para cada tipo de sentencia SQL. Es decir, existe un subanalizador capaz de analizar claúsulas de tipo *Select,* otro capaz de analizar *Tables,* pero no existía un subanalizador capaz de analizar *Vistas* (algo que los autores han subsanado como se muestra en la sección siguiente).

El analizador sintáctico, va consumiendo tokens, si encuentra algún token que se debe analizar por algún subanalizador (como por ejemplo: *select*) el sistema lanza el parser capaz de analizar una sentencia select, consumiendo previamente el token correspondiente.
Si el analizador sintáctico encuentra un orden incorrecto de tokens lanzará una excepción (**net.sf.jsqlparser.JSQLParserException.java**) indicando donde se encuentra el error sintáctico.

Las declaraciones (o statements) son el resultado del analizador sintáctico, es decir, son los objetos resultantes del análisis incluyendo la información estructurada presente en las diferentes sentencias SQL.

Por ejemplo, para una sentencia *SELECT* contiene los distintos campos presentes en la sentencia select utilizando la estructura de clases presentes en el paquete: **net.sf.jsqlparser.statement.select.**

4.1.3.3 Ampliaciones

- "CREATE VIEW"

Para permitir el análisis de vistas era necesaria la ampliación del sistema en 3 sentidos diferentes:

1. Analizador Léxico: el analizador léxico debía ser capaz de analizar la palabra "View". El primer paso es asignar al nuevo *token* un identificador numérico no usado hasta el momento.

2. Analizador Sintáctico: el analizador sintáctico al detectar el token: VIEW, debía ser capaz de ejecutar el subanalizador para vistas (también programado) que a su vez reutiliza el analizador para SELECTS (ya que una vista contiene un select). Esta ampliación se realizó incorporando un salto de código, es decir, una bifurcación que permita al sistema seguir tanto por la rama de TABLE como por la rama programada para VIEW. Como decimos, anteriormente cuando se detectaba la palabra CREATE, la palabra que el sistema esperaba a continuación era siempre TABLE, pero con esta ampliación el sistema admite que la palabra siguiente a CREATE pueda ser también la palabra VIEW.

3. Generación de las clases necesarias para la estructura de las vistas: Se creó una clase "Create View" que extiende la clase ya existente "Create Table". A continuación se explicarán las funcionalidades de esta clase y se enumerarán los atributos más importantes de la misma.

- Un objeto de tipo "View", esta clase se hizo a imagen de la clase "Table", y contiene principalmente el nombre de la vista.
- Un objeto de tipo Lista, llamado columDefinitions, y que también existe en "Create Table". Indica el nombre y tipo de cada una de las columnas de la Vista, obtenido del Select.
- El objeto de tipo "SubSelect" llamado "selectStatement". La clase SubSelect contiene a su vez un atributo de la clase SelectBody que es una interfaz que implementa la clase que forma el corazón del Select de una sentencia: PlainSelect. Esta clase contiene toda la información de la parte Select de la consulta: si tiene "DISTINCT", la lista de atributos referenciados, la expresión del "WHERE", la lista de columnas referenciadas en el "GROUP BY" o "ORDER BY" en caso de que existan o si tiene la expresión "HAVING". Todos los tipos de estos atributos tienen el método "public void accept (TipoVisitor nombre);"

Además, para facilitar la lectura de las sentencias, se añadieron más atributos que aportan información de utilidad, sobretodo de cara a posibles ampliaciones de este proyecto, como son: "MINUS", "INTERSECTION", "UNION" y "BASICO", dependiendo

de la aparición de cualquiera de estos operadores o su no aparición en el último caso. También se especifica si la sentencia tiene "Group By", "Where" o "Having". Esta información es extraída a través del objeto de tipo SubSelect, que comentamos anteriormente.

Por último la clase posee otros dos atributos que facilitan el manejo de la vista, son dos listas, la primera con el nombre de todos los hijos de la misma. Esto se hace gracias a la clase TableNamesFinder, implementada en la versión original de JSQLParser, que recoge el nombre de todas las tablas referenciadas en la parte FROM de un SELECT de una sentencia dada y la segunda con los nombres de los atributos referenciados en la parte SELECT, obtenidos a través de la clase PlainSelect.

