Universidad de Valladolid

E. T. S. DE INGENIERÍA INFORMÁTICA

Ingeniería Técnica en Informática de Gestión

Análisis de Jerarquías de Herencia en Java mediante ACF

Alumnos: Luis David Barrios Alfonso

Alberto San Martín López

Tutora: Yania Crespo González-Carvajal

Agradecimientos

Agradecemos la ayuda recibida por todas las personas que han colaborado de alguna forma en el desarrollo de este proyecto.

A nuestra tutora Yania Crespo por su inestimable ayuda y continuo interés, en el desarrollo del proyecto.

A Carlos Enrique Cuesta por su ayuda, facilitándonos material y orientación en los primeros pasos de la evolución de este proyecto.

A Félix Prieto por sus aportaciones sobre el análisis de contextos formales, y su ayuda en la compresión de dicho tema.

A nuestras familias, por su continuo apoyo y sacrificio, porque nunca perdieron la fe en nosotros.

Y por último, y no por ello menos importante a nuestros amigos, por no sólo estar en los buenos momentos, sino también aguantarnos cuando ni nosotros mismos nos aguantamos.

Resumen

El presente trabajo implementa una propuesta metodológica desarrollada dentro del marco del desarrollo de *frameworks de dominio* a través de técnicas de *Análisis de Conceptos Formales* (ACF), concretamente dentro de la primera fase de este proceso, análisis de herencia, mediante la utilización de una técnica matemática de organización de la información, ACF. Esta técnica nos permitirá representar la información obtenida en dicho análisis de una manera precisa, y a partir de ella obtener una representación conceptual adecuada que permita una interpretación de las posibles relaciones de herencia.

Para demostrar los resultados y la utilidad que se pueden obtener de la propuesta, así como discutir las limitaciones y errores que pudiera poseer, se aplicará a una serie de clases *Java* proporcionadas por el usuario final, las cuales después de la aplicación de una herramienta que obtenga la información necesaria para poder obtener una representación conceptual adecuada y de una forma automática, nos permite posteriormente interpretar los resultados obtenidos en dicha representación.

Abstract

The current study implements a methodological proposal, developed within the background of domain's *framework* developing through *Formal Concepts Analisys* (FCA), more exactly, inside of the first phase of this process, inheritance analisys, through the use of techniques of mathematical information arrangement, FCA. This technique allow us to illustrate the obteined information in these analisys in an exactly way and obtein an appropriate conceptual schema, which allow us to make an interpretation of the several inheritance relationships.

According to demonstrate the results and usefulness that can be obtained of such proposal, both to discuss its limits and errors which there woud be, the technique will be use over a *Java* classes' collection which are introduced by the final user. After using a tool which have obtein the necessary information to represent the appropriate conceptual schema in an automatic manner, we can interpretate the results of that representation.

Índice General

1 Introducción

1.1 Introducción y Motivación

Cada *Sistema de Información* (SI, *System Information*), ha sido analizado, desarrollado y mantenido para implementar de forma completa todas sus especificaciones técnicas y funcionales. Cada uno de sus componentes ha sido descrito en detalle y la forma en que ha sido desarrollado tiene una clara motivación. Como consecuencia, corregir errores, introducir nuevas funcionalidades, cambiar la arquitectura, migrar partes hacia otra plataforma, construir interfaces para la conexión con sistemas externos... son, por así decirlo, un *juego de niños*. Ésta sería una visión ideal de la situación de forma que, el término *Sistema Legado* debe traer a nuestra mente un sentimiento de admiración y respeto hacia los experimentados desarrolladores cuyas habilidades nos han dejado bellas piezas de ingeniería.

Desgraciadamente, este es un punto de vista poco realista y compartido:

Un sistema de información legado es un sistema de información que, de forma significativa, resiste modificaciones y cambios. Típicamente un sistema de información legado es enorme, con millones de líneas de código y más de 10 años de edad. [Brodie 1995]

La mayor parte de las tareas que hay que realizar durante el mantenimiento y la evolución de un sistema *software* se producen porque es necesario soportar los nuevos requisitos que aparecen en las organizaciones o porque se han de modificar los existentes. Si el sistema mantiene y procesa gran cantidad de datos persistentes, una cuestión que debe abordarse es determinar qué información debe ser almacenada y mantenida y cómo esta información puede ser utilizada en los diferentes contextos que se pueden presentar, según varían los requisitos. Así como que partes pueden ser reutilizadas y cuales no.

Sin embargo nos encontramos muchas veces con el problema de a que nivel podríamos reutilizar partes de un sistema, es decir que elementos podrían considerarse apropiados para garantizar un grado de reutilización adecuado. La reutilización ha sido uno de los objetivos de la Ingeniería del Software desde sus orígenes, y mejorar las posibilidades de reutilización fue uno de los acicates para la creación de las técnicas orientadas a objetos. No obstante, y a pesar de las evidentes ventajas que aporta en este sentido, parece ya claro que la clase constituye un elemento reutilizable (asset) de grano demasiado fino.

Para solventar esta dificultad se han propuesto técnicas de desarrollo basadas en assets de grano más grueso, como los *frameworks*, que han proporcionado éxitos notables, especialmente en el ámbito de las interfaces gráficas de usuario, donde estas técnicas tuvieron origen. Existen diferentes procesos en la construcción de dichos *frameworks* pero nosotros nos centraremos en [Prieto, Crespo, Marques y Laguna, 2003], donde se describe detalladamente un proceso de construcción de *frameworks de dominio* a través del uso de técnicas de *Análisis de Conceptos Formales* (ACF). En dicho proceso donde se combinan varias técnicas nos interesaremos por la primera de todas ellas donde se realiza un análisis de las jerarquías de herencia.

Puesto que no puede hacerse ningún cambio a un *Sistema de Información* antes de que hayamos obtenido un conocimiento detallado y preciso de sus aspectos funcionales y técnicos, existe una fuerte necesidad (de una forma científica, es decir, estricta y formal), de obtener algún tipo de información que aporte ideas sobre como está diseñado dicho sistema, para evolucionarlo en un mundo cuya complejidad tecnológica cambia constantemente.

1.2 Objetivos

Es por esto por lo que el presente proyecto fin de carrera se introduce dentro del proceso de desarrollo de *frameworks de dominio* a través de técnicas ACF, como un punto que se integra dentro del análisis de las jerarquías de herencia con objetivo de detectar las posibles relaciones de herencia en un código objeto de estudio aportado por el usuario final. Junto con las otras dos técnicas, análisis de clientela y análisis de dependencias funcionales, permite obtener toda la información necesaria para el proceso de desarrollo de *frameworks de dominio* planteado en [Prieto, Crespo, Marques y Laguna, 2003].

Se realizará un estudio de dichas jerarquías de herencia no sólo detectando las posibles relaciones que podrían darse dentro de un esquema de clases o interfaces sino también las ya existentes en dicho esquema, que junto con las otras dos técnicas de las que hablaremos más adelante (pero que no se aplicarán aquí pues nosotros sólo analizamos las jerarquías de herencia), se refinaran los resultados obtenidos en el desarrollo del *framework*. Con el análisis de las relaciones de herencia se consigue poner de manifiesto los aspectos comunes a un conjunto de aplicaciones iniciales del dominio, aspectos que deberemos recoger en el núcleo del *framework* en su primera versión, y separarlos de los elementos específicos de cada aplicación concreta. De forma que a través de una interpretación de la representación conceptual de las relaciones de herencia obtendremos dichos aspectos comunes.

Para ayudarnos en el proceso del análisis de las jerarquías de herencia dentro del marco de desarrollo de *frameworks* de dominio, se presentará una propuesta que intenta responder a algunas de las tareas que se han de realizar en dicho proceso, principalmente en la fase de *Representación Conceptual y Representación de la información*. Las principales tareas son: el análisis del código aportado por el usuario de donde se sacara toda la información relativa a la herencia, el análisis de esos datos presentándolos en una estructura donde quede bien definida las relaciones existentes entre las clases y su contenido con relación a la herencia, y la posterior representación conceptual de dichas relaciones con cuya interpretación lograremos tener una idea clara de las jerarquías de herencia existentes en dicho código y de las posibles que podría haber.

Para el análisis del código aportado será necesario el desarrollo de una herramienta o reconocedor de símbolos que a través de las expresiones regulares y de una gramática dada reconozca los elementos necesarios para el análisis de las relaciones de herencia. Dicha herramienta, llamada reconocedor o parser nos aportará información que deberemos tratar mediante técnicas matemáticas para el obtener dichas relaciones y poder interpretarlas.

La propuesta a la que hacíamos mención antes, consiste en abordar la consecución de dichas tareas aplicando técnicas de *Análisis de Conceptos Formales* (ACF), técnicas matemáticas de organización de la información. Las ventajas del uso de este enfoque se pueden expresar diciendo que es posible construir una herramienta, basada en ACF, automática, gráfica e interactiva. Esta herramienta va a hacer posible el soporte a la representación de la información obtenida en el análisis de los datos obtenidos a través del parser. Se va a poder marcar la información relevante para el análisis de las jerarquías de herencia en concordancia con las clases analizadas. En este trabajo se presenta de forma detallada el uso de ACF en la fase de r*epresentación gráfica conceptual* de los datos obtenidos viendo en que orden y de que manera se relacionan dichos datos determinando hasta dónde esta técnica es capaz de ayudarnos, y para ayudar a este fin, se construirá una herramienta que implemente este proceso (en concreto en la construcción del esquema conceptual donde ver las relaciones de herencia posible y reales) y que sirva de marco general y soporte para el resto de tareas.

Para finalizar, se empleará esta herramienta al estudio de las relaciones de herencia en las conjunto de clases facilitado por el usuario, que nos permitirá demostrar la utilidad de la aplicación de la técnica de ACF y las dificultades que el análisis de las jerarquías de herencia dentro del proceso de construcción de *frameworks de dominio* depara, pero que justifican sobradamente la investigación de nuevas técnicas y la construcción de nuevas herramientas que faciliten y sirvan de soporte a dicho proceso.

1.3 Marco de Trabajo

El método de *Análisis de Conceptos Formales* se utiliza principalmente en el análisis de datos, por ejemplo para investigación y procesamiento explicito de información. Tales datos serán estructurados en unidades, las cuales son abstracciones formales de conceptos del pensamiento humano permitiendo su pleno significado y una interpretación comprensible. Es por ello que esta técnica puede utilizarse dentro de numerosos campos como una técnica de extracción de conocimiento. Debido a que uno de los objetivos de este trabajo es la aplicación de la técnica de ACF dentro del marco del *Análisis de jerarquías de herencia*, inicialmente se partió de un proyecto anterior [DBRE., 2005] que utilizaba esta técnica dentro de un marco diferente. A su vez este proyecto se basaba en otro al que antes hemos referenciado y comparte ciertas similitudes con el nuestro [Pérez, 2002]. La aplicación final de este proyecto fue construida bajo un entorno *Linux*, en lenguaje *Eiffel*. El núcleo del programa era el encargado de la representación de las estructuras de datos y de los algoritmos (concretamente el algoritmo de *Bordat*) necesarios para aplicar la técnica de ACF. Por ello el objetivo inicial era la extracción y compresión de este núcleo como base de la futura aplicación del presente proyecto. A pesar de las ventajas que ofrece la utilización de software libre, este núcleo era demasiado dependiente de la plataforma y de las librerías utilizadas para su construcción; una restricción demasiado fuerte para una aplicación cuyas pretensiones son las de convertirse en un *framework* de trabajo para la aplicación de técnicas de ACF. Es por eso por lo que se optó por utilizar el proyecto de [DBRE, 2005] ya que aunque funcionalmente no comparte los mismo objetivos que el nuestro, también aplica las técnicas ACF del mismo modo que el proyecto de [Pérez, 2002] con lo que al fin y al cabo guarda similitudes en ese aspecto con el nuestro. Dicho proyecto, [DBRE, 2005], ha sido desarrollado enteramente en *Java* que es el mismo lenguaje con el que hemos elegido para la construcción del presente proyecto, debido a su enorme portabilidad y a que el código objeto de estudio estaría en *Java* se opto por dicho lenguaje. En principio se planteó realizar una migración de código *Eiffel* del proyecto de [Pérez, 2002], sin embargo debido a las diferencias que presentaban ambos lenguajes, se decidió por utilizar métodos nativos. Sin embargo no fue necesario debido a la aparición el año pasado por estas fechas del proyecto [DBRE, 2005] de forma que se facilitaron las cosas teniendo únicamente que reutilizar de una forma adecuada el núcleo de dicho proyecto adaptado a nuestro trabajo, pues las funcionalidades de uno y otro eran distintas. Todo esto se detalla en el capítulo 4, donde se habla de ello de una forma más extensa.

1.4 Definiciones, Acrónimos y Abreviaturas

A continuación daremos algunas definiciones y acrónimos de términos que se utilizarán de forma asidua en el desarrollo de la explicación de los diversos capítulos, de modo que permitan al lector una mejor comprensión de los mismos.

ACF (*Análisis de Conceptos Formales*): Técnica matemática de organización de la información. Este método es usado principalmente para el *análisis* de datos, por ejemplo para investigación y procesamiento explicito de información. Tales datos serán estructurados en

unidades, las cuales son abstracciones formales de conceptos del pensamiento humano permitiendo su pleno significado y una interpretación comprensible.

Framework: El *framework* puede ser definido como un conjunto de clases, generalmente algunas de ellas abstractas, y las colaboraciones que se establecen entre ellas, para proporcionar un diseño abstracto de las soluciones de un conjunto de problemas. [Johnson y B. Foot, 1988] y [Johnson y Russo, 1991].

Algoritmo de Bordat: Algoritmo aplicado en el presente proyecto para la obtención del retículo de *Galois* de la relación binaria de la matriz de contexto obtenida a través de las técnicas de ACF. Fue planteado en 1986 [Bordat, 1986] y construye el diagrama de *Hasse*. La idea del algoritmo es construir un retículo, G, comenzando con el ínfimo del conjunto, (E, f(E)), y a partir de ahí iremos generando todos los hijos que se añaden al retículo, G, y enlazados con sus padres. Este proceso de generación de hijos es repetido iterativamente para cada nuevo par de conceptos. Para cualquier referencia sobre el algoritmo nos remitimos a [Bordat 1986] o a [Pérez, 2002].

Retículo de Galois: En matemáticas, un **retículo**, **red** o **lattice** es un conjunto parcialmente ordenado en el cual todo subconjunto finito no vacío tiene un supremo y un ínfimo. El término "retículo" viene de la forma de los diagramas de *Hasse* de tales órdenes.

Diagrama de Hasse: es un cuadro simple de un conjunto parcialmente ordenado finito. Una arista dice de dos miembros x e y de un conjunto parcialmente ordenado S que "*y sigue a x*" si "$x \leq y$" y no hay elemento de S entre x e y. El orden parcial es entonces precisamente la clausura transitiva de la *relación de seguir*. El diagrama de *Hasse* de S se puede entonces definir abstractamente como el conjunto de todos los pares ordenados (x, y) tales que y sigue a x, es decir, el diagrama de *Hasse* se puede identificar con la *relación de seguir*. Concretamente, uno representa a cada miembro de S como un punto negro en la página y dibuja una línea que vaya hacia arriba de x a y si y sigue a x.

VCG freeBSD: La herramienta VCG lee una especificación textual y legible de un grafo y la visualiza. Si no han sido fijadas todas las posiciones de los nodos, las capas de de la herramienta del grafo utilizan una serie de heurística para reducir el numero de cruces, minimizando el tamaño de las aristas, centrando los nodos. La especificación del lenguaje de la herramienta VCG es cercanamente compatible con GRL, el lenguaje de la herramienta de las aristas, pero contiene muchas extensiones. La herramienta VCG te permite plegar dinámica o estáticamente regiones específicas del grafo. Trabaja sobre un entorno *UNIX/Linux* y está migrada de un entorno *Windows* 3.11.

Eiffel: Es un lenguaje de programación orientado a objetos, ideado en 1985 por *Bertrand Meyer*. Es un lenguaje centrado en la construcción de software robusto. Su sintaxis es parecida a la del lenguaje de programación Pascal. Una característica que lo distingue del resto de los lenguajes es que permite el diseño por contrato desde la base, con precondiciones, poscondiciones, invariantes y variantes de bucle, invariantes de clase y asertos. *Eiffel* es un lenguaje con tipos fuertes, pero relajado por herencia. Implementa administración automática de memoria, generalmente mediante algoritmos de recolección de basura. Las claves de este lenguaje están recogidas en el libro de *Meyer*, *Construcción de Software Orientado a Objetos*.

Java: es un lenguaje de programación orientado a objetos desarrollado por James Gosling y sus compañeros de *Sun Microsystems* al inicio de la década de 1990. A diferencia de los lenguajes de programación convencionales, que generalmente están diseñados para ser compilados a código nativo, Java es compilado en un bytecode que es ejecutado (usando normalmente un compilador *JIT*), por una máquina virtual *Java*. El lenguaje en sí mismo toma mucha de su sintaxis

de *C* y *C++*, pero tiene un modelo de objetos mucho más simple y elimina herramientas de bajo nivel como punteros.

Compilación JIT (*Just In Time*): la compilación en tiempo de ejecución (también conocida por sus siglas inglesas, *JIT*, *just-in-time*), también conocida como traducción dinámica, es una técnica para mejorar el rendimiento de sistemas de programación que compilan a *bytecode*, consistente en traducir el *bytecode* a código máquina nativo en tiempo de ejecución. La compilación en tiempo de ejecución se construye a partir de dos ideas anteriores relacionadas con los entornos de ejecución: la compilación a *bytecode* y la compilación dinámica.

JFC (Java Foundation Class): son parte de la API de Java compuesto por clases que sirven para crear interfaces gráficas de usuario para aplicaciones y *applets* de Java. Incluya los componentes Swing, soporte de aspecto y comportamiento conectable ("*look & Feel*"), API de accesibilidad, el API Java2D (sólo desde la JDK 1.2), y soporte "*Drag and Drop*" que proporciona la habilidad de arrastrar y soltar entre aplicaciones Java y aplicaciones nativas.

DTD (*Document Type Definition*): es una definición en un documento SGML ó XML que específica restricciones en la estructura del mismo. El DTD puede ser incluido dentro del archivo del documento, pero normalmente se almacena en un fichero ASCII de texto separado. La sintaxis de los DTD's para SGML y XML es similar pero no idéntica. La definición de un DTD específica la sintaxis de una aplicación de SGML o XML, que puede ser un estándar ampliamente utilizado como XHTML o una aplicación local. Los DTDs son generalmente empleados para determinar la estructura de un documento XML o SGML. Un DTD describirá típicamente cada elemento admisible dentro del documento, los atributos posibles y (opcionalmente) los valores de atributo permitidos para cada elemento. Es más, describirá los anidamientos y ocurrencias de elementos.

XML (*eXtensible Markup Language*): es un lenguaje extensible de etiquetas desarrollado por el World Wide Web Consortium (*W3C*). Es una simplificación y adaptación del SGML ("*Standard Generalized Markup Language"*) y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML). Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. XML no ha nacido sólo para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable. XML es una tecnología sencilla que tiene a su alrededor otras que la complementan y la hacen mucho más grande y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil.

UP (United Process): proceso de desarrollo de software y junto con el Lenguaje Unificado de Modelado UML, constituye la metodología estándar más utilizada para el análisis, implementación y documentación de un sistema software.

JavaCC: programa que genera un analizador sintáctico para *Java* a partir de una especificación léxica de la sintaxis del lenguaje *Java*. Similar al *Lex* y *Yacc*

1.5 Descripciones de Capítulos

A lo largo de este apartado vamos a ir haciendo un recorrido por los distintos capítulos que contendrán la documentación y explicaciones necesarias, para entender la evolución del desarrollo tanto del caso de estudio como de la herramienta final utilizada en él.

En el **Capítulo 2, *ACF como Apoyo en la Construcción de Frameworks de Dominio***, se introducen los conceptos básicos y definiciones del proceso de desarrollo de *frameworks* de dominio mediante las técnicas de ACF, marco donde encajaría nuestro proyecto, siendo una etapa del desarrollo de *frameworks* de dominio, así como las motivaciones que hicieron posible su estudio y los problemas que plantea su utilización.

En el **Capítulo 3, *ACF aplicado al Análisis de Jerarquías de Herencia***, se ilustra una breve introducción al *Análisis de Conceptos Formales*, y se expone la propuesta acerca de cómo esta técnica se puede aplicar a los diferentes pasos del proceso en el análisis de la jerarquía de herencia, centrándonos en la etapa de conceptualización para mostrar los datos referentes a las relaciones entre clases y características y para la construcción del retículo de *Galois*.

En el **Capítulo 4, *Caso de estudio*,** se detallarán los pasos seguidas para la realización del caso de estudio, desde los orígenes en que era una vaga idea hasta su realización. Se expondrán los principales resultados acerca del proceso y la herramienta desarrollada así como las pautas para la interpretación de los resultados, pasando por todas las fases y etapas del proceso. Los resultados concretos se expondrán en un documento aparte que se adjunta con el presente trabajo.

En el **Capítulo 5, Plan de Proyecto Software,** comenzaremos con los documentos asociados a la construcción de la herramienta. En este capítulo se esboza la visión, los requisitos no funcionales de la aplicación así como una aproximación al plan de actuación en concordancia con el proceso unificado (UP).

En el **Capítulo 6, Documento de Análisis,** se detallan y organizan las distintas funcionalidades, desde el punto de vista del usuario, que presentará la herramienta construida utilizando UML como lenguaje de descripción y el UP como la metodología de desarrollo. Se realizarán los casos de uso como documentos de requisitos funcionales, así como los primeros Diagramas de Clases que nos muestren la organización conceptual de la aplicación.

En el **Capítulo 7, Documento de diseño,** se detallará la arquitectura planteada para la construcción de la herramienta así como todos los Diagramas de Clases detallados y todas las pautas a seguir para realizar inmediatamente la implementación de la herramienta. Sin olvidarnos de la realización de los casos de uso, así como el diseño de la interfaz.

En el **Capítulo 8, Documento de usuario**, se encuentran las guías destinadas al usuario con respecto a la instalación y manejo de la aplicación.

En el **Capítulo 9, Conclusiones y Líneas Futuras**, todo proyecto debe llevar a un conjunto de conclusiones recopiladas a través de la experiencia del proceso de desarrollo de un producto, tanto positivas como negativas. Más allá de este conjunto de conclusiones se deben plantear posibles líneas de trabajo futuro y ampliaciones al trabajo realizado.

En el **Apéndice A, Contenido del CD-ROM**, se muestran los diferentes ficheros y su organización en el CD-ROM adjunto al presente trabajo.

2 ACF como Apoyo en la Construcción de Frameworks de Dominio

2.1 Introducción

En este capítulo veremos la relación existente entre nuestro proyecto fin de carrera, "Análisis de Jerarquías de Herencia en Java mediante ACF", con la construcción de *frameworks de dominio* con soporte del análisis de conceptos formales (ACF), y en concreto con la herramienta que hemos desarrollado, viendo como se integra en el proceso descrito en [Prieto, Crespo, Marques y Laguna, 2003].

El desarrollo de *frameworks de dominio* surge como respuesta a la necesidad de reutilización de software, ya que la gran mayoría de las nuevas aplicaciones y sistemas no son desarrolladas desde cero, sino al contrario, la gran mayoría suele apoyarse en programas ya existentes, siendo nuevas versiones, ampliaciones o modificaciones lo que en gran medida se realiza en la actividad de producción de software ya sea por necesidades del cliente, cambios en los requisitos iniciales etc. Como punto a destacar diremos que los sistemas que parten desde cero constituyen un mínimo porcentaje de todo el software desarrollado en el mundo.

La reutilización ha sido uno de los objetivos de la Ingeniería del Software desde sus orígenes, y mejorar las posibilidades de reutilización fue uno de los acicates para la creación de las técnicas orientadas a objetos. No obstante, y a pesar de las evidentes ventajas que aporta en este sentido, parece ya claro que la clase constituye un elemento reutilizable (*asset*) de grano demasiado fino como para garantizar niveles de reutilización adecuados [W-B y Johnson, 1990].

Para solventar esta dificultad se han propuesto técnicas de desarrollo basadas en *assets* de grano más grueso, como los *frameworks* [Johnson y B. Foot, 1988], que han proporcionado éxitos notables, especialmente en el ámbito de las interfaces gráficas de usuario, donde estas técnicas tuvieron origen.

Un ***framework*** puede ser definido como un conjunto de clases, generalmente algunas de ellas abstractas, y las colaboraciones que se establecen entre ellas, para proporcionar un diseño abstracto de las soluciones de un conjunto de problemas [Johnson y B. Foot, 1988] y [Johnson y Russo, 1991].

El *framework* captura las decisiones de diseño comunes a un tipo de aplicaciones, estableciendo un modelo común a todas ellas, asignando responsabilidades y estableciendo colaboraciones entre las clases que forman el modelo. Además, este modelo común contiene puntos de variabilidad, conocidos como puntos calientes[1] [Pree, 1995], capaces de albergar los distintos comportamientos de las aplicaciones representadas por el *framework*.

La reutilización se produce entonces en el proceso de instanciación del *framework*, en el que el desarrollador con reutilización proporciona la funcionalidad específica de su aplicación rellenando los puntos calientes.

[1] Puntos calientes, del inglés *hot spots*, también denominados *hooks*.

2.2 Clasificación de *Frameworks*

Aunque se han propuesto diferentes clasificaciones para caracterizar los tipos de *frameworks* [Johnson y B. Foot, 1988] , [Fayad y Schmidt, 1997], todas parecen coincidir en que se pueden distinguir dos tipos denominados *frameworks de aplicación* y *frameworks de dominio*:

- ***frameworks de aplicación**:* encapsula una capa de funcionalidad horizontal que puede ser aplicada en la construcción de una gran variedad de programas tales como *frameworks* que implementan las interfaces gráficas de usuario, otros dedicados al establecimiento de comunicaciones o procesamiento de documentos *XML* etc.
- ***frameworks de dominio:*** implementan una capa de funcionalidad vertical, correspondiéndose con un dominio de aplicación o una línea de producto. Estos deberán ser los más numerosos, y su evolución deberá ser también la más rápida, pues deben adaptarse a las áreas de negocio para las que están diseñados. En ellos será el ámbito donde se desenvuelva nuestra herramienta y donde nos centraremos en este capítulo.

También la forma en que se instancian sus puntos calientes permite clasificar los *frameworks*. Podemos distinguir *puntos calientes*, y por extensión de *frameworks*, *de caja blanca* y *puntos calientes de caja negra:*

- ***puntos calientes de caja blanca***: hablamos de ellos cuando su instanciación se realiza mediante técnicas de herencia, y por ello requiere de conocimientos sobre la implementación del *framework* por parte del desarrollador.

- ***puntos calientes de caja negra:*** se refiere a puntos calientes y *frameworks* en que la instanciación se realiza mediante composición e instanciación de parámetros genéricos. Su utilización es mucho más simple por cuanto sólo requiere seleccionar de entre un conjunto de opciones predefinidas. Sin embargo, y por los mismos motivos su construcción resulta mucho más difícil.

2.3 Ventajas e Inconvenientes

Todas las ventajas que se podrían esperar de la construcción de *frameworks* de dominio como modelos de reutilización provendrían del "reciclaje" del software tales como la reducción de los costes de desarrollo, el aumento de la calidad de los productos, el aumento de la productividad mediante la mejora de los tiempos en los que se desarrollan los nuevos proyectos informáticos, mejoras en las actividades de mantenimiento y soporte de aplicación, así como mejoras en las actividades de control y planificación por la reducción de desviaciones en los desarrollos. Sin embargo, a pesar de sus esperados beneficios, algunos obstáculos han impedido implantar, de manera exitosa y generalizada, modelos de reutilización basados en *frameworks*.

Las principales dificultades han venido dadas porque son difíciles de construir y no es sencillo aprender a utilizarlos. Prueba de ello es el buen número de trabajos dedicados a informar sobre la experiencia obtenida en la construcción y utilización de *frameworks*, como se puso de manifiesto en el "Electronic Symposium on Object-Oriented Application Frameworks" [Fayad, 2000].

Para comprender el origen de estas dificultades hay que tener en cuenta que la construcción de *frameworks* es un proceso puramente artesanal. Ello conduce a un desarrollo que requiere de grandes dosis de experiencia, es costoso y propenso a errores. Por ello, para generalizar la utilización de *frameworks* de dominio es necesario adoptar un enfoque más industrial, basado en estrategias y herramientas capaces de minimizar la inversión inicial necesaria y reducir los costes asociados al mantenimiento del *framework*.

En realidad esto no debería sorprendernos. Si diseñar un sistema es costoso, diseñar un sistema general reutilizable lo es mucho más. Un *framework* debe encerrar una teoría del dominio del problema y debería ser siempre el resultado de un análisis de dominio, sea explícito u oculto, formal o informal. El diseño de un sistema que cumpla sus requisitos y además encierre la solución para un amplio rango de problemas futuros, es un verdadero reto.

Debido a su coste y complejidad de desarrollo, los *frameworks* deberán ser construidos solamente cuando se advierte que muchas aplicaciones serán desarrolladas en un dominio específico, permitiendo que la inversión realizada en el desarrollo del *framework* se amortice al reutilizarlo. En resumen, por todas las razones anteriores, se estima que las situaciones en que es económicamente rentable afrontar la construcción de un *framework* son aquellas en que se dispone de varias aplicaciones del mismo dominio (o se deben producir en breve plazo) y se espera que se requieran nuevas aplicaciones con cierta frecuencia.

2.4 Un Proceso de Desarrollo de *Frameworks de Dominio*

Como hemos visto antes, los *frameworks* son aceptados como un *asset* de grano adecuado para fomentar el proceso de reutilización. Sin embargo, como hemos indicado, su éxito en la práctica ha sido bastante limitado fuera del ámbito de los *frameworks* de aplicación en que estas técnicas se originaron, y ello a pesar de que en estos días son escasos los sistemas completamente construidos desde cero.

Ello es debido como hemos visto no solo a su complejidad, tanto en el uso como en la fabricación sino también porque el *framework* no puede ser considerado como un producto final, sino que debe ser esencialmente evolutivo. Por esa razón se han propuesto varias estrategias para la fabricación de *frameworks* [Roberts y Johnson, 1997] y [Schmid y Johnson, 1999] que parten del desarrollo de una o varias aplicaciones del dominio, a partir de las cuales se inicia la construcción de un primer *framework*. Posteriormente, con la información proporcionada por las sucesivas instanciaciones, dicho *framework* va refinándose y transformándose.

Sin embargo, con este enfoque la información contenida en las aplicaciones ya desarrolladas se utiliza de un modo informal, como una forma de adquirir experiencia y conocimientos sobre el dominio, o simplemente como punto de partida para el desarrollo. Esto supone que buena parte de esa información puede ser utilizada de un modo insuficiente en un proceso que, en todo caso, requiere buenas dosis de habilidad y experiencia por parte del desarrollador, además de ser propenso a errores.

Y no sólo eso, tampoco con este enfoque se da un soporte adecuado para el posterior proceso de evolución de los *frameworks* de dominio. Los *frameworks* comparten con el resto de los artefactos software la necesidad de evolucionar para adaptarse a los cambios de la realidad que pretenden modelar. En este sentido las mismas técnicas aplicables al resto del software son susceptibles de utilización en el caso de los *frameworks*. Sin embargo en el caso de los *frameworks* surgen necesidades específicas relacionadas con la característica distintiva de éstos: su capacidad

para recoger el modelo común a un conjunto de aplicaciones que pretenden dar solución a un grupo de problemas relacionados. Es preciso entonces abordar la necesidad de un *framework* de evolucionar conforme al modelo de construcción evolutiva, ampliamente admitido en la actualidad, para mejorar su capacidad de ser instanciado en aplicaciones, y esta evolución debe ser planteada en dos dimensiones distintas:

- Necesidad de ampliar la capacidad de instanciación del *framework*.
- Necesidad de facilitar las instanciaciones que ya son posibles.

La primera versión del *framework* no es sino un primer diseño que debe ser ampliamente mejorado para ir adaptándose a las nuevas necesidades detectadas durante su utilización, ya sea mediante la introducción de nuevos puntos calientes o la reubicación de los ya existentes. Este tipo de mejoras no se restringen a las primeras fases del desarrollo del *framework*, sino que deben aplicarse siempre que se detecten nuevas necesidades que deben ser cubiertas por el mismo.

Conforme se detecta que el diseño del dominio recogido en el *framework* va estabilizándose, el interés se desplaza a la forma en que los puntos calientes han sido implementados, puesto que los conocimientos adquiridos en el desarrollo permiten ya establecer de un modo más preciso cuáles son las diferentes implementaciones necesarias para esos puntos calientes. Esta información permite abordar la refactorización[2] del mismo para, mediante técnicas de caja negra, posibilitar su instanciación simplemente mediante la elección entre un conjunto de opciones predefinidas.

Partiendo de esas ideas se propone en [Prieto, Crespo, Marques y Laguna, 2000] y en [Prieto, Crespo, Marques y Laguna, 2003] una estrategia de construcción de *frameworks* de dominio no tan "artesanal" y más sistemática, o como hemos dicho antes "industrial" de forma que aplicando técnicas de ACF se automaticen muchas de las tareas que son tan arduas y propensas a errores en el desarrollo de *frameworks*, así como la integración del modelo informal de construcción evolutiva a partir de las aplicaciones iniciales del dominio (con la utilización de herramientas basadas en técnicas ACF de representación del conocimiento).

Parte de la elaboración previa de un conjunto de aplicaciones del dominio. A diferencia de otras propuestas que utilizan estas aplicaciones como punto de partida para un proceso de generalización manual, o simplemente como medio informal de adquirir experiencia en el dominio en cuestión, en este proceso se apuesta por extraer los aspectos comunes a todas estas aplicaciones mediante herramientas automáticas basadas en técnicas formales. Esta primera versión del *framework* debe ser considerada como un producto intermedio, aunque ya utilizable, de un proceso más amplio que, también con la asistencia de herramientas adecuadas, debe contemplar el proceso completo de desarrollo del *framework*, con especial énfasis en su adaptación para responder a los cambiantes requisitos de las áreas de negocio en que estos artefactos software son finalmente utilizados.

En [Prieto, Crespo, Marques y Laguna, 2003] se propone un proceso de construcción de *frameworks* de dominio soportado por herramientas, que integra la estrategia anteriormente propuesta con las técnicas que facilitan la posterior evolución del *framework*, tanto para mejorar su capacidad de instanciar aplicaciones del dominio como para adaptarse a las variaciones que indudablemente han de producirse en los requisitos del mismo.

[2] Refactorización, del inglés *Refactoring*, es una forma especial de transformación de software Orientado a Objetos, que se caracteriza por no depender de la semántica del software a modificar, tener como objetivo refinar diseños y ser realizada mediante reestructuración y reorganización de clases y agrupaciones.

Dicha propuesta pasa por la utilización de herramientas basadas en el ACF, una técnica formal de representación del conocimiento que permite poner de manifiesto de forma automática la estructura subyacente en un conjunto de datos. La herramienta desarrollada en nuestro proyecto se integra en ese ámbito y podrá ser usada para la obtención de las relaciones de herencia como se verá más adelante.

El proceso consiste en que a partir de un proceso de desarrollo de *Frameworks de Dominio* como los que se describen en [Roberts y Johnson, 1997] o [Schmid y Johnson, 1999], introduciendo en la fase de construcción de la primera versión y en las posteriores fases de evolución, el uso de las técnicas basadas en el ACF para mecanizar la utilización de la información contenida en las aplicaciones iniciales y las instancias del *framework*.

Figura 2-1. *Diagrama del proceso de construcción completo.*

En una primera fase, las técnicas basadas en el ACF facilitan la detección de los aspectos comunes a un conjunto de aplicaciones iniciales del dominio, aspectos que se recogen en el núcleo del *framework*, y se separan de los elementos específicos de cada aplicación concreta. Estos elementos específicos dan lugar a los puntos de variabilidad que serán implementados mediante los puntos calientes. A esta forma de utilizar el ACF, propuesta inicialmente en [Godin y Mili, 1993], se denomina Análisis de Herencia.

En esta misma fase se aplica también otro tipo de análisis, que se denomina Análisis de Cliente, basado en el ACF, propuesto inicialmente en [Snelting y Tip, 1998]. Este análisis permite obtener una mejor composición de las clases mediante el reagrupamiento de sus métodos.

Para la posterior fase de evolución del *framework* se apuesta por utilizar la información disponible en las instancias del *framework* mediante herramientas y técnicas capaces de extraer los datos relevantes para guiar estos procesos de evolución. Las dos dimensiones de evolución que se ha mencionado anteriormente corresponderán con dos subprocesos diferenciados, y dos tipos de herramientas con objetivos distintos.

La necesidad de ampliar la capacidad de instanciación del *framework* se detecta cuando, al intentar instanciar una nueva aplicación, el modelo no presenta puntos de variabilidad suficientes para ello. La estrategia general consiste entonces en continuar la construcción de la aplicación, a partir de ese punto mediante técnicas convencionales, y aplicar posteriormente las técnicas basadas en ACF para analizar las relaciones de herencia y cliente. Con ello se obtienen nuevos puntos calientes, que implementados con técnicas de caja blanca, que proporcionarán nuevos puntos de variabilidad requeridos. De los nuevos puntos calientes dispondremos entonces de dos instancias

distintas, la que proporciona la nueva aplicación, y la correspondiente a las clases del *framework* que han salido de su núcleo.

El segundo tipo de proceso no debe ser disparado por una necesidad concreta, sino por la disponibilidad de un buen número de instancias del *framework.* Cuando la cantidad de instancias haga suponer que se dispone de representantes de un suficiente número de opciones de instanciación, podremos abordar su transformación hacia la tecnología de caja negra.

Un enfoque inocente podría suponer que basta con introducir las clases que instancian el punto caliente dentro del núcleo del *framework* como alternativas que deben ser seleccionadas mediante técnicas de composición, pero esa estrategia no es en absoluto suficiente.

Los puntos calientes de caja blanca pueden agrupar, y de hecho así suele ocurrir, varias dimensiones de variabilidad. Esto no es un problema, puesto que el desarrollador con reutilización debe comprender el punto caliente antes de instanciar su aplicación. Sin embargo es imprescindible que los puntos calientes de caja negra contengan una única dimensión de variabilidad, para que sus combinaciones se realicen en el desarrollo con reutilización, permitiendo así una mayor libertad en el proceso de instanciación y un menor número de opciones predefinidas.

Necesitamos, por lo tanto, herramientas y técnicas capaces de detectar los patrones de variación de las instancias del punto caliente, patrones que permitirán inferir qué métodos del punto caliente pueden ser agrupados en una dimensión de variabilidad. Para esto se utiliza también el ACF mediante una técnica que denominamos Análisis de Dependencias Funcionales.

De este modo podemos proponer la división del punto caliente en diferentes dimensiones de variabilidad, lo que redundará en una mayor facilidad para la posterior implementación mediante técnicas de caja negra.

En resumen, el proceso completo para el desarrollo de *frameworks* de dominio, descrito esquemáticamente en la Figura 1, se inicia a partir de un conjunto de aplicaciones del dominio sobre las que se aplican herramientas basadas en el ACF que analizan tanto las relaciones de herencia como las de cliente. Como consecuencia se obtienen las partes comunes de estas aplicaciones, que pasarán a formar el núcleo de la primera versión del *framework*, y los puntos de variabilidad que permitirán instanciar las distintas aplicaciones.

Esta primera versión del *framework* debe evolucionar en función de las necesidades detectadas en el posterior proceso de instanciación. Cuando en una de estas instanciaciones se descubran requisitos de variabilidad no contemplados en el *framework* se debe proceder a construir nuevas aplicaciones que implementen los requisitos solicitados y a unificarlas nuevamente con el *framework* disponible, mediante técnicas basadas en herencia y cliente, dando lugar a nuevos puntos calientes que dotan al *framework* de la variabilidad deseada.

Cuando el número de instancias disponibles de alguno de los puntos calientes del *framework* haga sospechar que todas sus opciones de variabilidad están ya implementadas, se debe proceder a su división en dimensiones independientes de variabilidad, aplicando en este caso las técnicas basadas en el análisis de dependencias funcionales.

3 ACF aplicado al Análisis de Jerarquías de Herencia

3.1 Introducción

El objetivo central del trabajo, como ya venimos diciendo, es el análisis de las jerarquías de herencia dentro de una colección de clases Java, a través de una herramienta que hemos desarrollado. En dicha herramienta, una vez que se ha efectuado la extracción de datos, aplica una serie de técnicas basadas en el *Análisis de Conceptos Formales* (ACF) [Ganter y Wille, 1999] , que nos permitirá obtener una visión sobre las relaciones existentes entre las clases de la colección.

Para ello, haremos uso fundamental del ACF, una técnica matemática de organización de la información utilizada en Ingeniería del Software y Representación del Conocimiento, que permite poner de manifiesto las abstracciones subyacentes en una tabla de datos, formalmente un contexto, mediante la construcción del retículo de conceptos, también conocido como retículo de *Galois*, asociado a ésta.

La actividad prevista del ACF en nuestro trabajo será:

1) En la fase de representación conceptual de los datos extraídos del código, de forma que mediante las características y atributos de las clases (atributos según notación ACF) y las clases (objetos en ACF) obtendremos la matriz de contexto donde aparecen las relaciones entre objetos y atributos.

2) En la fase de obtención y representación del retículo de *Galois* asociado al contexto; a través del algoritmo de *Bordat* extraeremos los conceptos formales gracias a los cuales podremos dibujar el retículo asociado al contexto como se vera mas adelante.

La técnica del ACF se ha aplicado con éxito en análisis de datos, recuperación de información y descubrimiento del conocimiento a partir de bases de datos [Stumme, Wille y Wille, 1988]. Como aplicación se ha desarrollado el programa TOSCANA [Vogt y Wille, 1995], que permite explorar de forma interactiva bases de datos. Por otra parte, el análisis formal de conceptos también ha sido objeto de estudio desde el campo de la deducción automática. Concretamente, el retículo de los conceptos de un contexto formal ha sido formalizado en Mizar [Schwarzweller, 2000].

A pesar de que el campo de aplicación del ACF es muy amplio, abarcando temas como la ingeniería inversa, reingeniería de bases de datos [DBRE, 2005], refactorización [Marticorena y Crespo, 2003], soporte a la construcción de *frameworks* [Prieto, Crespo, Marques y Laguna, 2003] etc. Nosotros nos hemos centrado en el análisis de las relaciones de herencia, que como se ha visto es uno de las piezas importantes en el puzzle que es la construcción de *frameworks*.

3.2 Análisis de Conceptos Formales

Antes de describir las soluciones tomadas en forma de algoritmos para implementar los puntos descritos en el apartado anterior realizaremos una pequeña introducción de la técnica matemática utilizada.

El Análisis de Conceptos Formales (AFC), introducido en 1982 por Rudolf Wille [Wille, 1982] y aparece totalmente desarrollado en [Ganter y Wille, 1999]. Es una técnica de aprendizaje capaz de extraer estructuras conceptuales de un conjunto de datos. Está basada en la idea filosófica de que un "concepto" consta de dos partes: su **extensión**, formada por todos los objetos que pertenecen a dicho concepto; y su **intensión**, que comprende todos los atributos compartidos por dichos objetos.

El marco en el que se establecen los conceptos se conoce como contexto formal. Consta de un conjunto de objetos, un conjunto de atributos o propiedades, y una relación que informa sobre los atributos que posee cada objeto. El conjunto de los conceptos de un contexto formal tiene estructura de retículo completo, lo que permite representarlos gráficamente como jerarquías conceptuales, posibilitando el análisis de estructuras complejas y descubriendo dependencias entre los datos.

La base del Análisis de Conceptos Formales es el estudio del retículo de *Galois* de la relación binaria entre el conjunto de objetos y el de atributos. Este retículo viene dado por la conexión de Galois[1] formada por dos aplicaciones entre conjuntos parcialmente ordenados, aplicaciones que son en cierto sentido "compatibles" con el orden definido en los conjuntos.

Definición 3.1 Llamaremos ***contexto formal*** C a una terna (O, A, I) donde O es un conjunto de objetos, A es un conjunto de atributos e $I \subseteq O \times A$ una relación binaria. Si $(o, a) \in I$ significa que el objeto o posee el atributo a (también lo notaremos por oIa).

En particular, consideraremos contextos formales finitos como contextos en los que los conjuntos de objetos y atributos son finitos y, además, el conjunto de atributos es no vacío. La relación binaria I facilita la *incidencia* del conjunto de atributos sobre el conjunto de objetos. Con ella podemos definir el siguiente par de aplicaciones, en cuyas definiciones recogemos las nociones de conjunto de atributos poseídos por ciertos objetos y conjunto de objetos que poseen ciertos atributos respectivamente:

Definición 3.2 Sea $\boldsymbol{X}$ un conjunto de objetos de un contexto $C = (O, A, I)$. La **intensión** de $\boldsymbol{X}$, que notamos por $\boldsymbol{X}^{\uparrow}$, es el conjunto de atributos comunes a todos los objetos de $\boldsymbol{X}$:

$$X^{\uparrow} = \{a \in A \mid oIa \ \ \forall\, o \in X\}$$

O lo que es lo mismo:

$$\varphi : \wp(O) \rightarrow \wp(A)$$
$$X \rightarrow X^{\uparrow} = \{a \in A \mid (o,a) \in I \ \ \forall\, o \in X\}$$

Definición 3.3 Sea $\boldsymbol{Y}$ un conjunto de atributos de un contexto $C = (O, A, I)$. La **extensión** de $\boldsymbol{Y}$, que notamos por $\boldsymbol{Y}^{\downarrow}$, es el conjunto de objetos que poseen todos los atributos de $\boldsymbol{Y}$:

$$Y^{\downarrow} = \{o \in O \mid oIa \ \forall\, a \in Y\}$$

[1] Evariste Galois fue el primero en analizar una situación de este tipo al estudiar la relación entre el conjunto de los cuerpos intermedios de una extensión de cuerpos E : F y el grupo de los automorfismos sobre E que fijan el subcuerpo F.