- "MINUS" e "INTERSECTION"

1. Analizador Léxico: el analizador léxico debía ser capaz de analizar la palabra "Minus" e "Intersection". El primer paso es asignar a los nuevos tokens identificadores numéricos no usados hasta el momento.

2. Analizador sintáctico: funciona igual que en el caso anterior, pero reconociendo el token "Minus" en un caso e "Intersection" en otro.

3. Generación de las clases necesarias para la estructura de las vistas: La clase "Minus" y la clase "Intersection" se hicieron sobre la base que ya existía para "Union". Por tanto, tienen los mismos atributos que esta clase. El más importante es la lista de atributos del Select a la izquierda del Minus (Intersection) y la lista de los atributos referenciados en el select a la derecha del Minus (Intersection). Se implementan también dentro de esta clase los métodos necesarios para saber si una sentencia de cualquiera de estos dos tipos tiene "HAVING", "WHERE" o "GROUP BY".

- "CREATE VIEW OR REPLACE"

1. Analizador léxico: Las palabras "OR" y "REPLACE" ya se encontraban dentro del léxico del sistema de partida, por lo que no hizo falta añadirlas ni reservar nuevos identificadores para ambas.

2. Analizador Sintáctico: El flujo del programa, tal y como estaba implementado, tras leer y reconocer VIEW esperaba un identificador, en este caso el nombre asignado por el usuario a la vista. Se modificó el flujo de forma que fuesen válidas tanto las sentencias en las que tras VIEW aparece el nombre de la vista, como aquellas en las que se especifica la opción REPLACE, es decir, que si la vista existe la reemplaza y si no existe la crea.

3. Generación de las clases necesarias para la estructura de las vistas: No es necesaria la creación de nuevas clases para reconocer este tipo de sentencias, sólo es necesario modificar el analizador sintáctico.

En general el sistema admite, tras la modificación, entre otras, sentencias con la siguiente sintaxis:

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1,...,tablaN **WHERE** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1,...,tablaN **WHERE** condiciones **AND** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1,...,tablaN **WHERE** condiciones **OR** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1 **UNION** tabla2 **WHERE** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1 **INTERSECTION SELECT** atributo1,…,atributoM **FROM** tabla2 **WHERE** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1 **MINUS SELECT** atributo1,…atributoM **FROM** tabla2 **WHERE** condiciones;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** * **FROM** tabla1 (INNER,NATURAL) **JOIN** tabla2 **ON** tabla1.atributo1 **=** tabla2.atributo1;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,**SUM**(atributoN) **FROM** tabla1 **ORDER BY** atributo1;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,atributoN **FROM** tabla1 **WHERE** tabla1.atributoN **>** 1 **GROUP BY** atributo1 **HAVING** atributo1 = **"**Nombre**"**;

CREATE VIEW nombreVista (col1,…,colN) **AS SELECT** atributo1,...,SUM(atributoN) **FROM** tabla1 **GROUP BY** atributo1 **HAVING SUM**(atributoN) < 1200;

CREATE VIEW OR REPLACE nombreVista **AS SELECT** atributo1,...,atributoN **FROM** tabla1 **MINUS** tabla2 **WHERE** cond;

La parte del WHERE, como siempre, es opcional.

Ejemplo de información obtenida utilizando JSQLParser

En el ejemplo se muestra un conjunto de sentencias SQL, vistas SQL. El orden en el que es leído el fichero de entrada debe ser inverso al orden en el que se generará el árbol. Es decir, si una vista referencia en su FROM a otra, la vista referenciada aparecerá antes que la vista que la ha referenciado.

Query1: "create view or replace animalOwner(ID,specie,aname) as select O.ID, P.specie, P.name from Owner O, pet P, petOwner PO where O.ID = PO.ID or P.code = PO.code";

Query2= "create view LessThan6(Id) as select ID from Owner where specie='cat'or specie='dog' group by ID having count(*)<6";

Query3= create view CatsAndDogsOwner(ID,aname) as select AO1.Id,AO1.aname from animalOwner AO1, animalOwner AO2 where AO1.Id = AO2.Id and (AO1.specie='dog' and AO2.specie='cat'or AO2.specie='dog' and AO1.specie='cat');”

Query4= "create view commonName(ID) as select distinct CDO1.ID from CatsAndDogsOwner CDO1, CatsAndDogsOwner CDO2 where CDO1.aname = CDO2.aname and CDO1.ID <> CDO2.ID;"

Query5= "create view NoCommonName(ID) as select ID from CatsAndDogsOwner C where not exists (select ID from commonName N where N.id = C.id);"

Querys6= "create view guest(ID,name) as select O.Id, O.name from Owner O, NoCommonName N, LessThan6 L where O.ID = N.Id and N.ID = L.ID;"

Entonces el árbol que representaría la jerarquía entre las vistas sería:

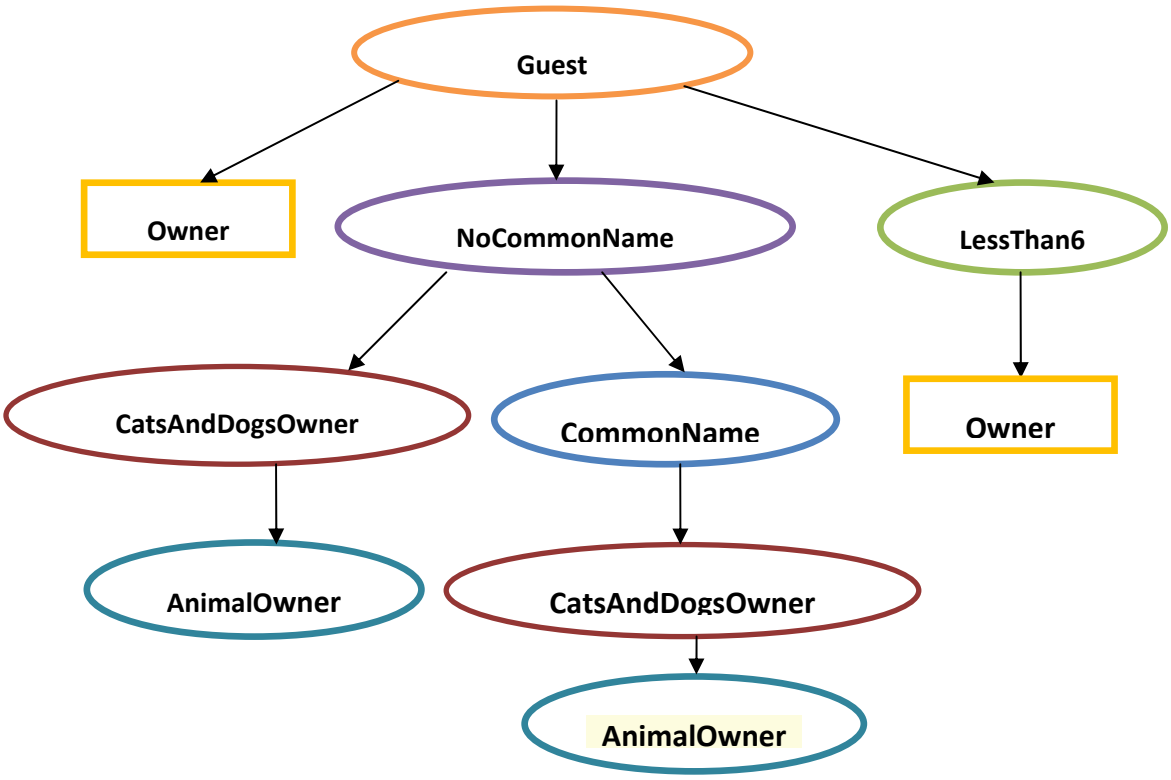

Figura 36 Árbol generado por JSQLParser

Y es esta información efectivamente la que devuelve JSQLParser. A través de la consola del entorno en el que lo programamos, Eclipse, podemos leer toda esta información, estructurada en distintos campos:

- En primer lugar el nombre de la vista.
- A continuación, el código de la sentencia SQL asociada a la vista.
- Para facilitar la depuración construimos una lista que contiene los nombres de las vistas o tablas referenciadas en el FROM.
- Otra lista en la que aparecen las columnas, los atributos, referenciados en la parte del *SELECT.*
- Tipo de la vista. Puede ser: *Básico, Intersection, Union y MInus*.
- Información relativa a si la vista contiene las claúsulas: *GROUP BY, WHERE y HAVING.*
- Y por último, la información relativa al subárbol que cuelga de esta vista. Es decir, sus hijos, con sus respectivos nombres, el código asociado a ellos, y los hijos de sus hijos, con la misma información para cada uno de ellos.
- En el caso de que se trate de una tabla, el código asociado es vacío.
- Si es una hoja, la lista de hijos aparece vacía.

Nombre: guest → **(RAÍZ DEL ÁRBOL)**

Codigo: CREATE VIEW guest(ID,name)AS (SELECT O.Id, O.name FROM Owner AS O, NoCommonName AS N, LessThan6 AS L WHERE O.ID = N.Id AND N.ID = L.ID)

Lista_Relaciones_From: [Owner, NoCommonName, LessThan6]

Lista_Expresiones_Select: [O.Id, O.name]

Tipo:BASICO

HASN´T GROUP BY

HAS WHERE

HASN´T HAVING

Hijos: [**Nombre**: Owner → **(PRIMER HIJO DE LA RAÍZ)**

Codigo: null

Hijos: [],

Nombre: NoCommonName → **(SEGUNDO HIJO DE LA RAÍZ)**

Codigo: CREATE VIEW NoCommonName(ID)AS (SELECT ID FROM CatsAndDogsOwner AS C WHERE NOT EXISTS (SELECT ID FROM commonName AS N WHERE N.id = C.id))

Lista_Relaciones_From: [CatsAndDogsOwner, commonName]

Lista_Expresiones_Select: [ID]

Tipo:BASICO

HASN´T GROUP BY

HAS WHERE

HASN´T HAVING

Hijos: [**Nombre**: CatsAndDogsOwner → **(PRIMER HIJO DE "NoCommonName")**

Codigo: CREATE VIEW CatsAndDogsOwner(ID,aname)AS (SELECT AO1.Id, AO1.aname FROM animalOwner AS AO1, animalOwner AS AO2 WHERE AO1.Id =AO2.Id AND (AO1.specie = 'dog' AND AO2.specie = 'cat' OR AO2.specie = 'dog' AND AO1.specie = 'cat'))

Lista_Relaciones_From: [animalOwner, animalOwner]

Lista_Expresiones_Select: [AO1.Id, AO1.aname]

Tipo:BASICO

HASN´T GROUP BY

HAS WHERE

HASN´T HAVING

Hijos: [**Nombre**: animalOwner → **(ÚNICO HIJO DE "CatsAndDogsOwner")**

Codigo: null

Hijos: []],

Nombre: commonName → **(SEGUNDO HIJO DE "NoCommonName")**

Codigo: CREATE VIEW commonName(ID)AS (SELECT DISTINCT CDO1.ID FROM CatsAndDogsOwner AS CDO1, CatsAndDogsOwner AS CDO2 WHERE CDO1.aname = CDO2.aname AND CDO1.ID <> CDO2.ID)

Lista_Relaciones_From: [CatsAndDogsOwner, CatsAndDogsOwner]

Lista_Expresiones_Select: [CDO1.ID]

Tipo:BASICO

HASN´T GROUP BY

HAS WHERE

HASN´T HAVING

Hijos: [**Nombre**: CatsAndDogsOwner → **(ÚNICO HIJO DE "CommonName")**

Codigo: CREATE VIEW CatsAndDogsOwner(ID,aname)AS (SELECT AO1.Id, AO1.aname FROM animalOwner AS AO1, animalOwner AS AO2 WHERE AO1.Id = AO2.Id AND (AO1.specie = 'dog' AND AO2.specie = 'cat' OR AO2.specie = 'dog' AND AO1.specie = 'cat'))