O lo que es lo mismo:

$$\psi : \wp(O) \rightarrow \wp(A)$$
$$Y \rightarrow Y^{\downarrow} = \{o \in O \mid (o,a) \in I \ \forall\, a \in Y\}$$

Una vez definidos los conjuntos de extensión e intensión definiremos una serie de propiedades que cumplen.

Proposición 3.4 Sea un contexto formal, $C = (O, A, I)$. Dados X, X_1, y X_2 conjuntos de objetos y Y, Y_1, y Y_2, conjuntos de atributos, se verifican las siguientes propiedades:

1. $X_1 \subseteq X_2 \Rightarrow X_2{}^{\uparrow} \subseteq X_1{}^{\uparrow}$
2. $X \subseteq X^{\uparrow\downarrow}$
3. $X^{\uparrow} = X^{\uparrow\downarrow\uparrow}$
4. $Y_1 \subseteq Y_2 \Rightarrow Y_2{}^{\downarrow} \subseteq Y_1{}^{\downarrow}$
5. $Y \subseteq Y^{\downarrow\uparrow}$
6. $Y^{\downarrow} = Y^{\downarrow\uparrow\downarrow}$
7. $X \subseteq Y^{\downarrow} \Leftrightarrow Y \subseteq X^{\uparrow} \Leftrightarrow X \times Y \subseteq I$

Usando estos operadores se establece la definición de concepto en un contexto formal como sigue:

Definición 3.5 Un **concepto** en un contexto formal $C = (O, A, I)$ es un par (X, Y) donde X es un conjunto de objetos de C, e Y es un conjunto de atributos de C, tales que $X^{\uparrow} = Y$ e $Y^{\downarrow} = X$. Denotaremos el conjunto de los conceptos formales asociado a $C = (O, A, I)$ por $\mathbf{G}(O, A, I)$.

Decimos que X e Y son la extensión y la intensión, respectivamente, del concepto (X, Y).

Veremos un ejemplo para dejar claro las definiciones vistas hasta ahora. Consideremos el conjunto de objetos {g: gato, s: sanguijuela, r: rana, m: maíz, p: pez} sobre los que se han observado las propiedades siguientes {N: necesita agua, A: es acuático, M: es móvil, P: tiene patas}, obteniéndose la relación dada por la siguiente tabla:

	N*ec. agua*	**A***cuático*	**M***óvil*	**P***atas*
g*ato*	**X**		**X**	**X**
s*anguijuela*	**X**	**X**	**X**	
r*ana*	**X**	**X**	**X**	**X**
m*aíz*	**X**			
p*ez*	**X**	**X**	**X**	

En nuestro ejemplo, $(\{g, s, r, m, p\}, \{N\})$ y $(\{s, r, p\}, \{N, A, M\})$ son conceptos, pero $(\varnothing, \{N, A, M, P\})$ no lo es, pues $\{N, A, M, P\}^{\downarrow} = \{r\} \neq \varnothing$. Observamos que el conjunto de conceptos de un contexto formal $C = (O, A, I)$ es no vacío, puesto que $(A^{\downarrow}, A^{\downarrow\uparrow})$ siempre es un concepto. Una afirmación similar se puede realizar sobre los subconjuntos de O respecto de las intenciones.

De forma que en nuestro ejemplo tendremos los siguientes conceptos:

({g, s, r, m, p}, {N})
({g ,s, r, p}, {N, M})
({g, r}, {N, M, P})
({s, r, p}, {N, A, M})

({r}, {N, A, M, P})

Los conceptos de un contexto pueden ser parcialmente ordenados de forma natural: un concepto *G*1 es "menor" que otro *G*2 cuando todos los objetos de *G*1 lo son también de *G*2.

En nuestro ejemplo, podemos visualizar la relación entre los conceptos del contexto mediante el siguiente diagrama:

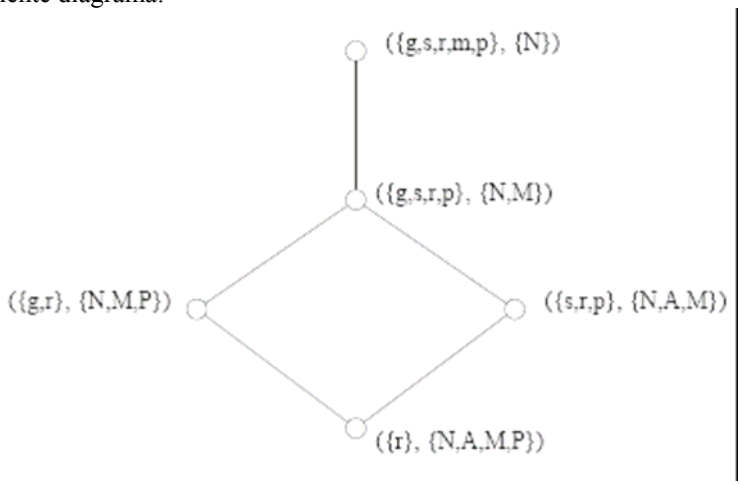

Figura 3-1. *Retículo de conceptos.*

Ahora definiremos la relación de orden entre los conceptos de un contexto formal y probaremos que el conjunto de los conceptos G(*O, A, I*), junto con esta relación tiene estructura de retículo completo.

Sobre **G**(*O*, *A*, *I*) podemos definir la relación de orden parcial mediante la siguiente fórmula en que (X_1, Y_1), (X_2; Y_2) ∈ **G**(*O*, *A*, *I*)**:**

Definición 3.6 *Sean* (X_1, Y_1) *y* (X_2, Y_2) *conceptos del contexto formal C* = (*O, A, I*)*. El concepto* (X_1, Y_1) *es subconcepto de* (X_2, Y_2)*, y se representa por:*

$$(X_1, Y_1) \leq (X_2, Y_2) \Leftrightarrow X_1 \subseteq X_2$$

$$(X_1, Y_1) \leq (X_2, Y_2) \Leftrightarrow Y_2 \subseteq Y_1$$

Para probar que G(*O, A, I*) es un retículo completo, tenemos que establecer que todo conjunto de conceptos posee ínfimo y supremo. Previamente, hay que ver que la intersección arbitraria de intenciones (resp. extensiones) es una intensión (resp. extensión).

Para ello bastaría con demostrar que dado un conjunto de índices naturales T, y conjuntos de objetos y atributos $X_t \subseteq O$, $Y_t \subseteq A$ se verifican las siguientes propiedades:

$$\left(\bigcup_{t \in T} X_t \right)^{\uparrow} = \bigcap_{t \in T} X_t^{\uparrow}$$

$$\left(\bigcup_{t \in T} Y_t \right)^{\downarrow} = \bigcap_{t \in T} Y_t^{\downarrow}$$

Ahora ya estamos en disposición de establecer los conceptos de ínfimo y supremo y de cómo todo conjunto de conceptos dispone de ambos y por tanto que G(*O*, *A*, *I*) es un retículo completo.

Teorema 3.7 (básico de los retículos de conceptos) El conjunto G(*O*, *A*, *I*) con la relación de orden definida es un retículo completo, en el que ínfimo y supremo viene dados por las fórmulas:

$$\mathbf{sup}(\ \{X_t, Y_t\} : t \in T\ ) = ((\bigcup_{t\in T} X_t)^{\uparrow\downarrow}, \bigcap_{t\in T} Y_t)$$

$$\mathbf{inf}(\{X_t, Y_t\} : t \in T\ ) = (\bigcap_{t\in T} X_t, (\bigcup_{t\in T} Y_t)^{\downarrow\uparrow})$$

La existencia de ínfimo y supremo para cualquier conjunto de conceptos nos permite en particular realizar la definición de las siguientes funciones:

$$\gamma : O \rightarrow G(O, A, I)$$
$$o \rightarrow \mathbf{inf}(X, Y)_{\{(X, Y) \in (O, A, I)\ |\ o \in X\}}$$

$$\mu : A \rightarrow G(O, A, I)$$
$$a \rightarrow \mathbf{sup}(X, Y)_{\{(X, Y) \in (O, A, I)\ |\ a \in Y\}}$$

Es fácil demostrar entonces que estas funciones admiten una notación mucho más simple, como la siguiente:

$$\forall\ o \in O, \gamma\ (o) = (o^{\uparrow\downarrow}, o^{\uparrow})$$
$$\forall\ a \in A, \mu\ (a) = (a^{\downarrow}, a^{\downarrow\uparrow})$$

Ello nos proporciona una forma práctica de determinar el mayor de los conceptos en cuya extensión aparece cierto objeto, o qué otros objetos comparten todos los atributos de uno dado.

Habitualmente los retículos de *Galois* son representados mediante su diagrama de *Hasse*, tal y como hacemos en este trabajo. En este diagrama cada punto o nodo representa un concepto formal y cada arco indica una relación de orden entre dos conceptos, donde el mayor de ellos está situado por encima del menor, con la restricción de que no exista ningún concepto intermedio.

Etiquetar los nodos del diagrama de *Hasse* con la descripción completa de su concepto asociado resulta poco legible, por ello habitualmente los objetos etiquetan el más bajo de los conceptos en cuya extensión aparecen, $\gamma\ (o)$, mientras que los atributos lo hacen con, $\mu\ (a)$, el más alto de los conceptos en cuya intensión están contenidos.

Hay que resaltar que desde esta representación, el retículo original y el propio contexto pueden ser reconstruidos, puesto que los conjuntos de objetos y atributos se obtienen directamente de los conjuntos de etiquetas, mientras que la matriz de incidencia se obtiene de la expresión

$$(o, a) \in I \Leftrightarrow \gamma\ (o) \leq \mu(a)$$

Así, un retículo obtenido de un contexto formal tiene un aspecto como el siguiente:

Ejemplo matriz de incidencia	atr_0	atr_1	atr_2	atr_3	atr_4
obj_0	✓	-	-	-	-
obj_1	-	-	✓	-	✓
obj_2	-	✓	-	-	-
obj_3	-	-	-	-	✓
obj_4	-	-	-	-	✓

Figura 3-2. *Ejemplo matriz de incidencia.*

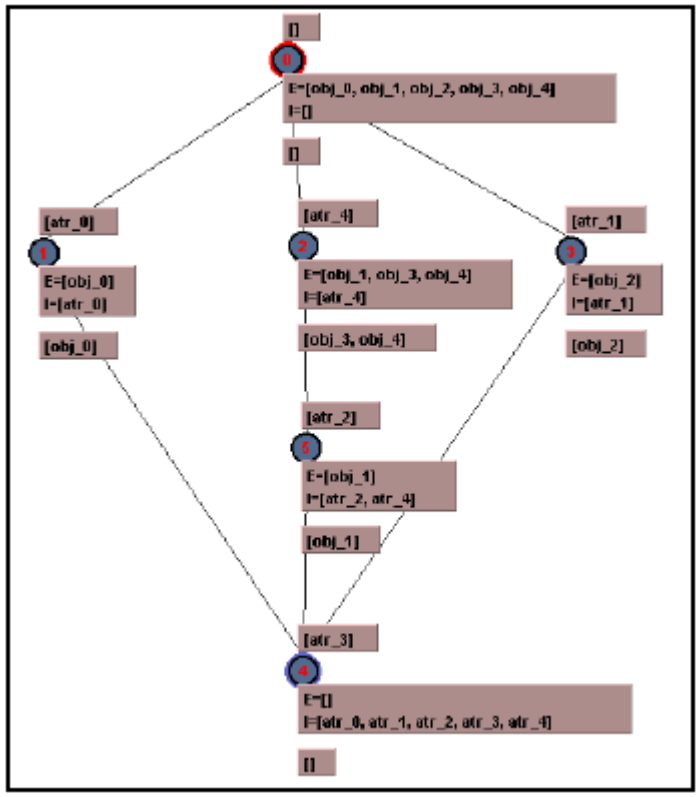

Figura 3-3. *Ejemplo retículo de conceptos*

En la **Figura 3-3** pueden verse los conceptos, en forma de nodos, obtenidos de la matriz de incidencia de la **Figura 3-2**. En cada nodo existen tres informaciones representadas en forma de cuadros: el cuadro central de cada nodo representa el concepto asociado al nodo, conjunto extensión e intensión; el cuadro superior de cada nodo la función $\gamma(o)$ y el cuadrado inferior $\mu(a)$, funciones descritas arriba.

Es evidente el interés de determinar qué parte del contexto determina realmente el *retículo de Galois*, y cual puede ser eliminada sin pérdida de información.

Se dice que un *contexto formal* está clarificado si dados $g, h \in O$ y $m, n \in A$ se verifican las siguientes afirmaciones:

$$\{g\}^{\uparrow} = \{h\}^{\uparrow} \Rightarrow g = h$$
$$\{\mathrm{m}\}^{\downarrow} = \{\mathrm{n}\}^{\downarrow} \Rightarrow \mathrm{m} = \mathrm{n}$$

Aún en el caso de contextos clarificados pueden existir atributos cuya presencia en el contexto no afecte a la forma final del *retículo de Galois* asociado al contexto. Como caso particular, los atributos que se aplican a todos los objetos y los objetos que tienen todos los atributos son innecesarios para la construcción del *retículo de Galois* asociado al contexto. A este tipo de atributos los denominamos *reducibles*.

El concepto más bajo en el retículo será $(A^{\downarrow}, A)$. Si $A^{\downarrow} = \varnothing$, esto es, si ningún objeto tiene todos los atributos (recíprocamente $O^{\uparrow} = \varnothing$) el contexto se llama reducido por filas (recíprocamente por columnas). Para simplificar se dice que el contexto está reducido.

3.3 Algoritmo de *Bordat*

El interés de la técnica se centra en la obtención, a partir de un contexto formal, del retículo completo asociado (lo denominaremos *retículo de Galois asociado* al contexto). Para ello se necesita tanto el conjunto de conceptos como las aristas, esto es, para cada concepto cuáles están directamente por encima o por debajo.

Para solucionar el problema que se ha expuesto, se ha optado por emplear el *Algoritmo de Bordat*, y así obtener el retículo asociado al contexto ya preparado. La implementación de este algoritmo se ha realizado siguiendo como guía la desarrollada en un Proyecto fin de carrera anterior [Pérez et al., 2002] y reutilizando el código del proyecto anterior en el que nos basamos [DBRE, 2005].

Para determinar el retículo de *Galois* asociado a un contexto tenemos que dado un contexto formal *C(O, A ,I)* cada concepto sobre su retículo de *Galois* asociado tiene la forma $(X^{\uparrow\downarrow}, X^{\uparrow})$ para algún $X \subseteq O$, y la forma $(Y^{\downarrow}, Y^{\downarrow\uparrow})$ para algún $Y \subseteq A$.

Además cada extensión es la intersección de extensiones de atributos, mientras que cada intensión es la intersección de intensiones de atributos.

A partir de la proposición anterior se puede proponer la fabricación de la lista de todas las extensiones del retículo asociado a un contexto mediante el siguiente algoritmo sencillo:

Paso 0 Añadir a la lista de las extensiones G
Paso m $\forall m \in M$, $\forall A$ en la lista de conjuntos añadir a la lista $A \cap \{m\}^{\downarrow}$

El algoritmo establecido aquí para determinar todos los conceptos, aunque es simple, no es adecuado para contextos grandes, ya que requiere recorrer la lista todas las veces. de ahí que abordemos la construcción de un algoritmo más eficaz, el *Algoritmo de Bordat*. Remitimos al lector al trabajo [Pérez et al., 2002] para solucionar cualquier duda que pudiera surgir acerca de su funcionamiento.

También se hace necesario el comentar la existencia de otros múltiples algoritmos que desempeñan la misma función que el algoritmo de *Bordat* para la obtención del retículo, tales como Tockit [Tockit, ref], ToscanaJ [Toscana, ref], o Galicia [Galicia, ref].

Debido a la multitud de algoritmos disponibles, la herramienta está diseñada de forma que en cualquier momento se pueda incorporar un algoritmo diferente como veremos en el diseño más adelante, así mismo y debido a que todavía no hay una DTD estándar para la exportación en xml de los contextos, se ha implementado en el mismo formato del anterior proyecto fin de carrera en el que nos basamos [DBRE, 2005]; además, también se ha incluido un exportador para Galicia [Galicia, ref] permitiendo así una verificación entre los retículos obtenidos, dejando para una futura línea de ampliación la comparación entre los distintos algoritmos existentes en el ACF para la construcción del retículo de *Galois*.

3.4 Representación conceptual de los datos extraídos del código

En este apartado describiremos como representamos nuestros datos en una matriz de contexto, es decir, que consideramos como objetos y que como atributos. También describiremos las incidencias y su significado.

Los datos que representaremos en la matriz de contexto o de incidencias los obtendremos a través del análisis objeto de estudio aportado por el usuario final. Dichos datos ocuparan lo que hasta ahora hemos venido llamando atributos y objetos. Para los objetos introduciremos las clases detectadas en el parser, ya sean finales, abstractas pero con algún método implementado etc. Sin embargo, cabe señalar que no se tendrán en cuenta las clases anidadas o "*nested class*" ya que no afectan a la herencia más que para la funcionalidad de la clase que las engloba, y será sobre ella sobre la que se detecte la herencia. En cuanto a los interfaces, sólo puede existir herencia entre ellos, incluso permitiéndose la herencia múltiple, ya que un no se puede instanciar un interfaz, únicamente implementarse, por lo que se hará el análisis por separado de las clases y de los interfaces. Concluyendo los objetos pueden ser o clases o interfaces pero no ambos a la vez, además no se tendrán en cuenta las clases anidadas. No confundir clases anidadas con varias clases independientes dentro de un mismo archivo.

Para los atributos tendremos métodos, atributos o *fields*, constantes reconocidos por el parser ya sean públicos, privados, protegidos, etc. Los privados aunque no podrán ser utilizados por las demás clases y no se heredarán, también las reconocemos pues es un elemento más que ayuda a la hora de construir el grafo e interpretarlo para averiguar quien hereda de quien, o las posibles relaciones de herencia que podría haber. Cabe hacer mención a que en la matriz de incidencias la clase "hija" que hereda del padre tendrá tanto sus características como las del padre, pauta que nos ayudara a descubrir relaciones de herencia en el código. La coincidencia de características también es indicio de una posible relación de herencia, aunque no sean en la realidad padre e hija, poniendo de manifiesto la idea de cómo podrían optimizarse dichas clases.

La matriz de incidencia vendra representada por una tabla de doble entrada donde en la entrada superior estarán las características detectadas en el código objeto de estudio, y las clases en la entrada izquierda, de forma que el cruce entre unas y otras señalara la incidencia denotándose con una "v" en caso de que exista dicha incidencia y con un cero en caso de no haberla.

Una incidencia en dicha matriz, entre un objeto que representa una clase y un atributo que representa una característica de dicha clase, significa que dicha característica es poseída por la clase en cuestión. De forma que todo el conjunto de incidencias será el conjunto de características que tienen las clases analizadas en cuestión. De forma que a través de un análisis de dichas incidencias, representadas en el retículo de *Galois*, induce una idea de las relaciones de herencia existentes en el

código así como de las posibles relaciones de herencia que se podrían extrapolar a través de la interpretación del anteriormente nombrado retículo de *Galois*.

3.5 Obtención y representación del retículo de *Galois.*

Una vez se ha obtenido la matriz de contexto no queda más que aplicar el algoritmo seleccionado para la obtención del retículo de *Galois*. Como hemos dicho anteriormente existen múltiples algoritmos para dicha construcción, utilizándose en este caso el algoritmo de *Bordat* por los motivos anteriormente descritos. Para la obtención del retículo reutilizamos del proyecto fin de carrera en el que nos basamos [DBRE, 2005] de forma que únicamente tenemos que pasarle la matriz de contexto obtenida en la fase anterior y dibujarlo.

En la reutilización, únicamente hemos tenido que construir un contexto compatible al que se emplea en [DBRE, 2005] y utilizarlo como caja negra, sin llevar a cabo modificación alguna.

Hay que remarcar que la matriz que hemos obtenido es totalmente compatible con el proyecto en el que nos basamos pues utilizamos el mismo formato. Si se utilizara un algoritmo distinto al de *Bordat*, habría que replantear la forma de obtener el retículo pues al reutilizar su código, estamos adaptándonos a su forma de obtener dicho retículo que por defecto era a través del algoritmo de *Bordat*.

En el capitulo posterior se vera como interpretar dicho retículo de *Galois*, ya aplicado al análisis de las jerarquías de herencia en *Java*.

En una futura línea de ampliación será necesaria la modificación en la construcción del retículo si se quiere añadir nuevos algoritmos, o la redefinición de los métodos que construyen el retículo.

Remitimos al lector al trabajo [DBRE, 2005] para solucionar cualquier duda que pudiera surgir acerca de su funcionamiento y construcción.

4 Caso de Estudio

4.1 Introducción

En este capítulo veremos una descripción global del desarrollo del proyecto, desde los inicios en que era una vaga idea, hasta su finalización como herramienta funcional. Hablaremos de las distintas partes que se pueden distinguir en el sistema, así como de las dificultades que nos fuimos encontrando en su desarrollo y como las fuimos superando. También veremos como encaja en el marco de trabajo del desarrollo de *frameworks de dominio* vista anteriormente, sin olvidarnos de su soporte teórico y su interpretación.

Otro punto a tratar y que no debemos de olvidar son las fases por las que ha ido pasando el proyecto, es decir su evolución. A lo largo del capitulo iremos describiendo las fase indicando las tareas realizadas, y los problemas que surgieron.

Recordaremos a grandes rasgos en que se basa nuestro proyecto. Hemos desarrollado una herramienta que a partir de código *Java*, 1.4 y 1.5 reconoce relaciones de herencia tanto las existentes como las que podrían darse, en una colección de clases dada. Para ello, hemos construido un *parser*[1] que reconoce código *Java* (1.4 y 1.5), obteniendo los datos necesarios para construir la matriz de incidencias o de contexto, y a partir de ahí mediante el algoritmo de *Bordat* construir el retículo de *Galois* asociado a dicho contexto. Una vez construido, mediante la interpretación de dicho retículo (como se vera más adelante) el usuario podrá deducir las posibles relaciones de herencia que pudieran construirse en dicho código.

En resumen, hablaremos primero de los comienzos, posteriormente de cada una de las partes que en que se puede dividir el proyecto, así como de las fases de evolución por las que ha ido pasando para finalizar hablando del soporte teórico que apoya el proceso y su interpretación.

4.2 Orígenes

Los comienzos siempre han sido difíciles y más si la tarea a realizar es algo que no se asemeja a nada que hayas hecho anteriormente. Nuestro proyecto, como venimos repitiendo, se basa en dos proyectos anteriores, el primero “Implementación de un sistema de diagnóstico software mediante Análisis de Conceptos Formales” [Pérez, 2002] y el segundo “Reingeniería de bases de datos mediante el análisis de conceptos formales” [DBRE, 2005]. Cabe señalar que este último también se basa en el anterior, con lo que no fue tan necesario basarse especialmente en el primero a la hora de meternos en harina, pero fue de gran utilidad a la hora de obtener documentación ya que continuamente se hacían referencias al proyecto [Pérez, 2002] desde el de [DBRE. 2005].

Las primeras dificultades surgieron ahí, ya que el proyecto de [Pérez, 2002] compartía gran numero de funcionalidades con nuestro proyecto, sin embargo con la diferencia de que estaba diseñado para código *Eiffel*, y enteramente implementado en dicho lenguaje. Aparte utilizaba una herramienta grafica (VGC Free BSD 1.30) para la representación de árboles que únicamente ofrecían funcionalidad para UNIX/Linux y Windows 3.11, algo que limitaba nuestro proyecto y desaprovechaba totalmente la portabilidad de *Java*, lenguaje en el que hemos desarrollado nuestro proyecto.

[1] Parser, analizador léxico sintáctico que reconoce cadenas de caracteres de un determinado lenguaje que confrontan con unos patrones o expresiones regulares dadas y realiza acciones determinadas por el usuario

De manera que nos obligaba a utilizar código nativo si queríamos reutilizar su código, o implementar desde cero dicho código en *Java* con lo que desaprovecharíamos las ventajas que nos ofrece la reutilización. Otro punto en cuestión es que nos limitaríamos a una plataforma UNIX/LINUX, pues es extraño encontrar hoy en día un sistema operativo Windows 3.1, además de ser totalmente absurdo desarrollar para un sistema operativo obsoleto, desaprovechando como hemos dicho la versatilidad que nos ofrece *Java* con su portabilidad.

Finalmente tuvimos un golpe de suerte y apareció en escena el proyecto [DBRE, 2005], proyecto que trataba sobre la reingeniería de bases de datos con técnicas ACF, desarrollado totalmente en *Java*, y que utilizaba el algoritmo de *Bordat*. Para nuestra satisfacción pintaba además el retículo de *Galois* con las *Java Foundation Class* (JFC) y más concretamente con el paquete *swing*, con lo que quedaba superada las limitaciones iniciales impuestas por el proyecto de [Pérez, 2002], pasando desde ese momento a ser únicamente fuente de documentación y nosotros volcarnos en la reutilización del proyecto [DBRE, 2005].

Una vez superada esa dificultad comenzamos la etapa de búsqueda de documentación sobre ACF, *frameworks*, etc. Encontrando bastante documentación aportada en su mayoría por nuestra tutora, Dr. Yania Crespo, y con aportaciones por otros profesores como Félix Prieto. Con esa documentación y la del proyecto [Pérez, 2002], el proyecto comenzó a dar sus primeros pasos, pudiendo distinguir en la herramienta dos partes claramente diferenciadas:

- Desarrollo del parser *Java*.
- Desarrollo del paquete ACF.

Y en el proyecto tres etapas:

- Análisis de las clases aportadas por el usuario
- Construcción de la matriz y el retículo de *Galois*.
- Interpretación por parte del usuario de dicho retículo.

Pasaremos a continuación a tratar cada una de las partes en que se podría dividir la herramienta, dejando las etapas para el final del capitulo, después de hablar previamente de las fases de evolución del proyecto.

4.3 Desarrollo del Parser

Los datos que alimentan a los distintos algoritmos que se aplican a la matriz de contexto para la obtención del retículo deben ser extraídos previamente del código fuente objeto del estudio. Esto requiere de una herramienta básica de análisis del código fuente dotada de algoritmos posteriores que identifiquen los datos relevantes para el tipo específico de análisis requerido. En definitiva un parser.

4.3.1 Primeras Aproximaciones al *Parser*

Para comenzar a estudiar el código fuente y poder extraer las características, clases, interfaces, etc. Una primera aproximación o prototipo consistió en diseñar e implementar el *parser* desde cero. Esta elección en principio parece la más rápida, ya que dicho *parser* solo tendría que reconocer los elementos necesarios para el proyecto, e ignorar el resto del código.

Esta aproximación presenta entre otras, las siguientes desventajas:

- Si no reconocemos toda la gramática del lenguaje es muy complicado detectar si el código fuente objeto de estudio, es valido. Aunque esto es uno de los requisitos del proyecto (código valido), otras soluciones consiguen detectar código no valido, con lo que tendremos más seguridad de que la aplicación funcione correctamente.

- Generar un *parser* desde cero conlleva generar a su vez un analizador léxico. Si dicho analizador léxico se implementa solo para que reconozca la parte de la gramática que nos interesa su ampliación o modificación posterior se vuelve muy costosa.

- Si el analizador léxico se implementa de forma general para que reconozca toda la gramática, a mayores de que seria costoso, existen herramientas (*lex*, *JavaCC*, etc.) en las que la implementación de dicha gramática es más fácil e incluso existen implementaciones de gramáticas para los lenguajes de programación más extendidos.

- Añadir otros *parsers* que reconozcan otros lenguajes consiste prácticamente en volver a implementar un *parser* nuevo.

Llegados a e este punto vemos que las desventajas de usar un *parser* creado desde cero son bastantes e importantes, por lo que se avalúan otras alternativas.

4.3.2 Herramientas para Generar *Parsers*.

Los criterios más importantes para encontrar una herramienta que generara *parsers* fueron los siguientes:

- Debía poder generar el código del *parser* en *Java*. Como el resto del proyecto está realizado en *Java*, el poder enlazar el *parser* con el resto del proyecto se realiza de una manera más cómoda y transparente.

- Que existan gramáticas implementadas para distintos lenguajes y sobre todo que exista para *Java*. De esta manera se ahorra el trabajo de implementar la gramática, solo hay que adaptarla a nuestras necesidades.

- Si la herramienta no solo aporta un reconocedor léxico, si no que permita la definición de reglas sintácticas, sin tener que usar otra herramienta (caso de ejemplo *lex* y *yacc*). Si solo se usa una herramienta en lugar de varias suele ser menos complicado el depurado del *parser*.

- Herramienta gratuita. Aunque existe una gran variedad de herramientas para poder crear el *parser*, el que sea gratuita como es obvio es una característica a tener en cuenta.

Después de valorar las distintas alternativas optamos por elegir la herramienta *JavaCC*, y reutilizar las gramáticas que ya existen implementadas para esta herramienta.

4.4 Desarrollo del Paquete ACF

Una vez finalizado el parser con todas las dificultades surgidas superadas, el siguiente paso evidente era el desarrollo del paquete que trataría los datos obtenidos del código suministrado por el usuario empleando técnicas de ACF. Su realización no era trivial pues aunque reutilizamos

gran parte del proyecto [DBRE, 2005], la gran dificultad residía en como "adaptarlo" a nuestro problema pues su uso en dicho proyecto se centraba en la reingeniería de bases de datos y la aplicación de ACF a dicho problema. Tras una gran cantidad de horas dedicadas a analizar la estructura y código de dicho proyecto, y ver como reutilizarlo en el nuestro, pudimos comprobar que las principales clases que daban la matriz de incidencias las podíamos reutilizar a través de relaciones de clientela. Sin embargo para construir dicha matriz de contexto adaptada a nuestras necesidades y salvaguardando la retrocompatibilidad con [DBRE, 2005] necesitábamos algunas otras clases de dicho proyecto, las cuales requerían de pequeñas modificaciones que tuvimos que realizar a través de la herencia, es decir heredar de dichas clases y redefinir algunos métodos, así como definir métodos nuevos necesarios. Con eso y con la introducción de clases nuevas pudimos reutilizar sin mayor problema las clases de dicho proyecto para la realización de nuestro bloque ACF.

Como vimos anteriormente, la herramienta debe ser capaz de extraer de forma automática la información proporcionada, tanto en las aplicaciones del dominio como en las instancias del *framework*; en este caso las suministra el usuario, cuando crea un proyecto nuevo en la herramienta, dando su localización y encargándose el parser de inspeccionarlas. También debe presentar dicha información de una forma suficientemente expresiva al desarrollador, ocultándole los detalles técnicos del mecanismo formal que las soporta. Cosa que hacemos, ya que una vez "parseado" el código lo que se hace es mostrar los resultados en forma de tablilla de celdas donde el usuario puede inspeccionar la matriz de contexto con total libertad viendo las incidencias existentes, esto en modo gráfico.

A	B	C	D	E	F
	atr0	atr1	atr2	atr3	atr4
Clase1	✓	✓	0	0	0
Clase2	✓	✓	✓	✓	0
Clase3	✓	✓	0	0	✓
Clase4	✓	✓	✓	✓	✓

Figura 4-1. *Matriz de incidencia en modo gráfico.*

El elemento central de las herramientas ACF tanto de nuestro proyecto como el de [DBRE, 2005] o el de [Pérez, 2002] está constituido por la implementación de uno de los algoritmos que proporcionan el retículo asociado a un contexto formal. Existe una gran variedad de algoritmos como vimos antes [Tockit, ref], [Toscana, ref], o [Galicia, ref], tanto de tipo general, como adaptados a un tipo particular de contexto formal, y se han realizado estudios comparativos entre ellos, como por ejemplo [Godin y Chau, 2000]. Para nuestra herramienta, nos hemos decantado por un algoritmo general, el algoritmo de *Bordat* [Bordat, 1986], ya que aparte de conjugar la facilidad de implementación con un rendimiento razonable, es el utilizado en los anteriores proyectos en los que nos basamos [Pérez, 2002] y [DBRE, 2005]. No obstante, la implementación está preparada para la sustitución de este algoritmo por otros más efectivos o más adaptados a datos con una configuración característica. Bastará para ello, únicamente definir un método nuevo para la obtención del retículo, donde se implemente el nuevo algoritmo y pintarlo.

La adopción del estándar *XML* facilita el procesamiento de estos ficheros, tanto de contexto como de retículo de *Galois*, aunque, desgraciadamente no disponemos de un DTD estandarizada que facilite el intercambio de información con herramientas de ACF producidas por otros autores.

Al no haber definida un DTD estándar (lo cual supuso una dificultad a la hora de elegir el formato a exportar, salvándola a través de dar la posibilidad al usuario de elegir el exportador) para guardar los datos referentes tanto a la matriz de contexto como al retículo se ha optado por salvaguardar la retrocompatibilidad con el proyecto anterior [DBRE, 2005]. De forma que se ha reutilizado el DTD definida en ese proyecto para la exportación de los datos de la matriz de contexto y retículo, así como también se ha adoptado el DTD utilizado en el proyecto [Galicia, ref], éste solo para la matriz de contexto, ya que ese proyecto es el que mayor funcionalidad proporciona de los que hemos mencionado. Se ha diseñado de tal forma que si se quiere incorporar nuevos exportadores con su respectivo DTD se pueda hacer de la forma más cómoda posible; simplemente habría que definir una clase que implemente el "`writer`" para el DTD que se quiere incorporar, y añadírselo al manejador de exportadores:

"`es.uva.pfc.herencia.io.SetExportContex.java`"

En el constructor. Veamos un ejemplo donde queramos introducir un nuevo exportador, **NuevoXMLWriter**:

```
private Vector exports;             // Vector de exportadores
                                    // con todos los
                                    // exportadores
                                    // disponibles

public SetExportContexto() {        // Constructor

      AbstractExportContextoWriter contex = null;
      exports = new Vector();

      contex = new GaliciaXMLWriter(); // Definimos el
      writer
                                       // para Galicia
      exports.add(contex);             // Añadimos el writer

      contex = new DBREXMLWriter();    //idem para DBRE
      exports.add(contex);

      // Añadimos el nuevo que quisiéramos meter
      contex = new NuevoXMLWriter();
      exports.add(contex);

}// SetExportContexto
```

También cabe la posibilidad, para una mayor reutilización de la clase `SetExportContexto` de añadir el nuevo exportador de forma dinámica en tiempo de ejecución. Un ejemplo seria:

```
SetExportContexto exportadores = new SetExportContexto();
AbstractExportContextoWriter contex = new NuevoXMLWriter();
exportadores.addExport(contex);
```

Señalar que para que el nuevo exportador sea compatible con la aplicación este debe implementar la clase abstracta `AbstractExportContextoWriter` (en el paquete `es.uva.pfc.herencia.io`).

Con el exportador de retículo estaríamos en caso similar.

```
Sintaxis:
[-nogui
        -nivel_salida nivel
        -fuentes archivoCodigo1 [archivoCodigo2 [...] ]
        -classpath directorio1 [directorio2 [...] ]
        -parser parserAUsar
        [(-out_contexto nombreArchivoSalida
                -out_format_contexto formatoSalida) ]
        [-overwrite_contexto ]
        [-algoritmo algoritmoGeneradorReticulo
                [-overwrite_reticulo ]
                [-mostrar_reticulo ]
                [(-out_reticulo nombreArchivoSalida
                        -out_format_reticulo formatoSalida)]
        ]
]
NOTA: con el modificador -nogui, hay que indicar al menos una de las siguientes:
 (-mostrar_reticulo, -out_reticulo, -out_contexto)

Donde los posibles parsers son (Nombre / descripcion):
java1.4         java 1.4
java1.5         java 1.5

Los formatos de archivos de salida para el contexto son:
GaliciaXML              Galicia XML
DBREXML         DBRE XML

Los algoritmo de generacion de reticulos son:
AlgoritmodeBordat               Algoritmo de Bordat

Los formatos de archivos de salida para el reticulo son:
DBREXML         DBRE XML

Los posibles niveles de salida son (Numero / Descripcion):
0               Mensajes de error
1               Mensajes esenciales
2               Mensajes informativos
3               Mensajes de depuracion
Un nivel de salida implica que tambien se mostraran los niveles inferiores
```

Figura 4-2. *Opciones del modo consola.*

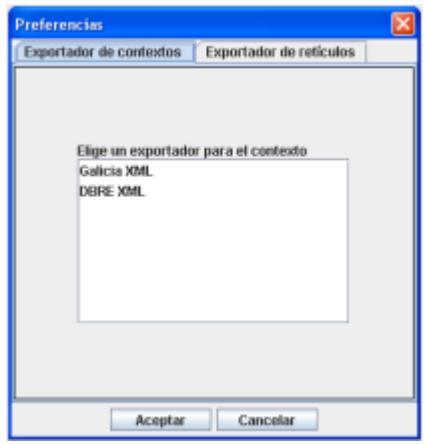

Figura 4-3. *Opciones en modo gráfico.*

Visto esto, es fácil ver como cuando por fin salga un DTD estándar podrá ser incorporada a la herramienta de una forma rápida y sencilla. Análogamente se podría hacer con los exportadores de retículo. Esto se deja para futuras ampliaciones, pero eso se detallara más adelante en la memoria.

Cabe señalar que a la hora de abrir un fichero acepta también de los dos formatos tanto de *Galicia* [Galicia, ref] como de *DBRE* [DBRE, 2005], detectándolo automáticamente según la extensión de cada archivo, siendo "`*.bin.xml`" "`*.setBin.xml`" para el contexto de cada uno de ellos respectivamente. Para el retículo, debido a que únicamente exportamos en compatibilidad con *DBRE* [DBRE, 2005], solamente tenemos el importador para dicho formato, no considerando ninguno más.

Una vez que tenemos la matriz de contexto y se ha ejecutado el algoritmo para pintar el retículo, el siguiente paso inmediato será el de representar dicho retículo. Inicialmente se nos proporciono una herramienta abierta de representación de grafos, la cual desconocíamos su forma de uso, con lo que hubo que documentarse en su uso, hasta que hizo aparición el proyecto [DBRE, 2005]. El gran problema como ya se comento en este capítulo (ver orígenes) era que en un principio para pintar el grafo se nos suministro dicha herramienta llamada VGC que representa grafos, sin embargo está desarrollada principalmente para UNIX/LINUX, migrada de Windows 3.11, con lo que al utilizarla se perdería toda la versatibilidad que da *Java* con la portabilidad. De forma que se opto por la misma solución que en [DBRE, 2005], pintar el retículo directamente en *Java* a través de su paquete *swing* y *graphics2d*. Con lo que solamente tuvimos que reutilizar sus clases a la hora de representar el retículo de *Galois*.

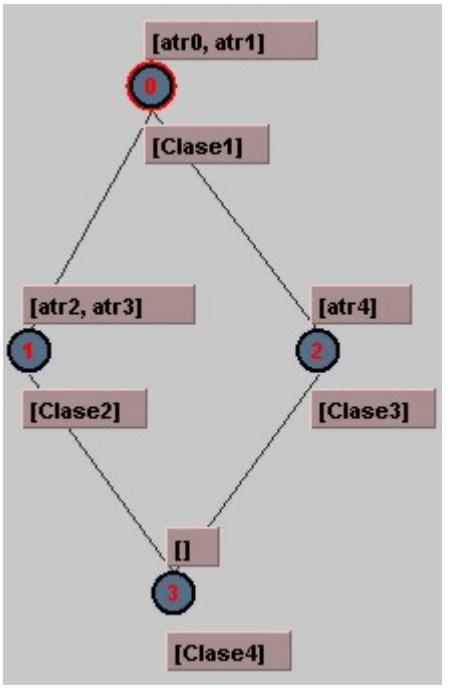

Figura 4-4. *Retículo de Galois de la matriz anterior.*

Aparte de la representación gráfica del retículo, será necesaria su análisis e interpretación y así sacar las conclusiones adecuadas conforme a la herencia. Hay que reseñar que dicho análisis requiere de conocimientos sobre la teoría matemática. Para una implantación efectiva en el desarrollo real de software, deben acercarse al desarrollador las conclusiones que se pueden extraer de los retículos Veremos un poco mas adelante en este mismo capitulo como se puede interpretar dicho retículo.

4.5 Fases de Evolución

En esta parte nos remitiremos al capítulo siguiente donde veremos toda la parte relacionada con la ingeniería del software, con los documentos de requisitos, la visión global, la planificación etc.

Únicamente decir que se ha seguido el proceso unificado en la manera que esto ha sido posible pues debido a que nuestra experiencia en el desarrollo de proyectos es bastante limitada,

reduciéndose a las prácticas de ingeniería del software 1 y 2, de forma que puede haber discrepancias en el tiempo dedicado a cada tarea etc.

4.6 Etapa 1: Análisis de Clases Java a través del Parser

En esta primera etapa el usuario le proporciona a la herramienta las clases o interfaces que serán objeto de estudio en el análisis de la jerarquía de herencia. Cabe señalar que se analizará tanto las posibles relaciones de herencia que se podrían obtener de esas clases, como la herencia en código propiamente dicha. Dicho de otra manera, las clases a analizar puede que hereden unas de otras o hereden de las clases propias de *Java* pero también, distintas clases pueden tener métodos idénticos sin que haya herencia en el código, con lo que podríamos encontrar posibles relaciones de herencia.

El usuario en un primer momento le proporciona al sistema las clases que son objeto de estudio, así como el parser que utilizará y el o los *classpath* donde encontrar las diversas clases a las que haga referencia, ya utilice la aplicación a través del interface gráfico o por medio de la línea de comandos. Dicho *classpath* será necesario para indicar donde comenzar a buscar las clases referenciadas en los "*import*" de cada clase.

La aplicación por defecto admite tres *parsers*, uno para *Java* 1.4 que está incompleto, otro que solo examina las clases para *Java* 1.5 y otro que examina los interfaces para *Java* 1.5, pero en cualquier momento se le podría integrar nuevos parser incluso sustituyendo a los que ya están.

Para incorporar nuevos *parsers* a la aplicación solo hay que compilar la clase (o clases) del nuevo *parser*, instanciar en el constructor de la clase `SetParsers` y volver a compilar esta clase. La aplicación carga dinámicamente todos los *parsers* que existan dentro de la clase `SetParsers`. Para asegurar la compatibilidad la clase principal de dicho "parser" (la que se instancia en `SetParsers`) debe implementar la clase abstracta `AbstractParser`. Ambas clases (`SetParsers`, `AbstractParser`) se encuentran en el siguiente paquete `es.uva.pfc.herencia.parser`. Así como los *parsers* que vienen con la aplicación. En el siguiente ejemplo se muestro como añadir un nuevo parser a la aplicación de forma permanente:

```
//Vector que contiene los parsers
//de la aplicación.
private Vector parsers;

//Constructor(es)
/**
 *  Constructor  por  defecto,  en  el  cual  se  crean  los
distintos
 * parsers.
 */
public SetParsers() {
      AbstractParser parser;
      parsers = new Vector();

      //Metemos los parsers conocidos
      parser = new JavaParser_1_4();
      parsers.add(parser);

      parser = new JavaParser_1_5();
```

```
            parsers.add(parser);

            parser = new JavaInterfacesParser_1_5();
            parsers.add(parser);

            //Instaciamos y añadimos el nuevo parser
            parser = new NuevoParser();
            parsers.add(parser;

    } //constructor
```

Para aumentar la reutilización de la clase `SetParsers`, también se puede introducir un nuevo parser dinámicamente en tiempo de ejecución a través del siguiente método `addParser(AbstractParser parser)`, un ejemplo de cómo se añadiría el *parser* sin tener que volver a compilar la clase `SetParsers`:

```
cjtoParsers = new SetParsers();
AbstractParser nuevo = new NuevoParser();
cjtoParsers.addParser(nuevo);
```

Una vez hecho eso, el parser entrara en acción buscando en cada clase métodos y atributos que confronten con las reglas que tiene establecidas. Describiremos su modo a proceder con respecto a las diferentes cosas que nos podemos encontrar dentro de una clase *Java*.

Antes de describir como actúa el *parser* con las distintas declaraciones de métodos, fileds, etc. hay que señalar que para nuestra aplicación hay que distinguir entre los interfaces y las clases, es decir, o se analizará la herencia solamente de las clases, o solamente de los interfaces. Por esta razón se han añadido dos *parsers*: Uno que trabaja solamente con las clases, y otro que trabaja solamente con los interfaces. La manera de proceder en ambos es similar. El *parser* que trabaja con los interfaces toma como nombres de clases los interfaces e ignora las clases, el otro *parser* trabaja al contrario ignorando los interfaces.

Internamente el "parser" se comunica con dos clases de la aplicación:

- `Matriz.java` (`es.uva.pfc.herencia.acf`).
- `GestorParser.java` ,(`es.uva.pfc.herencia.parser`).

Una de las funciones de `GestorParser` es ir ejecutando el *parser* en nuevos archivos sobre los cuales aparezcan referencias en el código y estén dentro de los directorios dados como "classpath". En el caso de trabajar con código *Java*, cuando se encuentre una sentencia del tipo `import es.uva.pfc.herencia.io.SetParser;` el *parser* pasará el parámetro de la sentencia a la clase `GestorParser`, esta clase identificará si es un fichero o un directorio (para el caso: `import es.uva.pfc.herencia.io.*;`).

- Si es un fichero lo añadirá si existe a la lista de archivos a examinar por el "parser".

- Si es un directorio añadirá todos los ficheros (que concuerden con la extensio) que se encuentren dentro de dicho directorio a la lista de archivos a examinar por el "parser".

Una vez seleccionado el "parser" (clases o interfaces) y comenzada su ejecución, el "parser" se comunica con un objeto de tipo `Matriz`, en el que va añadiendo (en cuanto encuentre una definición de clase) el nombre de la clase (ó interface dependiendo del *parser* seleccionado). Y las distintas características (métodos y campos) de dicha clase, tanto públicas como privadas. Las clases e interfaces anidadas (nested) son ignoradas.

Con la información recogida por el parser se construye la matriz de contexto como se describe en la siguiente etapa.

4.7 Etapa 2: Construcción de la Matriz de Contexto y del Retículo de *Galois*

La matriz de incidencia o de contexto formal se construirá utilizando las técnicas básicas de ACF. A partir de dicha matriz se construye de forma algorítmica el retículo de *Galois*, representable mediante su correspondiente diagrama de *Hasse*, que contiene toda la información original, si bien organizada de un modo que pone de manifiesto la estructura de los datos.

Las filas de la matriz de incidencia representan objetos, sus columnas atributos[2], y la incidencia, la presencia de un atributo en determinado objeto. Los nodos del retículo, denominados conceptos formales, están entonces constituidos por un par de conjuntos de objetos y atributos que se determinan mutuamente. La relación de orden que constituye el retículo viene determinada tanto por la relación de inclusión entre conjuntos de objetos como por la relación de contención entre conjuntos de clases.

La aplicación de nuestra herramienta formal requiere por tanto definir la forma en que será construida la matriz de incidencia, determinando qué interpretaremos como objetos y atributos respectivamente, y la forma en que el retículo debe ser posteriormente interpretado, lo cual lo veremos en el siguiente apartado. Diferentes matrices de incidencia permiten poner de manifiesto diferentes estructuras en los datos originales. En esta sección ilustraremos de manera informal el modo en que esta técnica permite realizar los análisis de herencia mencionados anteriormente.

Para ilustrar el funcionamiento del Análisis de Herencia, consideremos el conjunto de clases de una biblioteca estándar que pudiéramos crear y que proporcionan la funcionalidad básica de entrada y salida de datos. La **figura 4-5** contiene una representación de este conjunto de clases, simplificado a efectos de facilitar la comprensión de la técnica.

[2] Los términos objeto y atributo son utilizados habitualmente en la teoría de ACF en este sentido, y no tienen relación alguna con los términos homónimos de la Orientación a Objetos.

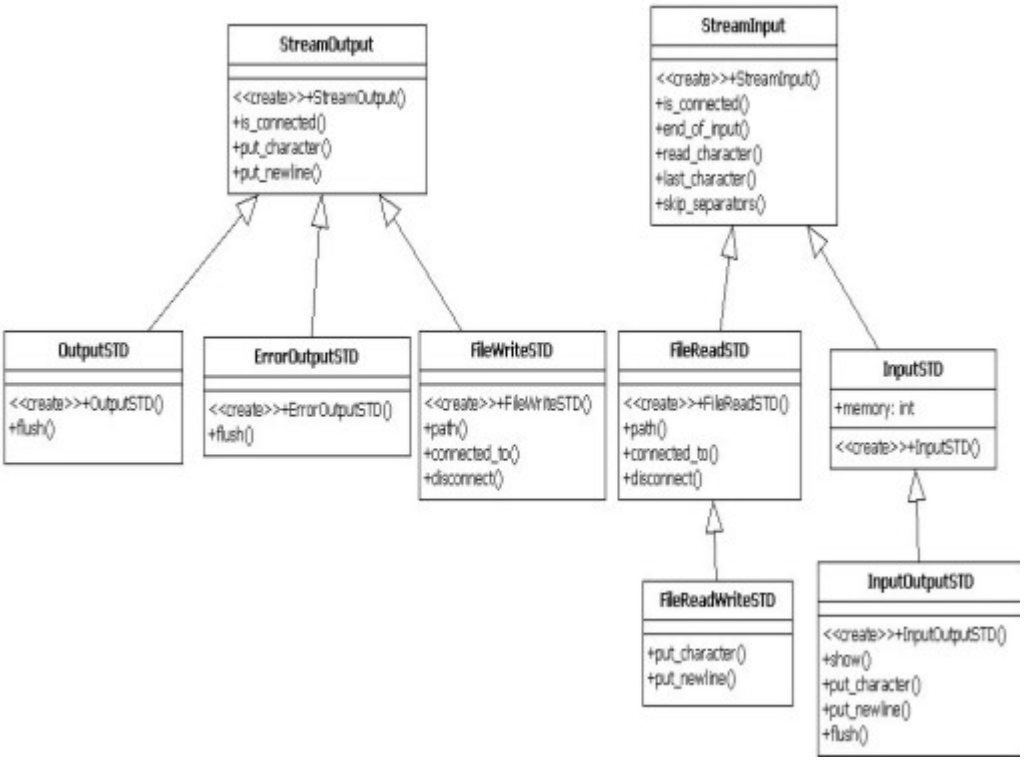

Figura 4-5. *Versión simplificada de algunas clases de la biblioteca estándar.*

	is_connected	put_character	put_newline	end_of_input	read_character	last_character	path	connected_to	disconnect	Skip_separators	flush	show	memory
ErrorOutputSTD	1	1	1								1		
FileReadSTD	1			1	1	1	1	1	1	1			
FileReadWriteSTD	1	1	1	1	1	1	1	1	1	1			
FileWriteSTD	1	1	1				1	1	1				
InputOutputSTD	1	1	1	1	1	1				1	1	1	1
InputSTD	1			1	1	1				1			1
OutputSTD	1	1	1								1		
StreamInput	1			1	1	1				1			
StreamOutput	1	1	1										

Figura 4-6. *Matriz de incidencia de las clases de la biblioteca estándar.*

Mientras que la **figura 4-6** representa la matriz de contexto formal asociada a ese conjunto de clases. Debido a su extensión ponemos una versión escrita de ella en lugar de la que se obtendría con el programa. En ella, las filas representan las clases a analizar, mientras que sus columnas representan las características[3] de éstas y la incidencia, es decir la presencia de una característica en una clase, evaluada teniendo en cuenta sus redefiniciones.

Explicaremos brevemente que hemos considerado como atributos y como objetos. Como objetos de la matriz de incidencia hemos considerado las clases que nos aporta el usuario, desde clases "simples" (es decir clases que únicamente contengan métodos y atributos) hasta clases anidadas dentro de otras clases, pasando por clases abstractas, finales, sincronizadas… No obstante no hemos considerado los interfaces pues una clase en *Java* no heredara de un interfaz, sino que los implementa, sin embargo un interfaz puede heredar de otro u otros interfaces, sería interesante introducir esta variante, pues a través de los interfaces se consigue la herencia múltiple en *Java*, sin embargo dicha tarea superaba la envergadura del proyecto y se deja como futuras ampliación a dicho proyecto.

Como atributos de la matriz de contexto hemos considerado los métodos, tanto constructores como propios de la clase y que no son constructores, constates, y atributos (en este caso haciendo referencia a la orientación a objetos) ya sean públicos, privados o protegidos. Aunque los métodos privados no se heredan, por funcionalidad interna les reconocemos.

4.8 Etapa 3: Interpretación del Retículo conforme a las Relaciones de Herencia

Con la matriz obtenida el siguiente paso inmediato es analizar el retículo obtenido mediante la aplicación del algoritmo (en nuestro caso el de *Bordat*) y ver las posibles relaciones de herencia que podría haber en el código objeto de estudio.

En la **figura 4-7** hemos obtenido el diagrama de *Hasse* del retículo de *Galois* obtenido a partir de la matriz de incidencia. Los nodos del diagrama representan conceptos formales del retículo, mientras que las etiquetas ilustran la presencia de clases y características, objetos y atributos, en los mismos. El criterio utilizado para el etiquetado es el habitual en este tipo de diagramas: Las clases etiquetan el concepto más bajo en que aparecen, mientras que los métodos lo hacen con el más alto.

[3] Con características hacemos referencia tanto a métodos como a los denominados "*fields*" (Campo o atributo) en *Java*.

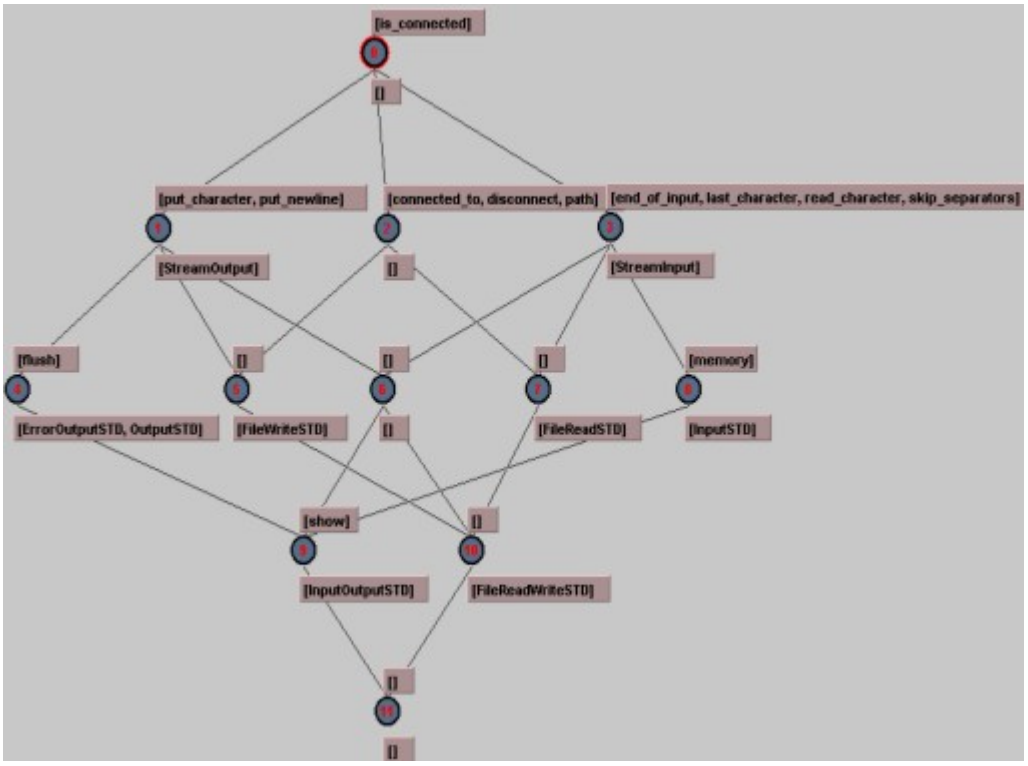

Figura 4-7. *Retículo de Galois referente a clases de la biblioteca estándar.*

Con un análisis adecuado, dicho diagrama permite poner de manifiesto hechos como los siguientes:

- La aparición de un nodo etiquetado por una clase por debajo del que representa a otra nos indica que todas las características de la primera están presentes en la segunda, por lo que podemos representar a la primera clase como heredera de la segunda. Así en el ejemplo podemos representar FileReadWrite como heredera de FileWriteSTD y FileReadSTD. Sin embargo, en el diagrama de clases se puede observar que únicamente hereda de FileReadSTD, con lo que ahí habria una posible relación de herencia multiple. Lo mismo pasa con InputOutputSTD, se puede interpretar que hay una posible herencia multiple de InputSTD y OutputSTD (o ErrorOutputSTD), y sin embargo realmente hereda de InputSTD.

- Existen nodos que representan varias clases del conjunto original, y con ello ponen de manifiesto que, con la información recogida en el contexto, las clases son equivalentes. En nuestro ejemplo esto es lo que ocurre con ErrorOutputSTD y OutputSTD.

- Algunos de los nodos no corresponden a clase alguna del conjunto original, pero su presencia proporciona un punto de entrada único para una característica presente en la jerarquía de herencia. En nuestro ejemplo se sugieren varias clases de este tipo, como la que permitiría una definición abstracta para el método is_connected.

- Otros nodos que no corresponden a clase o característica alguna del conjunto original pueden constituir una abstracción útil, en el sentido de que recogen los ancestros comunes a varias clases. Así en nuestro ejemplo, se sugiere la oportunidad de introducir una clase que herede de StreamInput y StreamOutput y sirva de ancestro para clases como InputOutputSTD y FileReadWrite.

Como hemos comprobado, la interpretación del retículo no es algo trivial y precisa de conocimientos matemáticos por parte del usuario, con lo que sería deseable un mayor acercamiento al desarrollador, facilitándole su labor a través de módulos de diagnostico que proporcionen informes en una terminología más cercana al usuario final. Dicho modulo sería una interesante futura ampliación con lo que facilitar la tarea al desarrollador.

Junto con el análisis de herencia, existen otras dos actividades que se complementan y permiten refinar los resultados de la técnica anterior estructurando la información sobre la forma en que las distintas entidades presentes en el código utilizan las características de su clase base. Así como detectar patrones regulares en la forma en la que se instancian los distintos métodos de un punto caliente de caja blanca. Esta información permitirá detectar en el punto caliente diversas dimensiones de variabilidad que lo componen, y ello facilitará su implementación con técnicas de caja negra. Ambas técnicas junto con el análisis de herencia forman parte del proceso en la elaboración de *frameworks* de dominio mediante técnicas de ACF visto anteriormente y son respectivamente el análisis de clientela y el análisis de dependencias funcionales. Nosotros nos hemos quedado en la primera dejando las otras dos técnicas para futuros proyectos y líneas de investigación.

5 Plan de Proyecto Software

5.1 Introducción

El objetivo de este Plan de Desarrollo Software es definir las actividades en términos de fases e iteraciones requeridas para desarrollar la herramienta objetivo del proyecto.

El documento está organizado en las siguientes secciones:

- Visión General del Proyecto — proporciona una descripción del propósito, alcance y objetivos del proyecto, estableciendo los artefactos que serán producidos y utilizados durante el proyecto.
- Organización del Proyecto — describe la estructura organizacional del equipo de desarrollo.
- Gestión del Proceso — explica los costos y planificación estimada, define las fases e hitos del proyecto y describe cómo se realizará su seguimiento.

5.2 Visión General

5.2.1 Propósito, Alcance y Objetivos

El siguiente capítulo provee una visión global del enfoque de desarrollo propuesto y aplicado en la realización del presente Proyecto Fin de Carrera. En él se detallará la los tiempos estimados cada fase del proceso de desarrollo como idea aproximada de la duración del proyecto. Hay numerosas necesidades que justifican dichas estimaciones y que serán puntos a completar el capítulo:

- Identificar, priorizar y justificar las necesidades de dichos recursos (en nuestro caso básicamente los recursos se reducen al tiempo de los alumnos que realizan el proyecto, y a los recursos software y hardware disponibles).
- Definir un inventario de tareas e hitos que deberán llevarse a cabo para la consecución del proyecto.
- Realizar un estudio de los riesgos, cuantificar su impacto y planificar y preparar los planes de contingencia que minimizarán los efectos de los riesgos.

El proyecto ha sido ofertado por Yania Crespo González-Carvajal siguiendo en la medida de lo posible una metodología de Proceso Unificado (Unified Process) (UP), y por tanto con una aproximación del sistema dirigido por los casos de uso, y encuadra su realización dentro del periodo comprendido entre Octubre de 2004 a Agosto del 2006 para la Escuela Técnica Superior de Ingenieros Informáticos de la Universidad de Valladolid, concretamente para la carrera de Ingeniero Técnico en Informática de Gestión.

El objetivo es la realización de una herramienta que permita aplicar una propuesta metodológica planteada para el proceso de *Desarrollo de frameworks de dominio mediante ACF*, y más concretamente de una de sus técnicas, el análisis de jerarquías de herencia. Así como la aplicación de dicha herramienta en el estudio de código *Java* introducido por el usuario para un posterior análisis de las relaciones de herencia existentes, así como las posibles relaciones que se podrían dar, para descubrir el alcance de la propuesta.

Además no conviene olvidar que el presente trabajo está basado inicialmente en dos proyectos anteriores [Pérez et al., 2002] y [DBRE, 2005].

5.2.2 Características

Como la propuesta planteada se basa en una técnica matemática denominada *Análisis de Conceptos Formales* (ACF), desarrollaremos una aplicación capaz de trabajar en ese marco de conocimiento con la consigna de que sea fácilmente adaptable a la introducción de múltiples tipos de algoritmos que utilicen las estructuras de conceptos básicas del ACF, así como a la introducción de nuevos formatos de archivo tanto para la entrada como la salida de la aplicación, véase:

a) Construcción de retículos de *Galois* de un contexto formal a partir del cual pueda extraerse conocimiento implícito.

b) Extracción de relaciones de herencia como conocimiento implícito del contexto formal previo cálculo del retículo de *Galois* a través de la interpretación de dicho retículo por parte del usuario. Este conocimiento servirá de apoyo en la búsqueda de relaciones de herencia existentes en el código objeto de estudio, así como las posibles relaciones que podrían darse en dicho código, de forma que facilite una posterior refactorización de dichas clases.

c) Permitir la incorporación de nuevos algoritmos así como nuevos formatos de archivo tanto para la importación como exportación, mientras no haya una DTD estándar definida. Cuando en un futuro próximo exista, se pueda incorporar de una manera sencilla como formato estándar por defecto.

d) Permitir la incorporación de nuevos parser de una forma fácil y sencilla, de forma que en cualquier momento se pueda incorporar un parser nuevo para cualquier lenguaje especificado.

e) Sencilla incorporación con otras etapas de proceso de construcción de *frameworks de dominio*.

f) Como objetivo de la propuesta planteada, sería la obtención del retículo de *Galois* y (como conocimiento implícito) que exprese las relaciones de herencia que podrían darse.

g) Aplicación de la herramienta a una serie de clases que constituyen el código objeto de estudio.

5.2.3 Restricciones Técnicas

En principio no existen restricciones técnicas importantes que podamos mencionar. Se ha decidido construir la aplicación en lenguaje *Java* por ser la tecnología más utilizada en este momento y su gran capacidad de portabilidad, aparte de facilitar la reutilización de código del

proyecto [DBRE, 2005]. De forma que será necesario la versión 1.5 de Java para el correcto funcionamiento del sistema. Pudiendo analizar ahora mismo código *Java* 1.5 y 1.4 (este ultimo no con todas las funcionalidades del 1.5). La aplicación esta desarrollada en *Java* y para análisis de código *Java*, código objeto de estudio. Sin embargo como hemos dicho antes en cualquier momento se puede incorporara un parser nuevo que reconozca cualquier otro lenguaje.

5.2.4 Entregables del Proyecto

A continuación se indican y describen cada uno de los artefactos que serán generados y utilizados por el proyecto y que constituyen los entregables. Esta lista constituye la configuración de UP desde la perspectiva de artefactos, y que proponemos para este proyecto.

Es preciso destacar que de acuerdo a la filosofía de UP (y de todo proceso iterativo e incremental), que todos los artefactos son objeto de modificaciones a lo largo del proceso de desarrollo, con lo cual, sólo al término del proceso podríamos tener una versión definitiva y completa de cada uno de ellos. Sin embargo, el resultado de cada iteración y los hitos del proyecto están enfocados a conseguir un cierto grado de completitud y estabilidad de los artefactos. Esto será indicado más adelante cuando se presenten los objetivos de cada iteración.

1) **Plan de Desarrollo del Software**

   Es el presente documento.

2) **Lista de Riesgos**

   Este documento incluye una lista de los riesgos conocidos y vigentes en el proyecto, ordenados en orden decreciente de importancia y con acciones específicas de contingencia para su mitigación.

Requisitos

3) **Especificación de requisitos del cliente (SRS)**

   Este documento define la visión del producto desde la perspectiva del cliente, especificando las necesidades y características del producto. Constituye una base de acuerdo en cuanto a los requisitos del sistema.

4) **Modelo de Casos de Uso**

   El modelo de Casos de Uso presenta las funciones del sistema y los actores que hacen uso de ellas. Se representa mediante Diagramas de Casos de Uso.

5) **Especificaciones de Casos de Uso**

   Para los casos de uso que lo requieran (cuya funcionalidad no sea evidente o que no baste con una simple descripción narrativa) se realiza una descripción detallada utilizando una plantilla de documento, donde se incluyen: precondiciones, poscondiciones, flujo de eventos, requisitos no-funcionales asociados. También, para casos de uso cuyo flujo de eventos sea complejo podrá adjuntarse una representación gráfica mediante un Diagrama de Secuencia centrado en el sistema.

6) **Especificaciones Adicionales**

   Este documento capturará todos los requisitos que no han sido incluidos como parte de los casos de uso y se refieren a requisitos no-funcionales globales. Dichos

requisitos incluyen: requisitos legales o normas, aplicación de estándares, requisitos de calidad del producto, tales como: confiabilidad, desempeño, etc., u otros requisitos de ambiente, tales como: sistema operativo, requisitos de compatibilidad, etc.

Análisis y diseño

7) Modelo de Análisis

Este modelo establece la realización de los casos de uso en clases, estableciéndolo en una representación en términos de análisis (sin incluir aspectos de implementación).

8) Modelo de Diseño

Este modelo establece la realización de los casos de uso en clases, pasando desde una representación en términos de análisis (sin incluir aspectos de implementación) hacia una de diseño (incluyendo una orientación hacia el entorno de implementación), de acuerdo al avance del proyecto. Incluye el modelo estático con Diagramas de Clases y un modelo dinámico con Diagramas de Secuencia. Asi como el diseño del interfaz.

Implementación

9) Modelo de Implementación

Este modelo es una colección de componentes y los subsistemas que los contienen. Estos componentes incluyen: ficheros ejecutables, ficheros de código fuente, y cualquier otro tipo de ficheros necesarios para la implantación y despliegue del sistema.

10) Casos de Prueba

Cada prueba es especificada mediante un documento que establece las condiciones de ejecución, las entradas de la prueba y los resultados esperados. Estos casos de prueba son aplicados como pruebas de regresión en cada iteración. Cada caso de prueba llevará asociado un procedimiento de prueba con las instrucciones para realizar la prueba.

Gestión del proyecto

11) Plan de Iteración

Es un conjunto de actividades y tareas ordenadas temporalmente, con recursos asignados, dependencias entre ellas. Se realiza para cada iteración, y para todas las fases.

12) Evaluación de Iteración

Este documento incluye le evaluación de los resultados de cada iteración, el grado en el cual se han conseguido los objetivos de la iteración, las lecciones aprendidas y los cambios a ser realizados.

Destinados al usuario

13) Manual de Instalación

Este documento incluye las instrucciones para realizar la instalación del producto.

14) Material de Apoyo al Usuario Final

Corresponde a un conjunto de documentos y facilidades de uso del sistema, incluyendo: Guías del Usuario, Guías de Operación, Guías de Mantenimiento y Sistema de Ayuda en Línea.

15) Producto

Los ficheros del producto empaquetados y almacenados en un CD-ROM con los mecanismos apropiados para facilitar su instalación. El producto, a partir de la primera iteración de la fase de Construcción, es desarrollado incremental e iterativamente, obteniéndose una nueva versión al final de cada iteración.

Este documento recoge concretamente los artefactos 1), 2) y 11) de la lista de entregables.

5.3 Organización del Proyecto

5.3.1 Recursos Humanos

Al tratarse de un Proyecto Fin de Carrera el único personal disponible presente en todas las fases del proceso de desarrollo, será:

- La tutora del proyecto, Carmen Hernández Díez, profesora del departamento de Informática de la Universidad de Valladolid, que desempeñará el rol de cliente y usuario de la aplicación permitiendo la obtención de los requisitos y funcionalidades deseadas de la herramienta a construir, así como la revisión de los resultados obtenidos del estudio de la base de datos. Todo ello gracias a una serie de reuniones celebradas desde el inicio del proyecto.

- Los alumnos encargados de la realización del proyecto fin de carrera, Luis David Barrios Alfonso y Alberto San Martín López, alumnos de último curso de Ingeniería Técnica en Informática de Gestión de la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Valladolid. Con una experiencia modesta en metodologías de desarrollo, herramientas CASE y notaciones, UML en particular, y en el proceso de desarrollo UP, así como en el desarrollo mediante el lenguaje *Java*. Asumiremos todos los roles posibles dentro del UP, Jefe de proyecto, Analista de Sistemas, Analista – Programador, Ingeniero del software y Responsable de pruebas. Como consecuencia de esto se producirá una concentración de tareas en las mismas personas pero dividiendo el areas donde actuar cada uno, de forma que se dividió la aplicación en dos partes claramente diferenciadas el parser, y la aplicación utilizando técnicas ACF. Cabe destacar que aunque los dos acaparamos todos los roles, establecimos un equipo de trabajo y colaboración en el desarrollo de las dos partes, con la correspondiente comunicación entre ambos.

5.3.2 Recursos Hardware

Como recursos hardware disponibles tenemos los laboratorios y equipos proporcionados por la Escuela de Informática de la Universidad de Valladolid incluyendo la conexión de alta velocidad de Internet. Además los alumnos cuentas con sus respectivos equipos en los domicilios

de cada uno que soportará la mayor parte de trabajo necesario para la realización del proyecto. Debido a la distancia que nos separa, ya que uno vive en Zamora, Luis David Barrios Alfonso, y otro en Valladolid, Alberto San Martín López, fue de vital importancia la comunicación a través de Internet, y de teléfono, para el desarrollo del presente proyecto.

Las características técnicas de los equipos de los alumnos son:

- AMD Athlon 64 3200+ a 2.01 GHz, 1 Gb RAM, 80 Gb disco duro con Microsoft Windows Xp y Ubuntu Linux 6.01.

- AMD Seprom 3100+ a 1.8 GHz, 1 Gb RAM, 80 Gb disco duro con Mandriva Linux 2006.

Con respecto a los equipos de los laboratorios nos movemos en rangos inferiores a los equipos descritos arriba, tanto en entornos Windows como Unix.

5.3.3 Recursos Software

Debido a la condición de alumno y a que se trata de un Proyecto Fin de Carrera (es decir no existe un aporte económico para sufragar la compra de software especializado) se utilizarán en la medida de lo posible software de libre distribución, aparte del software que provee la Universidad de Valladolid y que está instalado en los equipos de los laboratorios. En concreto, se ha empleado los siguientes programas:

- Eclipse SDK-3.1.2 con J2SDK 1.5.08 como editor de desarrollo *Java*.

- Microsoft Word y OpenOffice Writer como editores para la creación de los documentos creados para la entrega del proyecto (Word bajo licencia estudiantil).

- StarUML y ArgoUML como constructor de los diagramas que servirán para el análisis y diseño de la aplicación.

5.4 Gestión del Proceso

5.4.1 Análisis de Riesgos

A continuación realizaremos un pequeño estudio de los posibles riesgos que pueden afectar al desarrollo del proyecto provocando fallos en la planificación. Lo primero es identificar los riesgos a los que nos enfrentamos y después cuantificarlos para valorarlos y de esta manera posibilitar su control mediante el desarrollo de los planes de contingencia. Dichos planes se realizarán de forma que nos permitan evitar el riesgo o en el peor de los casos si éste se produce, mitigar su efecto en la planificación.

5.4.1.1 Identificación y Cuantificación de los Riesgos

Detallaremos en la siguiente lista los riesgos identificados:

1. **Disponibilidad temporal**. Al comienzo de la realización del proyecto los alumnos estaba a la espera de la realización de unas prácticas en empresas que limitarían considerablemente el tiempo que se dedicara al proyecto. Esta situación podría comprometer las fechas fijadas provocando un aumento considerable en la realización del proyecto.

2. **Mala planificación temporal (Falta de experiencia).** Debido a la poca experiencia en planificación temporal de los alumnos unido a la indeterminación en este tipo de proyectos en los que es necesario un periodo de aprendizaje del marco de conocimiento en el que se engloban, es posible que la estimación de tiempos no sea completamente realista, tomándose decisiones equivocadas y estimando mal las fechas, produciéndose desajustes en la finalización de las actividades.

3. **Tamaño del equipo de desarrollo.** El equipo de desarrollo está constituido únicamente por 2 integrantes. Esto puede provocar una carga excesiva de trabajo que provoque retrasos en la planificación. Además de por sí, aunque sea más de uno los integrantes del grupo, limita la paralelización de numerosas actividades lo que se traduce en un incremento de la duración del proyecto.

4. **Cambios en los requisitos.** En nuestro caso los requisitos estaban bien definidos desde un principio, consecuencia de la claridad en el objetivo del proyecto y de una serie de entrevistas iniciales entre la tutora y el alumno. Sin embargo, no conviene obviar este riesgo pues siempre es posible que se produzcan cambios en los requisitos (no drásticos) que puedan provocar un ligero incremento del tiempo de desarrollo.

5. **Análisis y diseño erróneos.** Es necesario un entendimiento claro de todos los requisitos y objetivos del proyecto recogidos en las fases iniciales para que el análisis y diseño del sistema partan de una base correcta. Sin embargo, esto no significa que ambos estén libres de errores. Por lo tanto realizar un análisis y diseño del sistema correcto desde un principio elimina la necesidad de revisiones continuas y evita que los errores se propaguen a las siguientes fases donde su factor de impacto en la planificación aumenta conforme nos acercamos al final del proyecto.

6. **Desconocimiento de las herramientas de desarrollo.** Por lo habitual el aprendizaje y habituación de las herramientas utilizadas en la construcción del proyecto será rápido y productivo, sin embargo conviene curarse en salud preparando un plan de contingencia que evite al alargamiento innecesario del proyecto.

En la **Tabla 5-1** se muestran los diferentes riesgos encontrados cuantificados y ordenados por el tamaño de la incidencia que pueden provocar en el desarrollo del proyecto. Esta ordenación nos permitirá concentrar nuestra atención, aumentando los recursos que dedicamos al control de unos riesgos en detrimento de los recursos que dedicamos a otros recursos menos decisivos.

Riesgo	Probabilidad	Impacto	Incidencia
Disponibilidad temporal	0.9	7	6.3
Mala planificación temporal	0.4	6	2.4
Tamaño del equipo de desarrollo	0.7	3	2.1
Análisis y diseño erróneos	0.3	5	1.5
Cambios en los requisitos	0.2	2	0.4
Desconocimiento de las herramientas de desarrollo	0.2	2	0.4

Tabla 5-1 *Cuantificación de los riesgos.*

Probabilidad		Impacto	
Muy baja	[0.0 – 0.2)	Despreciable	[0 – 3)
Baja	[0.2 – 0.4)	Marginal	[3 – 5)
Media	[0.4 – 0.6)	Critico	[5 – 8)
Alta	[0.6 – 0.8)	Catastrófico	[8 – 10)
Muy alta	[0.8 – 1.0]		

Tabla 5-2 *Valores de probabilidad e impacto de los riesgos.*

5.4.1.2 Planes de Contingencia

A continuación detallaremos las soluciones que deben adoptarse para evitar la aparición del riesgo, y en caso de que esto no sea posible resolver la situación minimizando el impacto o la ocurrencia potencial del riesgo en la planificación.

1. **Disponibilidad temporal.** Realizar una planificación con plazos suficientemente amplios para que absorban estas situaciones.

2. **Mala planificación temporal (Falta de experiencia).** Prolongar la jornada de trabajo, incluyendo tanto laborales como festivos y fines de semana para recuperar el tiempo que se haya perdido en la planificación errónea. Sin embargo, hay que tener en cuenta que un sobreesfuerzo continuado, lejos de ayudar, puede incluso repercutir negativamente, por mermar la capacidad de concentración y la resistencia del desarrollador. También, el acceso a documentación actualizada y foros de conocimiento como la asistencia del tutor pueden ayudar al ajuste correcto de la planificación.

3. **Tamaño del equipo de desarrollo.** Distribuir el trabajo sin imponer jornadas excesivamente largas, que a la larga redundarán en la elongación de la planificación.

4. **Cambios en los requisitos.** Realizar todas las entrevistas necesarias con el cliente hasta conseguir el conjunto completo de requisitos (si es necesario realizar prototipos gráficos para ayudar a la captura de los mismos).

5. **Análisis y diseño erróneos.** Entender y comprender perfectamente el conjunto capturado de requisitos mediante la realización del análisis y del diseño antes de realizar cualquier implementación de ellos. Si es necesario realizar varias revisiones de los mismos contrastándolos con el cliente.

6. **Desconocimiento de las herramientas de desarrollo.** Realizar una labor de búsqueda de bibliografía adecuada, actualizada y de calidad que facilite la labor de aprendizaje.

5.4.2 Plan de Proyecto

En esta sección se presenta la organización en fases e iteraciones y el calendario del proyecto.

5.4.2.1 Plan de Fases

El desarrollo se llevará a cabo sobre la base de fases con una o más iteraciones en cada una de ellas. La siguiente tabla muestra una la distribución de tiempos y el número de iteraciones de cada fase (para las fases de Construcción y Transición es sólo una aproximación preliminar, aparte de una estimación inicial optimista para todas las fases).

Fase	Nº de Iteraciones	Duración
Fase de Inicio	1	5 semanas
Fase de Elaboración	2	8 semanas
Fase de Construcción	3	30 semanas
Fase de Transición	2	6 semanas

Tabla 5-3 *Duración de las fases del UP*.

Los hitos que marcan el final de cada fase se describen a continuación:

- **Fase de inicio.** En esta fase se desarrollarán los requisitos del producto desde la perspectiva del usuario, los cuales serán establecidos en documento SRS (*Software Requirement Specification*). Los principales casos de uso serán identificados. La aceptación del cliente/usuario del documento SRS y del Plan de Desarrollo modificado marcan el final de esta fase.

- **Fase de Elaboración.** En esta fase se analizan los requisitos y se desarrolla un prototipo de arquitectura (incluyendo las partes más relevantes y/o críticas del sistema) y un prototipo de la interfaz gráfica. Al final de esta fase, todos los casos de uso correspondientes a requisitos que serán implementados en la primera versión de la fase de Construcción deben estar analizados y diseñados (en el Modelo de Análisis/Diseño).

  La revisión y aceptación del prototipo de la arquitectura y de la interfaz gráfica del sistema marca el final de esta fase. La primera iteración tendrá como objetivo la realización preliminar en el Modelo de Análisis/Diseño de los casos de uso identificados y especificados, también permitirá hacer una revisión general del estado de los artefactos hasta este punto y ajustar si es necesario la planificación para asegurar el cumplimiento de los objetivos. La segunda tendrá como objetivo el refinamiento de todos los productos artefactos obtenidos en la iteración anterior, consiguiendo un análisis y diseño completo y detallado que elimine errores en el desarrollo de las siguientes fases. Ambas iteraciones tendrán una duración de cuatro semanas (estimación inicial optimista).

- **Fase de Construcción.** Durante la fase de construcción se terminan de analizar y diseñar todos los casos de uso, refinando el Modelo de Análisis/Diseño. El producto se construye sobre la base de 3 iteraciones, cada una produciendo una versión que implementará un grupo de subsistemas de la aplicación: 1ª Iteración: subsistema gráfico principal, 2ª Iteración: subsistema ACF (con algoritmos de cálculo de retículos y construcción de la matriz de contexto), subsistema gráfico para retículos, y 3ª Iteración: subsistema del parser, con las reglas correspondientes a la propuesta planteada y el acceso del subsistema ACF a los datos del parser. A la aplicación se le aplicarán las pruebas pertinentes y se validará con el cliente/usuario. Se comienza la elaboración del material de apoyo al usuario. El hito que marca el fin de esta fase es la versión con toda la capacidad operacional del producto, lista para ser entregada a los usuarios para pruebas beta.

- **Fase de Transición.** En esta fase se entrenará al usuario en la nueva aplicación y se completará el proyecto con la realización del caso de estudio que nos permitirá sacar las

conclusiones adecuadas a la propuesta inicial planteada. El hito que marca el fin de esta fase incluye la entrega de toda la documentación del proyecto con los manuales de instalación y todo el material de apoyo al usuario, la finalización del entrenamiento de los usuarios, el empaquetamiento del producto y el documento que recoge el caso de estudio.

Para finalizar remarcaremos que es un estimación inicial optimista y que los tiempos pueden variar con respecto a lo que se estimó inicialmente.

5.5 Resultado de la Planificación

El plan de proyecto que se obtuvo de la planificación, aún con las elongaciones propuestas consecuencia de los riesgos planteados, se quedó corto. La fecha de entrega del proyecto tuvo que ser aplazada unos 12 meses y las fases alargadas, sobre todo la de implementación y transición. Hay que aclarar que en esos meses se incluye, vacaciones, fiestas locales, desplazamientos ausencias etc. Con la dificultad añadida de que los miembros del grupo somos cada uno de una ciudad lo cual dificultaba los días de reunión y obligaba establecer una comunicación a distancia por medio de Internet o el teléfono que es mucho mas lenta que una reunión en persona.

Aparte existía el riesgo de la realización de prácticas en empresa por parte de ambos alumnos, la cual se cumplió; sin embargo, se contaba con que se pudiera extraer una pequeña jornada de trabajo dedicada al proyecto durante la realización de las prácticas, situación que no se dio la mayoría de los días en que estas duraron. Además surgieron nuevos riesgos no controlados: a) la dificultad del marco de conocimiento en el que está englobado el proyecto, b) el tamaño, complejidad y estilo a veces desorganizado e incoherente de los proyectos fin de carrera en que nos basamos, c) así como la naturaleza investigadora de la propuesta planteada (hace necesario pautas de interpretación correctas para el retículo, estudiando todas las posibilidades que ofrece la herencia), d) aparición del proyecto [DBRE, 2005] en agosto de ese año, lo cual nos hizo dar marcha a tras en gran cantidad de artefactos sobre todo de diseño y algún prototipo ya creado, replanteándonos la nueva situación (descartamos la utilización de métodos nativos en la reutilización del proyecto de [Pérez, 2002] y optar por la reutilización directa del código de [DBRE, 2005] lo que llevaba a una nueva fase de análisis. Todo esto hizo que los tiempos planteados fueran insuficientes. No obstante, siempre es posible extraer algo positivo de los errores cometidos, y es el conocimiento adquirido en la realización y planificación del proyecto que servirá de experiencia en tareas futuras.

6 Documento de Análisis

6.1 Introducción

El siguiente documento recogerá los artefactos construidos durante el desarrollo del proyecto referentes a los requisitos y análisis del sistema, concretamente el 3), 4), 5), 6) (ver **Apartado 5.2.4**). La herramienta está destinada al desarrollador software que este desarrollando *frameworks de dominio* mediante técnicas ACF en la etapa preliminar de análisis de herencia, o simplemente para aquel desarrollador que quiera analizar determinado código para una posterior reutilización, u optimización del mismo viendo posibles relaciones de herencia existentes en dicho código. En cualquier caso será necesario que el usuario final tenga conocimientos en la interpretación de grafos, en este caso en la aplicación al análisis de jerarquías de herencia. La herramienta ofrece los resultados en forma de matriz de contexto y de retículo de *Galois*, ambos exportables en formato *XML* compatible con el proyecto anterior [DBRE, 2005] y en caso particular, el contexto compatible con la herramienta *Galicia* [Galicia, ref].

El documento está organizado en las siguientes secciones:

Propósito, alcance y objetivos del sistema — describe el propósito, alcance y objetivos del sistema.

Sistema actual — describe la situación actual del sistema a desarrollar, es decir cómo se realizan en la actualidad las tareas que soportará el nuevo sistema.

Sistema propuesto — describe todos los detalles en términos de requisitos funcionales y no funcionales, restricciones de diseño y el modelo de Casos de Uso con sus especificaciones.

Requisitos funcionales — describe en lenguaje natural la funcionalidad de alto nivel del sistema. El objetivo es establecer un catálogo de requisitos.

Requisitos no funcionales — describe en lenguaje natural los requisitos de usuario no relacionados directamente con la funcionalidad del sistema.

Modelo de Casos de Uso — presenta las diferentes formas detalladas en las que los usuarios interactúan con el sistema.

6.2 Propósito, Alcance y Objetivos del Sistema

El proyecto consiste en el análisis de jerarquías de herencia dentro de un código implementado en *Java 1.5* objeto de estudio detectando tanto las relaciones de herencia existentes en el código como las posibles relaciones de herencia que podrían realizarse, todo ello dentro del marco de desarrollo de *frameworks de dominio* mediante técnicas de *Análisis de Conceptos Formales*, y concretamente el la primera etapa del proceso, utilizando para ello una herramienta construida para tal propósito.

Recordemos que la herramienta se construye con el objetivo de dar soporte e implementar la propuesta metodológica planteada en capítulos anteriores y citada antes, *Desarrollo de frameworks de dominio mediante ACF*. A su vez debe servir como una herramienta con una arquitectura abierta e implementaciones genéricas para facilitar su adaptación en la incorporación

de nuevos *parsers*, algoritmos para la construcción de retículos de *Galois*, y exportadores e importadores de archivos de entrada y salida.

El ámbito de trabajo será un ordenador personal en donde se instalará la herramienta y funcionará de forma autónoma.

6.3 Sistema Actual

El método de *Análisis de Conceptos Formales* es usado principalmente para el análisis de datos, por ejemplo para investigación y procesamiento explicito de información, en inteligencia artificial, ingeniería del software, etc. Tales datos serán estructurados en unidades, las cuales son abstracciones formales de conceptos del pensamiento humano permitiendo su pleno significado y una interpretación comprensible. Es por ello que esta técnica puede utilizarse dentro de numerosos campos como una técnica de extracción de conocimiento. Debido a que uno de los objetivos de este trabajo es la aplicación de la técnica de ACF dentro del *Análisis de Jerarquías de Herencia*, inicialmente se partió de un proyecto anterior [Pérez et al., 2002] que utilizaba esta técnica dentro de un marco similar al nuestro. La aplicación final de este proyecto fue construida bajo un entorno *UNIX/Linux*, en lenguaje *Eiffel*. El núcleo del programa era el encargado de la representación de las estructuras de datos y de los algoritmos (concretamente el algoritmo de *Bordat*) necesarios para aplicar la técnica de ACF. Por ello el objetivo inicial era la extracción y compresión de este núcleo como base de la futura aplicación del presente proyecto. A pesar de las ventajas que ofrece la utilización de software libre, este núcleo era demasiado dependiente de la plataforma y de las librerías utilizadas para su construcción como ya se comento anteriormente (ver apartado 4.2 orígenes); una restricción demasiado fuerte para una aplicación que estará en consonancia con otras existentes para el desarrollo *frameworks de dominio*, y otras que realizan tareas similares pero en ámbitos distintos. Es por eso por lo que se optó por utilizar *Java* como lenguaje alternativo en la construcción de la aplicación del presente proyecto, debido entre otras ventajas, a su enorme portabilidad.

En principio se planteó la reutilización del código *Eiffe*l a través de los métodos nativos de *Java*, en un intento de evitar una migración de dicho código a *Java*. Sin embargo como ya se comento en el capitulo 4, gracias a la aparición del proyecto [DBRE, 2005] no fue necesario realizar tal cosa. Dicho proyecto aplicaba técnicas ACF dentro del marco de la reingeniería del software, se basaba también en el proyecto [Pérez, 2002] y lo más importante incorporaba todo el núcleo del citado proyecto en *Java*.

Sin embargo lo que por un lado lo que parecía inicialmente todo ventajas, llevaría consigo un incremento en el número de horas en el análisis de dicho proyecto, sin contar que gran cantidad de trabajo ya realizado quedaría sin uso. Sin embargo, a pesar del inconveniente del tiempo, el resto sería todo ventajas, pues el nuevo proyecto estaba enteramente desarrollado en *Java* y estábamos en posesión de una amplia documentación sobre el mismo, con lo que se podría ahorrar gran cantidad de tiempo en la fase de desarrollo pudiendo reutilizar gran parte de dicho proyecto. Aparte, toda la experiencia adquirida al empezar con el proyecto [Pérez, 2002], facilitaría una rápida compresión del núcleo de [DBRE, 2005] así como su análisis. Por lo que inclino la balanza para la utilización de dicho proyecto.

En cuanto al parser que realizaría el análisis del código objeto de estudio, existía un proyecto [Rodríguez y Gil, 1999] en el que se desarrollaba una herramienta para la herencia múltiple de clases en *Java*, en la que se desarrollaba un parser a través de *JavaCC*. Debido a que existía una gran dependencia del parser con el resto de las clases que integraban la aplicación fue del todo imposible su uso, sin embargo nos dio pie a desarrollar dicho parser desde el principio

apoyados en la herramienta *JavaCC*, programa que genera un analizador sintáctico para *Java* a partir de una especificación léxica de la sintaxis del lenguaje *Java*. Similar al *Lex* y *Yacc*.