Lista_Relaciones_From: [animalOwner, animalOwner]

Lista_Expresiones_Select: [AO1.Id, AO1.aname]

Tipo:BASICO

HASN´T GROUP BY

HAS WHERE

HASN´T HAVING

Hijos: [**Nombre**: animalOwner → **(ÚNICO HIJO DE "CatsAndDogsOwner")**

Codigo: null

Hijos: []] → **(FIN DEL SUBÁRBOL QUE CUELGA DE "CatsAndDogsOwner")**

] → **(FIN DEL SUBÁRBOL QUE CUELGA DE "CommonName")**

], → **(FIN DEL SUBÁRBOL QUE CUELGA DE "NoCommonName")**

Nombre: LessThan6 → **(TERCER HIJO DE LA RAÍZ "Guest")**

Codigo: CREATE VIEW LessThan6(Id)AS (SELECT ID FROM Owner WHERE specie = 'cat' OR specie = 'dog' GROUP BY ID HAVING count(*) < 6)

Lista_Relaciones_From: [Owner]

Lista_Expresiones_Select: [ID]

Tipo:BASICO

HAS GROUP BY

HAS WHERE

HAS HAVING

Hijos: [**Nombre**: Owner

Codigo: null

Hijos: []] → **(FIN SUBÁRBOL QUE CUELGA DE "LessThan6")**

] → **(FIN HIJOS DE LA RAÍZ. FIN ÁRBOL)**

4.2. Problemas encontrados

A lo largo de la implementación de la aplicación, fueron surgiendo ciertos problemas que se detallarán a continuación.

- A la hora de implementar las distintas **estrategias** de depuración, quizá fue la de *Divide & Query* la que nos dio más problemas. La búsqueda manual de errores no nos supuso gran dificultad, dado que el usuario tiene total libertad a la hora de explorar los nodos que él crea convenientes. La estrategia *Top Down* no fue tan trivial, pero tras entender su funcionamiento, no fue muy complicado adaptarlo a la implementación. La búsqueda de *Divide & Query* nos llevó algo más de tiempo, ya que el algoritmo es un poco más complejo y elaborado. Tras varias pruebas y consultas en foros y páginas de internet, finalmente dimos con la solución.

- Como se comentó en el apartado de Especificación, una herramienta que decidimos dejar de lado fue ***Hibernate,*** dado que estuvimos un tiempo intentando adaptarla al proyecto y no tuvimos éxito. En su lugar utilizamos los archivos XML para guardar los árboles con sus atributos en un momento determinado.

- Otro problema que se nos presentó fue el intentar ejecutar la orden ***source*** de SQL mediante cualquiera de los métodos ofrecidos por el conector de JDBC (*execute, executeQuery*). Esta orden carga en MySQL el fichero .sql que contenga las vistas que queramos depurar. Con cualquiera de los métodos mencionados anteriormente obteníamos un error y no conseguíamos ejecutar la orden con éxito. Tras documentarnos en varios foros, encontramos que *source* no es estándar de SQL, es específico de MySQL, entonces el conector no reconocía como válida la orden. La solución fue simple, almacenar las vistas en una estructura local y ejecutarlas una a una en lugar de hacerlo con la orden *source*.

- Con la **parte gráfica** también encontramos algunos problemas:

  Para mostrar el árbol de vistas se ha usado el componente de Java *JTree*. Esta estructura tenía una serie de limitaciones:

  - Hemos tenido que crear un modelo propio que permitiera escuchar un evento sobre el árbol, esto es necesario para poder saber cuando el usuario ha pinchado sobre un nodo, saber qué nodo concreto es y poder llevar a cabo las acciones pertinentes sobre él.
  - Como usamos diferentes iconos para mostrar el estado del nodo, hemos tenido que ampliar el *DefaultTreeCellRenderer* del árbol para que permitiera mostrar iconos en el nodo.

  Al escoger la estrategia *Top Down*, se muestra una tabla con los hijos del nodo en el que nos encontramos y un combo para elegir el estado de cada hijo. La celda del

componente Java *JTable* no muestra por defecto nada que no sea del tipo String, con lo cual hemos tenido que definir un nuevo modelo para ella que extendiera al modelo abstracto y que mostrara un combo en la segunda columna de cada fila.

También hemos tenido que extender el *JDesktopPane* para poder añadir a la ventana principal barra de *scroll* en caso necesario.

- Otro problema fue adaptar el **JSQLParser** a las necesidades de nuestro proyecto:

JSQLParser está formado por dos módulos fundamentales: el analizador léxico y el analizador sintáctico.

El analizador léxico contiene un listado de palabras reservadas y es capaz de detectarlas, clasificar identificadores y finalmente devolver un listado de tokens (cada tokens se representa internamente por un número entero positivo). De esa forma, el analizador sintáctico, recorre el listado de tokens, generando así la estructura de clases resultante.

Para ampliar el léxico y reglas sintácticas en este analizador (reconocimiento de nuevos tokens como “view” por ejemplo), es necesario ampliar y modificar el flujo de análisis en ambos módulos. En el analizador léxico es necesario, reservar la cadena de caracteres deseada (“view” por ejemplo) lo que conlleva tener que programar el reconocimiento carácter a carácter de la misma. No es tarea sencilla, ya que hay cadenas que son subcadenas de otras, con lo que hay que tener mucho cuidado evitando solapamientos, con el fin de que se reconozcan todas las palabras unívocamente.

Para hacer más accesible la ampliación del léxico, hemos reprogramado el control de solapamiento entre cadenas para las palabras reservadas. En la versión inicial de JSQLParser es sencillo cometer un error a la hora de añadir nuevas palabras. Utilizando un método más sencillo basado en testigos, hemos simplificado este tipo de ampliaciones que pueden servir para futuras versiones del sistema.

El sistema de testigos tiene el objetivo de ir reconociendo las palabras reservadas carácter a carácter y funciona de la siguiente manera para cada palabra reservada:

- Existe una variable entera que se va incrementando y va indicando cuantos caracteres de la palabra reservada se han encontrado.
- Si en algún momento se encuentra un carácter no esperado se reinicia el testigo y se indica que no es una palabra reservada.
- Si se llega al final del testigo, el sistema (al igual que hacía en la versión anterior) devuelve el identificador de esa palabra reservada para su posterior procesamiento sintáctico.

Acto seguido, para poder generar una estructura de clases lógica, hay que generar en primer lugar la subestructura de clases capaz de envolver al nuevo tipo de sentencias que a partir del paso anterior ya son reconocidas ("create view "en nuestro ejemplo) y modificar el flujo de análisis del analizador sintáctico asociando la generación de esa estructura cuando la nueva palabra reservada se reconoce.

La primera dificultad surge debido a que la versión existente está limitada en cuanto a las operaciones SQL que se pueden realizar. Principalmente, en la versión inicial del parser es imposible el reconocimiento de vistas SQL. Por ello ha sido necesaria su modificación para adaptarlo a las necesidades de nuestro proyecto, reutilizando la codificación dada para tablas.

El primer paso fue ampliar el analizador léxico para que reconociera como palabra reservada "*Create View*", posteriormente para hacer más potente el analizador aumentando el número de sentencias que es capaz de reconocer, se incorporaron al léxico : *union*, *intersection* y *minus*.

La mayor complejidad al modificar el JSQLParser es la comprensión inicial del código del analizador léxico. Al principio el código resulta incomprensible, pero poco a poco, basándonos en los ejemplos de su página web (http://jsqlparser.sourceforge.net/), los problemas se fueron solventando. Una vez comprendido, proseguimos con una batería de pruebas de diferentes sentencias SQL, para conocer cuales era capaz de analizar sintácticamente el código disponible y que ampliaciones harían falta para abarcar las características de nuestro proyecto y dotarlo de más potencia léxica.

En referencia a la inclusión de *union*, *intersection* y *minus*, se hizo de tal manera, que el análisis sintáctico ofrece información adicional, indicando si la sentencia en cuestión es de tipo "UNION","INTERSECTION","MINUS" o "BÁSICA".