6.4 Sistema Propuesto

6.4.1 Visión General

La herramienta se diseñará con la intención de ofrecer un entorno gráfico sencillo e intuitivo que permita la construcción de contextos formales rápidamente, a través del análisis de un flujo de entrada constituido por un conjunto de clases o de interfaces en lenguaje java 1.5, y que sirvan como entrada al algoritmo que construirá el retículo de conceptos donde se pondrán de manifiesto las relaciones de herencia existentes, así como las posibles relaciones de herencia que podríamos observar. Se debe realizar de una forma lo suficientemente genérica de forma que en un futuro se pueda incorporar nuevos algoritmos y analizadores de la forma sencilla.

Deberá permitir la visualización rápida del retículo de conceptos construidos así como de la matriz de contextos construida a partir de los datos obtenidos en el análisis del código objeto de estudio. Además permitirá guardar tanto el contexto, como el grafo obtenido en cualquier momento con formato XML para su posterior manipulación. Al no existir una DTD estándar para ACF, se guardara la compatibilidad con el proyecto [DBRE, 2005], así como se facilitara la incorporación de nuevos exportadores con el fin de incorporar la DTD estándar cuando haga su aparición. Del mismo modo, el formato de entrada de archivos será también en XML, pudiéndose abrir tanto retículos como matrices de contextos en los mismos formatos en que se pueden exportar y guardando la retrocompatibilidad con el proyecto [DBRE, 2005].

Todo ello para facilitar el estudio de las jerarquías de herencia en *Java* a través de ACF, que se realizará utilizando la herramienta construida. Teniendo en cuenta que el producto se desarrollará en *Java* no existirá ninguna restricción en cuanto a la arquitectura o Sistema Operativo de la máquina donde trabajará la aplicación (siempre y cuando dicha máquina tenga instalada una JVM –Java Virtual Machina en su versión 1.5 destinada a ese Sistema Operativo).

A continuación se describen los requisitos del sistema en forma de plantillas, a través de los cuales se especifica la funcionalidad que requiere el sistema, definiendo sus límites o limitaciones y especificando su comportamiento.

Los rangos de valores para el campo de **Prioridad** son:

- **Urgente.** Es un requisito que debe implementarse con la mayor brevedad posible.
- **Hay presión.** Aunque su implementación es un objetivo a medio plazo no debe perderse de vista.
- **Puede esperar.** Es un objetivo que se prevé cumplir a largo plazo, una vez se hayan alcanzado todos los demás.

6.5 Requisitos Funcionales

Número	REQ-001
Descripción	Permitir el análisis de una colección de clases o interfaces suministradas por el usuario, para la obtención de datos relevantes para la construcción del contexto formal.
Prioridad	Urgente.
Caso de Uso	CU-01 y CU-02
Comentarios	Dicho análisis se realizará mediante un parser que habrá que construir y que realizará un análisis léxico y sintáctico extrayendo la información relevante para la construcción del contexto, como son las características, y las clases así como las relaciones de herencia en el código.

Número	REQ-002
Descripción	Construcción de un contexto formal con la información obtenida en el requisito REQ-001
Prioridad	Urgente.
Caso de Uso	CU-01
Comentarios	Implementación de las estructuras de datos necesarias de los conceptos básicos asociados a la definición de contexto formal.

Número	REQ-003
Descripción	Modificación del contexto formal del requisito REQ-002 a través del análisis de nuevas clases o interfaces incorporadas a las que ya se habían analizado en REQ-001
Prioridad	Hay presión.
Caso de Uso	CU-03
Comentarios	Conlleva un reanálisis de las clases ya analizadas así como de las nuevas con el fin de detectar nuevas características generando un nuevo contexto formal REQ-002

Número	REQ-004
Descripción	Poder elegir el formato del archivo XML para la exportación de un contexto o un retículo de una lista de exportadores disponibles
Prioridad	Hay presión
Caso de Uso	CU-05
Comentarios	Actualmente constara de 2 exportadores para contextos, Galicia y DBRE, y 1 exportador para retículos, DBRE.

Número	REQ-005
Descripción	Guardar el contexto formal creado en REQ-002 en un fichero de datos XML con el formato elegido en REQ-004.
Prioridad	Hay presión.
Caso de Uso	CU-04
Comentarios	Se utilizara la definición de la DTD escogida en REQ-004.

Número	REQ-006
Descripción	Construcción de un retículo de *Galois* mediante el algoritmo de *Bordat* a partir del contexto formal de REQ-002.
Prioridad	Urgente.
Caso de Uso	CU-07
Comentarios	Implementación de las estructuras de datos necesarias de los conceptos básicos asociados a la definición de retículo.

Número	REQ-007
Descripción	Guardar el retículo construido en REQ-006 en un fichero de datos XML con el formato elegido en REQ-004.
Prioridad	Hay presión.
Caso de Uso	CU-06
Comentarios	Se utilizara la definición de la DTD escogida en REQ-004.

Número	REQ-008
Descripción	Visualización de la matriz de contexto formal obtenida en REQ-002 con una presentación clara de la información asociada a ella.
Prioridad	Urgente.
Caso de Uso	CU-09
Comentarios	Se construirá en forma de tabla de doble entrada, con los objetos a la izquierda y los atributos arriba, marcando con una señal dentro de la tabla cuando exista incidencia entre atributos y objetos.

Número	REQ-009
Descripción	Visualización del retículo de *Galois* obtenido en REQ-006 con una presentación clara de la información asociada a él.
Prioridad	Urgente.
Caso de Uso	CU-08
Comentarios	Debe existir movilidad entre los nodos para facilitar la visión y comprensión del retículo, escondiendo los conjuntos de intensión y extensión de los conceptos, y mostrándolos cuando se haga “click” en ellos con el ratón.

Número	REQ-010
Descripción	Poder cargar un archivo de datos XML previamente guardado con la información referida a un contexto formal o a un retículo
Prioridad	Puede esperar.
Caso de Uso	CU-01 y CU-07
Comentarios	Al abrir un contexto formal posteriormente se podra aplicar el REQ-006, y posteriormente REQ-007 ó REQ-005. Se empleara las DTDs Galicia y DBRE.

Número	REQ-011
Descripción	El usuario elegirá el analizador sintáctico que se empleara en el REQ-001.
Prioridad	Urgente.
Caso de Uso	CU-02
Comentarios	-

6.6 Requisitos No Funcionales

Número	REQ-012
Descripción	Toda la información generada por los diferentes algoritmos en el marco de la aplicación debe proporcionarse al usuario de la forma más sencilla y organizada posible.
Prioridad	Urgente.
Caso de Uso	Todos.
Comentarios	La información suministrada es indispensable para la trazabilidad de los algoritmos aplicados.

Número	REQ-013
Descripción	Utilización de *Java* como lenguaje de programación debido a sus excelentes características como su portabilidad, independencia de plataforma, simplicidad, seguridad,...
Prioridad	Urgente.
Caso de Uso	Todos.
Comentarios	-

Número	REQ-014
Descripción	Construcción de una arquitectura lo suficientemente abierta para su adaptación a nuevos algoritmos tanto en el ámbito del *Análisis de Conceptos Formales*, así como para la incorporación de nuevos parsers, y nuevas DTDs tanto para la importación como para la exportación
Prioridad	Urgente.
Caso de Uso	Todos.
Comentarios	Este requisito facilitará la adaptación de la aplicación ante la incorporación de dichos elementos.

Número	REQ-015
Descripción	La aplicación debe ser lo suficientemente práctica como servir de apoyo en el análisis de una colección de clases o interfaces y cuyo objetivo es el análisis de la jerarquía de herencia para la obtención de las partes comunes entre ellas en un proceso de desarrollo de *frameworks de dominio*.
Prioridad	Urgente.
Caso de Uso	Todos.
Comentarios	-

Número	REQ-016
Descripción	La aplicación debe constar de un modo de operación grafico y un modo de operación de línea de comandos.
Prioridad	Hay presión.
Caso de Uso	Todos.
Comentarios	Debe salvaguardar la funcionalidad de todos los requisitos.

Número	REQ-017
Descripción	La aplicación debe constar de un modo de operación grafico y un modo de operación de línea de comandos.
Prioridad	Hay presión.
Caso de Uso	Todos.
Comentarios	Debe salvaguardar la funcionalidad de todos los requisitos.

Número	REQ-018
Descripción	Deberá conservar la retrocompatibilidad en el archivo de datos XML empleado en el proyecto [DBRE, 2005] tanto para guardar como para cargar contextos o retículos.
Prioridad	Urgente.
Caso de Uso	Todos
Comentarios	Se utilizara la definición de la DTD del proyecto [DBRE, 2005].

6.7 Restricciones y Otras Consideraciones

La versión de *Java* que se empleara en el parser o analizador sintáctico es la versión 1.5, con todas las funcionalidades activas. También se ha desarrollado, pero sólo en esqueleto, un parser para la versión 1.4 de forma que para futuras incorporaciones de nuevos analizadores sintácticos de posteriores versiones de *Java* se siga ese mismo esquema. Con esto ponemos de manifiesto que sólo se podrá escoger el parser 1.5 para un perfecto funcionamiento.

Únicamente se analizaran o clases o interfaces, no ambos a la vez, es decir a la aplicación se la dará una serie de clases, o una serie de interfaces, no una mezcla de ambas.

Las clases o interfaces a analizar deben tener todos nombres distintos, independientemente del paquete en el que estén, es decir no se contemplara la duplicidad de nombres en paquetes distintos, ya que el analizador las consideraría la misma clase e induciría a una errónea interpretación de la jerarquía de herencia.

El conjunto de clases o interfaces que se vaya analizar por parte del usuario para detectar las posibles relaciones de herencia, debe ser valido, esto es, debe compilar perfectamente sin errores ni avisos o “warnings”, cumpliendo los estándar de *Java*.

A la hora de cargar un contexto formal, no se podrá modificar el analizador sintáctico, ya que si lo intentamos, como no hay información relevante al parser aplicado ni a la localización de las clases analizadas aparecerá vació. Lo mismo pasa al cargar un retículo de *Galois*, donde únicamente tendremos información relevante a dicho retículo.

La existencia de clases anidadas (Nested Class) dentro de las clases objeto de estudio no será tenida en cuenta ya que éstas influyen en funcionalidad únicamente dentro de las clases en que están contenidas, y a la hora de estudiar las relaciones de herencia, realmente a la que afecta es a la clase huésped que las contiene, y esa si que es tenida en cuenta. Hay que distinguir aquí qué es clase anidada y qué no lo es; si nos encontramos con una clase en la que antes de que acabe su especificación encontramos otra clase ahí tendremos un anidamiento de clases, mientras que si lo que tenemos es en un único archivo clase una especificación de clase detrás de otra, ahí no tendremos anidamiento de clases y eso si que se podría analizar.

La matriz de contexto formal representada a través de una tabla de doble entrada no admitirá modificación alguna que no sea a través de un nuevo análisis de clases, sin embargo si se permitirá el reordenamiento de columnas con el fin de facilitar al usuario una mejor visión de la misma pudiendo agrupar los atributos en los que tenga especial interés. Ya que cuando la tabla que representa la matriz adquiere dimensiones exageradas puede resultar incomodo el tener que andar desplazándose a través de barras de desplazamiento.

La aplicación únicamente representara un contexto formal por vez con su retículo de *Galois* correspondiente, es decir es mono – contextual. No pudiéndose representar más de un contexto formal a la vez. Ya que aunque en [DBRE, 2005] se puede trabajar con varios contextos a la vez, con Galicia no, por lo que se guarda la compatibilidad con ambos pero únicamente trabajando con un solo contexto formal.

En el modo consola para poder obtener el retículo de *Galois*, es necesario estar en un entorno operativo de ventanas, ya que sino no se podrá pintar el contexto, es decir que si estamos ejecutando la aplicación remotamente a través de un telnet, o un ssh, nos avisara de que no puede pintar el retículo.

6.7.1 Documentación del Usuario y Sistemas de Ayuda

Debido a que se trata de una aplicación específica con pocas opciones y muy concretas destinada a un usuario especializado la ayuda en línea que se proporcionara será la misma que podrá encontrar en el manual de usuario del cual trataremos en un capitulo, más adelante dentro del presente texto.

6.7.2 Diccionario de Datos

Debido a la extensión (más de 100 páginas) del diccionario de datos, éste será incluido directamente en el CD-ROM adjunto con la memoria del proyecto.

6.8 Modelo de Casos Uso

En esta sección se describen los escenarios y casos de uso que describen el sistema. Esta sección contiene la especificación funcional completa del sistema.

6.8.1 Descripción general de los actores

Usuario final: es el actor encargado de interactuar en primera persona con el sistema y destinatario de la aplicación. Se encarga de la realización del proceso del *Análisis de Jerarquías de herencia* aplicado a una serie de clases o interfaces que el mismo proporcionará y que serán el objeto del estudio, para la construcción del retículo de *Galois.* Podremos interpretar tanto las relaciones de herencia existentes, como las posibles relaciones de herencia que podrían darse, sacando los puntos comunes entre las distintas clases o interfaces. Todo ello apoyándose en las tareas semiautomáticas que es capaz de ofrecer la herramienta construida dentro del proceso de *Análisis de Jerarquías de herencia* dentro del marco de desarrollo de *frameworks de dominio* utilizando técnicas de *Análisis de Conceptos Formales* tanto en la herramienta como en el marco de desarrollo.

6.8.2 Escenarios

a) Abrir un contexto formal.

Descripción General: El usuario final puede abrir un contexto forma de un grupo de contextos formales almacenados en un archivo de datos XML (ese grupo puede ser de uno o mas contextos, pero sólo abrirá uno).

Actor: Usuario final.

Flujo de eventos:

1. El usuario final selecciona el menú **Archivo > Abrir** y selecciona el tipo de fichero con extensión *.setBin.xml* o *Bin.xml* según quiera abrir un archivo con compatibilidad DBRE o Galicia.

2. El usuario final selecciona el nombre del fichero que desea abrir y pulsa **Aceptar**.

3. El usuario selecciona el contexto que desea abrir del grupo de contextos que aparecerá en el archivo de datos cargado.

4. El programa recupera la información acerca de los contextos almacenada en el fichero, construyendo las estructuras de datos necesarias para representar esos contextos y visualizarlos de forma gráfica mediante una tabla de incidencia.

b) Analizar una colección de clases o interfaces.

Descripción: El usuario final desea analizar una serie de clases o interfaces *Java* que el mismo suministrará al sistema indicando su localización, así como el classpath y el parser a utilizar. El sistema obtendrá información relevante de ese análisis con el que poder construir la matriz de contexto. En versión gráfica, todo ello se realizaría en una ventana y en línea de comandos en una a través de los argumentos en el comando.

Actor: Usuario final.

Flujo de eventos:

1. El usuario final seleccionara **Archivo** > **Nuevo** (si realiza esta tarea en modo gráfico si lo realiza en modo línea de comandos únicamente tendrá que utilizar los parámetros -nogui). Introduce el classpath donde se encontraran las referencias a las clases, la versión del parser a utilizar y las clases o interfaces a analizar.

2. El usuario final da **Aceptar**, una vez ha introducido esos datos. En modo comandos no hará falta este paso.

3. El sistema inicializa el parser seleccionado con los datos introducidos y comienza el análisis recopilando la información necesaria para construir conceptualmente la matriz de contexto.

c) Crear un contexto formal:

Descripción: El usuario final puede crear un contexto a partir del análisis de una serie de clases o interfaces. Se hará automáticamente después de haber analizado dichas clases o interfaces. Se representara una matriz de incidencias en modo gráfico con un número objetos y atributos según el número de clases y características detectadas. Se marcara en dicha tabla donde exista incidencia entre un objeto y un atributo concreto.

Actor: Usuario final.

Flujo de eventos:

1. Previamente se ha dado el escenario a) o b).

2. El programa construye las estructuras de datos necesarias para representar el contexto.

3. Se representa la matriz conceptual, de una forma visual, obtenida del paso anterior marcando en una tabla como objetos las clases analizadas, como atributos, las características detectadas, y marcando entre ellos la incidencia en caso de existir.

d) Modificar contexto formal:

Descripción: El usuario final puede modificar el contexto formal creado mediante el análisis de nuevas clases que se añadirán a las ya analizadas; supondrá un nuevo reanálisis de las ya analizadas para ver nuevas incidencias con las nuevas introducidas. Únicamente esto se podrá hacer en modo gráfico, en modo línea de comandos no se podrá realizar.

Actor: Usuario final.

Flujo de eventos:

1. El usuario introducirá en el menú **Archivo** > **Modificar Parser** las nuevas clases a analizar, los nuevos classpath y podrá volver a elegir el parser que quiere utilizar.

2. El sistema añadirá las nuevas clases a la lista de las que ya tenia y aplicará los escenarios b) y c) con los nuevos datos obtenidos.

e) Seleccionar formato para la exportación de contextos y retículos:

Descripción: El usuario final podrá seleccionar el formato del archivo en el que se guardaran los datos del contexto construido o del retículo construido. Se seleccionaran de una lista de exportadores disponibles. En dicha lista será el lugar donde se cargaran automáticamente las futuras nuevas incorporaciones de DTDs. La forma de proceder será distinta según estemos en modo gráfico o en modo consola. Si no selecciona ninguno, por defecto serán DBRE tanto para contextos como para retículos.

Actor: Usuario final.

Flujo de eventos:

1. El usuario seleccionara directamente el formato de los exportadores a través de parámetros como se vera en el manual de usuario si el modo a proceder es por línea de comandos. Si es a través de modo gráfico seleccionará en **Ver** > **Opciones** el exportador deseado para contextos y el exportador deseado para retículos de una lista de posibles exportadores.

2. El usuario pinchara en **Aceptar** si está en modo grafico si no lo está ya habría acabado en la etapa anterior.

3. El sistema establecerá los exportadores seleccionado en el sistema, si no hay ninguno seleccionado, utilizara por defecto el formato utilizado en el proyecto [DBRE, 2005].

f) Guardar contexto formal

Descripción: El usuario final podrá guardar un contexto formal en un archivo de datos XML con el formato seleccionado en el escenario e). Previamente tiene que existir dicho contexto formal, con lo que se han tenido que dar los escenarios a) ó b) y c).

Actor: Usuario final.

Flujo de eventos:

1. El usuario selecciona en el menú **Archivo** > **Guardar Contexto**, en modo línea de comandos será con otros parámetros como se vera en el manual de usuario.

2. A continuación selecciona el directorio donde almacenar el fichero y teclea el nombre de éste. Tras ello el ingeniero pulsa **Aceptar**.

3. El programa construirá el fichero XML basándose en las estructuras de datos que representan a todos los contextos. Le añadirá la extensión *.setBin.xml* o *.Bin.xml* según se trate de DBRE o de Galicia respectivamente; y se lo añadirá en caso de que el usuario no lo haya introducido.

g) Abrir retículo de *Galois*

Descripción: El usuario final puede cargar en el sistema un retículo, que previamente se haya construido, a partir de un fichero de datos XML en el que se admite el formato DBRE. En dicho archivo de datos puede existir un grupo de retículos, más de uno, de forma que el usuario deberá seleccionar cual de todos los almacenados seleccionar.

Actor: Usuario final.

Flujo de eventos:

1. El usuario seleccionará el archivo de datos a cargar a través del menú **Archivo** > **Abrir** donde aparecerá un menú de selección de archivos.

2. El sistema abrirá el archivo con extensión *.setLat.xml* y mostrara un dialogo al usuario con los distintos retículos que contiene dicho archivo.

3. El usuario seleccionara un retículo del grupo de retículos disponibles en dicho archivo.

4. El programa recupera la información acerca del retículo seleccionado, construyendo las estructuras de datos necesarias para representar ese retículo y visualizarlo de forma gráfica mediante un grafo.

h) Construir Retículo de *Galois*:

Descripción: El usuario final puede construir un grafo reticular que represente al retículo de *Galois* de un contexto dado. Tienen que haberse dado los escenarios a) ó b) y c) ó d)

Actor: Usuario final.

Flujo de eventos:

1. El usuario selecciona la opción del menú **Acción > Construir Retículo > Algoritmo de *Bordat***. Éste será el algoritmo que utilice para el cálculo del retículo (la aplicación está abierta para la introducción de más algoritmos). Tras ello el ingeniero pulsa **Aceptar**.

2. El programa aplicará el algoritmo utilizado construyendo las estructuras de datos necesarias para representar al retículo y lo visualizará en forma de grafo.

i) Guardar Retículo de *Galois:*

Descripción: El usuario final podrá guardar un retículo de *Galois* en un archivo de datos XML con el formato seleccionado en el escenario e). Previamente tiene que haberse construido el retículo de *Galois* correspondiente al contexto formal que estemos tratando, es decir tiene que darse el escenario h).

Actor: Usuario final.

Flujo de eventos:

1. El usuario selecciona en el menú **Archivo** > **Guardar Retículo**, en modo línea de comandos será con otros parámetros como se vera en el manual de usuario.

2. A continuación selecciona el directorio donde almacenar el fichero y teclea el nombre de éste. Tras ello el ingeniero pulsa **Aceptar**.

3. El programa construirá el fichero XML basándose en las estructuras de datos que representa al retículo. Le añadirá la extensión *.setLat.xml* y se lo añadirá en caso de que el usuario no lo haya introducido.

j) Visualizar retículo de *Galois*:

Descripción: El usuario final puede visualizar en una ventana el grafo reticular que represente al retículo de *Galois* de un contexto dado si éste ya fue calculado anteriormente por el escenario a) ó b) y c) o d).

Actor: Usuario final.

Flujo de eventos:

1. El usuario seleccionará en el menú **Ver** > **Retículo de *Galois***.

2. El sistema seleccionará el retículo construido en el escenario h) y lo visualizará.

k) Visualización del contexto formal:

Descripción: El usuario final puede visualizar en la ventana de GUI la matriz de contexto formal ya calculado anteriormente por el escenario a) ó b) y c) o d).

Actor: Usuario final.
Flujo de eventos:

1. El usuario seleccionará en el menú Ver > Matriz de Contexto.

2. El sistema seleccionará el contexto construido en los escenarios a) ó b) y c) o d) y lo visualizara.

6.8.3 Diagrama del Modelo de Casos de Uso

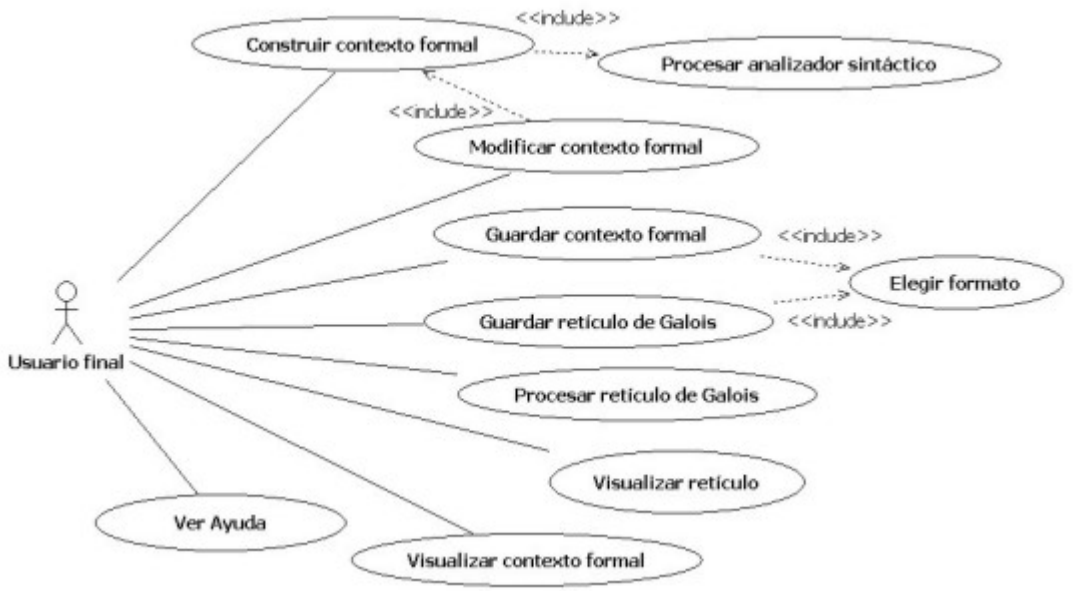

Figura 6-1 *Diagrama de Casos de Uso.*

A continuación pasamos a describir los Casos de Uso de nuestro sistema de forma detallada:

6.8.4 Descripción de los Casos de Uso

CU-01 <Construir contexto formal>

Descripción breve:

Mediante este Caso de Uso contemplamos la apertura o creación de un contexto formal, así como modificación, a partir del análisis de una serie de clases o interfaces introducidas por el propio usuario. Por lo tanto, el Caso de Uso incluye los escenarios siguientes: a), b), c),d) j).

Actor:

Usuario final.

Precondición:

Para ejecutar los escenarios c), y j) antes debe haberse aplicado los escenarios b) ó a).

Flujo de eventos:

Flujo básico:

1. El caso de uso comienza cuando se ejecuta el escenario a), abrir un contexto formal o b) analizar una colección de clases o interfaces, con distintas acciones a realizar según sea uno u otro.

2a. Si se da el escenario a)

   I. El usuario seleccionara el archivo de datos donde esté almacenado el contexto que quiere cargar.

   II. Se leerá dicho archivo cargándose su contenido.

   III. El usuario seleccionara que contexto cargar del archivo de datos.

2b. Si se da el escenario b)

   I. Se ejecutara el caso de uso **CU-02**, procesar analizador sintáctico.

3. Con los datos obtenidos de los dos pasos anteriores la aplicación construirá la matriz de contexto.

4. Una vez construida la matriz de contexto, la aplicación visualizara dicha matriz en forma de tabal de doble entrada, teniendo a un lado las clases (objetos) y al otro las características (atributos) señalando en el cruce de ambas con una “v” cuando haya incidencia y con un 0 cuando no la haya.

Flujos alternativos:

Escenario a), abrir un contexto formal:

<Archivo erróneo #1>

1. El archivo que se pretende abrir no existe o está defectuoso, o se carece de los permisos necesarios.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Formato XML erróneo #1>

1. Al abrir un fichero que no posea el formato XML con la DTD definida para el almacenaje de un grupo de contextos formales.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

Poscondición:

El contexto se ha creado correctamente y es visualizado por pantalla.

Relaciones:

Este relacionado con el Caso de Uso **CU-02** mediante una relación de <<include>>, ya que para crear un contexto si no es cargándose a través de la lectura de un archivo de datos tiene que ser a través del análisis de clases suministradas por el actor.

CU-02 <Procesar analizador sintáctico>

Descripción breve:

Mediante este Caso de Uso contemplamos el análisis de las clases o interfaces aportadas por el actor, teniendo lugar el escenario b).

Actor:

Usuario final.

Flujo de eventos:

Flujo básico:

1. El actor seleccionará las clases o interfaces a analizar.

2. El actor seleccionará el analizador sintáctico a emplear.

3. El actor seleccionará el analizador sintáctico a emplear.

4. El actor introducirá el o los directorios donde buscar las referencias realizadas en las clases o interfaces a analizar.

5. La aplicación comenzara analizar dichas clases obteniendo los datos necesarios para la construcción del contexto formal. Esto es las características, las clases, la relación de incidencia entre ambas así como las relaciones de herencia en código que existan en dichas clases.

6. Dicha información se guardara en el sistema para su posterior uso en los escenarios b) y d).

Flujos alternativos:

Escenario b), analizar una colección de clases o interfaces:

<Archivo(s) erróneo(s) #1>

1. El archivo o los archivos que se pretende analizar no existen o están defectuosos o se carece de los permisos necesarios.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Directorio no válido #2>

1. Al mirar el directorio introducido por el actor, vemos que no existe o se carece de los premisos necesarios.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

Poscondición:

Se han analizado las clases o interfaces suministradas guardándose la información obtenida para su posterior uso.

CU-03 <Modificar contexto formal>

Descripción breve:

El usuario quiere modificar el contexto formal ya creado por medio de la introducción de nuevas clases o interfaces. Mediante este Caso de Uso contemplamos la modificación del contexto formal construido a través de la incorporación de nuevas clases o interfaces a analizar, a las que ya se habían incorporado de forma que se tendrán que volver a analizar para descubrir nuevas incidencias con las nuevas clases o interfaces introducidas. Una vez construido el nuevo contexto formal se visualizara. Contempla los escenarios d), b), c).

Actor:

Usuario final.

Precondición:

Para ejecutar los escenarios c) antes debe haberse dado el b), lo mismo pasa con el d) que requerirá de la existencia de un analizador sintáctico ya definido con las clases que ya habían sido analizadas y el directorio de búsqueda utilizado.

Flujo de eventos:

Flujo básico:

1. El sistema carga los archivos ya analizadas por el analizador sintáctico, así como el directorio de búsqueda y el tipo de analizador utilizado.

2. Se muestra por pantalla dicha información.

3. El actor realiza las modificaciones pertinentes, tales como añadir clases o interfaces nuevos, según se este analizando una cosa u otra, quitar clases, cambiar de analizador sintáctico, añadir nuevos directorios de búsqueda, quitar alguno de los que ya estaban…

4. Fijar los cambios aplicados por el actor.

5. Aplicar caso de uso **CU-01**

Flujos alternativos:

Escenario b), analizar una colección de clases o interfaces:

<Archivo(s) erróneo(s) #1>

1. El archivo o los archivos que carga el actor no existen o están defectuosos, o no hay permisos para acceder a ellos.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Directorio no válido #2>

1. El o los directorios cargados, ya sean nuevos o antiguos ya no existen o no hay permisos para acceder a él.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

Poscondición:

Se han analizado las clases o interfaces suministradas guardándose la información obtenida para su posterior uso.

Relaciones:

Este relacionado con el Caso de Uso **CU-01** mediante una relación de <<include>>, ya que una vez redefinido los nuevos archivos a analizar se procede a la construcción del contexto formal previo análisis del nuevo conjunto de archivos.

CU-04 <Guardar contexto formal>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario f) *guardar contexto formal*, donde una vez que tenemos un contexto formal construido se guardara en el sistema con toda la información relevante. También contempla el escenario e) *Seleccionar formato para la exportación de contextos y retículos*.

Actor:

Usuario final.

Precondición:

Debe existir un contexto formal antes de poder aplicar este caso de uso, en otras palabras debe haberse dado antes el escenario c).

Flujo de eventos:

Flujo básico:

1. Aplicamos caso de uso **CU-05**.
2. El actor selecciona en el sistema el lugar donde quiere guardar el contexto formal.
3. El actor introduce el nombre que quiere que tenga su archivo.
4. La aplicación guarda en el sistema el archivo de datos con la información relativa al contexto formal.

Flujos alternativos:

Escenario f), guardar contexto formal:

<Directorio no valido #1>

1. El directorio destino que selecciona el actor no existen o están defectuosos, o no hay permisos para acceder a él.
2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Nombre no válido #2>

1. El nombre introducido por el actor no es valido.
2. La herramienta informará al actor mediante un mensaje por pantalla avisando que ha introducido un nombre no válido.

<Fallo en escritura #3>

1. La aplicación no ha podido guardar en el sistema el archivo con la información por falta de espacio en el lugar de almacenamiento.

2. La herramienta informará del fallo al actor mediante un mensaje por pantalla.

Poscondición:

Se obtiene un archivo de datos XML donde se ha guardado la información referente al contexto formal.

Relaciones:

Este relacionado con el Caso de Uso **CU-05** mediante una relación de <<include>>, ya que se tendrá que dar antes dicho caso de uso para la obtención del formato en que quiere guardar el contexto formal, el actor.

CU-05 <Elegir formato>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario e) *Seleccionar formato para la exportación de contextos y retículos*, donde el actor elegirá el formato de los archivos para su exportación de una lista de formatos disponibles. Pudiéndolo hacer en cualquier momento sin la necesidad de esperar a que este creado un contexto o un retículo.

Actor:

Usuario final.

Flujo de eventos:

Flujo básico:

1. La herramienta cargara todos los formatos disponibles para la exportación de archivos de datos XML.

2. El actor escogerá un formato entre los disponibles para contexto y otro para retículo.

3. La herramienta registrara la elección del actor para su posterior uso.

Flujos alternativos:

Escenario e), Seleccionar formato para la exportación de contextos y retículos:

<Elección vacía #1>

1. El actor no ha escogido ningún formato para la exportación.

2. La herramienta tomará por defecto la DTD del proyecto anterior [DBRE, 2005] tanto para contexto como para retículo.

Poscondición:

Se obtiene el formato para el archivo de datos XML donde se guardaran o el contexto o el retículo.

CU-06 <Guardar retículo de *Galois*>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario i) *guardar retículo de Galois*, para el que previamente tiene que haberse construido un retículo de *Galois*, escenario h). Se guardara en el sistema con toda la información relevante. También contempla el escenario e) *Seleccionar formato para la exportación de contextos y retículos.*

Actor:

Usuario final.

Flujo de eventos:

Flujo básico:

1. Aplicamos caso de uso **CU-05**.
2. El actor selecciona en el sistema el lugar donde quiere guardar el contexto formal.
3. El actor introduce el nombre que quiere que tenga su archivo.
4. La aplicación guarda en el sistema el archivo de datos con la información relativa al retículo de *Galois*.

Flujos alternativos:

Escenario i), guardar retículo de Galois:

<Directorio no valido #1>

1. El directorio destino que selecciona el actor no existen o están defectuosos, o no hay permisos para acceder a él.
2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Nombre no válido #2>

1. El nombre introducido por el actor no es valido.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que ha introducido un nombre no válido.

<Fallo en escritura #3>

1. La aplicación no ha podido guardar en el sistema el archivo con la información por falta de espacio en el lugar de almacenamiento.

2. La herramienta informará del fallo al actor mediante un mensaje por pantalla.

Poscondición:

Se obtiene un archivo de datos XML donde se ha guardado la información referente al retículo de *Galois*.

Relaciones:

Este relacionado con el Caso de Uso **CU-05** mediante una relación de <<include>>, ya que se tendrá que dar antes dicho caso de uso para la obtención del formato en que quiere guardar el retículo de *Galois*, el actor.

CU-07 <Procesar retículo de *Galois*>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario g) *abrir retículo de Galois, y h) construir retículo de Galois*, donde como el propio nombre indica se construirá el retículo ya sea a través de un flujo de entrada de archivo de datos XML o a través de un contexto formal ya creado. Una vez creado se mostrara, contemplando el escenario j) *visualización del retículo de Galois.*

Actor:

Usuario final.

Precondición:

Para que se pueda dar este caso de uso es necesario que previamente se haya cargado correctamente el archivo fuente donde este almacenado el retículo, escenario g) o que exista un contexto formal del que poder construir dicho retículo, escenario c).

Flujo de eventos:

Flujo básico:

1. Si se da el escenario g), *abrir retículo de Galois,*

   I. El actor introducirá la localización del archivo de datos a cargar.

   II. La herramienta cargará el archivo seleccionado.

III. El actor escogerá el retículo que se cargara en el sistema entre los almacenados en el archivo.

2. Sino es que se esta dando el escenario h), construir el retículo a partir de un contexto formal.

   I. A partir del contexto formal construido previamente, la aplicación ejecuta el algoritmo de *Bordat* para la obtención del retículo.

   II. La herramienta construirá todas las estructuras de datos necesarias y presentará los resultados por pantalla en forma de grafo reticular.

3. Una vez calculado el retículo el actor puede:

   a. Eliminar la ventana de presentación del retículo. Si desea volver a verla deberá ejecutar el escenario j) en el caso de uso **CU-08**.

   b. Guardar dicho retículo en un archivo de datos, aplicando el caso de uso **CU-06**.

   c. Generar un nuevo contexto formal, o modificarle según el escenario c) y d) en cuyo caso finalizaría este caso de uso.

Flujos alternativos:

Escenario g), abrir retículo de Galois:

<Archivo no valido #1>

1. El archivo seleccionado o no existe, o está dañado, o no se tienen los permisos adecuados para tratarle.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

<Formato XML erróneo #2>

1. Al abrir un fichero que no posea el formato XML con la DTD definida para el almacenaje de retículo de conceptos disponibles.

2. La herramienta informará al actor mediante un mensaje por pantalla avisando que no se ha podido realizar la operación.

Poscondición:

Se obtiene el retículo de *Galois*, visualizado en una ventana aparte, referido al contexto formal que estemos tratando o al referido en el fichero que hemos cargado.

CU-08 <Visualizar retículo>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario j) *visualizar retículo de Galois.* De forma que para ello ya tiene que estar construido dicho retículo, escenario h), caso de uso **CU-07.** Se aplica para cuando hemos cerrado la ventana donde se podía observar dicho retículo, después de haberlo construido.

Actor:

Usuario final.

Precondición:

Para que se pueda dar este caso de uso es necesario que previamente se haya construido correctamente el retículo de *Galois*, escenario h).

Flujo de eventos:

Flujo básico:

1. El actor le comunica a la herramienta su deseo de volver a ver el retículo de *Galois* que previamente ha construido.

2. La herramienta vuelve a dibujar el grafo reticular a partir del retículo ya procesado.

Flujos alternativos:

Escenario g), abrir retículo de Galois:

<Retículo desaparecido #1>

1. Se ha cerrado el archivo de donde se ha cargado el retículo.

2. Se deshabilita la opción de visualizar retículo.

Poscondición:

Se visualiza el retículo de *Galois* referido al contexto formal que estemos tratando o al referido en el fichero que hemos cargado.

CU-09 <Visualizar contexto formal>

Descripción breve:

Mediante este Caso de Uso contemplamos el escenario k) *visualizar contexto formal.* De forma que para ello ya tiene que estar construido dicho contexto, escenario c), caso de uso **CU-01**. Dibujara la matriz de incidencias como tabla de doble entrada.

Actor:

Usuario final.

Precondición:

Para que se pueda dar este caso de uso es necesario que previamente se haya construido correctamente el contexto formal, escenario c).

Flujo de eventos:

Flujo básico:

1. El actor le comunica a la herramienta su deseo de volver a ver el contexto formal que previamente ha construido.

2. La herramienta vuelve a dibujar la matriz de incidencia a partir del contexto ya procesado.

Poscondición:

Se visualiza la matriz de incidencia referido al contexto formal que estemos tratando o al referido en el fichero que hemos cargado.

CU-10 <Ver ayuda>

Descripción breve:

Mediante este Caso de Uso se visualiza el manual de usuario para cualquier duda que pueda tener el actor en el uso de la aplicación.

Actor:

Usuario final.

Flujo de eventos:

Flujo básico:

1. El actor le comunica a la herramienta su deseo de obtener ayuda en el manejo de la propia herramienta.

2. La herramienta cargar el manual de usuario donde se describe su utilización.

Flujos alternativos:

<Manual no disponible #1>

1. El manual no esta disponible en el sistema por la razón que sea.

2. Se avisa al actor de dicho problema mediante un mensaje por pantalla y se termina la operación.

Poscondición:

Se visualizan los contenidos de ayuda referentes al manejo de la herramienta.

7 Documento de Diseño

7.1 Introducción

El siguiente documento recogerá los artefactos construidos durante el desarrollo del proyecto referentes al diseño del sistema, concretamente el 8), y 10) (ver **Apartado 5.2.4**) estando el 9) en el CD-ROM y 12), 13), 14) y 15) tratados más adelante. En el diseño se hará hincapié en los requisitos no funcionales detectados en el análisis tratando de conseguir una arquitectura abierta que garantice características como adaptabilidad, extensibilidad o reutilización. La descripción del pseudo-código del principal algoritmo aplicado a la extracción de conocimiento sobre los contextos formales proviene del proyecto [DBRE, 2005] con lo que nos remitiremos a él directamente. Pasa lo mismo con las DTD's utilizadas, ya que una es la del proyecto [DBRE, 2005], descrita ampliamente en su documentación, y otra es la de Galicia [Galicia, ref] descrita en la página Web de dicho proyecto.

El documento está organizado en las siguientes secciones:

Arquitectura actual — describe la arquitectura del sistema de [DBRE, 2005] y [Pérez, 2002], proyectos en los que nos basamos.

Arquitectura propuesta — documenta el modelo de diseño del sistema para el nuevo sistema planteado en [Prieto, Crespo, Marques y Laguna, 2003].

Diagrama de clases — describe de una forma estática el sistema desarrollado, se plantearan en subsistemas debido a la extensión del mismo.

Diagrama de secuencia — documenta de una forma dinámica el sistema desarrollado.

Casos de prueba — donde veremos como se comporta la aplicación, comprobando si los resultados obtenidos ante determinados casos de prueba son realmente los que debería dar.

Interfaz de usuario — donde hablaremos del diseño de la interfaz viendo como esta planteada en el sistema desarrollado.

7.2 Arquitectura actual del sistema

En el sistema actual podemos encontrar dos arquitecturas distintas, ya que nos basamos en 2 proyectos distintos que guardan similitudes tanto funcionales como de diseño. Analizaremos tales arquitecturas y plantearemos una propia para el sistema software a desarrollar.

7.2.1 Arquitectura planteada por [Pérez, 2002]

Recordemos que el presente trabajo se basa en parte en otro proyecto [Pérez et al., 2002] que destinaba el uso del *Análisis de Conceptos Formales* para evaluar un sistema software. La arquitectura desarrollada para la implementación de dicho proyecto se basaba en una arquitectura por capas relativamente sencilla dividida en 3 subsistemas básicos como puede verse en la siguiente:

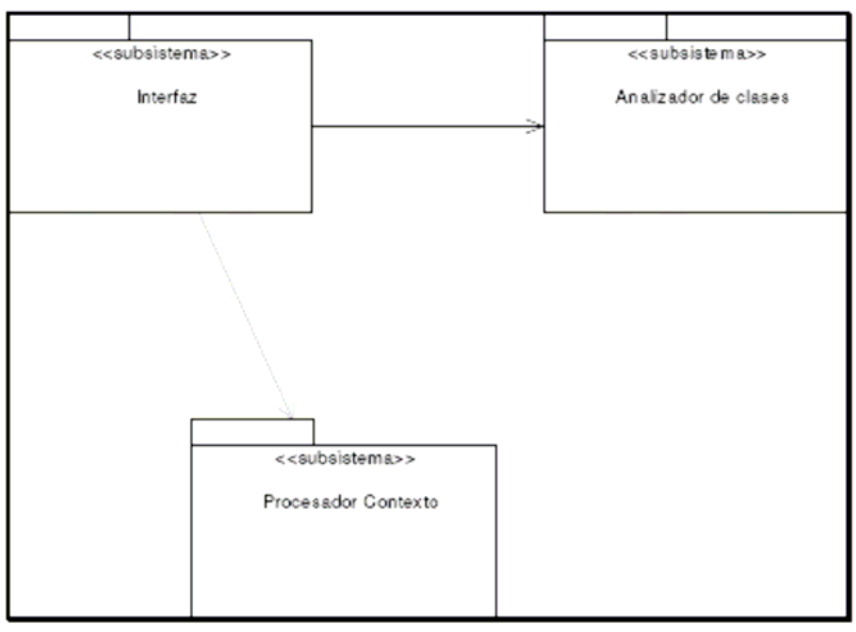

Figura 7-1 *Arquitectura PFC [Pérez et al., 2002].*

En este diagrama se pueden apreciar las tres partes claramente diferenciadas en que se descompone el sistema:

- El subsistema *Interfaz*, como la capa más alta, se encarga de toda la comunicación de entrada y salida con el usuario. Es quien se encargará de llamar a los otros dos subsistemas, como capas inferiores al mismo nivel, para ejecutar las tareas que tienen asignadas.

- El subsistema *Analizador de clases* se encarga de realizar un análisis de las clases, para obtener de ellas la información que permita construir el contexto formal.

- El subsistema *Procesador Contexto* se encarga de procesar un contexto formal para obtener su retículo de *Galois* asociado.

Para el sistema de [DBRE, 2005] se reutiliza y adapta el subsistema *Procesador Contexto* que será el responsable de modelizar las nociones matemáticas referentes al *Análisis de Conceptos Formales*, sus estructuras de datos y algoritmos. El nuevo sistema por tanto, seguiría la arquitectura por capas planteada inicialmente sólo que será adaptada y ampliada en el proyecto [DBRE, 2005] para cumplir con los requisitos no funcionales especificados en ese mismo proyecto.

7.2.2 Arquitectura planteada por [DBRE, 2005]

Debido a que el proyecto de [DBRE, 2005] se basa en el de [Pérez, 2002], como ya se ha comentado anteriormente, con la peculiaridad de que está completamente implementado en *Java*, garantizando así la portabilidad y facilitándonos la reutilización del módulo comentado anteriormente encargado de procesar tanto la matriz como el retículo a través de técnicas ACF. Gracias a esa peculiaridad la integración en nuestra aplicación fue más sencilla que tener que adaptar el anterior proyecto implementado en *Eiffel*.

La arquitectura del sistema [DBRE, 2005] responde a una arquitectura tradicional por capas, con un nivel inferior compuesto por el entorno de *Java* y un nivel superior dedicado a la interacción entre los usuarios y la plataforma.

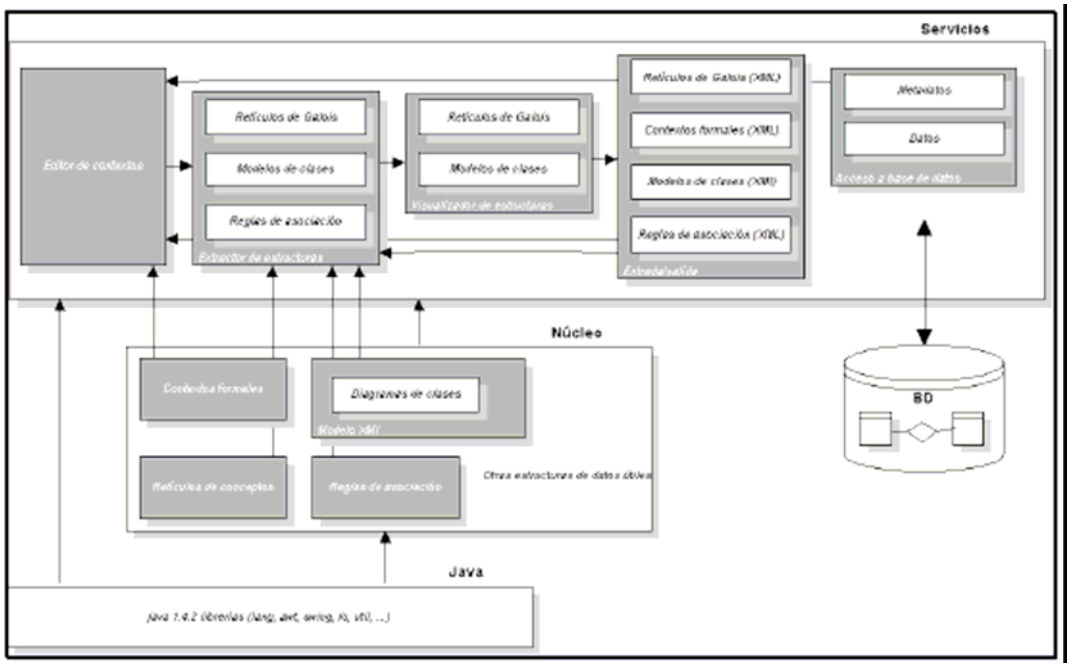

Figura 7-2 *Arquitectura en capas del sistema DBRE.*

Se identifican dos niveles bien diferenciados: el *núcleo* y la capa de *servicios*. El *núcleo* constituye el soporte relativo a construcción y manipulación de contextos, retículos de conceptos, modelos de clases y reglas de asociación, para que la capa de *servicios* pueda efectuar su trabajo. La capa de *servicios* utilizará las funciones que ofrece el *núcleo* para la implementación de los métodos de alto nivel del *Análisis de Conceptos Formales* y del proceso de *Ingeniería Inversa*, que permitirán extraer conocimiento de los contextos formales.

7.3 Arquitectura Propuesta

A continuación presentamos la estructura global de la arquitectura del sistema y una breve descripción de la asignación de funcionalidad de cada subsistema, así como, la descripción de la arquitectura software elegida para el sistema (patrón arquitectónico).

El nuevo sistema se ha diseñado como una plataforma abierta y su implementación se realizará en *Java* garantizado una alta portabilidad del sistema entero. Se ha mostrado una particular atención en el diseño de la arquitectura para lograr una serie de características deseables como adaptabilidad, extensibilidad y reutilización. La arquitectura del sistema se basa en la planteada en [Prieto, Crespo, Marques y Laguna, 2003] para el desarrollo de *frameworks de dominio* basado en técnicas ACF. Dicha arquitectura tiene por objetivo el garantizar una construcción modular de la herramienta, que facilite la sustitución de algunos módulos, y la reutilización de otros en diferentes procesos; establece un diseño basado en una arquitectura tipo tubería (pipeline) en la que es importante establecer el formato de la información que se intercambia entre los procesos. La decisión de diseño en este caso es establecer el intercambio mediante documentos XML. El formato de la información de estos documentos se define mediante una DTD propia (aun no hay un estándar) que permite representar contextos formales y retículos de *Galois.*

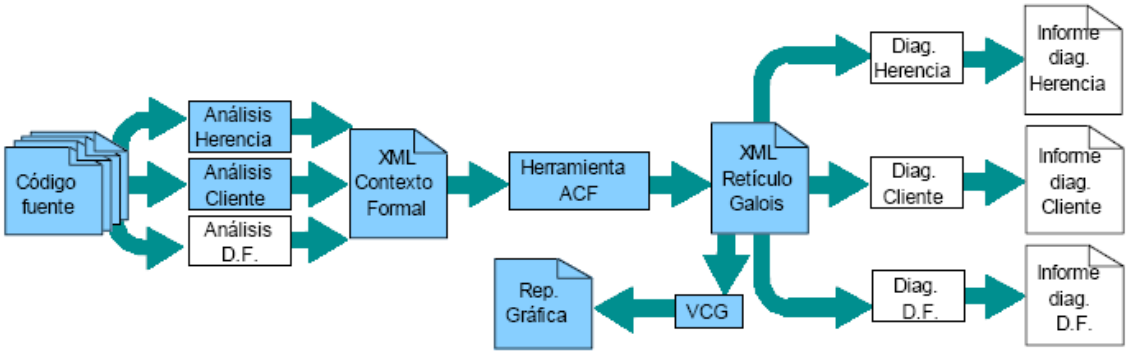

Figura 7-3 *Arquitectura planteada para el proceso de desarrollo de frameworks de dominio.*

Nuestra aplicación no llegará tan lejos quedándose únicamente en el análisis de herencia, de forma que analizando un código fuente se aplicará un análisis de herencia, sacando una matriz de contexto formal tanto visual como en XML (en caso de guardarla) una vez construida dicha matriz mediante la aplicación de técnicas ACF obtendremos el retículo de *Galois* (que también podremos guardar en XML) que representaremos reutilizando el motor gráfico del proyecto [DBRE, 2005] basado en el paquete swing de *Java.*

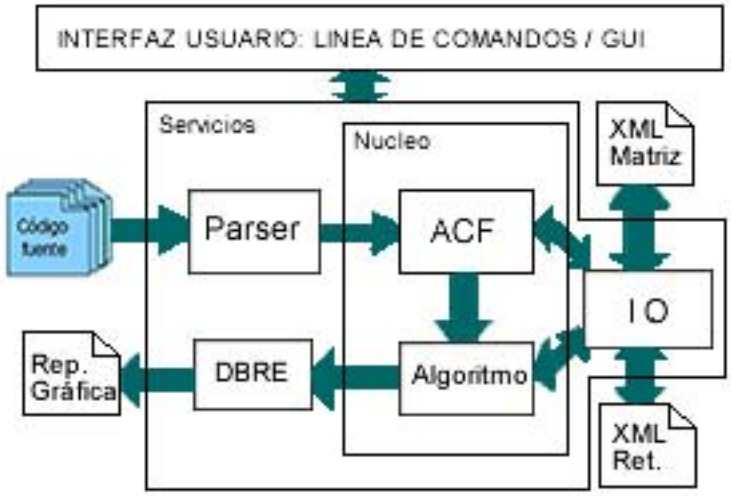

Figura 7-4 *Arquitectura del sistema.*

Nuestra arquitectura se apoya en el diseño en tubería visto anteriormente en referencia al análisis de herencia, pero un poco diferente teniendo como resultado 3 capas:

- Núcleo: Constituido por ACF y Algoritmo que serán los encargados de realizar todas las operaciones relacionadas con el ACF para la obtención de la matriz de contexto y el retículo de *Galois* en otras palabras constituye el soporte relativo a construcción y manipulación de contextos, retículos de conceptos, para que la capa de *servicios* pueda efectuar su trabajo. ACF recibe como entrada los datos obtenidos del análisis léxico-sintáctico en el subsistema Parser, construirá dicha matriz, y procesará el retículo cuando lo estime conveniente el usuario. También se comunicaran en sentido bidireccional con el subsistema IO, para la importación/exportación de la matriz de contexto y retículo de *Galois*, de forma que servirán de comunicación para las futuras herramientas para el desarrollo de *frameworks de dominio* mediante ACF, como se vio al principio.

- Servicios: Constituido por los subsistemas Parser, IO y DBRE, siendo este último una reutilización del motor gráfico del proyecto [DBRE, 2005] para la representación gráfica del retículo de Galois. Esta capa utilizará las funciones que ofrece el *núcleo* para la implementación de los métodos de alto nivel del *Análisis de Conceptos Formales* y del proceso de *Análisis de herencia* que permitirán extraer conocimiento de los contextos formales. El subsistema Parser será el encargado del análisis léxico-sintáctico del código objeto de estudio introducido por el usuario. Con los datos obtenidos, ACF construirá la matriz de contexto y posteriormente el retículo. El subsistema IO es el encargado de la gestión de los importadores y exportadores que servirán como flujo de entrada o de salida de los archivos de contexto.

- Interfaz de usuario: La cual será la capa que permanezca en contacto directo con el usuario y sobre la que éste se comunicara para realizar alguna función sobre el programa. Puede hacerlo de dos maneras distintas, a través del subsistema línea de comandos, donde tendremos un interfaz de modo texto consistente en introducid una orden con una serie de parámetros a la hora de ejecutar el programa. La otra forma de interactuar con el sistema es a través del GUI, o interfaz gráfico de usuario. Un entorno de ventanas, cómodo e intuitivo para el usuario donde podrá realizar cualquier operación que permita la herramienta.

Gracias al diseño específico de la arquitectura, los componentes de la capa de *servicios* adquieren un alto grado de integración. Por un lado, confían en los servicios prestados por la capa inferior, el *núcleo*, lo cual simplifica la comunicación y el intercambio de datos entre los componentes. Y por otro lado, los componentes están dirigidos por una interfaz gráfica de usuario o por una línea de comandos que facilita la utilización conjunta de ambas capas.

Esta arquitectura modular, facilita las modificaciones tanto en el conjunto de métodos algorítmicos implementados en la capa de *servicios* como en las estructuras de datos específicas usadas en el *núcleo*. Es decir, los subsistemas se desarrollarán para que estén perfectamente definidos de forma que se limite el efecto de cualquier decisión de diseño en el resto del sistema.

A continuación mostramos un diagrama indicando la organización en subsistemas y paquetes del -sistema implementado, **Figura 7-5.**

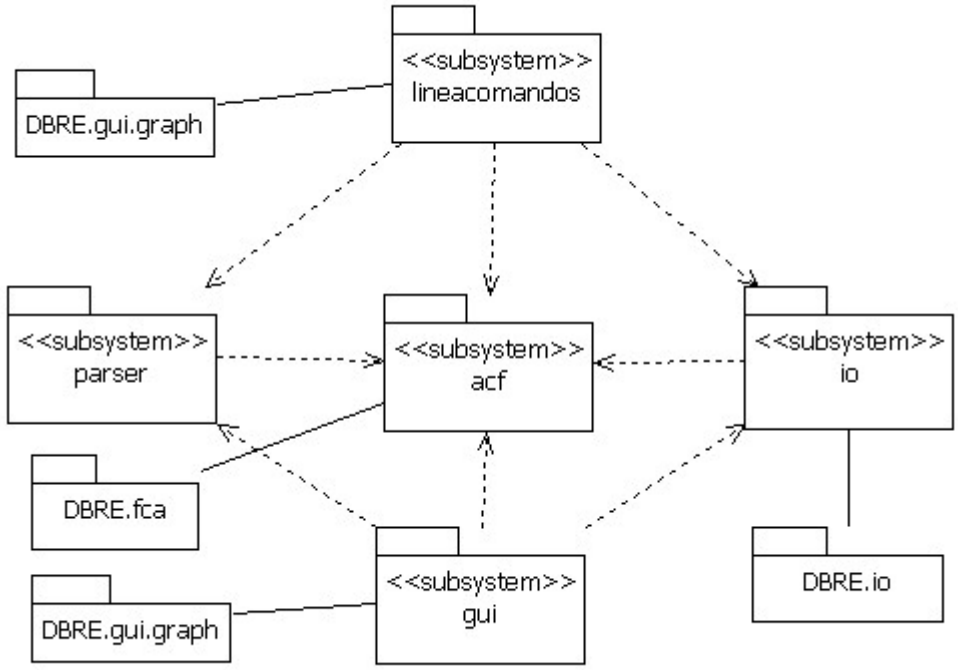

Figura 7-5 *Organización en subsistemas del sistema.*

La mayor parte de los subsistemas corresponden con la trazabilidad directa de los modelos conceptuales obtenidos del análisis efectuado. Se optó por construir los modelos con cierta abstracción en forma de clases abstractas o interfaces, de forma que se fueran respondiendo a las necesidades de implementación de los requisitos no funcionales, facilitando la labor del proceso de diseño. Es ahora, cuando se completará dicha abstracción proporcionando en todos los subsistemas un conjunto de interfaces y clases abstractas que ayuden a la separación entre las especificaciones, operaciones públicas de un tipo de datos y su implementación en una estructura concreta de datos.

Podemos detallar las responsabilidades de cada subsistema:

> **Subsistema *algorithm***. Encargado de la gestión de los diversos algoritmos que se emplearán en el sistema. Implementará el patrón *Strategy*. Esta incluido dentro del Subsistema ACF. Y por tanto no está representado en dicho diagrama.
>
> **Subsistema acf**. Encargado de la representación de todas las nociones matemáticas y su representación en estructuras de datos complejas que manejará la aplicación, concernientes al *Análisis de Conceptos Formales*. Reutiliza el paquete DBRE.fca del proyecto [DBRE, 2005] para asegurar la compatibilidad de estructuras, que era uno de los requisitos establecidos al principio de todo.
>
> **Subsistema *gui***. Encargado de la interacción del usuario con la aplicación representando tanto las ventanas de aplicación de cada subsistema como las clases encargadas de visualizar las estructuras obtenidas de la aplicación de los diversos algoritmos.

Subsistema *linea de comandos*. Encargado de la interacción del usuario con la aplicación a través de la consola del sistema. Representará una funcionalidad parcial permitiendo únicamente una parte de las que ofrece el subsistema gui.

Subsistema *io*. Responsable de la persistencia de la aplicación con respecto a las principales estructuras de datos. Ya sea para la entrada o para la salida de archivos de datos XML. Utiliza el paquete DBRE.io del proyecto [DBRE, 2005] ya que uno de los requisitos es guardar la compatibilidad con dicho proyecto en estructuras y formato de archivos.

Subsistema *parser*. Encargado del análisis de las clases o interfaces introducidas en el sistema por el usuario, obteniendo datos relevantes para la construcción de la matriz de contexto. Esta diseñado con la consiguiente estructura de clases abstractas e interfaces de forma que facilite la incorporación de nuevos parser de una forma rápida y sencilla.

El alto nivel de abstracción genérica proviene de uso riguroso de los tipos de datos abstractos de las diversas estructuras relacionadas con el *Análisis de Conceptos Formales*: conjunto extensión e intensión, conceptos, retículos, etc. Por cada uno de ellos se ha especificado una colección de primitivas y asignado a una clase abstracta o interfaz (siguiendo el lenguaje de desarrollo *Java*. Esto reduce la dependencia sobre la implementación de las estructuras de datos. Además, de esta forma se pueden usar implementaciones alternativas para favorecer una característica particular del resultado del servicio (e.g. eficiencia, consumo bajo de memoria, etc.).

Una aproximación similar se ha utilizado para el diseño de las estructuras de extracción de conocimiento. En lugar de representar algoritmos alternativos como métodos software de una única clase, se ha decidido emplear patrones de diseño (Patrón *Strategy*) que favorezcan la aplicación de múltiples tipos de algoritmos o de varias implementaciones de un mismo tipo de algoritmos sobre una estructura. De esta forma cada algoritmo está representado por una clase heredera de una clase base que represente a un tipo de algoritmo. Este diseño permite la integración de nuevos procedimientos algorítmicos de forma relativamente sencilla. Cualquier algoritmo heredará la información relativa de su categoría específica mientras se beneficia de un amplio rango de servicios ofrecidos por las capas inferiores de la arquitectura. Lo mismo ocurre con los distintos exportadores tanto para contextos como para retículos, así como para los *parsers* e importadores.

7.3.1 Topología del sistema

No será necesario definir un conjunto de diagramas de despliegue o de componentes para describir la asignación del software al hardware. La configuración del hardware del sistema es simple, un solo componente, la aplicación, que se ejecuta en una única máquina llamando sólo a las bibliotecas oportunas del lenguaje de desarrollo con el que está implementado el sistema. No existe comunicación con otros elementos, bibliotecas o componentes externos a excepción de los subsistemas mencionados anteriormente de DBRE que están integrados directamente en la aplicación. El ejecutable estará formado por un único archivo (con extensión jar) con todas las clases compiladas del sistema y todos los ficheros gráficos de la aplicación.

7.3.2 Gestión de la persistencia

La aplicación debe ser capaz de interactuar con dos tipos de ficheros de entrada/salida que contienen dos tipos de estructuras de datos diferentes. En concreto debe ser capaz de: 1) leer/escribir un contexto formal creado a través de la aplicación mediante el análisis de una serie de clases o interfaces; al guardar la compatibilidad con el proyecto [DBRE, 2005] en dicho fichero se pueden guardar un grupo o conjunto de ficheros, de forma que al cargarlos, el usuario deberá elegir

que contexto formal abrirá, ya que nuestra aplicación solamente trabaja con un contexto formal como ya se describió anteriormente. 2) leer/escribir un retículo calculado a partir de un contexto formal. Como pasaba con los contextos, debido a la misma razón que antes, ocurrirá que el usuario deberá elegir el retículo que desea cargar del conjunto de posibles retículos que se guardan en dicho archivo. El formato para 1) y 2) será un formato XML con una DTD que se detalla en [DBRE, 2005], también se incorporó la posibilidad de cargar archivos XML con la DTD del proyecto Galicia [Galicia, ref] para contextos formales. Se remite al lector a la bibliografía para comprender dicha DTD. Esta última DTD se incorporó en la fase pruebas que se efectuaron con el programa con la intención de comprobar los resultados que se obtenían en la herramienta con los del proyecto Galicia.

Para trabajar sobre estos ficheros y poder crearles se desarrolló dos DTDs en [DBRE, 2005] con los cuales se pueda obtener toda la información que se necesita de dichos contextos y/o retículos. Esto tiene una doble utilidad, por una parte permite controlar cuando un documento generado por la herramienta o por otra aplicación cumple con los requisitos necesarios para poder ser empleado por la aplicación presentada, y por otra permite ver como se guardan las informaciones obtenidas.

7.3.3 Aspectos de rendimiento y tamaño

Cuando se trabaja con un volumen de datos relativamente grande, el rendimiento de los algoritmos planteados puede deteriorarse tanto en tiempo como en espacio de memoria consumida. Se ha optado por potenciar el tiempo de respuesta en detrimento del espacio en memoria utilizado. Por tanto, los algoritmos serán implementados teniendo en cuenta esta circunstancia, de tal forma que se hará especial hincapié en la utilización de las estructuras de datos básicas adecuadas para cada momento eligiéndolas según su ventaja con respecto a los tiempos de acceso (e.g. tablas de dispersión versus vectores).

7.4 Diseño de los Subsistemas

En esta sección se detallarán los aspectos de diseño de cada subsistema. Se establecen los límites entre los subsistemas y se especifican sus interfaces. Para cada paquete hablaremos de su funcionalidad, sus relaciones y sus responsabilidades, pudiendo verlas en mayor profundidad en los apartados posteriores donde se describe los diagramas de clases. Con respecto al resto de los diagramas de secuencia del sistema, los definiremos más adelante viendo el comportamiento del sistema, así como la pertenencia de las clases a su paquete correspondiente y la interacción entre las clases. El diseño detallado de cada clase no se reflejará en el presente documento debido a la extensión que podría alcanzar dicho diagrama. Sin embargo se presentaran un diagrama de clases para cada subsistema así como para cada diagrama de secuencia en el apartado siguiente. Se creará una documentación en el CD-ROM anexo para facilitar su consulta en el momento que el lector estime conveniente.

7.4.1 Subsistema *acf*

Este subsistema es el encargado de representar todos los conceptos matemáticos dentro del marco de *Análisis de Conceptos Formales* asociados a la representación de contextos formales y retículos de conceptos. En dicho subsistema está contenido el subsistema *algorithms* encargado de la aplicación del algoritmo de *Bordat* para la construcción del retículo. Se detallará más adelante.

En esta subsistema también se encuentra la clase `Matriz`, encargada de representar el conjunto de clases (o interfaces) y características como un contexto.

7.4.2 Subsistema *algorithms*

En este subsistema se recogen todos los algoritmos de procesamiento de retículos y de contextos, así como las clases que los empaquetan para poder acceder a ellos de forma dinámica. En nuestro caso el único algoritmo que se utiliza es el algoritmo de *Bordat*, obtenido del proyecto [DBRE, 2005]. Para conseguir un retículo a partir de un contexto formal.

Para acceder a los algoritmos se utiliza la clase `SetAlgorithms`, dicha clase proporciona un método `descripciones()`, obteniendo un `String[]` compuesto por cada una de las descripciones de los distintos algoritmos, y con el índice de dicha descripción seleccionar el algoritmo mediante el método: `getAlgorithm(int indice)`, y poder aplicarlo.

Excepciones:

- `IndiceAlgorithmNoValidoException`: Cuando se intenta obtener un algoritmo cuyo índice no es válido.

7.4.3 Subsistema *io*

Este subsistema engloba todas las operaciones de entrada/salida, en nuestro caso estas operaciones se refieren a la importación/exportación de retículos y contextos, las operaciones de proporcionar archivos y directorios que puede necesitar el *parser*, así como los mensajes de salida para el usuario en modo consola (línea de comandos).

Exportar retículo/contexto.

Mediante las clases `SetExportReticulo` y `SetExportContexto` respectivamente, accedemos a los distintos formatos de exportación.

Obteniendo a través de los `<<interface>> AbstractExportReticulo` y `AbstracExportContexto`, un exportador al formato seleccionado. Mediante el `método setWriter(Writer destino)`, se selecciona donde se guardara el retículo/contexto, y con los métodos :

```
writeBinaryRelation(BinaryRelation contexto)
writeConceptLattice(ConceptLattice reticulo)
```

Escribimos el contexto/retículo en el `Writer`, seleccionado

Para seleccionar un formato las clases `SetExport…` proporcionan el método `descripciones()`, donde obtenemos un `String[]`, con las descripciones de dichos exportadores. Con índice de dicha descripción en el array obtenemos el exportador mediante el método `getExport(int indice)`, con dicho exportador

Excepciones:

Al realizar las operaciones de exportación se pueden dar las siguientes excepciones:

- `IndiceExportImportNoValidoException`: Si se intenta acceder a un exportador cuyo índice no existe.
- `NoWriterSelectedExcepton`: Si se intenta exportar y no se ha seleccionado un `Writer` de destino

- También se pueden dar Excepciones estándar de error de Entrada/Salida, y de formato no valido.

Importar retículo/contexto.

Para importar un retículo o un contexto la mecánica es la misma que la descrita anteriormente para la exportación.

Las clases que engloban los exportadores son `SetImportReticulo` y `SetImportContexto` respectivamente.

La manera de acceder a los distintos formatos se realiza mediante los `<<interface>> AbstractImportReticuloReader` y `AbstractImportContextoReader`, respectivamente.

Los métodos para obtener el retículo o el contexto son:

```
getSetRelation() : SetBinaryRelation
getSetConceptLattice() : SetConceptLattice
```

Las clases `SetBinaryRelation` y `SetConceptLattice`, definen un conjunto de contextos y de retículos respectivamente (ver documentación [DBRE, 2005]).

La manera de seleccionar un formato de importación es la misma que la detallada para los exportadores.

Excepciones:

Las excepciones son las mismas que los exportadores excepto `NoWriterSelectedExcepton` que es sustituida por `NoReaderlectedExcepton`, cuyo significado es el mismo.

Mensajes de salida al usuario en modo consola.

Los mensajes de salida para el usuario, se agrupan en distintas categorías o niveles. Para gestionarlos se utiliza la clase `OutMensajesUsuario`. Esta clase agrupa los mensajes según el nivel dado para cada mensaje, y los muestra o no dependiendo del nivel de salida que se desee mostrar. Por defecto estos mensajes son enviados a la salida estándar y a la salida estándar de error. Aunque se pueden seleccionar otros `java.io.PrintStream`, donde mandar los mensajes.

Excepciones:

La única excepción, a parte de las estándar de error de entrada/salida, es `NivelSalidaNoValidoException`, que se genera cuando para un mensaje se da un nivel no válido, o se quieren mostrar los mensajes de un nivel no válido.

Archivos solicitados por el parser.

El parser a medida que vaya estudiando el código puede encontrar referencias a otros archivos o directorios (e. g. sentencia `import` en java), en ese momento el parser solicita esos archivos, la disponibilidad de esos archivos y de los que se están usando se representa mediante la clase `ArchivosFuente`, dicha clase permite tanto introducir archivos, como directorios, en cuyo caso se buscarán todos los archivos cuya extensión concuerde con la requerida por el parser en el

directorio dado. Si un archivo o directorio no existe, estos se marcarán como no disponibles.

Excepciones

Las excepciones lanzadas por esta clase se refieren a los datos dados a la hora de comenzar el parser. Estos son: el archivo o archivos donde comenzar dicho parser, y el directorio o los directorios a partir de los cuales buscar los archivos referenciados en el código (classpath).

- `NoExisteClassPathException`: Si el directorio dado como classpath no existe o no es accesible.
- `FuenteInicialNoEncontrada`: Si alguno de los archivos donde comenzar el parser no existe o no es accesible.

7.4.4 Subsistema *lineacomandos*

Este subsistema se compone de las operaciones y tareas que la aplicación lanzará ejecutándose en modo consola. La clase principal se denomina `Aplicacion`, es la encargada de capturar y procesar los parámetros de la línea de comandos a través de la clase `ParserLineaComandos` guardando las distintas opciones en un objeto de tipo `OpcionesLineaComandos`.

A partir de esta información la aplicación realiza las distintas tareas. Estas pueden ser:

- Realizar el parser sobre el código.
- Exportar el contexto.
- Generar el retículo.
- Exportar el retículo.
- Mostrar el gráfico del retículo en una ventana.

El método que utiliza `ParserLineaComandos` para comunicar a `Aplicacion`, que hay errores en los distintos modificadores y parámetros de la línea de comandos, es a través de excepciones, estas se describen a continuación

Excepciones:

- `LinComandosParametroNoValidoException`: Si se da un parámetro o modificador de la línea de comandos que no se reconoce.
- `LinComandosNoParserException`: Si no se ha seleccionado ningún parser, o el seleccionado no es un parser válido.
- `LinComandosNoFuentesException`: Si no se ha dado ningún nombre de archivo (o no es valido) donde comenzar el parser.
- `LinComandosNoclassPathException`: Si no se ha dado ningún directorio donde se encuentran los archivos de código.
- `LinComandosNoNivelSalidaException`: Si se ha dado el modificador de seleccionar el nivel de los mensajes de salida que se mostraran, y el nivel dado no es válido.
- `LincomandosNoSalidaException`: Cuando no se da ninguna opción que conlleve alguna tarea para realizar por la aplicación.
- `LinComandosNoArchivoSalidaContextoException`: Si se ha seleccionado exportar el contexto y no se ha dado ningún nombre de archivo válido.

- `LinComandosNoAlgoritmoException`: Si no se ha seleccionado ningún algoritmo de generación de retículo, y es necesario generar el retículo.
- `LinComandosNoFormatoSalidaContextoException`: Si el formato seleccionado para el exportador de contexto no es válido.
- `LinComandosNoArchivoSalidaReticuloException`: Si se ha seleccionado exportar el retículo y no se ha dado ningún nombre de archivo válido.
- `LinComandosNoFormatoSalidaReticuloException`: Si el formato seleccionado para el exportador de retículo no es válido.

7.4.5 Subsistema *parser*

En este subsistema se engloban los distintos *parsers*, que operan sobre el código fuente objeto de estudio.

Para acceder a dichos *parsers*, a su control y a los resultados se utiliza la clase `ControladorParser`. Esta clase interactúa con `GestorParser`, (y con `Matriz` englobada en el subsistema *acf*) que es la encargada de mantener un listado de todos los ficheros que se han examinado, y de localizar y mandar examinar nuevos ficheros si se encuentra su referencia dentro del código.

El acceso a los distintos *parsers* se realiza a través de la clase `SetParsers`, que en su definición engloba todos los *parsers* que están declarados en la aplicación. Esta clase también permite añadir un *parser* nuevo de forma dinámica a través del método `addParser( AbstractParser parser)` para facilitar así su reutilización. Este método también existe en `ControladorParser`, y cuya finalidad es la misma.

Los *parsers* definidos por la aplicación son los siguientes:

- `JavaParser_1_5`: *Parser* que reconoce las clases con sus características para código *java 1.5*.
- `JavaParser_1_4`: *Parser* **incompleto** para código *java 1.4* incluido como ejemplo de cómo añadir nuevos *parsers*.
- `JavaInterfacesParser_1_5`: *Parser* que reconoce los interfaces con sus características para código *java 1.5*.

Estos *parsers* están definidos en la clase `SetParsers`, la aplicación a través de la clase `ControladorParser` accede a ellos dinámicamente a través del método `descripciones()`, que obtiene un `String[]` con las descripciones de los parsers. Mediante este array, obtiene el índice del parser según su descripción, y este es seleccionado a través de `setParser(int indice)` en `ControladorParser`.

Una vez que el *parser* termine, y si no ha habido ningún error se puede acceder al contexto.

Los pasos para realizar el *parser* son dos: primero una inicialización del *parser* en la que se comprueba que el *parser* puede comenzar, y después una ejecución en la que se realiza el *parser* para cada fichero seleccionado o que se encuentre referenciado y dentro del *classpath*.

Excepciones:

- `ParserNoInicializadoException`: Si se intenta ejecutar el parser sin haber sido inicializado previamente.
- `NoArchivosFuenteException`: Si no se puede acceder (o este no existe) a alguno de los archivos donde comenzar el parser.
- `NoClassPathException`: Si no se ha definido un directorio (o directorios) a partir del cual buscar los archivos referenciados, o los archivos iniciales del código sobre el que realizar el parser.
- `ParseException`: Cuando se encuentra algún error en el código analizado.

7.4.6 Subsistema *gui*

Este subsistema es el responsable de representar el interfaz gráfico de usuario (GUI), se encargará de hacer de puente entre la aplicación y el usuario, permitiendo a este último interactuar con la herramienta.

Se ha construido un sistema de interfaz gráfico de usuario siguiendo el patrón vista/controlador. Separando totalmente la capa de interfaz de la de dominio logrando así una mayor independencia de la misma. Veamos la estructura de un patrón vista/controlador comparándolo con nuestro diseño.

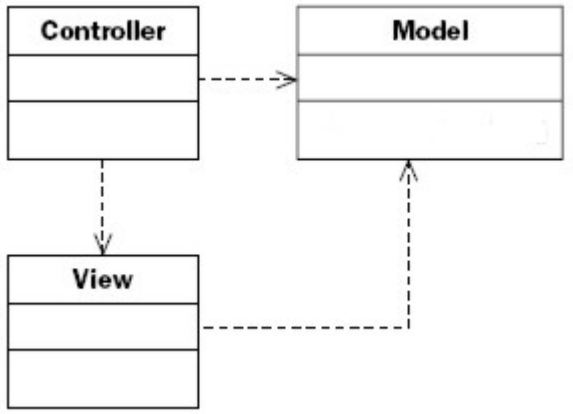

Figura 7-6 *Modelo vista/controlador.*

En la **figura 7-6**, se puede observar un típico modelo vista controlador donde se ven claramente las dependencias entre la vista y el modelo así como entre el controlador y la vista, y el controlador y el modelo, de forma que un cambio en el modelo afectará a la vista y al controlador, es decir que si estamos trabajando con un modelo de figuras geométricas por ejemplo, un cambio en las dimensiones de una figura, provocara un cambio en la vista que representa gráficamente a esa figura, de forma que ésta deberá cambiar adaptándose a las nuevas dimensiones de la figura. Por otro lado un cambio en el modelo puede propiciar un cambio en el controlador que por defecto manejara los eventos, es decir que un cambio en el modelo puede provocar que se dispare un evento en el controlador. Así mismo, un cambio en la vista, por ejemplo, el usuario ha modificado alguna característica de la representación de una figura, provocará que se dispare un evento en el controlador.

Ya centrándonos en nuestra herramienta, el interfaz está diseñado de la siguiente manera, asemejándose a dicho patrón. (Ver **figura 7-7**).En dicha figura podemos observar que la ventana principal que haría típicamente de vista la hemos separado en dos, los menús y capturadores de eventos harán de controlador (`FramePrincipalOpt`) mientras que el panel de contenido de

dicha ventana realizará las funciones de vista a través de la clase `VistaContexto`, mostrándose ahí la matriz de incidencia y abriendo una ventana nueva para los retículos. Ambas tienen relaciones de dependencia con `ModeloContexto` que no es más que una lista enlazadaza de `Contextos`, que será la clase que modele tanto la matriz de incidencia como el retículo de *Galois*.

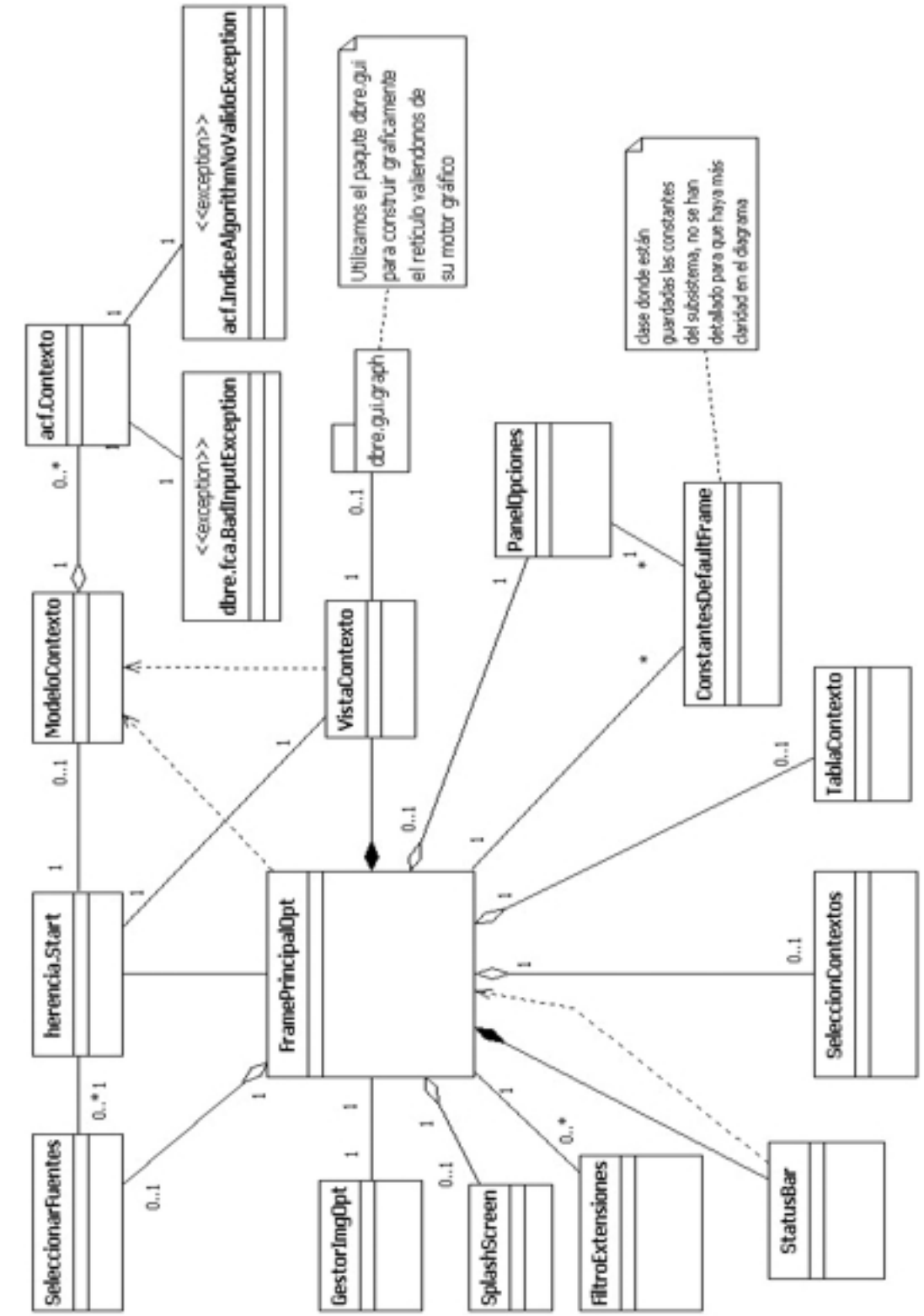

Figura 7-7 *Diagrama de clases simplificado del subsistema GUI.*

Implementamos entre `ModeloContexto` y `FramePrincipalOpt` y `VistaContexto` un patrón Observador de las librerías *Java* (`Observable` y `Observer` del paquete `java.util`) de forma que en el momento en que cambie el `ModeloContexto`, ya sea porque se ha introducido un modelo nuevo o porque el ya existente ha sido modificado, actualice tanto `VistaContexto`, mostrando cualquier cambio en el modelo, como `FramePrincipalOpt`. Por otro lado resaltaremos que Contexto lanzará dos excepciones, `BadInputData` procedente del subsistema de `dbre.fca` del proyecto [DBRE, 2005], e `IndiceAlgoritmoNoValido` del paquete `acf`. La clase `herencia.Start` aparte de ser la clase que lance todo el sistema, es la aquella que nos proporcionará acceso a las distintas partes del patrón (`ModeloContexto`, `FramePrincipalOpt`, `VistaContexto`) permitiéndonos insertar un modelo nuevo a través de él, así como recuperar a través de dicha clase el modelo con el contexto con el que estemos trabajando. Si nos fijamos en el resto de clases, son bastante explicativos sus nombres con respecto a su función, teniendo `SeleccionFuentes` para introducir los datos con que se iniciará el *parser*, consistiendo esta clase en un menú gráfico donde seleccionaremos el *parser* de entre los disponibles (se cargarán dinámicamente), añadiremos el *classpath* donde buscar las clases a las que hace referencia en el código a analizar, y las clases o interfaces que vamos a analizar.

`GestorImgOpt` será la clase encargada del manejo de las imágenes de la aplicación; cuando nos referimos a imágenes, nos referimos a los iconos cargados tanto en la ventana como en los menús, así como la imagen de Carga. Su funcionamiento básicamente se encarga de devolver la ruta de los recursos. De forma que si queremos cambiar alguna imagen tan solo tendremos que sustituir un recurso por otro que tenga el mismo nombre, o si queremos introducir alguna imagen nueva únicamente tendremos que especificar en dicha clase la nueva ruta. `SplashScreen` cargara una imagen sobre la autoría del proyecto, así como su título y demás información relevante cuando seleccionemos en el menú de Ayuda “Acerca de”. Esa misma imagen servirá de fondo mineras se carga la aplicación, pero no a través de esa clase sino en la misma clase lanzadora, `Start`. `FiltroExtensiones` será un filtro para los archivos de datos que deseemos cargar o guardar. `StatusBar` forma parte de la ventana principal y reflejara el estado en que se encuentre la aplicación, teniendo el valor “Listo” cuando se haya cargado todo y se esté a la espera de que el usuario realice alguna interacción con la herramienta. `SeleccionContextos` es una clase que se utiliza para cuando se acaba de cargar un archivo externo que pudiera contener más de un contexto o retículo; en ese caso se genera una ventana emergente (`SeleccionContextos`) donde el usuario selecciona el contexto que desee cargar. En caso de no seleccionar ninguno y pinchar con el ratón en aceptar se cargará por defecto el primer contexto que contenga el archivo. `TablaContexto` se encarga de la construcción de la tabla para la visualizar la matriz de contexto. `ConstantesDefaultFrame` contiene las constantes que se utilizaran en `FramePrincipalOpt` y en `PanelOpciones` siendo esta última la encargada de visualizar una ventana emergente donde el usuario podrá seleccionar un formato para la guardar la matriz de incidencias o el retículo de *Galois* en un archivo de datos XML Cargará los exportadores disponibles, en una lista donde el usuario podrá escoger el más adecuado. Si no se escoge ninguno, por defecto se cargará en compatibilidad con [DBRE, 2005].

Recordamos al lector que el diagrama de clases mostrado corresponde a la versión simplificada ya que por motivos de tamaño no se ha podido incluir el diagrama completo. Si desea obtener un mayor detalle del mismo, le remitimos al CD-ROM donde están contenidos todos los diagramas a un nivel de detalle completo.

7.5 Diagramas de Clases (Modelo Estático)

A continuación hablaremos de las distintas clases que conforman la aplicación, mostrando los diagramas de clases, y digo diagramas ya que para que quede una idea más clara y concisa se ha dividido el diagrama general en partes, de forma que podamos ver cada parte enmarcada dentro de las funciones que se pueden dar dentro de la aplicación. Hablaremos de todas menos las correspondientes al interfaz gráfico, ya que éstas se detallaron claramente tanto en diagrama y funcionalidad en el apartado anterior. Comentaremos por último al lector que en los diagramas, la multiplicidad es la que aparece y si no aparece es porque es una relación de 1 a 1.

7.5.1 Línea de comandos

La estructura principal de la aplicación en el contexto de que se ha ejecutado en modo consola, es decir desde la línea de comandos es la que se puede contemplar en la **figura 7-8**.

Las principales características, de la aplicación ya se mencionaron anteriormente en la descripción de los subsistemas, en este punto se muestran las clases y sus relaciones, pasamos a describir brevemente las principales clases.

- `Star`: Es la clase lanzadora de la aplicación.
- `Aplicacion`: Es la clase principal para el consola. Captura los parámetros y dependiendo de sus valores realiza las tareas pertinentes.
- `ParserLineaComandos`: Interpreta los parámetros de la línea de comandos, detectando los errores.
- `OpcionesLineaComandos`: Guarda las distintas opciones y valores de los parámetros que se han dado desde la línea de comandos.
- `OutMensajesUsuario`: Clase intermedia que procesa los mensajes informativos y de error para usuario, y dependiendo de un nivel los muestra o no. Los principales niveles de los mensajes son: ERROR, ESENCIALES, INFORMATIVOS, DEPURACION.
- `ControladorParser`: Esta clase se encarga de toda la gestión con los *parsers* de código, y de generación del retículo.
- `SetParsers`: Engloba todos los *parsers* de código que al aplicación utiliza.
- `AbstractParser`: Clase abstracta mediante la cual los distintos *parsers* son utilizados. Si se quisiera añadir un nuevo *parser* a la aplicación (dinámicamente o recompilando), este *parser* tendría que implementar esta clase.
- `JavaParser_1_4`: *Parser* para código *Java 1.4* **incompleto.**
- `JavaParser_1_5`: *Parser* para código *Java 1.5*, que solo examina las clases y sus características.
- `JavaInterfacesParser_1_5`: *Parser* para código *Java 1.5*, que solo examina los interfaces y sus características.
- `SetAlgorithms`: Clase que engloba todos los algoritmos, en este caso los algoritmos de generaciónd de retículo de *Galois*, a partir de un contexto formal.
- `LatticeAlgorithm`: Clase abstracta que define un algoritmo de generación de retículos de *Galois* (esta clase pertenece al proyecto [DBRE, 2005]).
- `BordatLatticeAlgorithm`: Implementación del algoritmo de *Bordat* (esta clase pertenece al proyecto [DBRE, 2005]).
- `SetExportContexto`: Clase que engloba todos los exportadores a los distintos formatos de un contexto formal.