- Por último, quizá el problema que más tiempo nos costó solucionar fue el de intentar abrir **varios árboles en la misma ejecución** del programa. Tras muchas ejecuciones en modo paso a paso y tras explorar los valores de las variables, descubrimos que una de ellas no se inicializaba correctamente, causando que el programa no respondiera y evitando la opción de abrir un segundo árbol que sustituyera al que se encontraba en curso.

5. Reparto de tareas

Hemos repartido las tareas, basándonos en el patrón de arquitectura del software denominado modelo *Vista Controlador*, separando los datos de la aplicación, la interfaz del usuario y la lógica de control en tres componentes diferenciados.

El modelo es la representación de la información con la cual el sistema opera. En nuestro proyecto, los datos se gestionan mediante un Sistema Gestor de Bases de Datos. La vista presenta el modelo en un formato adecuado para interactuar, es decir, la interfaz de usuario. El controlador responde a eventos, concretamente a las acciones del usuario, e invoca peticiones al modelo y a la vista.

El flujo que sigue el control generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de alguna forma (por ejemplo, el usuario pulsa un botón del menú para realizar alguna acción)
2. El controlador recibe (por parte de los objetos de la vista) la notificación de la acción solicitada por el usuario. El controlador gestiona el evento que llega.
3. El controlador accede al modelo, actualizándolo, posiblemente modificándolo de forma adecuada a la acción solicitada por el usuario.
4. El controlador delega a los objetos de la vista la tarea de desplegar la interfaz de usuario. La vista obtiene sus datos del modelo para generar la interfaz apropiada para el usuario donde se reflejan los cambios en el modelo. El modelo no debe tener conocimiento directo sobre la vista. El controlador no pasa objetos de dominio (el modelo) a la vista aunque puede dar la orden a la vista para que se actualice.
5. La interfaz queda a la espera de nuevas interacciones del usuario, comenzando el ciclo nuevamente.

El modelo encapsularía la parte en la que los datos de entrada, es decir, el archivo de sentencias SQL (las vistas) se analiza sintácticamente y como consecuencia de este proceso se crea el árbol de dependencias entre las diferentes vistas, que son los nodos del árbol y las tablas que aparecen como hojas del mismo.

El control, que es la clase principal del proyecto, es el encargado de pedir al resto de clases, los datos que necesite. Así mismo, se encarga de llamar al formulario, el cual pide los datos necesarios al usuario para poder conectarse a la base de datos, y una vez que estos datos han sido verificados, el usuario podrá seleccionar un archivo de vistas, el que se quiera depurar, el modelo lo analizará sintácticamente y a continuación se mostrará el resultado en forma de árbol en la interfaz gráfica, es decir, estos datos son accesibles desde la vista. El nodo raíz es la vista principal y los hijos son las vistas referenciadas por otras. A partir de este momento, el usuario interactúa con

la interfaz gráfica cambiando el estado de los nodos, o eligiendo diferentes algoritmos para recorrer el árbol.

La ampliación de JSQLParser y la generación del árbol de vistas como resultado del análisis sintáctico de las mismas han sido realizadas por María Fernández. La parte del control, que enlaza la parte del análisis sintáctico con la interfaz, la conexión con la base de datos y el algoritmo *Divide & Query* han sido realizados por Laura Fernández. Y por último, la interfaz gráfica y el algoritmo *Top Down* y depuración básica han sido realizados por Aleixa Helguera.

La asignación del trabajo se produjo en la primera reunión que tuvimos con nuestro director de proyecto, en Noviembre del pasado año. Desde ese momento, se fueron planteando objetivos para cada reunión que hacíamos todos los lunes de cada semana. Se nos proponían tareas a cada una de nosotras y en la siguiente reunión se probaban y se proponían nuevos objetivos para la semana siguiente.

Hemos cumplido con bastante éxito todos los plazos establecidos y con el trabajo semanal, lo que nos ha permitido terminar el código del proyecto a mediados del mes de Marzo, gracias en gran parte a la constancia de nuestro director, que nos ha guiado cuidadosamente en todo este proceso.