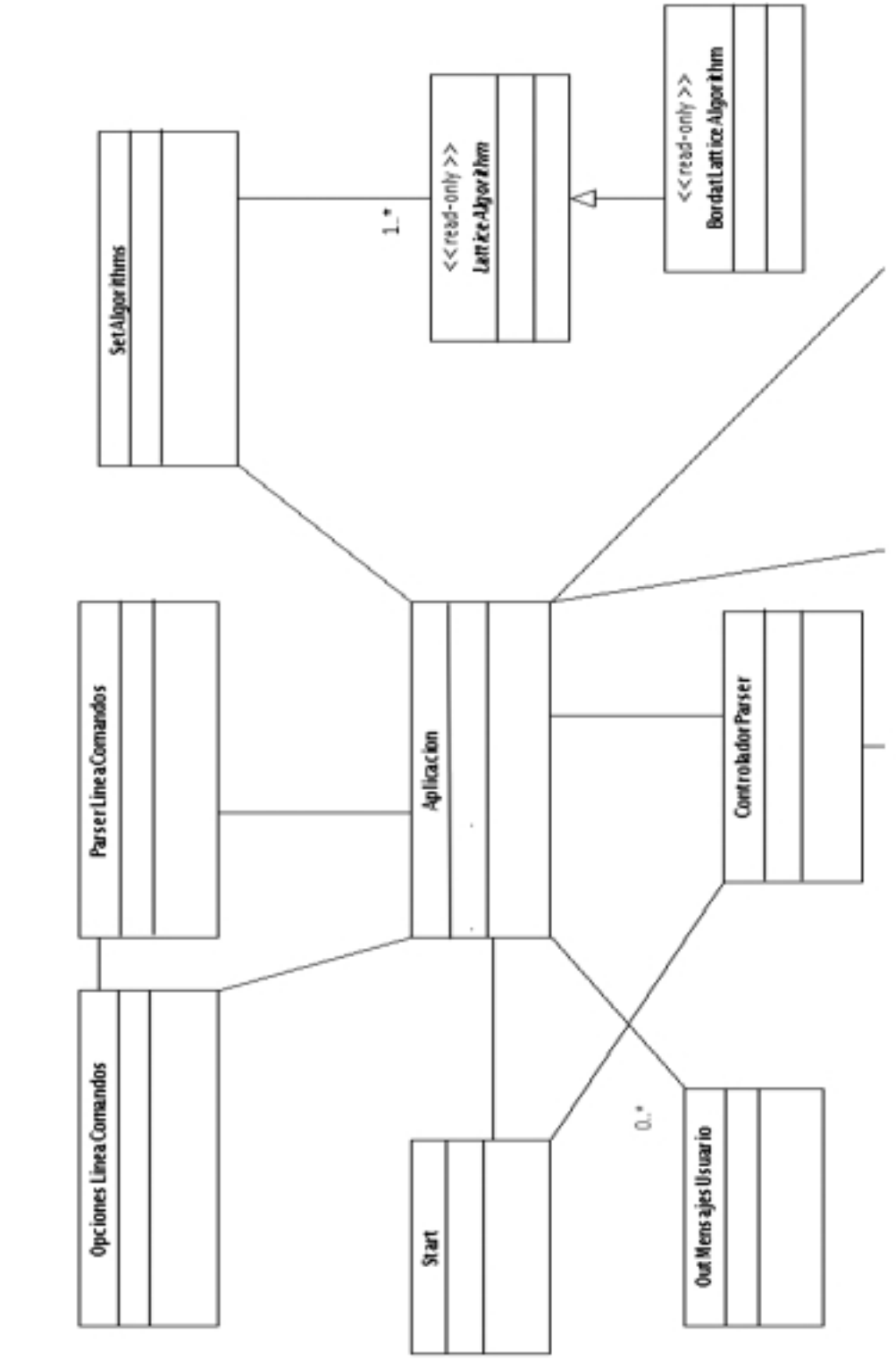

Figura 7-9 *Diagrama de clases en el marco de línea de comandos (parte 1)*

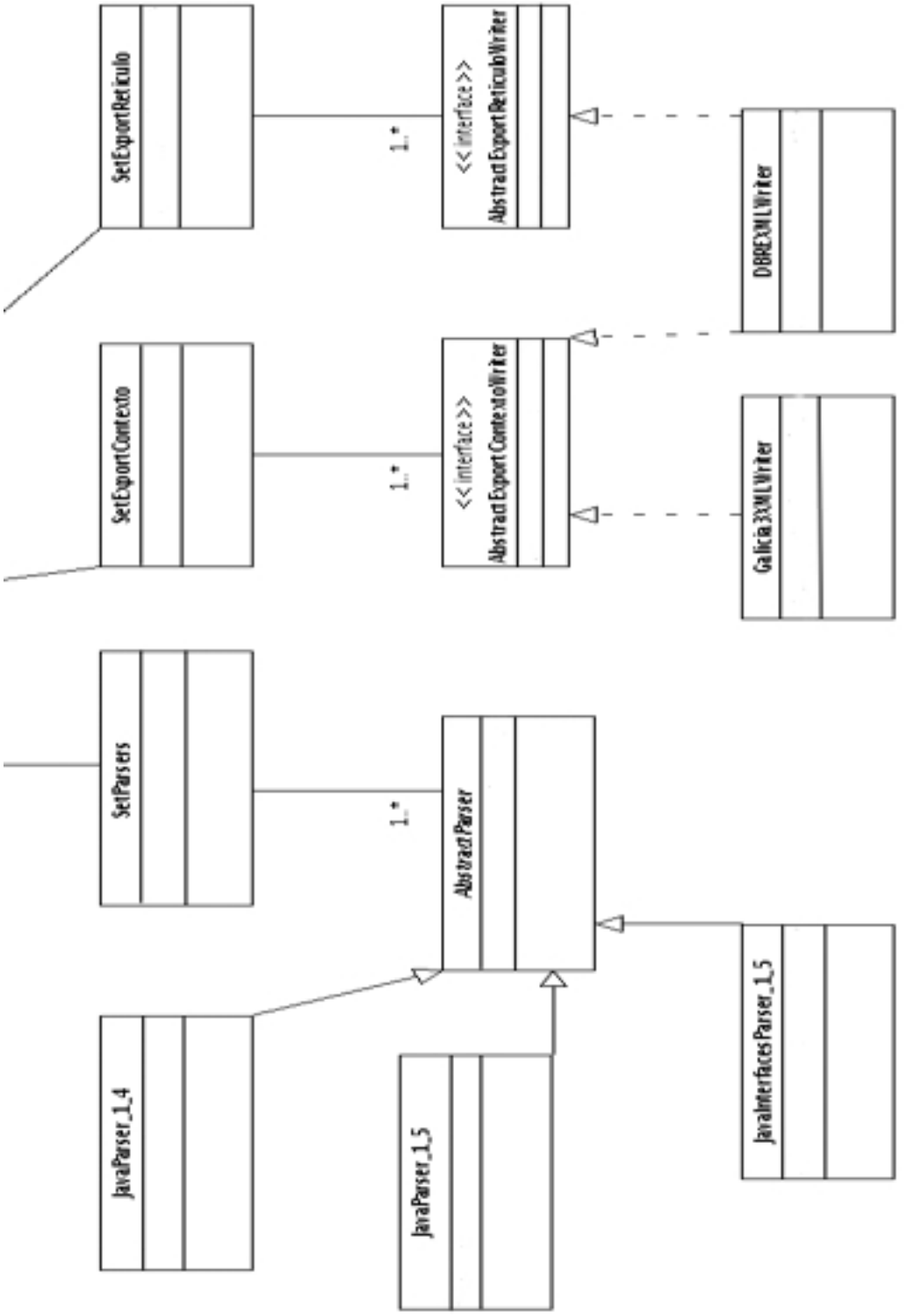

Figura 7-10 *Diagrama de clases en el marco de línea de comandos (parte 2)*

- `AbstractExportContextoWriter`: Interface mediante el cual la aplicación maneja un exportador de contexto seleccionado. Si se quisiera añadir un nuevo exportador de contexto a la aplicación (dinámicamente o recompilando), este tendría que implementar este interface.
- `Galicia3XMLWriter`: Exportador para el formato del programa "Galicia 3".
- `SetExportReticulo`: Clase que engloba todos los exportadores a los distintos formatos de un reticulo de *Galois.*
- `AbstractExportReticulo`: Interface mediante el cual la aplicación maneja un exportador de reticulo seleccionado. Si se quisiera añadir un nuevo exportador de retículo a la aplicación (dinámicamente o recompilando), este tendría que implementar este interface.
- `DBREXMLWriter`: Exportador para el formato del proyecto [DBRE, 2005].

Por motivos de claridad algunas clases no han sido representadas en el diagrama anterior, por eso se presentan los siguientes subdiagramas, divididos en distintos escenarios.

7.5.2 Generar Contexto

En este subdiagrama se representa las clases con las que se relaciona ControladorParser para poder generar un contexto. Esto se muestra en la **figura 7-11**.

Una breve descripción de las clases nuevas:

- `GestorParser`: Clase que actúa de forma intermedia entre el *parser* seleccionado, los archivos de código a examinar, y la matriz del contexto.

- `ArchivosFuente`: Clase que engloba todos los archivos que se examinan del código fuente, y busca los referenciados por el *parser*. Si lo referenciado es un subdirectorio buscará todos los archivos que coincidan con la extensión para el *parser* en ejecución, y los añadirá a la lista para que GestorParser se los proporcione al *parser*.

- `Matriz`: Clase que representa el contexto formado por las clases y las características

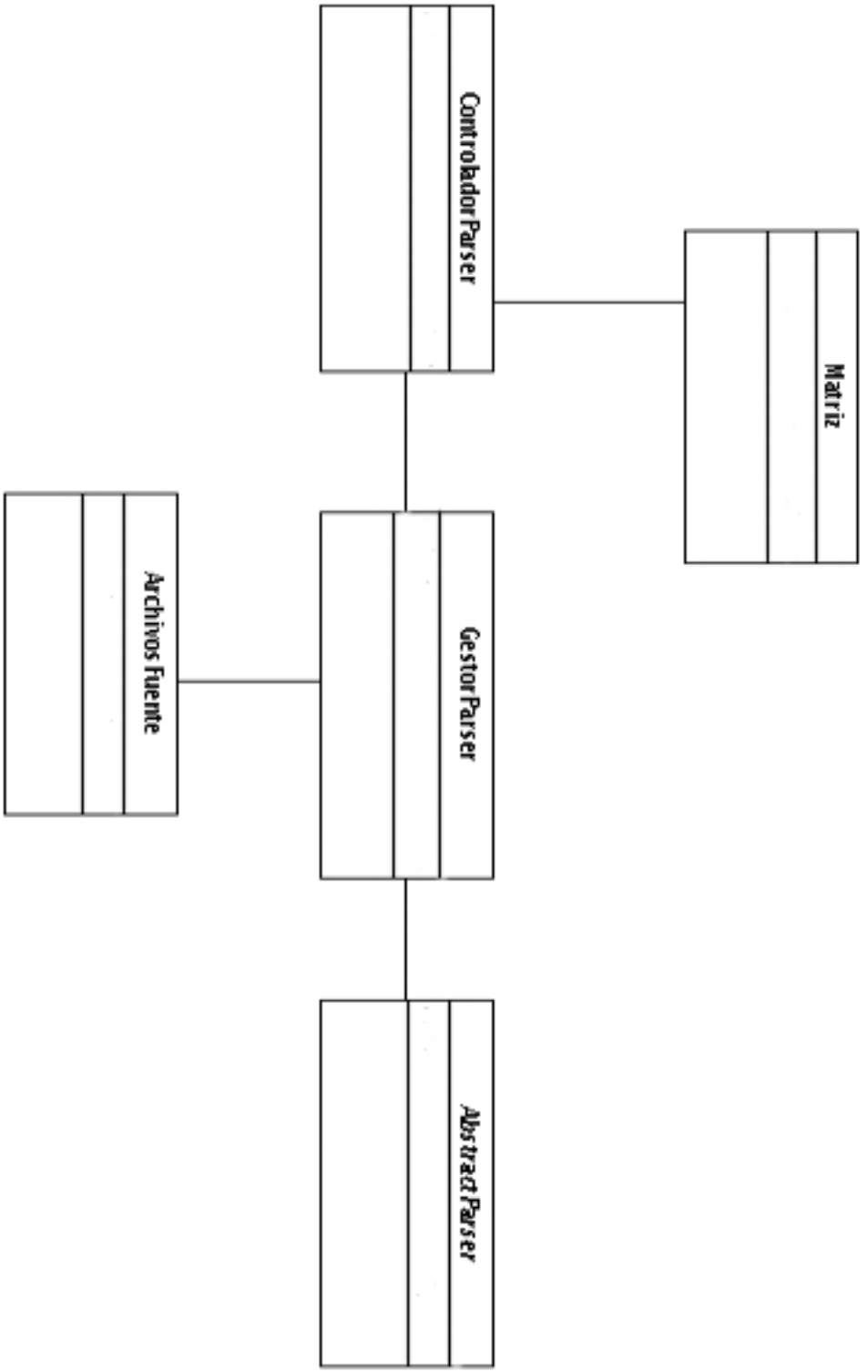

Figura 7-11 *Diagrama de clases en el escenario: Generar Contexto.*

7.5.3 Guardar un Contexto Formal

En este subdiagrama se representa las clases con las que se relaciona `Aplicacion` para poder guardar un contexto formal. Lo podemos observar en la **figura 7-12**.

7.5.4 Generar un Retículo de *Galois*.

En este subdiagrama se representa las clases con las que se relaciona `Aplicacion` para poder generar un retículo a partid de un contexto formal. Lo podemos observar en la **figura 7-13**.

Una breve descripción de las clases nuevas:

- `ConceptLattice`: Implementación del proyecto [DBRE, 2005], de un retículo de *Galois*.

7.5.5 Exportar Retículo

En este subdiagrama se representa las clases con las que se relaciona `Aplicacion` para poder exportar un retículo generado a partir de un contexto formal. Lo podemos observar en la **figura 7-14**.

7.5.6 Visualizar el Retículo de *Galois*

En este subdiagrama se representa las clases con las que se relaciona `Aplicacion` para poder visualizar un retículo ya generado. . Lo podemos observar en la **figura 7-15**.

Una breve descripción de las clases nuevas:

- *LatticeGraphFrame*: Clase del proyecto [DBRE, 2005] que muestra el grafo de un retículo de *Galois*, en una ventana.

- *AdaptadorLattice*: Clase asociada a *LatticeGraphFrame* que maneja el evento de cerrar ventana y termina la aplicación.

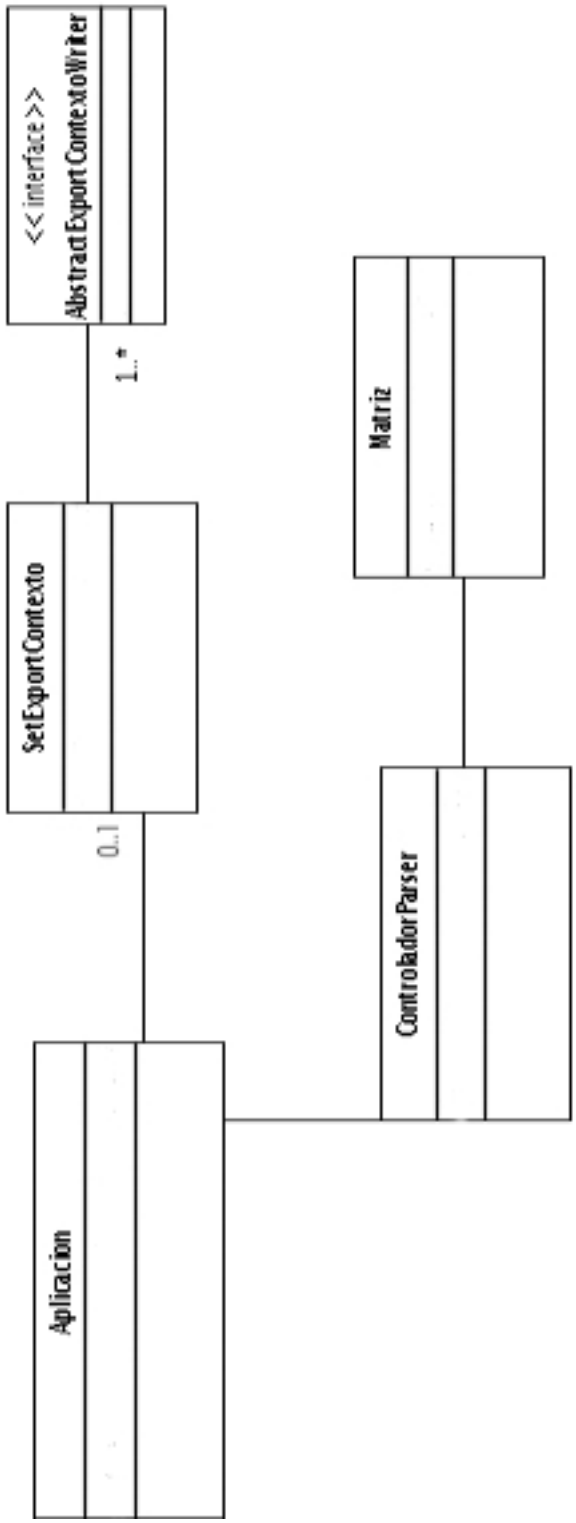

Figura 7-12 *Diagrama de clases en el escenario: Guardar contexto formal*

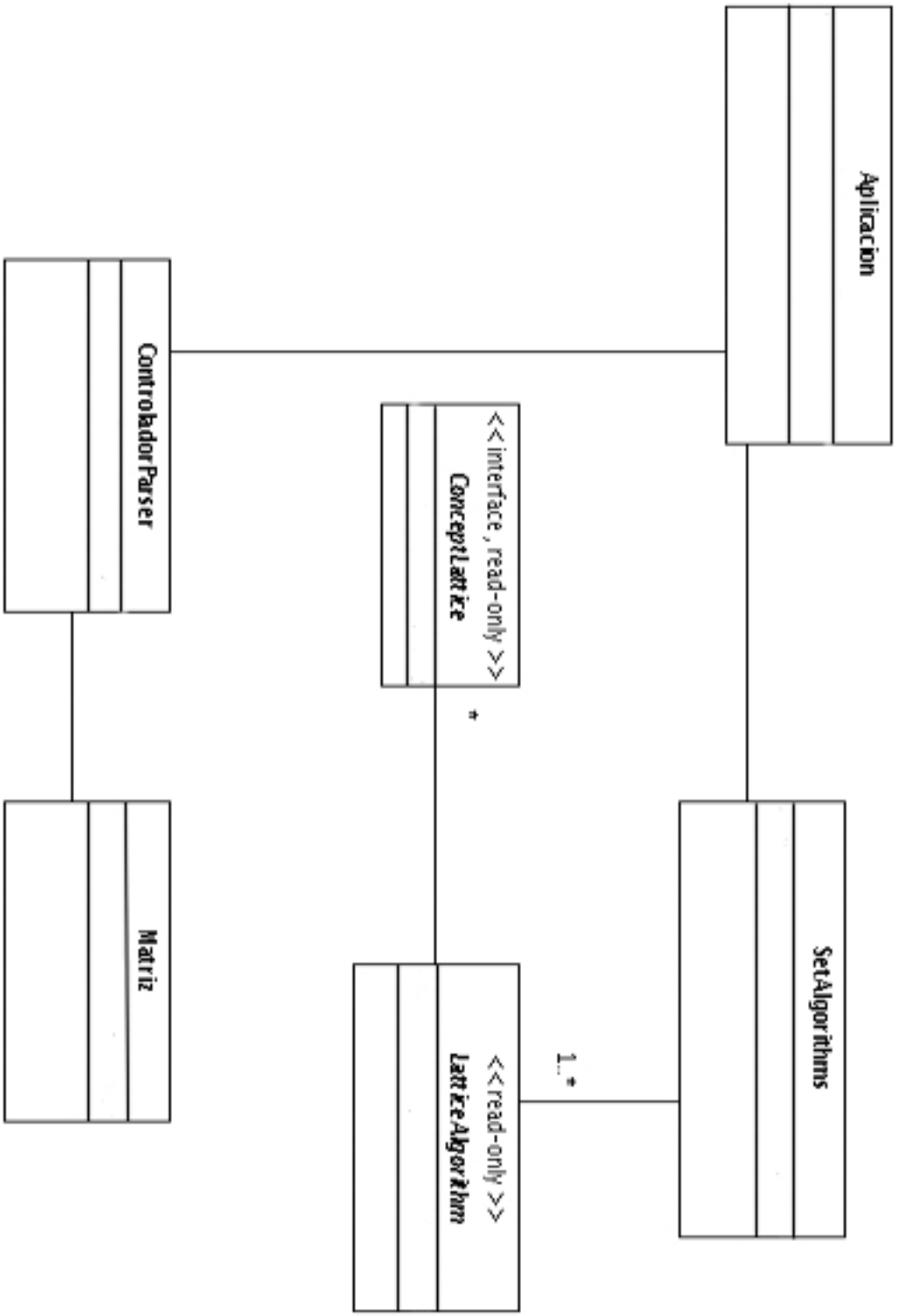

Figura 7-13 *Diagrama de clases en el escenario: Generar retículo de Galois.*

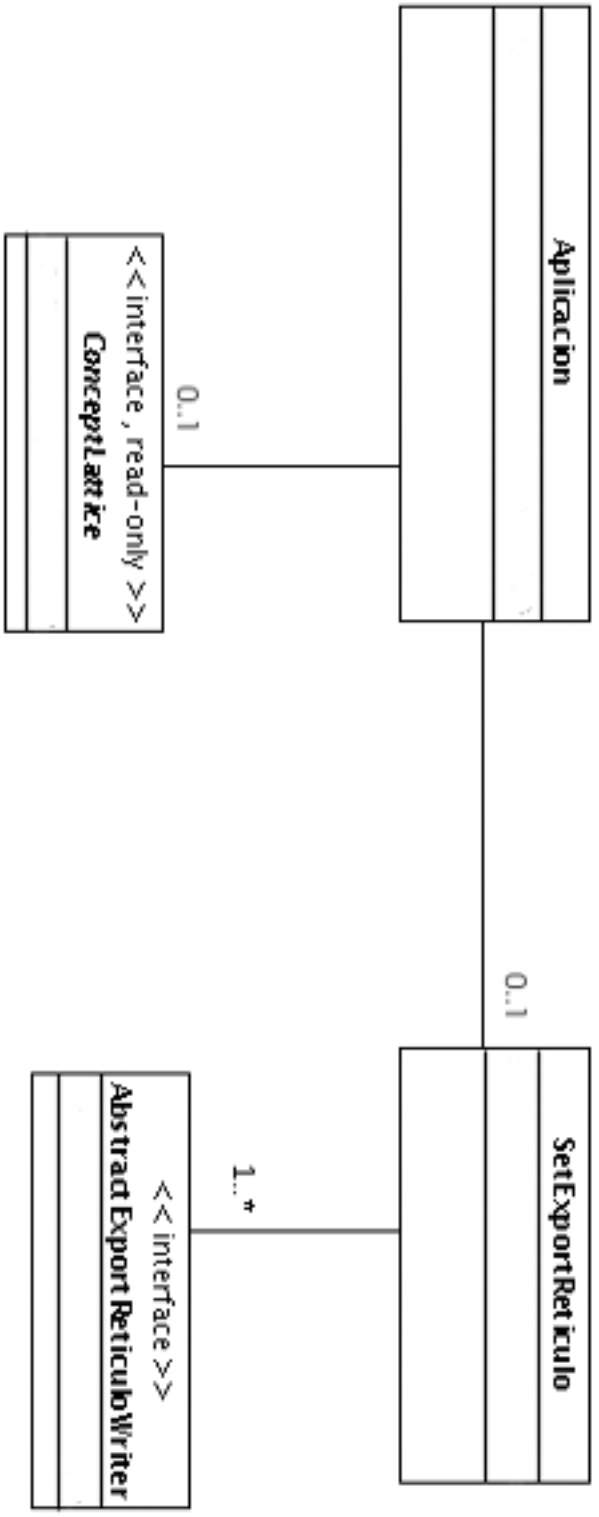

Figura 7-14 *Diagrama de clases en el escenario: Guardar retículo.*

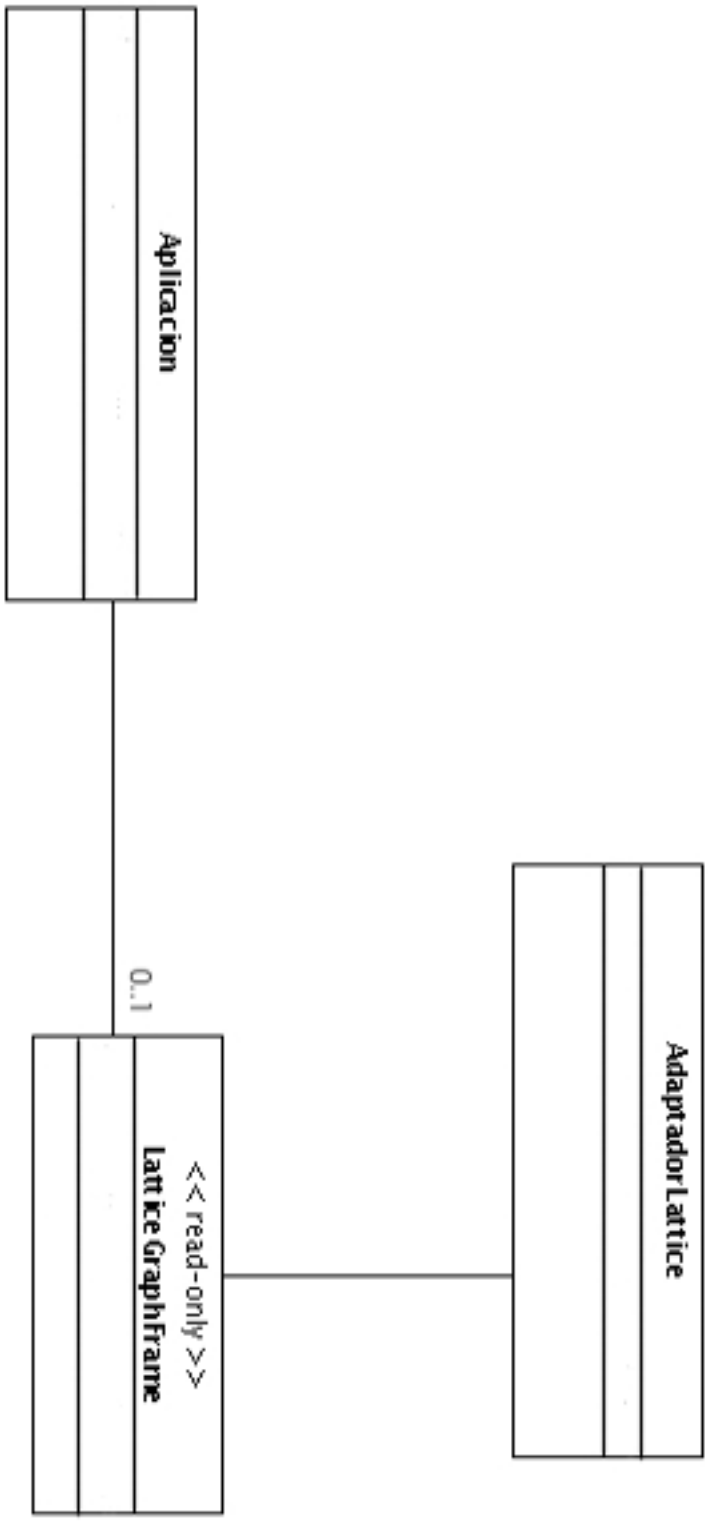

Figura 7-15 *Diagrama de clases en el escenario: Mostrar retículo*

7.6 Diagramas de Secuencia (Modelo Dinámico)

A continuación detallaremos cada uno de los diagramas de secuencia correspondientes con la representación directa de los conceptos obtenidos de los Casos de Uso. Ayudarán a entender mejor el funcionamiento de la aplicación y el diseño escogido para ella. Para cada diagrama, realizaremos los comentarios pertinentes que estimemos oportunos con el fin de facilitar una rápida comprensión de los mismos. Recordaremos al lector que dichos diagramas se encuentran en el CD-ROM de la memoria pudiendo encontrar las imágenes de dichos diagramas sin cortes y a un tamaño adecuado.

7.6.1 Arranque del Sistema

El primer diagrama de secuencia (ver **figura 7-16** y **figura 7-17**) muestra el proceso de creación de objetos correspondiente a la ejecución de la herramienta en modo gráfico o en modo de línea de comandos. Lo primero que realiza el sistema es comprobar que tenemos la versión de *Java* adecuada, si no es así saldrá informado de no tiene la versión adecuada y por tanto no podrá ejecutar la herramienta.

A continuación crea un objeto Aplicación si estamos ejecutando el programa en modo consola, o uno de clase `Start` en un hilo distinto al hilo de ejecución normal para evitar los puntos muertos en el tratamiento de eventos. `Start` será el encargado de lanzar el sistema así como de dar acceso a las distintas partes del modelo, en un modelo vista/controlador. Dicho objeto es el responsable de crear el interfaz gráfico, creando un objeto `FramePrincipalOpt` que será la ventana principal y que actuara de controlador a través de sus menús y barra de herramientas.

El objeto `Start` también creará un objeto `VistaContexto` que será tendrá el papel de vista correspondiente al modelo vista/controlador. Dicho Objeto no será más que una extensión del panel de contenido donde se representaran los datos concernientes a la matriz de incidencias de un contexto formal, así como el responsable de crear en una ventana aparte el retículo cuando éste sea construido. Una vez creado dichos objetos, el objeto de la clase `Start` creara un objeto `ControladorParser` que será quien controle las acciones en el análisis léxico-sintáctico, rellenando el objeto de la clase Matriz que inicialmente se creará vació, y como ya hemos dicho, rellenado según finalice el análisis del código objeto de estudio.

Finalmente, el objeto de la clase `Start` creará un objeto `ModeloContexto` que no será más que una lista enlazada de contextos, inicialmente vacía, que tiene adjuntos como observadores tanto el objeto de la clase `VistaContexto` como el objeto de la clase `FramePrincipalOpt`, siendo ambos como venimos diciendo la vista y el controlador respectivamente del modelo vista/controlador que utilizamos.

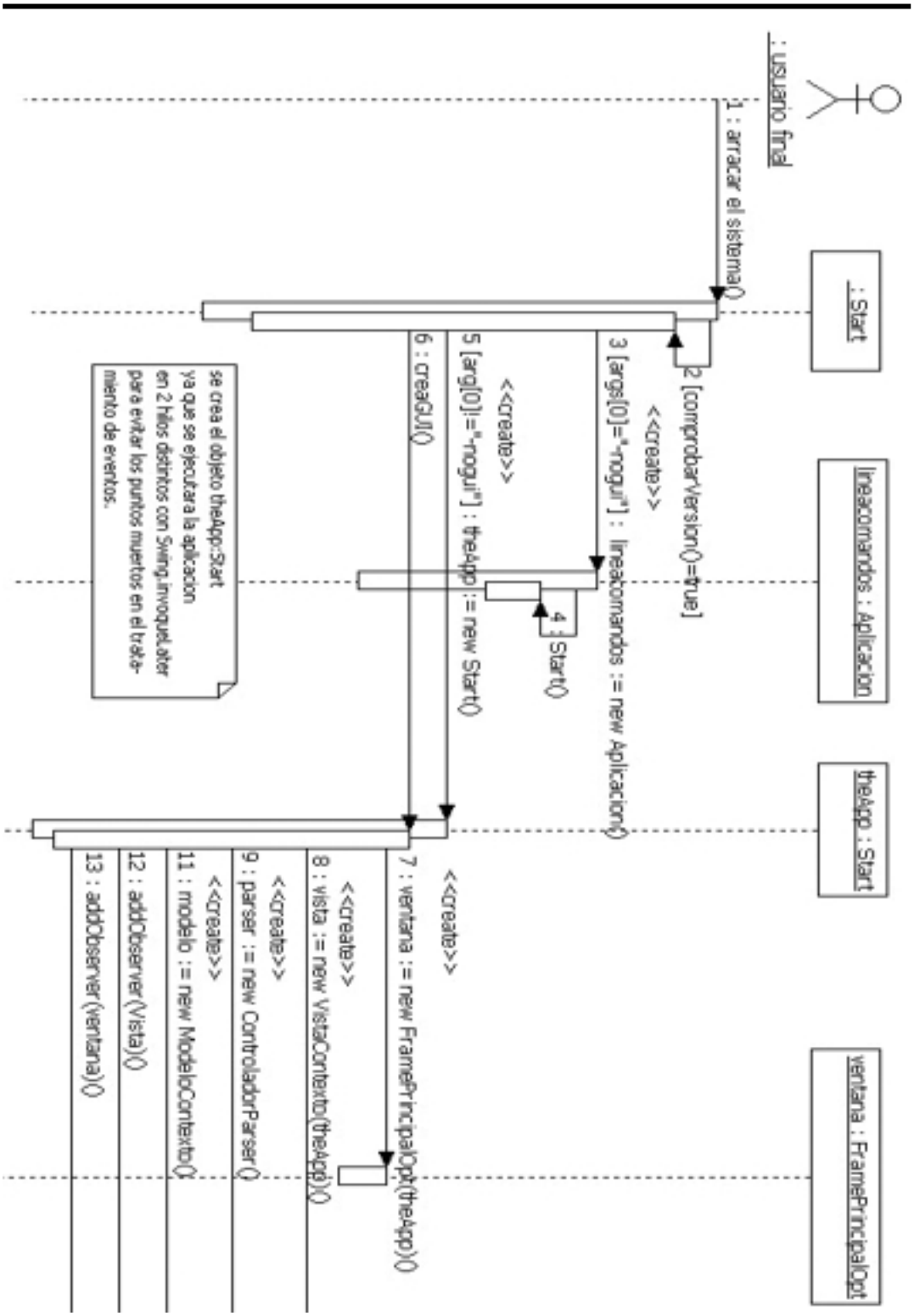

Figura 7-16 *Diagrama de secuencia del arranque del sistema (1 de 2)*

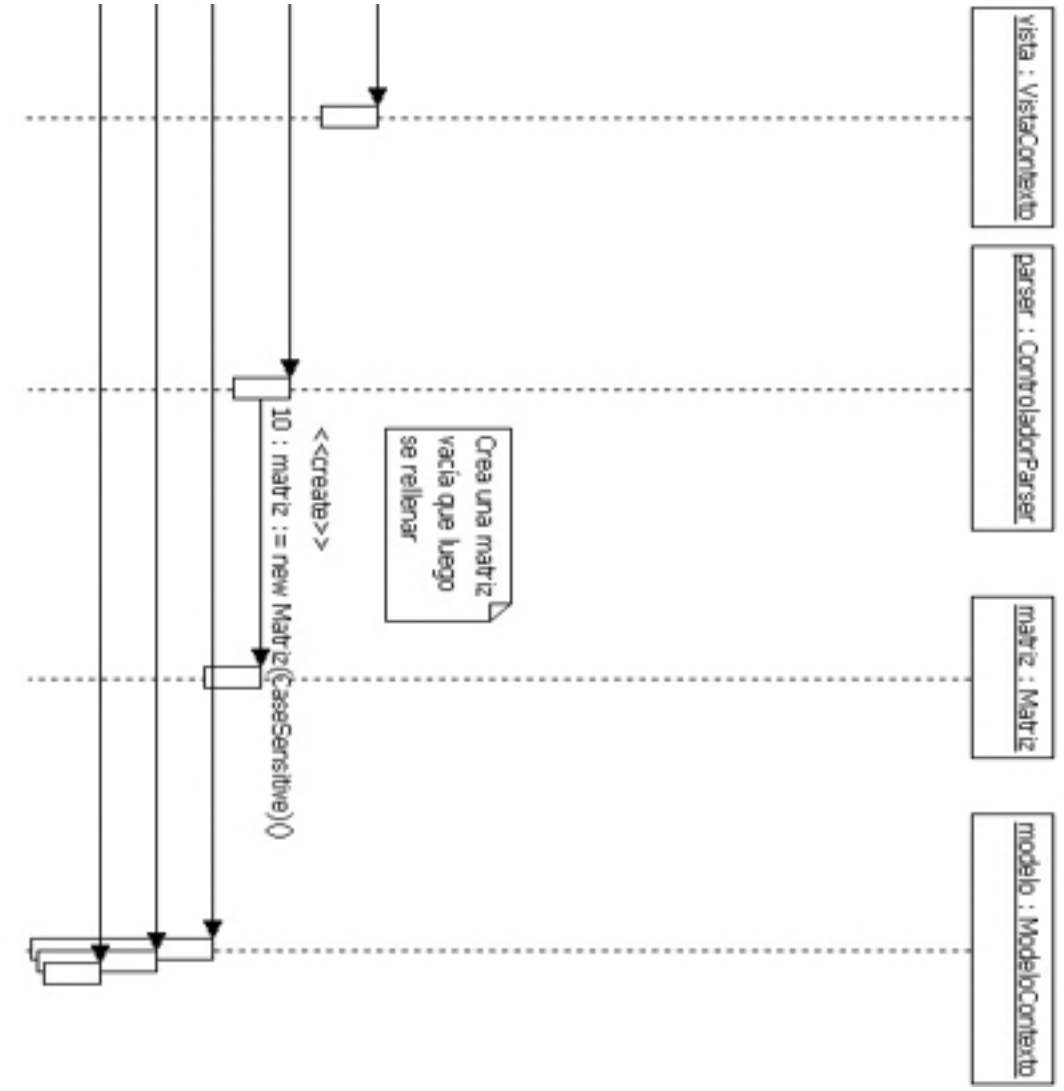

Figura 7-17 *Diagrama de secuencia correspondiente al arranque del sistema (2 de 2)*

7.6.2 Procesar un Contexto

En el siguiente diagrama estudiaremos la construcción de un contexto formal a partir del análisis léxico-sintáctico de una colección de clases o interfaces dadas por el usuario trabajando en un entorno gráfico de ventanas. Dicho diagrama de secuencia se corresponderá con los casos de uso **CU-01**, **CU-02** y **CU-09**.

Hemos dividido el siguiente diagrama en dos partes para una mejor comprensión del mismo, teniendo por una parte la interacción del actor con la ventana principal y luego la interacción de dicho actor con el objeto de la clase `SeleccionarFuentes`.

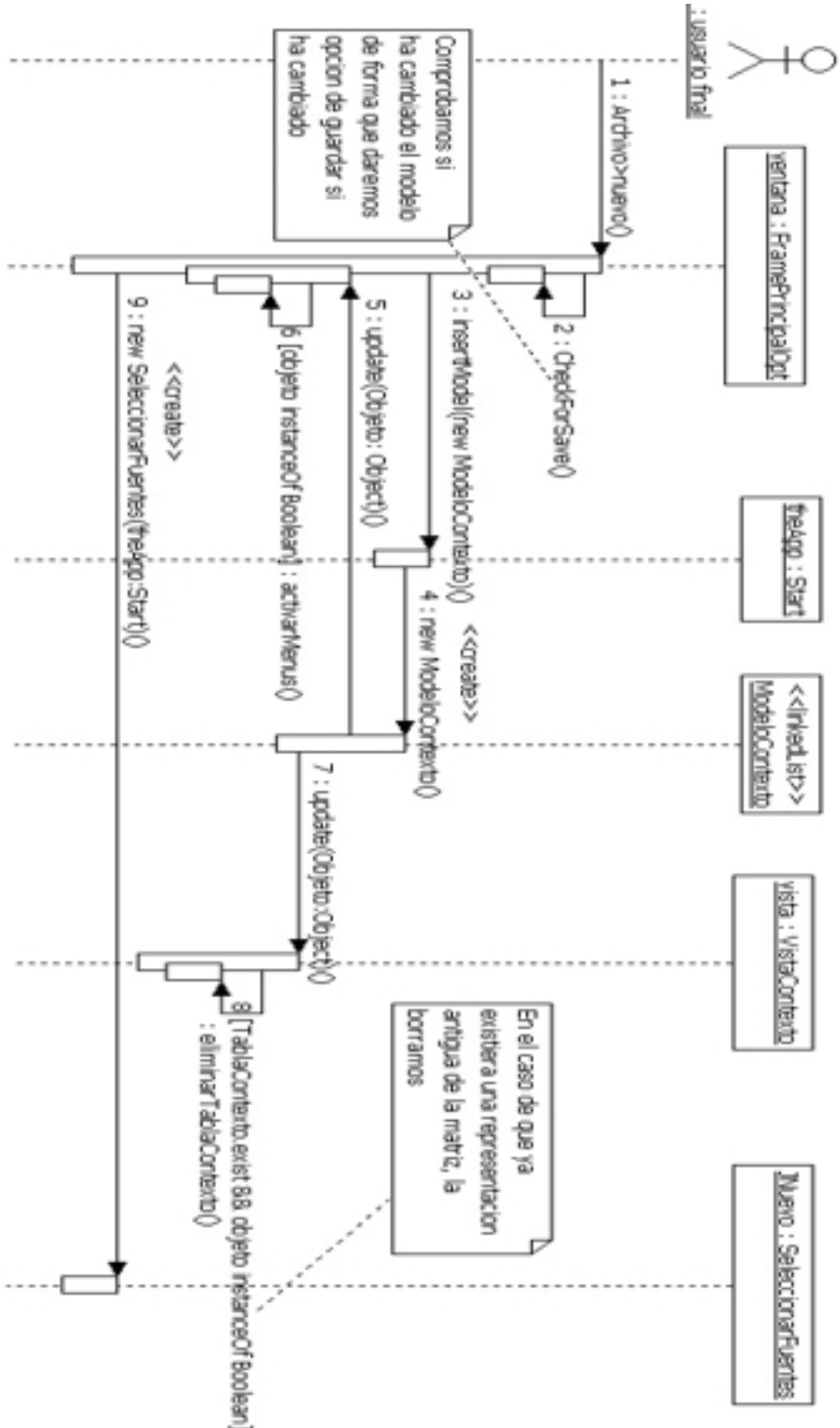

Figura 7-18 *Procesamiento de un contexto formal(parte 1).*

En primer lugar, en cuanto el actor selecciona construir un nuevo proyecto, se comprueba si el modelo ha cambiado o no, ya que puede ser la primera vez que seleccione nuevo o por el contrario puede tener ya algún contexto formal construido, de forma que si ya existe un contexto formal y no ha sido guardado (por lo tanto hay cambios en el modelo que no están guardados) se le preguntara al actor si desea guardar el archivo o no, pasando en caso afirmativo al diagrama de secuencia de guardar contexto referente al caso de uso **CU-04**.

Una vez hecha esa operación, se inserta un modelo nuevo a través del objeto de la clase `Start`, para ello se crea un objeto de la clase `ModeloContexto`, que no es más que una lista enlazada inicialmente vacía. En el momento que es creado dicho objeto se informa a las clases observadoras a través del mensaje `update(objeto: Object)`. En cuanto lo reciben tanto el objeto ventana (de la clase `FramePrincipalOpt`) como el objeto vista (de la clase `VistaContexto`), ventana actualizará los menús de ella misma (deshabilitando determinadas opciones) y vista borrara la representación del contexto anterior, en caso de existir. Señalaremos que aunque luego se cancele la operación, no se recuperaran dichos datos, por eso se avisa al usuario a la hora de crear un nuevo contexto formal que perderá la información del ya construido (si este existe) si éste ha cambiado y no ha guardado los cambios.

Por último el objeto ventana de la clase `FramePrincipalOpt` creará un objeto JNuevo de la clase `SeleccionarFuentes` que no será más que una nueva ventana donde el usuario introducirá los datos relevantes al *parser* que va a seleccionar, el directorio *classpath* donde buscar las clases para las posibles referencias existentes dentro del código (en los `import`) así como las clases o interfaces para analizar. Hay que destacar que a la hora de crear dicho objeto es necesario el pasarle como argumento el objeto parser de la clase `ControladorParser`, al que tendremos acceso a través del objeto theApp, para que Jnuevo establezca los valores introducidos por el usuario, en dicho objeto parser. En el proceso de creación del objeto JNuevo se cargaran los *parser* disponibles en el sistema, de forma que si hubiéramos incorporado alguno nuevo inmediatamente lo detectaría. Como ya vimos en el capítulo 5, es bastante sencilla la incorporación de nuevos *parser* sin que por ello se vea afectado el resto del sistema.

En la **figura 7-19** y **figura 7-20** podemos ver la segunda parte del diagrama de secuencia del procesamiento de una contexto formal a partir del punto donde lo habíamos dejado. Una vez el actor ha introducido los datos anteriormente citados, el objeto JNuevo mandará el mensaje `setParser(indiceLenguaje: int)` con el índice del lenguaje seleccionado ya sea para clases o para interfaces, le mandará también el mensaje para que establezca el *classpath* que haya introducido el usuario así como ira recorriendo el *array* de archivos que haya introducido el usuario pasándoselos al objeto *parser*. Si alguno de estos campos está vacío, el programa avisará al usuario de ello a través de un mensaje en pantalla.

Una vez disponga de todos los datos, iniciaremos el objeto parser con dicho valores mediante el paso del mensaje `iniciarParser()`. Acto seguido crearemos un nuevo contexto en el modelo, para ello accedemos al modelo a través de theApp, objeto de la clase `Start`, y a continuación añadiremos un nuevo contexto (de la clase del mismo nombre). Dicho contexto será el responsable de ejecutar el *parser* y obtener la matriz de incidencias del contexto formal que estamos construyendo. Una vez que tengamos el nuevo contexto con la matriz de incidencias, el objeto modelo de la clase `ModeloContexto` avisara a sus dos observadores (vista y ventana) pasándolos el objeto contexto, de forma que vista pintará con dicho objeto la matriz de contexto (ver **figura 7-21**) y ventana activará los menús correspondientes que permitirán operaciones con dicho contexto tales como construir el retículo de *Galois* asociado, guardar contexto, guardar retículo, o modificar *parser*. Los veremos más adelante.

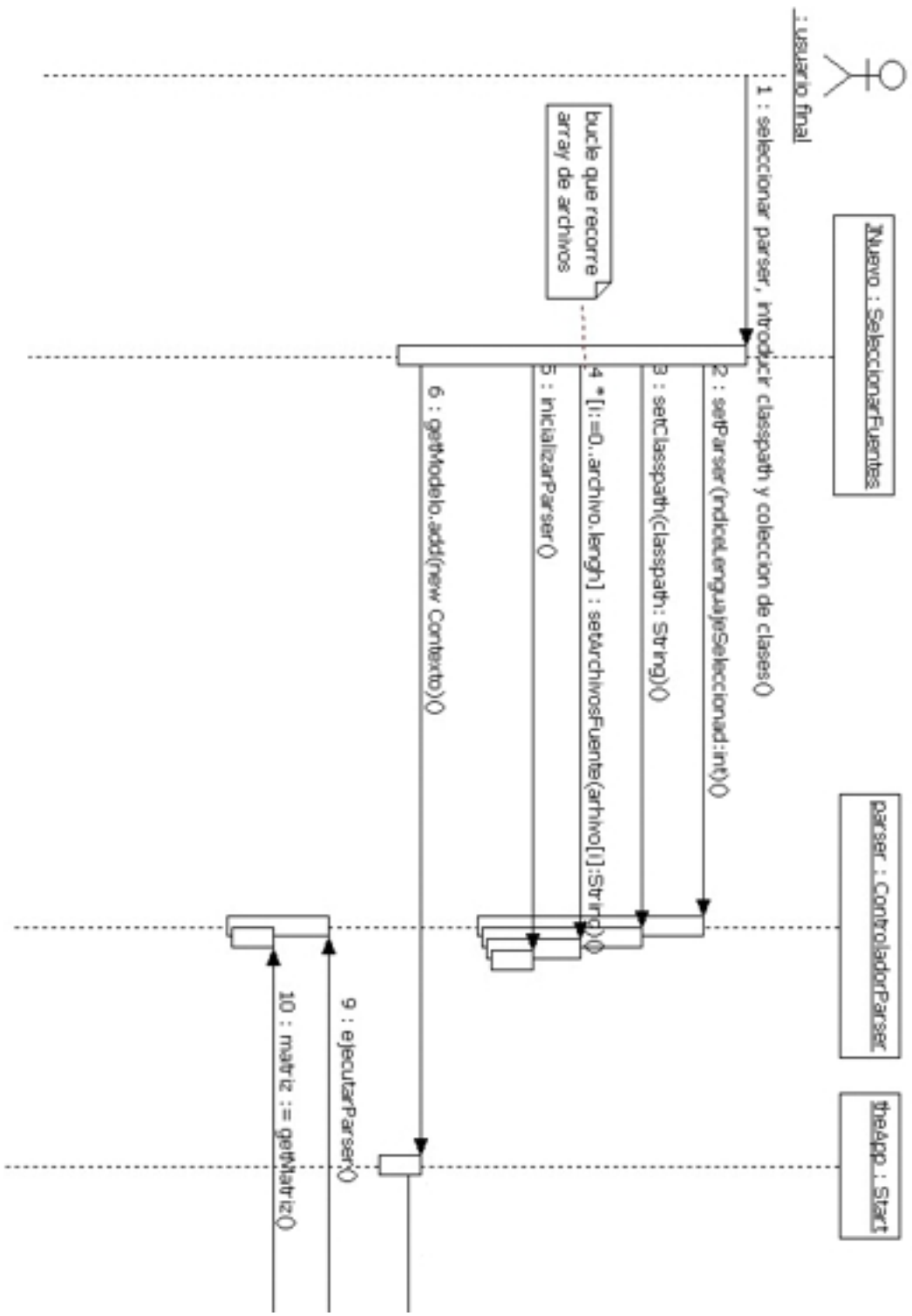

Figura 7-19 *Procesamiento de un contexto formal (parte 2).*

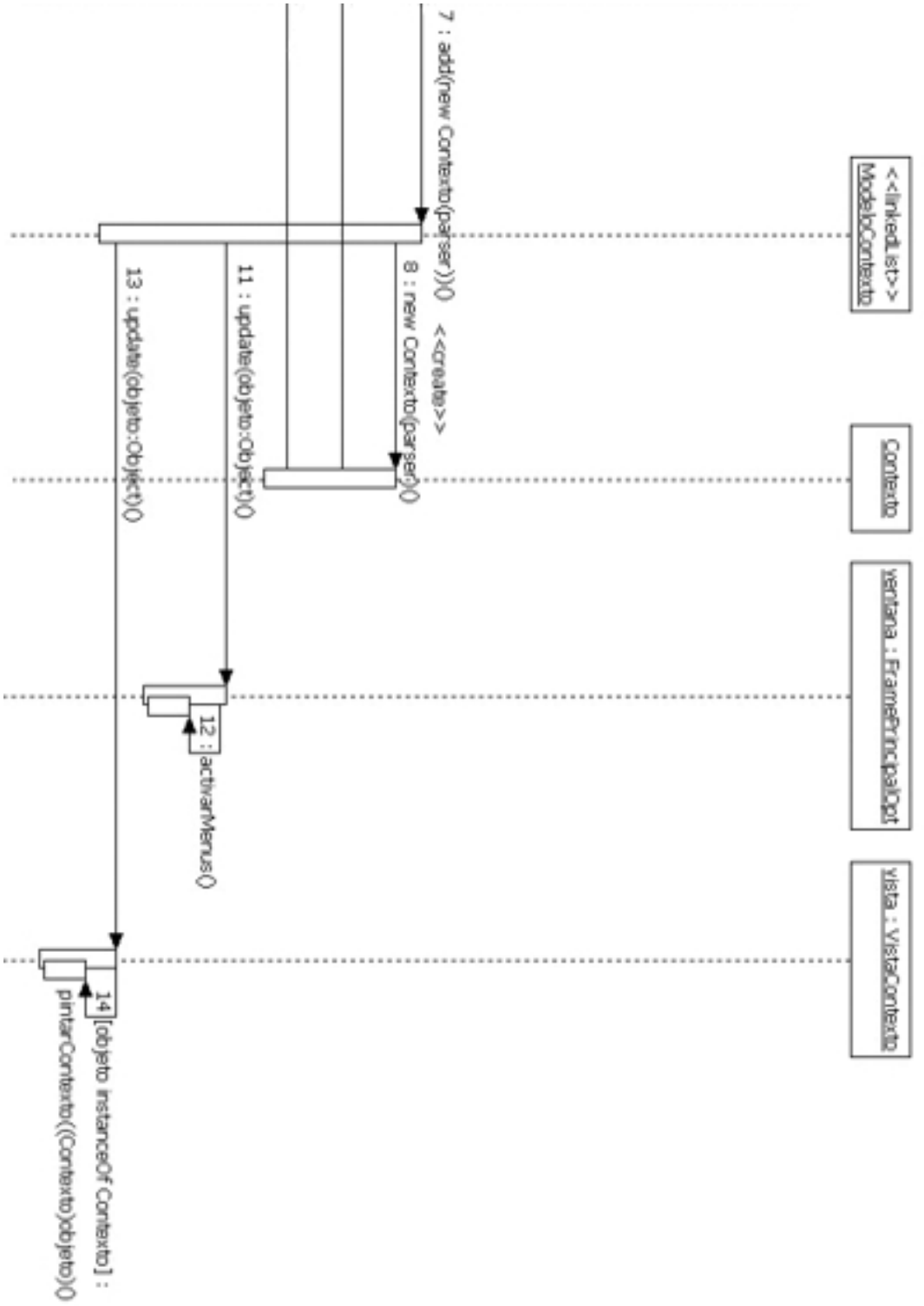

Figura 7-20 *Procesamiento de un contexto formal (parte 3).*

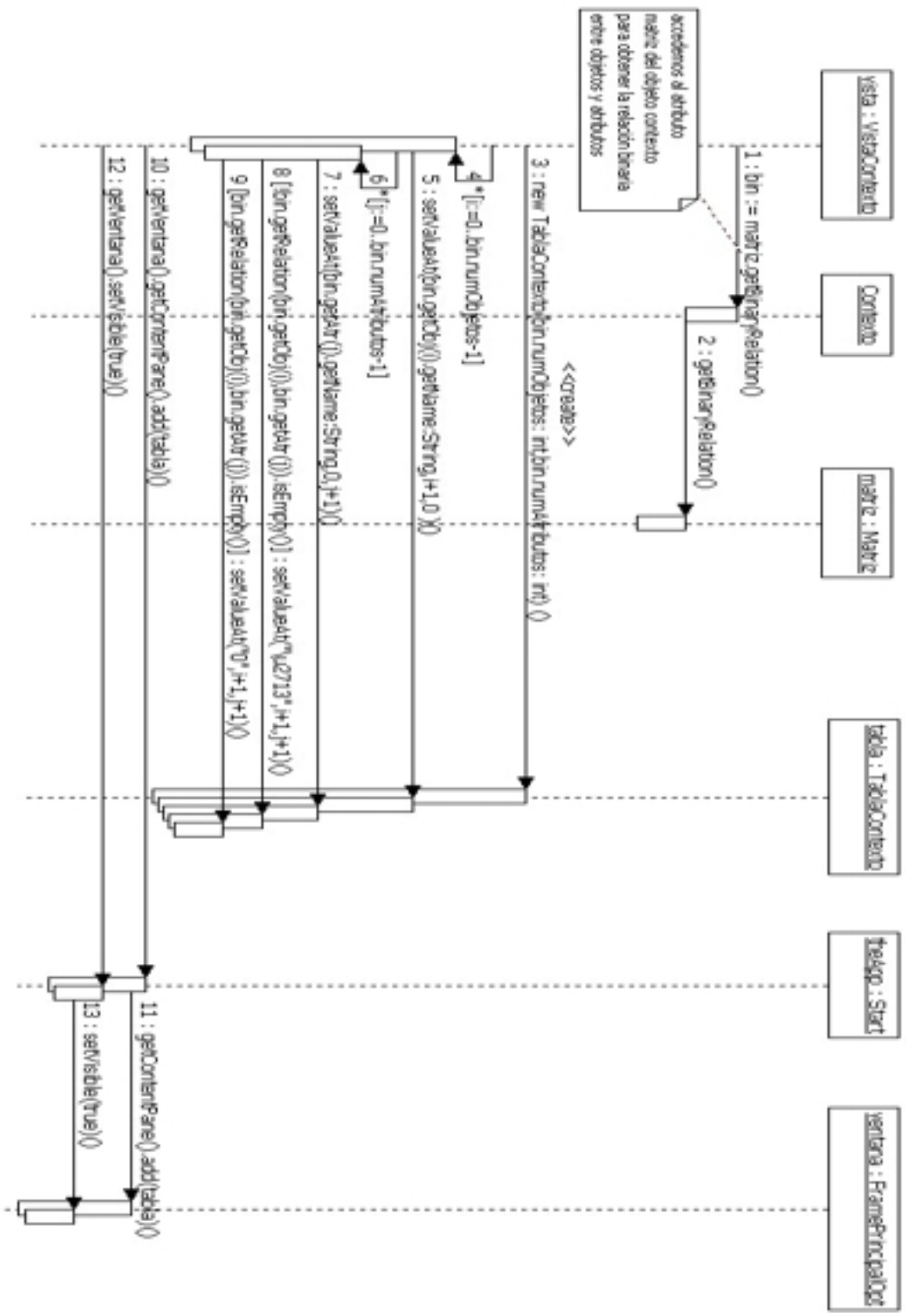

Figura 7-21 *Pintar un contexto formal.*

7.6.3 Cargar un Contexto Formal

En dicho diagrama se refleja el proceso en la construcción de un contexto formal o de un retículo de *Galois* a partir de la carga de un archivo de datos XML con un formato compatible con los disponibles en el sistema. Corresponde a los casos de uso: **CU-01**, **CU-02**, **CU-07**, **CU-08**, **CU-09**; todos ellos correspondientes a la construcción y visualización de un contexto formal o un retículo de *Galois*.

Como nota a destacar recomendaremos al lector encarecidamente el estudio de este diagrama a partir de diagrama contenido en el CD-ROM pues sus debido a sus dimensiones, su reducción a un tamaño B5 incluso dividido como los otros diagramas puede dificultar en grado sumo la comprensión del mismo. Aun así se plasmará en dicho documento, **figura 7-22** y **7-23**, con el fin de guiar la descripción del mismo.

El proceso comienza como el anterior, haciendo un chequeo del modelo para detectar los posibles cambios no guardados que haya podido realizar el usuario, dando la posibilidad de guardarles, si no los ha guardado y no los guarda se perderán. Se crearán los dos objetos que manejarán los exportadores disponibles, dos vectores uno de la clase `SetImportContexto` y otro de la clase `SetImportReticulo`, pues todavía no sabremos que tipo de archivo introducirá el actor y necesitamos las extensiones de los formatos compatibles con el sistema para los filtros en la selección de archivos. Con lo que cargaremos las extensiones de los exportadores disponibles. A continuación mostraremos un menú de selección de archivos, para que actor elija el archivo de datos XML que cargará. Una vez seleccionado se confrontará la extensión de dicho archivo con las disponibles para saber que tipo de archivo es, y la manera a proceder con él. Si es un archivo de contexto con un formato compatible con los disponibles en el sistema, se creará una entidad de tipo `AbstractImportContextoReader;` realmente no se crea una entidad de dicha clase ya que es abstracta y tiene métodos sin implementar, sino que se creará de una clase que heredará de `AbstractImportContextoReader` implementando los métodos de dicha clase; sin embargo utilizamos ese tipo, y así lo reflejamos en el diagramas, ya que es el genérico para todos sin depender de los formatos disponibles del sistema. De forma que si en un futuro si se añade una nueva DTD, únicamente habrá que crear una clase que implemente dichos métodos adaptados a esa nueva DTD, sin que repercuta en el resto de la aplicación. De forma análoga se realizará si se quiere cargar un retículo. Dicho objeto, será el “lector” del archivo, de forma que le adjuntaremos el archivo seleccionado, y se cargará. Lo primero que mirará será el número de contextos o retículos almacenados en dicho archivo de datos, ya que al ser uno de los formatos del sistema compatible con el proyecto [DBRE, 2005] (y recordemos que dicho formato podía guardar múltiples contextos o retículos mientras que nuestra herramienta trabaja únicamente con un solo retículo o contexto), pues se debe seleccionar cual de entre los posibles se cargará.

Una vez seleccionado el contexto o retículo a cargar, el siguiente paso será introducir un modelo nuevo a través del objeto theApp, de forma que al hacer eso se actualizarán los observadores (vista y ventana) borrando el anterior contexto si éste había sido construido en el caso de vista y se modificaran los menús deshabilitando las opciones correspondientes en el caso de ventana. Una vez hecho esto, se cargará la relación escogida por el usuario, ya sea un contexto o un retículo. Una vez aquí, los caminos a proceder son distintos. De forma que si se ha cargado un contexto, a partir de la relación obtenida crearemos un objeto contexto, y los pasos a seguir serán los mismos que antes cuando procesamos un contexto formal, pudiendo realizar después cualquier opción como si se tratara de un contexto construido desde cero, mientras que si es de un retículo de quien se trata, únicamente se pintará en una ventana nueva, pudiendo únicamente visualizarle como se establecía en la especificación de requisitos.

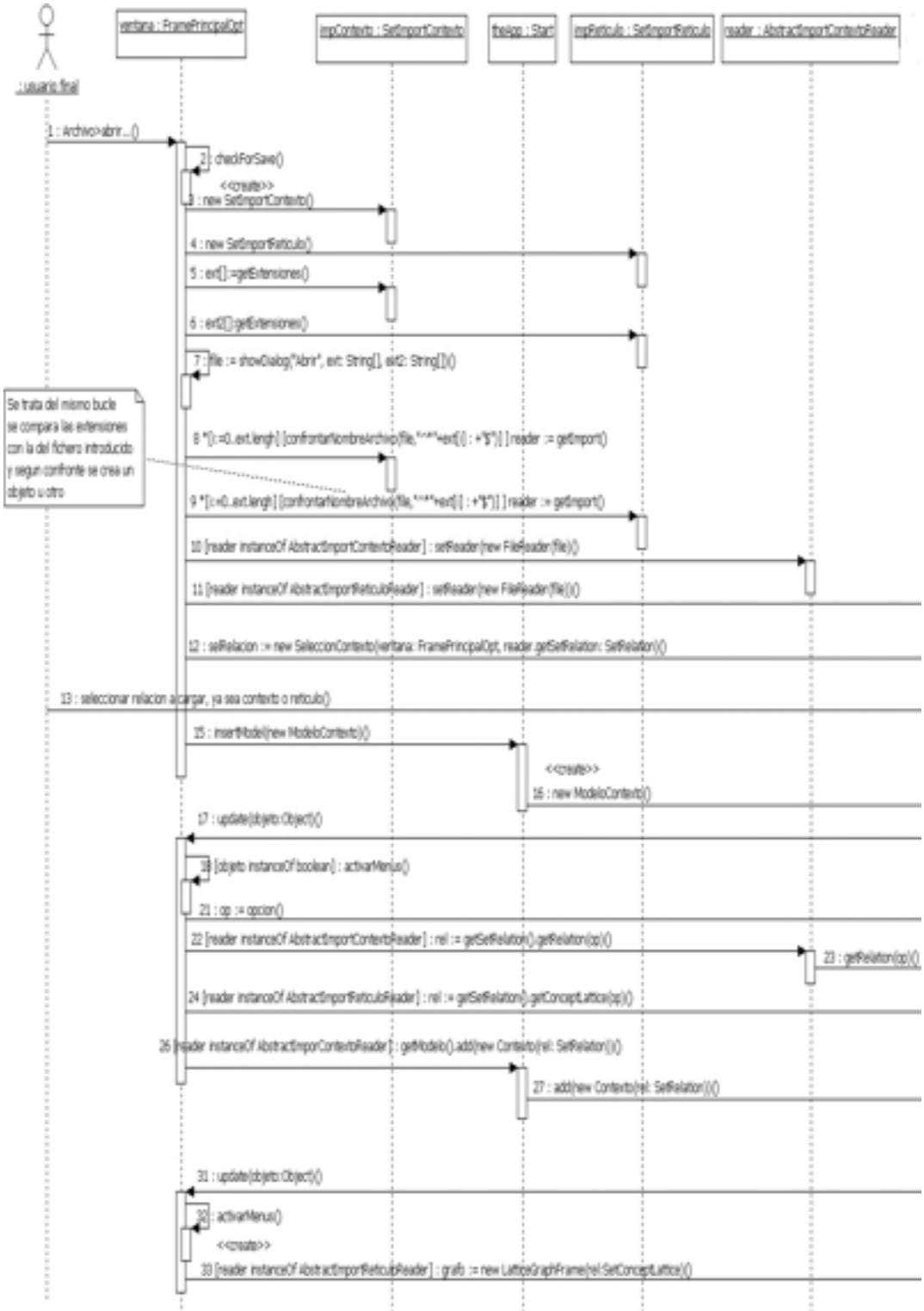

Figura 7-22 *Proceso de carga de un archivo de datos en el sistema (parte 1).*

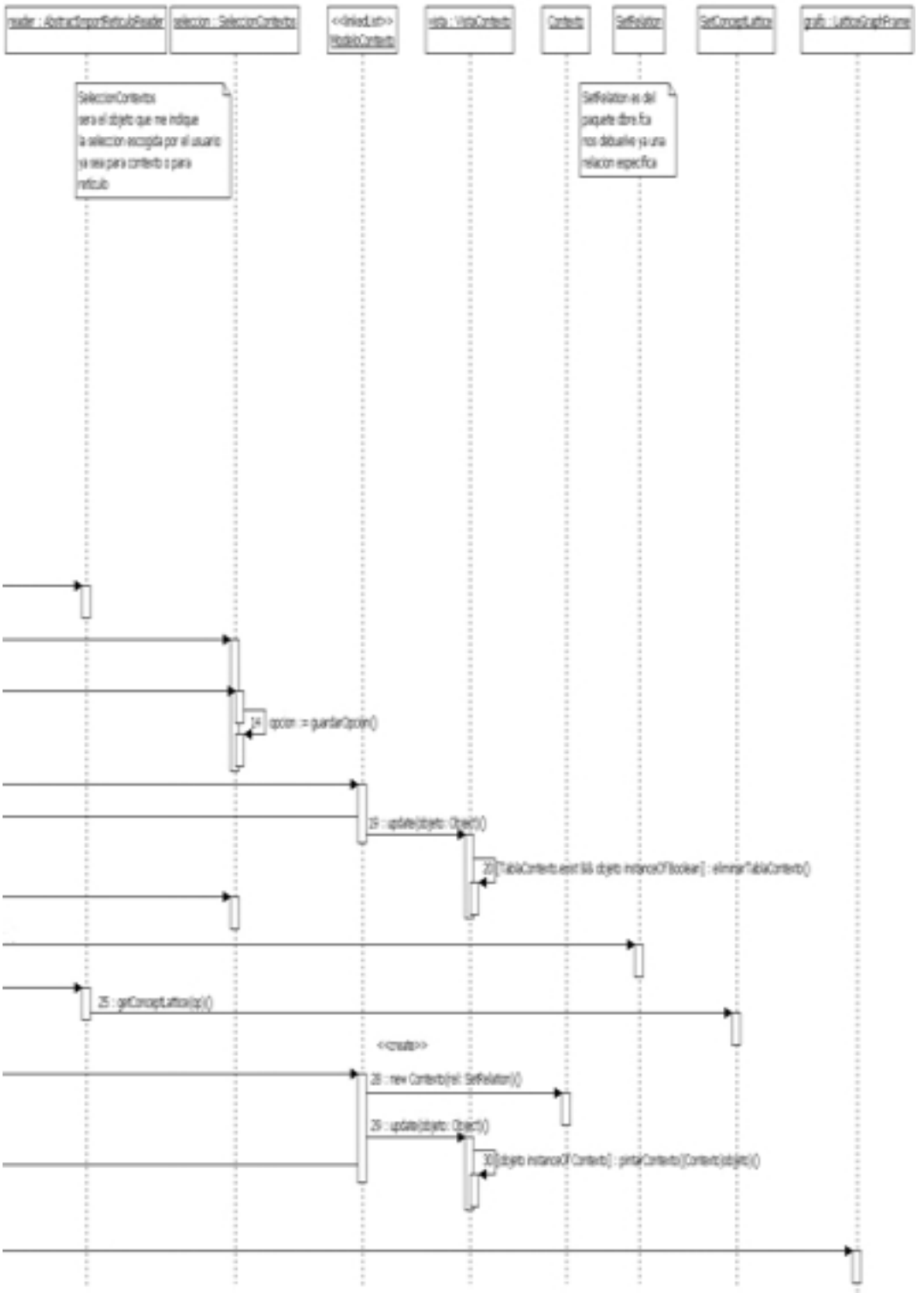

Figura 7-23 *Proceso de carga de un archivo de datos en el sistema (parte 2).*

.

7.6.4 Modificar un Contexto

En dicho diagrama se refleja la posibilidad de poder modificar la matriz de contexto a través de la inclusión de nuevas clases o interfaces analizar junto a unas ya analizadas. Contempla el caso de uso **CU-03**, en el que como ya se vio necesita que previamente se haya analizado un conjunto de clases o interfaces, de no ser así, al seleccionar modificar *parser* será como si hubiéramos seleccionado nuevo trabajo. La secuencia de dicho diagrama se puede observar en la **figura 7-24**.

Lo primero que se hace, como venimos haciendo hasta ahora, es analizar si el modelo ha cambiado, es decir ver si se han guardado o no las modificaciones que pudieran existir en un modelo cambiado ya que al modificar *parser* introduciremos un nuevo modelo con su respectivo contexto. Una vez hecho eso, se procederá con las acciones análogas a las de "procesar contexto formal", es decir, se creara un nuevo modelo vacío, dicho modelo avisara a sus dos observadores, vista y ventana, para que actualicen los menús en caso de ventana, deshabilitando determinadas opciones, y vista por su parte se encargara de borrar la posible visualización del contexto anterior.

Una vez hecho eso, el siguiente paso que en "procesar contexto formal" sería el de construir un objeto de seleccionar fuentes y rellenarle con los datos referentes al análisis del conjunto de clases, aquí únicamente tendremos que hacer una llamada a dicho objeto, haciéndole visible de nuevo. Dicho objeto guardará todavía los valores anteriormente introducidos por el actor, de forma que al hacer visible de nuevo la ventana que representa a dicho objeto el actor podrá modificar dichos datos añadiendo nuevas clases, o eliminando clases de dicho conjunto. Y no sólo eso, también podrá modificar el *classpath*, sino incluso podrá cambiar de opción en el *parser* que había seleccionado anteriormente, pudiendo cambiar a uno diferente.

Cabe destacar que si dicho objeto, JNuevo de `SeleccionarFuentes`, no existe, ya sea porque en el sistema todavía no se ha creado ningún contexto formal, se creará uno nuevo, sin ningún dato introducido por el actor, como si se tratara de un trabajo nuevo, construyendo un contexto formal desde cero.

Una vez modificado dicho, con las opciones nuevas que el actor haya estimado convenientes, se procederá al procesamiento de dicho contexto formal, analizando las clases nuevas (o interfaces en el caso de que estemos analizando estos últimos) junto con las ya anteriormente analizadas. Habrá que analizar de nuevo éstas, pues al introducir nuevas clases (o interfaces) pueden existir nuevas incidencias entre la clase nueva y métodos de otras clases ya detectados, surgiendo nuevos conceptos. El proceso para realizar esto será exactamente el mismo que se realiza en "procesar un contexto formal" a partir de cero, únicamente con nuevos datos añadidos a los que ya teníamos. Por ese mismo detalle, una vez hecho visible el objeto JNuevo de la clase `SeleccionarFuentes`, se continuará en el diagrama de "procesar un contexto formal", cuando el actor introducía los datos referentes al *classpath*, conjunto de archivos a analizar, y el *parser* que utilizará. Sólo que en este caso, introducirá las modificaciones oportunas sobre los ya existentes. Después de eso los pasos son los mismos.

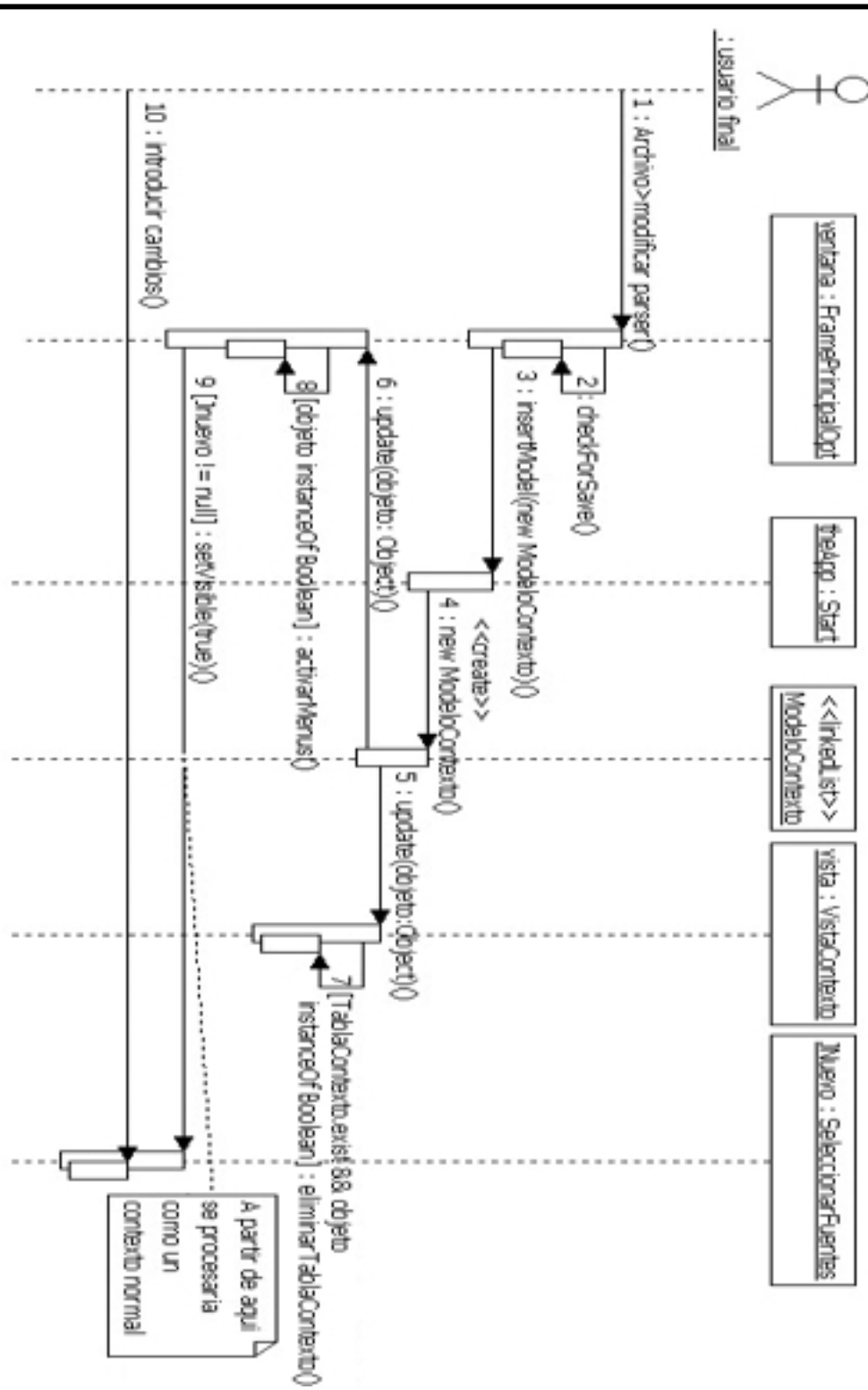

Figura 7-24 *Modificar un contexto formal*.

7.6.5 Construir un Retículo de *Galois*

En dicho diagrama se refleja la construcción de un retículo de *Galois* a partir de la matriz de incidencias de un contexto formal ya procesado. Contempla los casos de uso **CU-07** y **CU-08**, en el que como ya se vio necesita que previamente se haya procesado un contexto formal, habiendo obtenido su matriz de incidencias; de no ser así, no estará habilitada la opción de construir dicho retículo. La secuencia de dicho diagrama se puede observar en las **figuras 7-25** y **7-26**.

Comienza el diagrama cuando el actor selecciona la opción de construir un retículo a partir del algoritmo de *Bordat* (es el único disponible en este momento); ya se vio en capítulos anteriores como se podría integrar fácilmente nuevos algoritmos de forma que no sólo estuviera disponible el de *Bordat*. Una vez realizada esta tarea, la ventana principal (objeto ventana de la clase `FramePrincipalOpt` que realiza las veces de controlador) llama al objeto vista para que pinte el retículo, pudiendo acceder a la vista a través del objeto theApp de la clase `Start`. Vista a su vez necesitará el contexto con el que estamos trabajando, para ello recuperará el objeto modelo de la clase `ModeloContexto` a través del objeto de clase `Start`; dicho modelo no es más que una lista enlazada de `Contextos`, con lo que el siguiente paso será localizar dicho contexto. Como nuestra herramienta sólo trabaja con un contexto por vez, dicho contexto estará el primero de todos. Sin embargo aunque solamente se trabaja con un sólo contexto, tiene sentido diseñar modelo como una lista enlazada pues de esa manera en una futura ampliación se podría trabajar con múltiples contextos. Una vez localizado el contexto, se consigue el retículo a través del paso del mensaje `getRetículo(Algoritmo: int)` a dicho contexto, devolviendo éste un objeto de tipo `ConceptLattice`. De forma que es dicho contexto el responsable de devolver un retículo a vista para que pueda representarlo. Para ello, crea un objeto de tipo `SetAlgorithms` que será el que maneje los posibles algoritmos disponibles en el sistema. Contexto le pasará el índice del algoritmo escogido y un objeto `BinaryRelation` obtenido a través del atributo matriz que contendrá la matriz de incidencias correspondiente al contexto formal con el que estamos trabajando. Con ellos, `SetAlgorithms` creará un objeto de la clase `LatticeAlgorithm` del paquete `dbre.algotithms` del proyecto [DBRE, 2005]. En ese momento, el objeto contexto le mandará el mensaje `doAlgoritm()` a `LatticeAlgorithm`, de forma que este último aplica el algoritmo de *Bordat* para la construcción de un retículo de *Galois* mediante el uso de dicho algoritmo. `LatticeAlgoritm` devolverá un objeto de clase `ConceptLattice` en el momento en que contexto le mande el mensaje `getLattice()`, y a continuación realizará una simplificación del mismo cuando contexto le pase el mensaje a `ConceptLattice` de `fillSimplify()`. Una vez ya tenemos el retículo, únicamente restará pintarlo. Para ello se reutiliza el motor gráfico de [DBRE. 2005], de forma que desde el objeto vista se crea un objeto grafo de clase `LatticeGraphFrame` correspondiente al paquete `dbre.gui.graph`. Antes de hacer visible dicho objeto con la construcción del diagrama de *Hasse* del retículo en cuestión, es necesario activar los menús de la ventana principal, para ello se le manda al objeto ventana de clase `FramePrincipalOpt` de que active los menús y a continuación ya se hace visible el objeto grafo de la clase `LatticeGrapthFrame` mostrando una visualización de dicho diagrama de *Hasse*.

Por último destacar que los objetos de las clases `LatticeAlgorithm`, `ConceptLattice` y `LatticeGraphFrame` son pertenecientes al proyecto [DBRE, 2005], consistiendo las dos primeras como los objetos necesarios para la obtención de un retículo compatible para su representación mediante el motor gráfico del proyecto anteriormente mencionado. Y no sólo será necesario que sea compatible para su visualización sino también para su posterior almacenamiento en un archivo de datos XML con formato compatible con [DBRE, 2005].

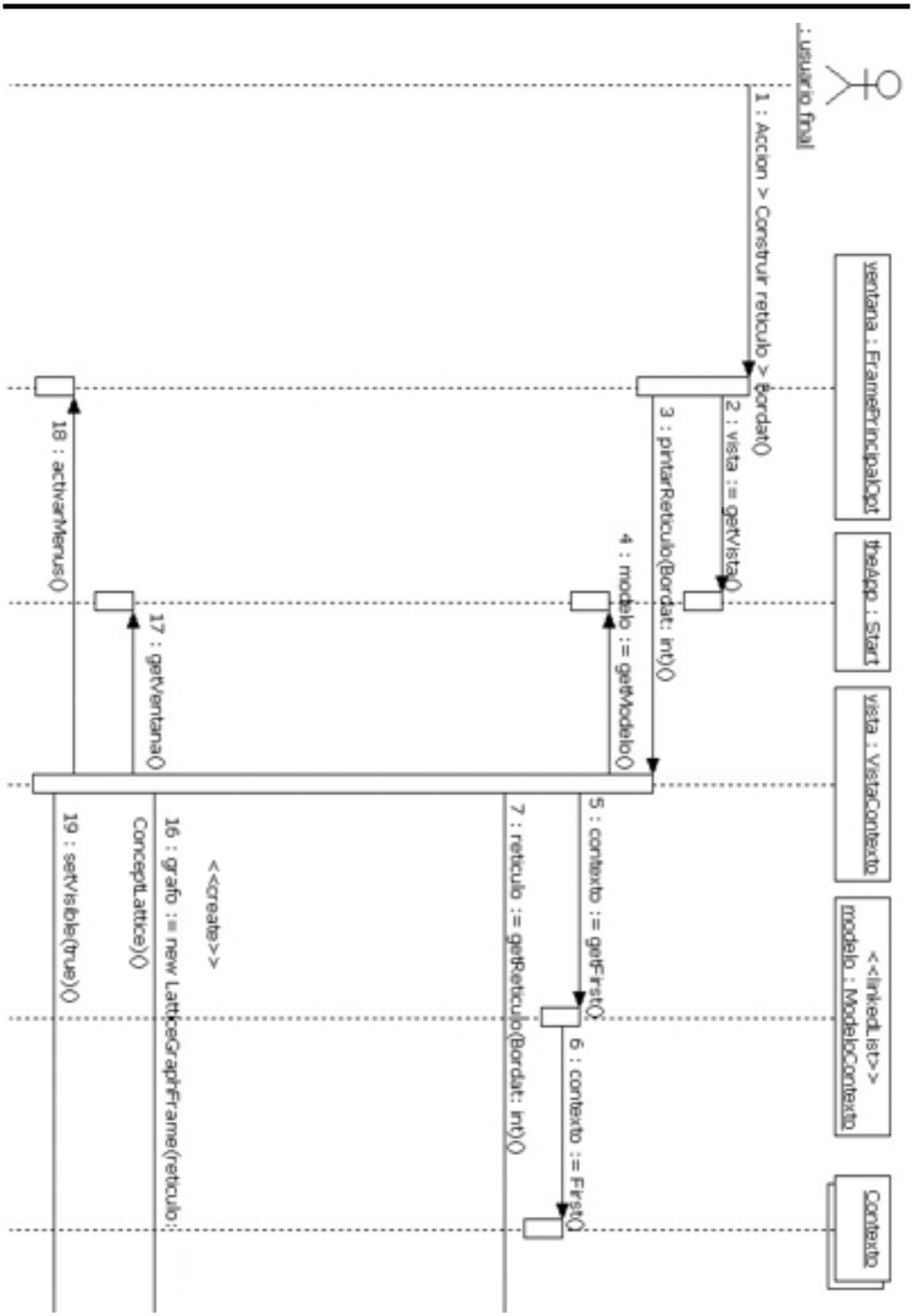

Figura 7-25 *Procesar un retículo de Galois (parte 1).*

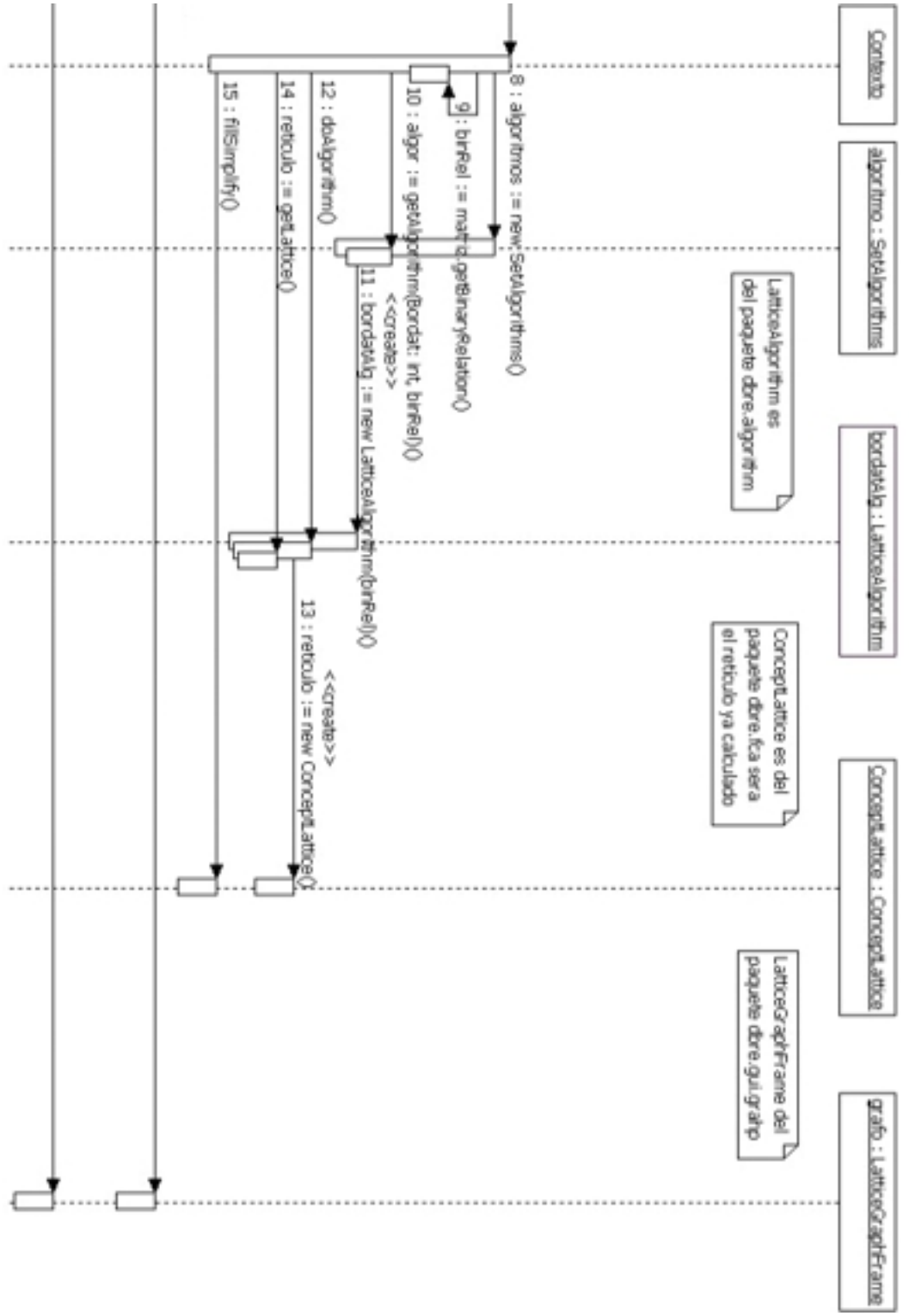

Figura 7-26 *Procesar un retículo de Galois (parte2).*

7.6.6 Elegir Formato para la Exportación

En dicho diagrama se refleja la elección por parte del actor, usuario final, del formato del archivo de datos XML para su exportación. Contempla el caso de uso **CU-05**, en el cual se seleccionaba dicho formato; si no se selecciona ninguno, la aplicación toma por defecto los exportadores compatibles con el proyecto [DBRE, 2005]. La secuencia de dicho diagrama se puede observar en la **figura 7-27**.

Dicho diagrama comienza cuando el actor selecciona "opciones" en el menú "ver" de la ventana principal (objeto ventana de clase `FramePrincipalOpt`). Seguidamente el objeto ventana creará un objeto de clase `PanelOpciones` donde se mostrará los exportadores disponibles para guardar la información relativa a la matriz de incidencias del contexto formal con el que estemos trabajando, o con la información relativa al retículo de *Galois* que podamos construir. Para ello, cuando creamos un objeto de clase `PanelOpciones`, este mostrará al actor una ventana con dos listas, una de los exportadores disponibles para contextos formales y otra para retículo de *Galois*. Dichas listas se crearán a partir de las descripciones de los exportadores devueltos por los objetos de clase `SetImportContexto` y `SetImportRetículo`, clases que serán los responsables de manejar los exportadores disponibles. Para ello es necesario que previamente se hayan creado dichos objetos antes de obtener las descripciones.

`SetImportContexto` y `SetImportRetículo` no son más que vectores de exportadores con los métodos relativos al manejo de los mismos. Destacando entre tales métodos el que utilizamos en esta opción, `descripciones()`, que devuelve un array de `Strings` con la descripción de cada exportador.

Una vez que el objeto de `PanelOpciones` ha obtenido las descripciones de los exportadores existentes para salvaguardar los contextos o retículos, se acaba de construir la ventana y se hace visible desde la ventana principal, para que así el actor selecciones el formato que buenamente estime apropiado. Si dicho proceso de selección no se lleva a cabo, o simplemente no selecciona ninguno en el proceso, la aplicación, por defecto, tiene establecido como parámetros para la exportación el uso de los formatos compatibles con [DBRE, 2005] junto con otros parámetros definidos por defecto en `ConstantesDefaultFrame`, tales como la elección por defecto del algoritmo de *Bordat* para la construcción de retículos, etc.

Una vez hecha dicha selección por parte del actor, la ventana principal establecerá dicha elección como formato de exportación sustituyendo al que esté por defecto. Cabe destacar que tal selección de formato se puede realizar en cualquier momento de la ejecución de la aplicación, sin tener que existir previamente ni retículos, matrices de incidencia etc.

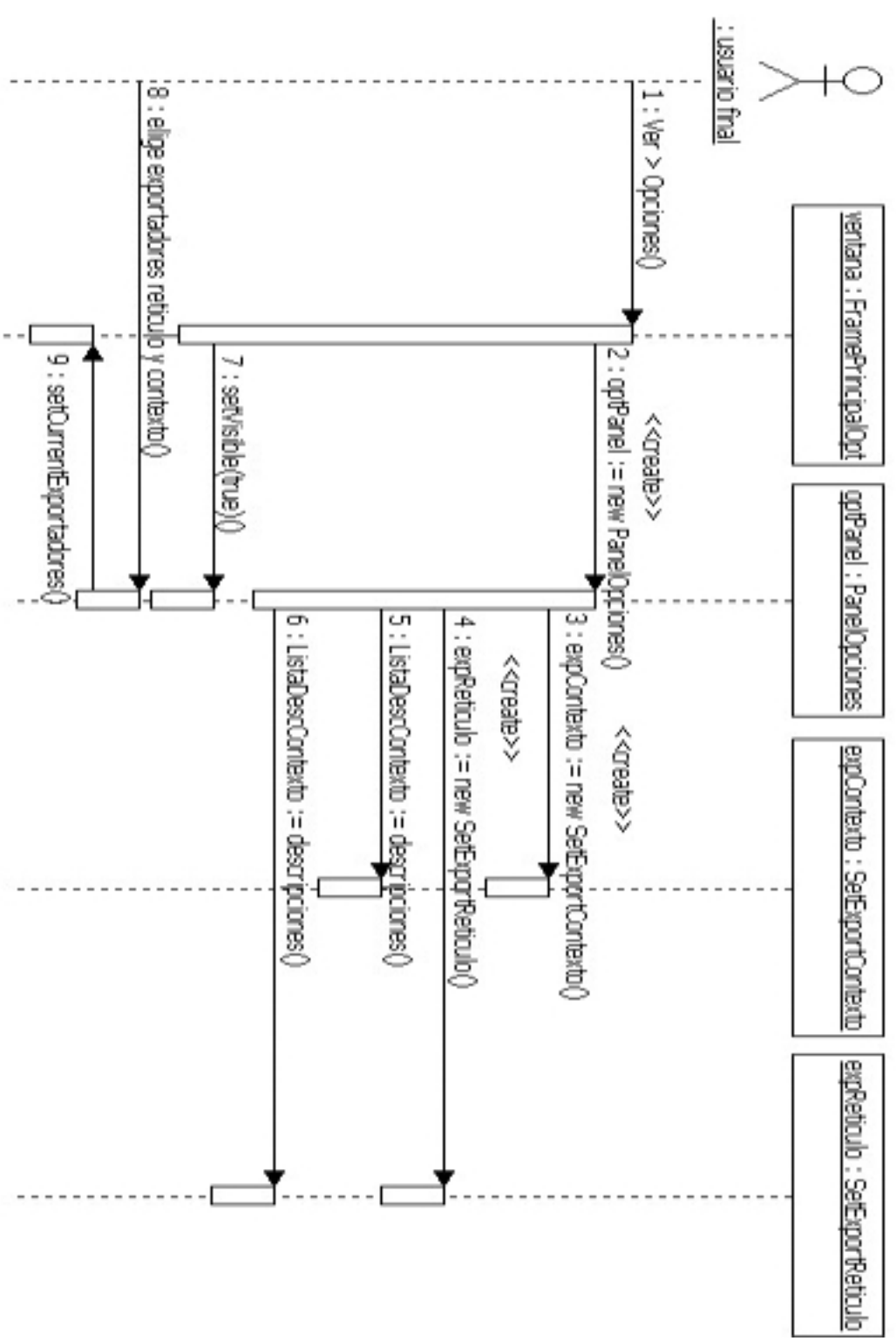

Figura 7-27 *Selección de formato para la exportación de retículos y contextos.*

7.6.7 Guardar el Retículo de *Galois* y la Matriz de Incidencias

En dicho diagrama se refleja el proceso de guardar un retículo de Galois ya construido o de la matriz de incidencia de un contexto formal con el que estemos trabajando. Contempla los casos de uso **CU-04** y **CU-06**, en el que como ya se vio necesita que previamente se haya procesado un contexto formal, habiendo obtenido su matriz de incidencias, o la construcción de un retículo de *Galois* a partir de dicha matriz ya construida; de no ser así, no estará habilitada la opción de guardado ni para retículo ni para contexto.