Desde esa fecha, tras una semana sin reunión, comenzamos a trabajar en esta memoria, y nuestro nuevo objetivo es completarla a finales del mes de Mayo.

6. Conclusiones y trabajo futuro

Al comienzo de este trabajo nos planteamos el problema de la depuración de vistas correlacionadas en SQL. Este objetivo se ha alcanzado mediante la implementación de un sistema que utiliza la técnica de la depuración declarativa aplicada a SQL. El prototipo permite cargar ficheros fuente conteniendo las vistas a depurar, y construye un árbol que puede ser navegado bien de forma manual o mediante alguna de las estrategias que incorpora. El sistema interacciona con la base de datos cuya instancia estamos considerando, lo que permite analizar las respuestas obtenidas por cada vista y comprobar su validez. Como resultado de este proceso la aplicación finaliza señalando una de las vistas consideradas como fuente del error.

Aunque la utilidad de la herramienta debe ser comprobada mediante el uso en entornos reales, nuestra experiencia indica que puede ser útil en aplicaciones que utilizan gran cantidad de vistas (por ejemplo para elaborar informes complejos).

La principal dificultad surge cuando se consideran vistas cuya respuesta puede constar de gran cantidad de filas; aunque la vista inicial puede tener asociada una respuesta "pequeña", las vistas intermedias a menudo generan cantidades de datos demasiado grandes para ser verificadas.

Posibles soluciones para este problema, que constituyen parte del trabajo futuro, pueden ser:

- Utilizar un generador de casos de prueba para SQL que genere tablas con el menor número de tuplas posible.
- Afinar el proceso de depuración distinguiendo distintos tipos de errores (respuestas perdidas, respuestas erróneas), lo que puede utilizarse para restringir el número de filas consideradas.

Otras extensiones para este proyecto, pueden ser:

- Dar soporte a más sistemas gestores para bases de datos, como Oracle, SQLServer, Access, etc.
- Ampliar la herramienta para aceptar expresiones, cláusulas, DML, DDL, etc. Que no acepta nuestro depurador actualmente.
- El punto anterior nos lleva a plantearnos la posibilidad de poder cambiar tanto la base de datos como el código de las vistas en tiempo real.
- Incluir la opción de deshacer varios pasos dados, opción *Undo.*
- Permitir depurar el lenguaje de programación incrustado en Oracle PL/SQL.
- Ampliar el control en el solapamiento de palabras reservadas en JSQLParser, siguiendo por ejemplo el sistema de testigos ya usado en este proyecto y explicado en el punto 4.2 [Problemas encontrados].

7. Bibliografía

R. Caballero. *Técnicas de diagnóstico y depuración declarativa para lenguajes lógico-funcionales*. Tesis Doctoral. Junio 2004.

Daum, Berthold. *Eclipse 3 para desarrolladores Java.* Anaya Multimedia-Anaya Interactiva, 07/2005.

G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method*. The Journal of Logic Programming 4(3):177{198, 1987.

J.W. Lloyd. *Declarative Error Diagnosis*. New Generation Computing 5(2):133-154,1987.

H. Nilsson y J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121-150, 1997.

H. Nilsson. *How to look busy while being lazy as ever: The implementation of a lazy functional debugger*. Journal of Functional Programming 11(6):629-671.

B. Pope y L. Naish, *Practical Aspects of Declarative Debugging in Haskell 98*, In Proc. PPDP03, ACM Press, 230-240, 2003.

E.Y.Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Mass.,1982.

A. Tessier y G. Ferrand. *Declarative Diagnosis in the CLP Scheme*. En P. Deransart, M. Hermenegildo y J. MaÃluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming, Chapter 5, 151-174. Springer LNCS 1870, 2000.

Páginas de interés

Página de JSQLParser: http://jsqlparser.sourceforge.net/

Ayuda para interfaz gráfica: http://chuwiki.chuidiang.org/

Para información sobre elementos como Hibernate, los distintos algoritmos, Java, MySQL, etc: http://www.wikipedia.org/