La secuencia de dicho diagrama se puede observar en las **figuras 7-28** y **7-29** donde se contempla el proceso de guardar un contexto. Para el proceso de guardar un retículo es análogo, con los mismos mensajes con la salvedad de que los objetos son los respectivos para retículo. De forma que aunque no mostramos el diagrama de secuencia para guardar un retículo ya que como hemos dicho antes, los pasos son los mismos pero cambiando los objetos, se da una descripción detallada junto a la de guardar un contexto.

El proceso comienza cuando el actor selecciona en la ventana principal (objeto de la clase `FramePrincipalOpt`) la opción de guardar el contexto formal con el que esta trabajando. Acto seguido, el objeto ventana de clase `FramePrincipalOpt,` crea un objeto de clase `SetExportContexto` (`SetExportReticulo` para la opción de guardar retículo) que será el manejador de exportadores de contexto (manejador de retículos para el caso de guardar retículo).

Una vez hecho esto, la ventana recuperará el "escritor" para el archivo de datos que guardará la información relativa al contexto mediante el método `getExport(indice: int)`, pasando como argumento el índice donde estará la opción que identifique al exportador seleccionada por el actor; sino ha elegido ningún exportador, por defecto se utilizará el que guarda compatibilidad con [DBRE, 2005] (sería análogo para el caso de guardar un retículo de *Galois*). A continuación, `SetExportContexto` creará un objeto de clase `AbstractExportContextoWriter` (`AbstractExportRetículoWriter` para el caso de guardar un retículo de *Galois*) que será quien realmente escriba en el fichero de salida. Hay que aclarar que el objeto que se creará es de clase `DBREXMLWriter` o `Galicia3XMLWriter` que implementarán los métodos abstractos de `AbstractExportContextoWriter` (`AbstractExportRetículoWriter` para el caso de guardar retículo).

Sin embargo, a la hora de manipular dichos exportadores, utilizararemos un objeto `AbstractExportContextoWriter` (`AbstractExportReticuloWriter` para el caso de guardar retículo de *Galois*) a través de una conversión de tipos en la llamada de `getExport(indice: int)` convirtiendo el exportador elegido en dicha clase, ya que de esa manera obtenemos la genericidad suficiente como para poder incluir en un futuro nuevos exportadores sin que tenga la mayor repercusión en la aplicación.

Una vez hecho esto, obtendremos la descripción y la extensión de archivos reconocidos por el "escritor", a través del envío de los mensajes `getExtensionFicheroContexto()` (`getExtensionFicheroReticulo()` para el caso de guardar un retículo de *Galois*) y descripciones() a dicho "escritor". Con la extensión y la descripción del "escritor" construiremos un filtro para la ventana de elección de archivo donde se guardará el contexto (análogamente retículo).

El actor introducirá a continuación el nombre del archivo en el directorio escogido para tal menester, de forma que el objeto de clase `FramePrincipalOpt` confrontará dicho nombre con la intención de ver si contiene o no la extensión, añadiéndosela en caso de no haberla introducido. Si el archivo introducido por el actor ya existe, se le pedirá confirmación para sobrescribirlo. En caso de ser afirmativa la confirmación se procederá a ello y en caso de no ser así se cancelará la operación.

Una vez se tiene confirmación positiva, o confirmación de que el archivo no existe en el sistema procedemos a su escritura; para ello, si el contexto a sido construido desde cero se obtendrá del *parser* asociado a dicho contexto un objeto de clase `BinaryRelation` (reutilización de `dbre.fca` para salvaguardar la compatibilidad con [DBRE,2005]) y se mandará como argumento del mensaje `writeBinaryRelation(bin: BinaryReltion)` para el objeto de clase `AbstractExportContextoWriter` que será el que proceda a escribir la información acerca de la matriz de incidencias. Previamente habremos adjuntado a tal objeto el archivo destino elegido por el usuario a través del metodo `SetWriter(file: FileWriter)`. Si la matriz por el contrario ha sido cargada de un archivo, se obtendrá el objeto de clase `BinaryRelation` a partir de la matriz obtenida de dicho archivo.

Para el caso de guardar un retículo, pasara lo mismo que pasaba con el contexto en el tratamiento de exportadores. De forma, que aunque estemos manejando un objeto de clase `AbstractExportRetículoWriter`, realmente quien escribirá en el fichero de salida será un objeto `DBREXMLWriter` o `Galicia3XMLWriter` que implementarán los métodos abstractos de `AbstractExportRetículoWriter`. Sin embargo, a través de una conversión de tipos en la llamada de `getExport(indice: int)` se convierte el exportador elegido en un objeto de tipo `AbstractExportRetículoWriter`, ya que de esa manera obtenemos la genericidad suficiente como para poder incluir en un futuro nuevos exportadores sin que tenga la mayor repercusión en la aplicación.

Una vez establecido el exportador los pasos a proceder serán básicamente los mismos, con la distinción de que en vez de tener que obtener un objeto de tipo `BinaryRelation` (del paquete `dbre.fca` del proyecto [DBRE, 2005]), en este caso necesitaremos un objeto de tipo `ConceptLattice` (también del paquete `dbre.fca` del proyecto [DBRE, 2005]) que representará conceptualmente al retículo. Una vez obtenido dicho objeto de clase `ConceptLattice`, únicamente tendremos que comunicar al objeto "escritor" que escriba en el fichero que habremos adjuntado previamente con el mensaje `setWriter(file: FileWriter)`, a través del mensaje `writeConceptLattice(ret: ConceptLattice)`.

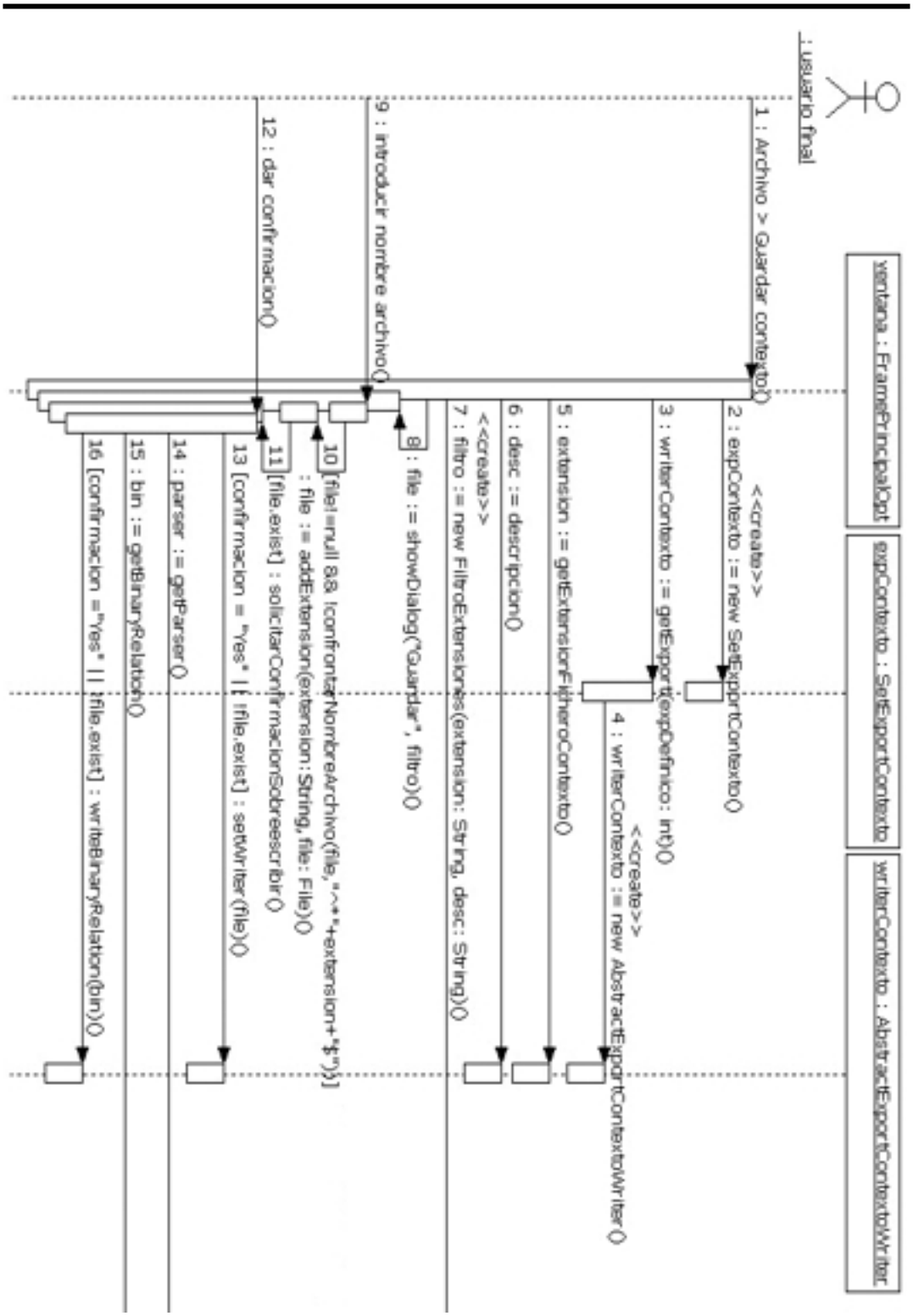

Figura 7-28 *Guardar un contexto formal (parte 1).*

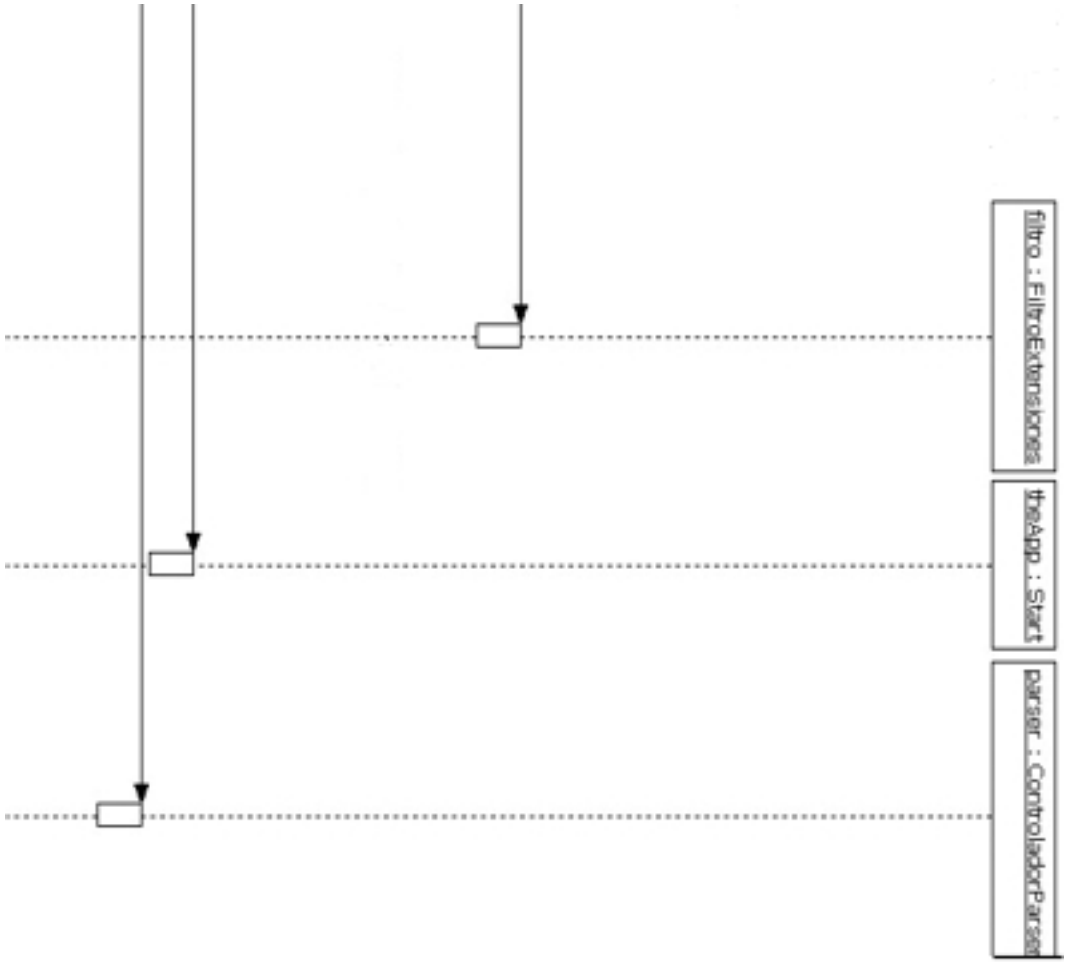

Figura 7-29 *Guardar contexto formal (parte 2).*

7.6.8 Arranque del Sistema de Línea de Comandos

En modo de línea de comandos la forma de proceder es la misma que en las descritas anteriormente, exceptuando que en este caso, la interacción del usuario en cuanto a seleccionar opciones solo se produce al principio al dar la orden en la línea de comandos, por lo tanto ya no aparece un modelo vista controlador como tal, sino una captura de los parámetros y ante los cuales se crea una lista de tareas que realizar. En dicha tareas ya no interviene el usuario, solamente para ver los resultados obtenidos, ya sean mensajes de error, o datos. Solo se exceptúa si el usuario opta por mostrar el retículo, en cuyo caso la aplicación terminará cuando se cierre la ventana. En consecuencia toda la intervención del usuario durante la ejecución de la aplicación viene dada por capturar un evento.

Los escenarios y los diagramas de secuencia que se muestran a continuación (**figura 7-30**) se refieren a situaciones ya referidas en los apartados anteriores, pero sin la intervención del usuario, y no se detallaran en profundidad.

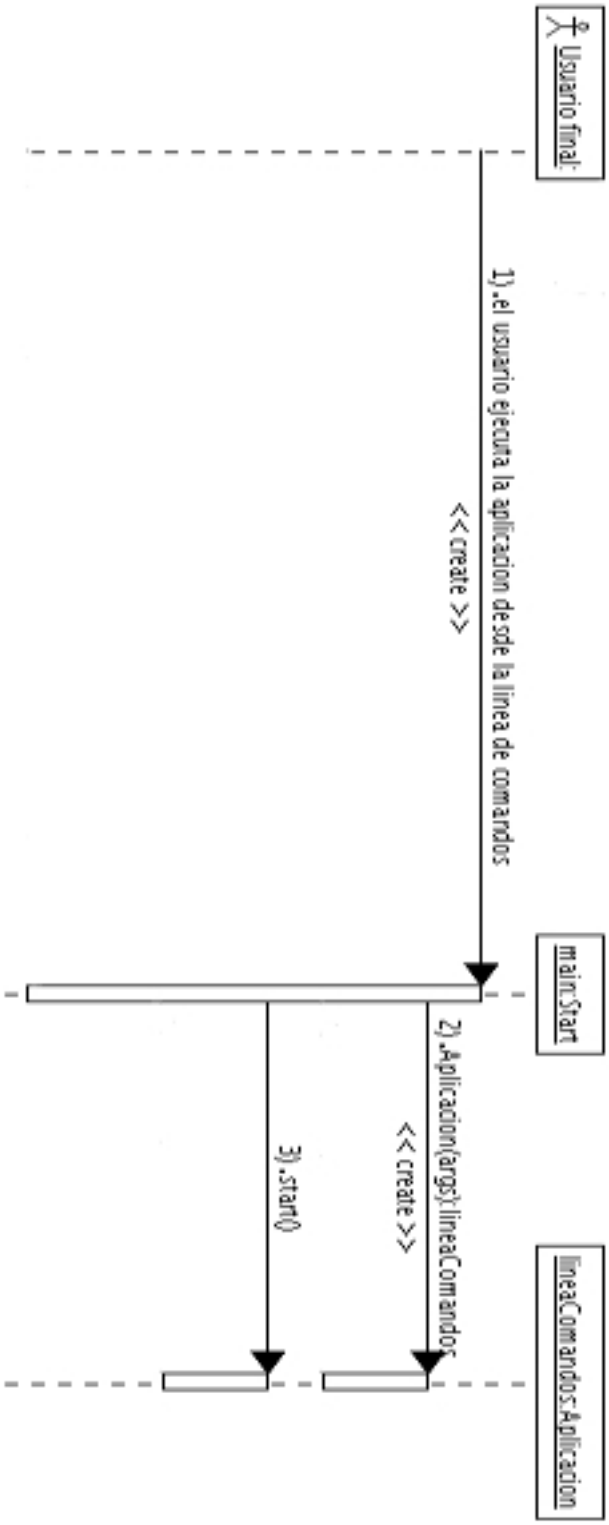

Figura 7-30 *Arrancar la aplicación en modo consola.*

7.6.9 Comienzo de la Ejecución.

Para poder ver con más detalles este diagrama remitimos al lector a la documentación que existe en el soporte CD, en la que se verá con más claridad el diagrama. Sin embargo hemos incluido el boceto con el fin de que sea orientativo en la descripción del mismo. Ver **figura 7-31**.

- En este diagrama se representa el comienzo del hilo principal de la aplicación en modo consola, son las inicializaciones y creaciones de objetos necesarios para comenzar a ejecutar tareas. Pasando a detallarse las siguientes actuaciones.

7.6.10 Procesamiento de los Parámetros de la Línea de Comandos.

En este diagrama se representa el procesamiento de las opciones y parámetros dados en la línea de comandos. Ver **figura 7-32 7-33**.

Este proceso esta formado por un bucle y una estructura de tipo "case", en la que cuando concuerda un parámetro se guardan los modificadores o datos de usuario en la un objeto de tipo `OpcionesLineaComandos`. El encargado de realizar estas operaciones es un objeto de tipo `ParserLineaComandos`.

Para saber como se detectan los errores en los parámetros de la línea de comandos ver el apartado anterior en el que se describe el subsistema *lineacomandos*.

7.6.11 Definición y Realización de las Tareas Necesarias.

En el siguiente diagrama de secuencia se representa la realización de las distintas tareas. Las tareas a realizar ya se conocen debido al procesamiento de las opciones de la línea de comandos. Ver **figuras 7-34 y 7-35.**

Los pasos principales ante una ejecución son:

- Generación del contexto a partir de ejecutar el *parser* mediante la clase `ControladorParser`.
- Exportar el contexto a un formato determinado.
- Generar el retículo de *Galois* a partir del contexto.
- Exportar el retículo.
- Mostrar el retículo.

Estos pasos ahora se representan de forma general, pasando a detallarse más profundamente en los siguientes apartados

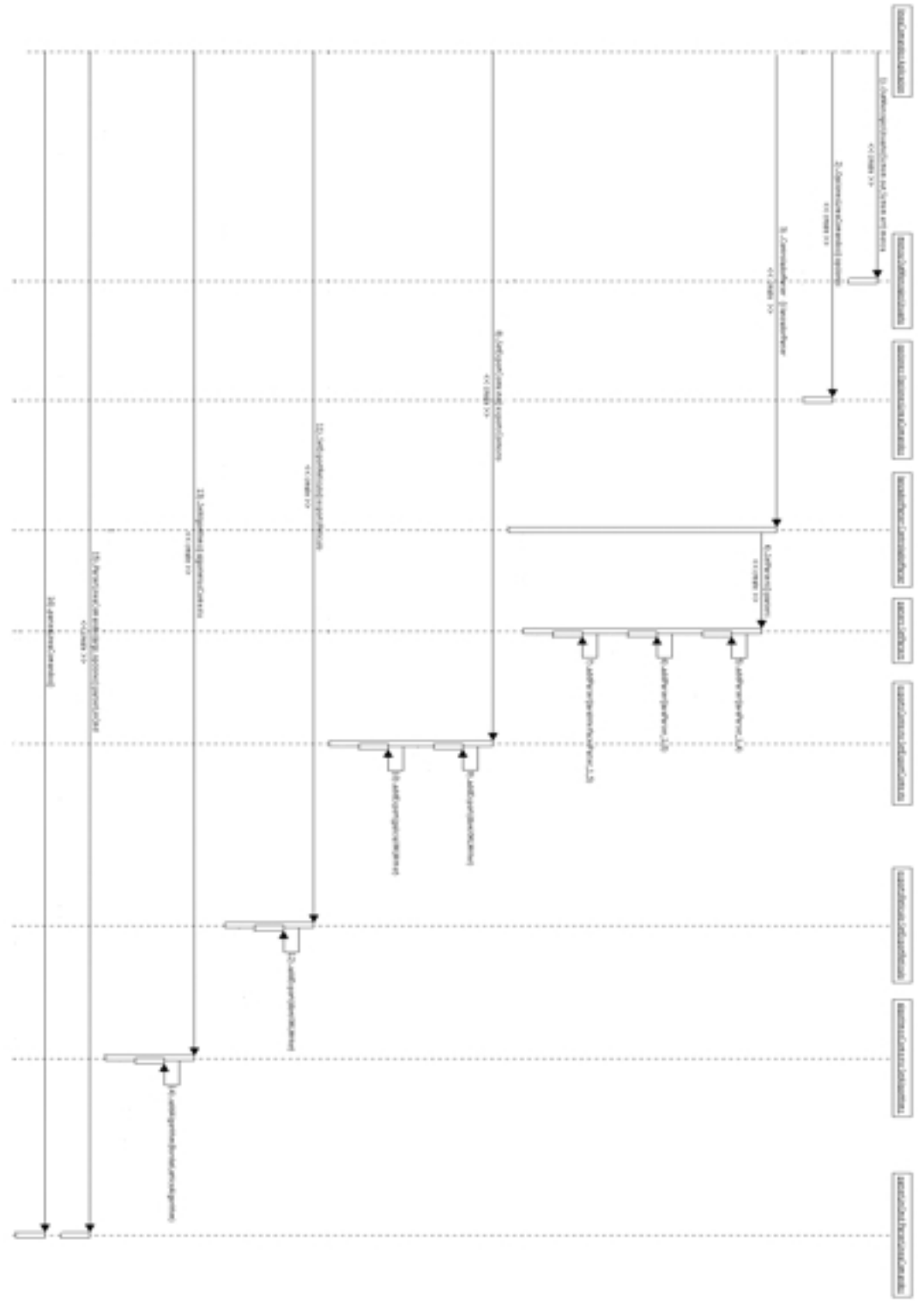

Figura 7-31 *Diagrama de secuencia de inicializar línea de comandos*

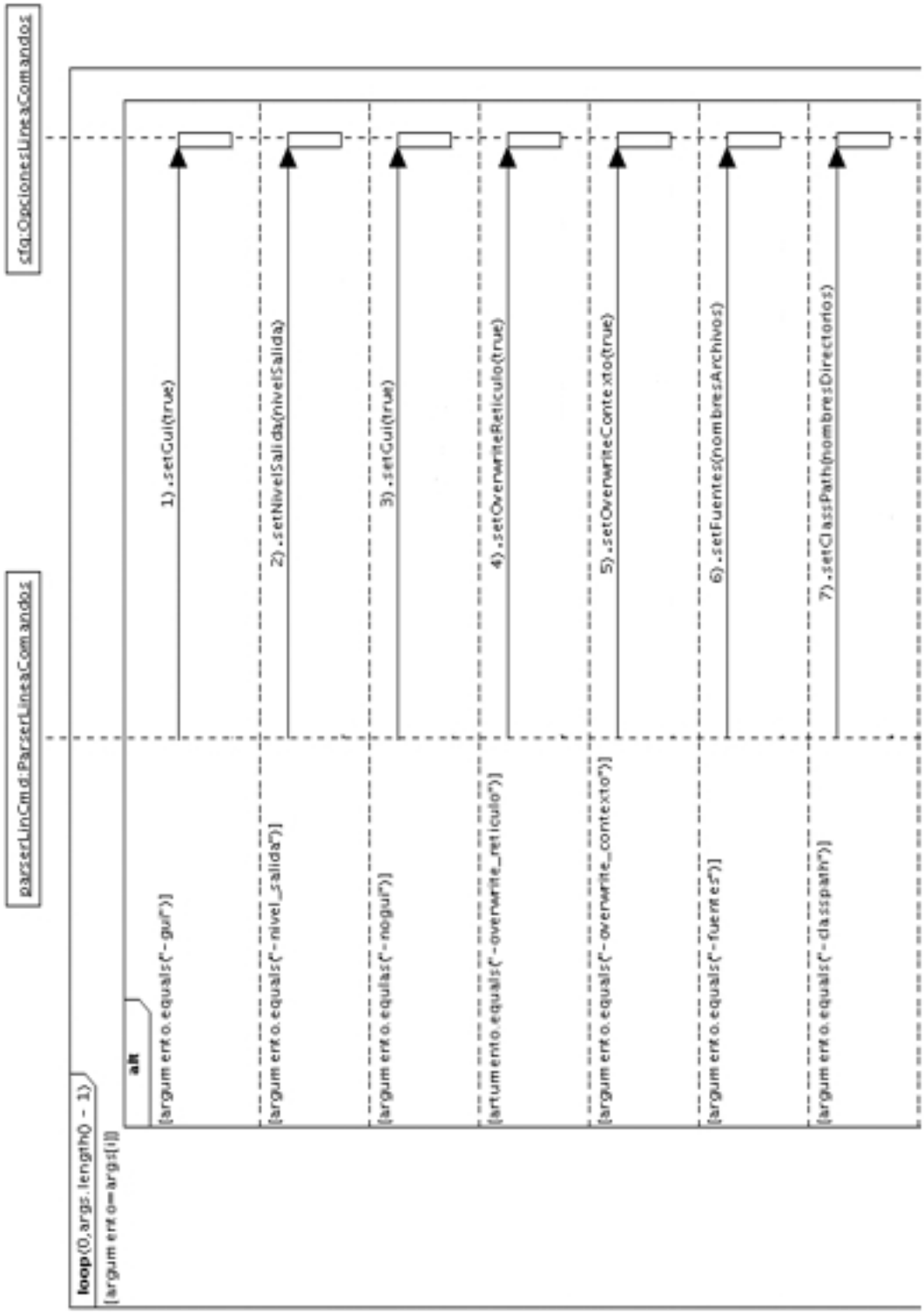

Figura 7-32 *"Parsear" línea de comandos (1 de 2).*

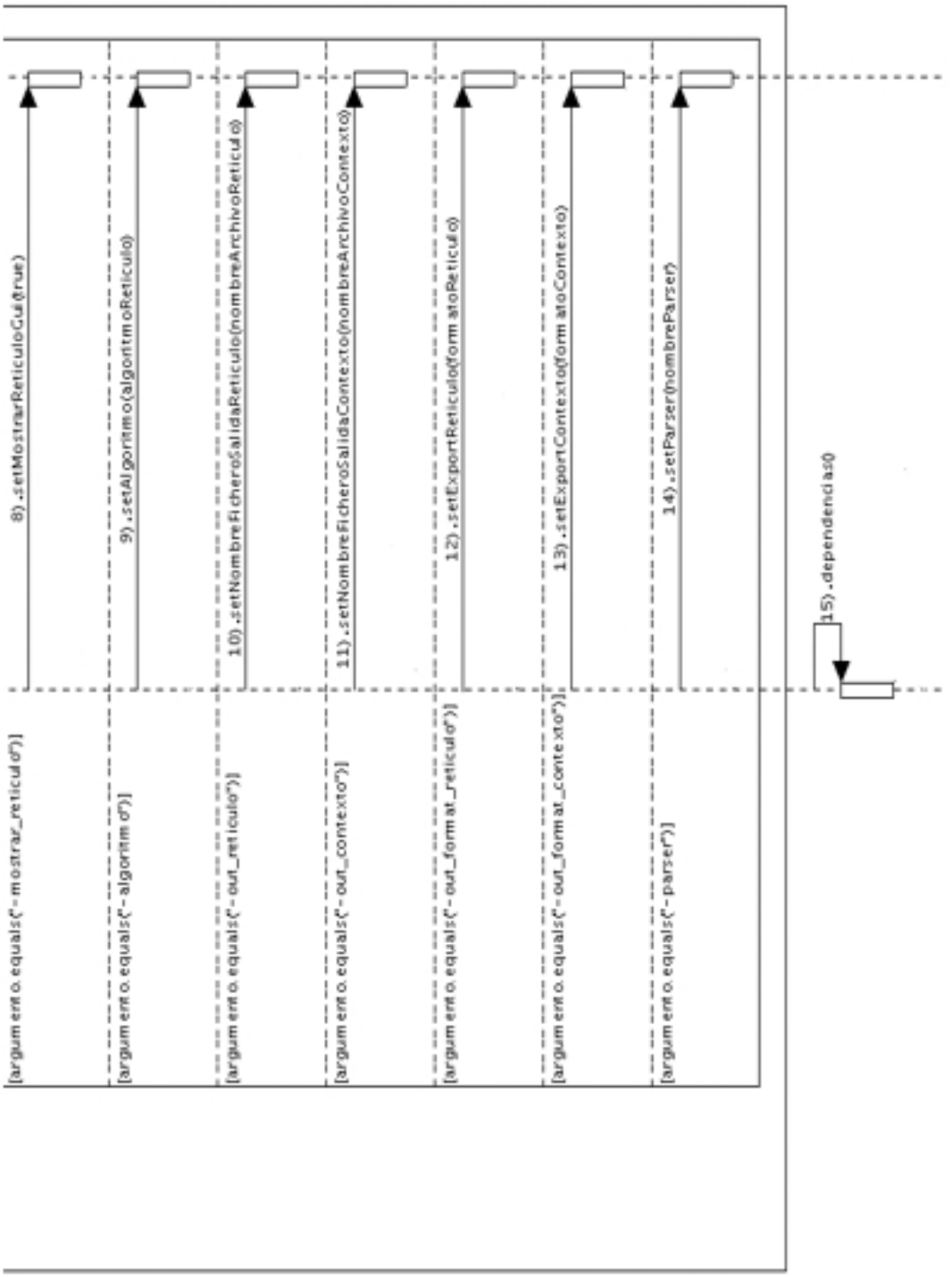

Figura 7-33 *"Parsear" línea de comandos (2 de 2)*

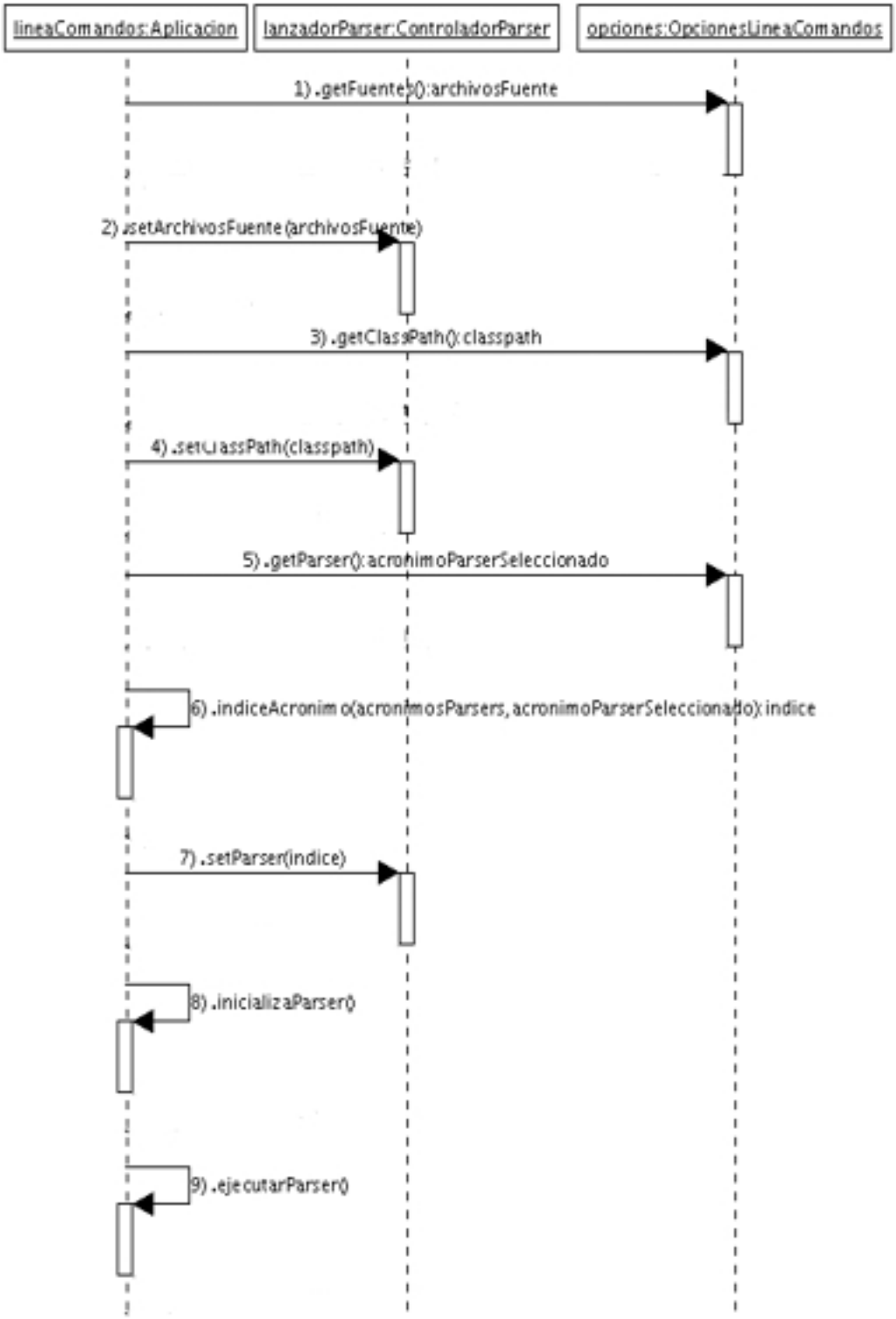

Figura 7-34 *Definir y ejecutar tareas (1 de 2)*

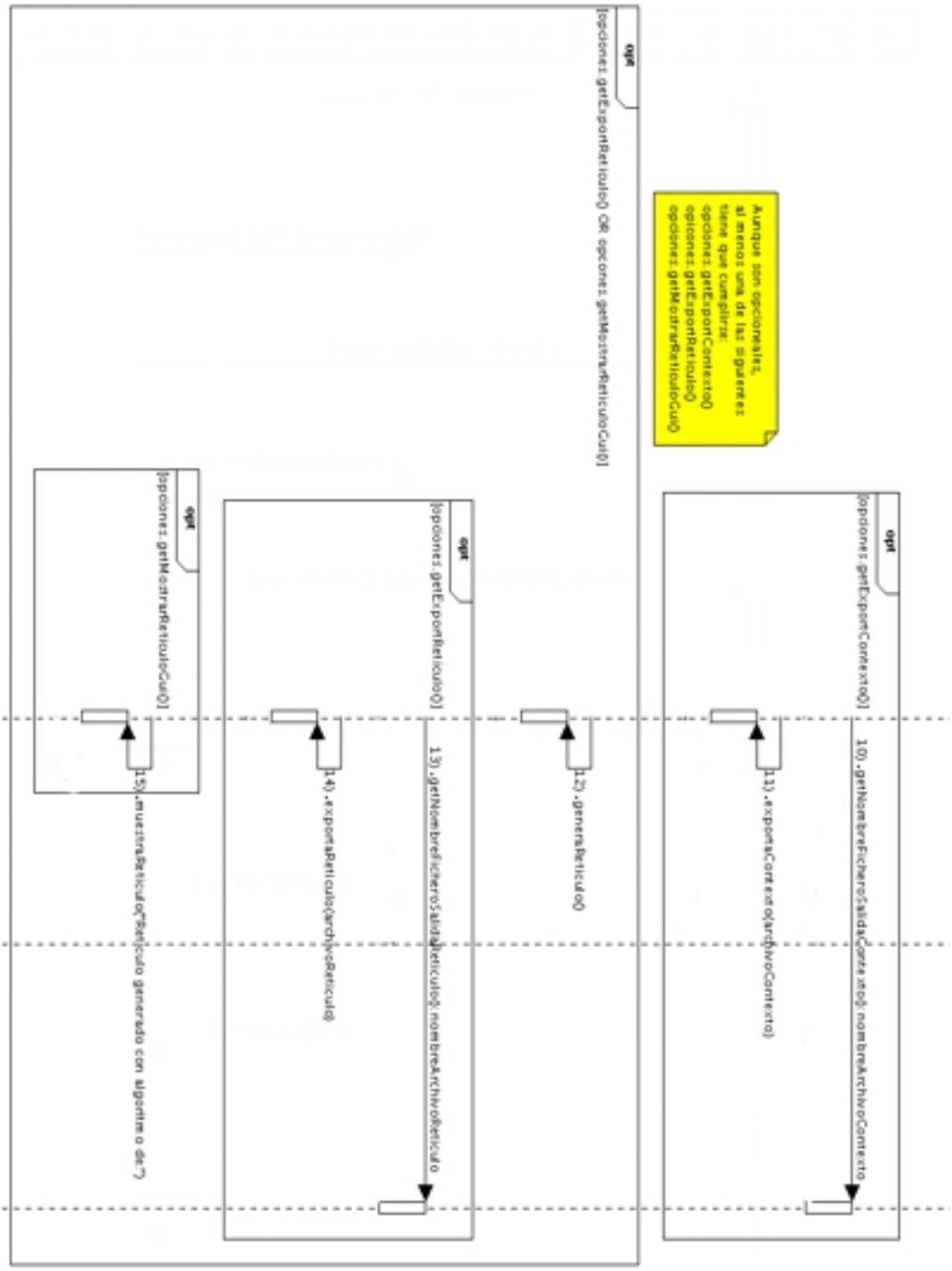

Figura 7-35 *Definir y ejecutar tareas (2 de 2)*

7.6.12 Inicializar Parser.

La inicialización del *parser* es una tarea previa a la ejecución del mismo, en la que se detectan los posibles errores de partida antes de comenzar a examinar código, así como la inicialización de los distintos valores, que hasta este momento no se tenían. Los posibles errores son del tipo de archivos o directorios no accesibles, *parsers* no compatibles con el código, etc. La inicialización de valores se refiere sobre todo a que `ControladorParser` comienza a enlazar y comunicar distintos objetos (de tipo `GestorParser`, `Matriz`, `AbstractParser`, `ArchivosFuente`)que son necesarios para la realización del *parser*, o lo que es lo mismo la obtención del contexto formal. Ver **figuras 7-36** y **7-37**.

7.6.13 Ejecutar *Parser*: Generar un Contexto Formal.

La obtención del contexto formal, es el proceso por el cual se "parsean" los archivos de código y se obtienen sus clases (o interfaces) y sus características. Estas formaran el contexto o lo que es lo mismo una relación binaria. Ver **figuras 7-38, 7-39, 7-40**.

En este diagrama se representa la manera en que el *parser* actúa ante las principales sentencias que se encuentra en el código, y como `GestorParser`, hace de puente entre `ArchivosFuente`, `AbstractParser` y `Matriz`, ante las distintas posibilidades que se dan al examinar el código, controlados todos por la clase `ControladorParser`.

7.6.14 Exportar un Contexto Formal.

Con el siguiente diagrama de secuencia se ve como se realiza la exportación de un contexto formal a un formato previamente obtenido desde la línea de comandos. Las descripciones básicas de las clases se presentaron en puntos anteriores (ver subsistema *io*). Ver **figuras 7-41 y7-42**.

7.6.15 Generar Retículo de *Galois* a partir de un Contexto Formal.

Aunque la aplicación solo genere retículos por medio del algoritmo de *Bordat*, se buscan los posibles algoritmos, y el usuario tiene que seleccionar uno. De esta manera sería factible poder añadir nuevos algoritmos a la aplicación.

En el siguiente diagrama se representa la manera de obtener el retículo a partir del contexto formal. Ver **figuras 7-43 y 7-44**.

7.6.16 Exportar un Retículo de *Galois*.

La manera de proceder para exportar el retículo es similar a la manera de exportar el contexto formal. Se puede ver en el siguiente diagrama. Ver **figura 7-45**.

7.6.17 Visualizar un Retículo de *Galois*.

Para mostrar el retículo de *Galois* se utiliza la clase `LatticeGraphFrame` perteneciente al proyecto [DBRE, 2005]. El único añadido ha sido un controlador nuevo (`AdaptadorLattice`), que captura el evento de cerrar la ventana y termina la aplicación.

En el siguiente diagrama de secuencia se muestra dicho proceso. Ver **figura 7-46**.

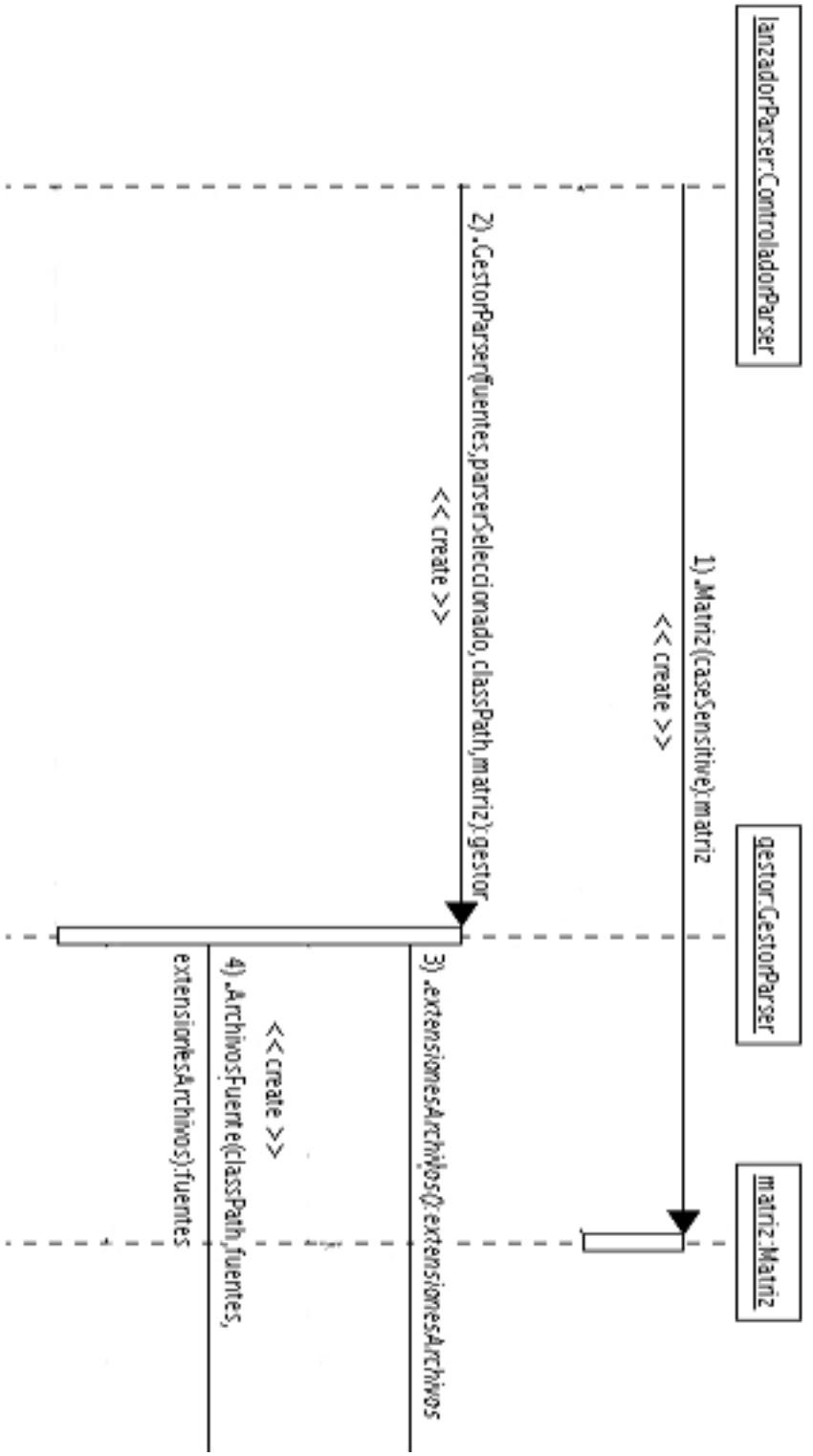

Figura 7-36 *Inicializar parser (1 de 2)*

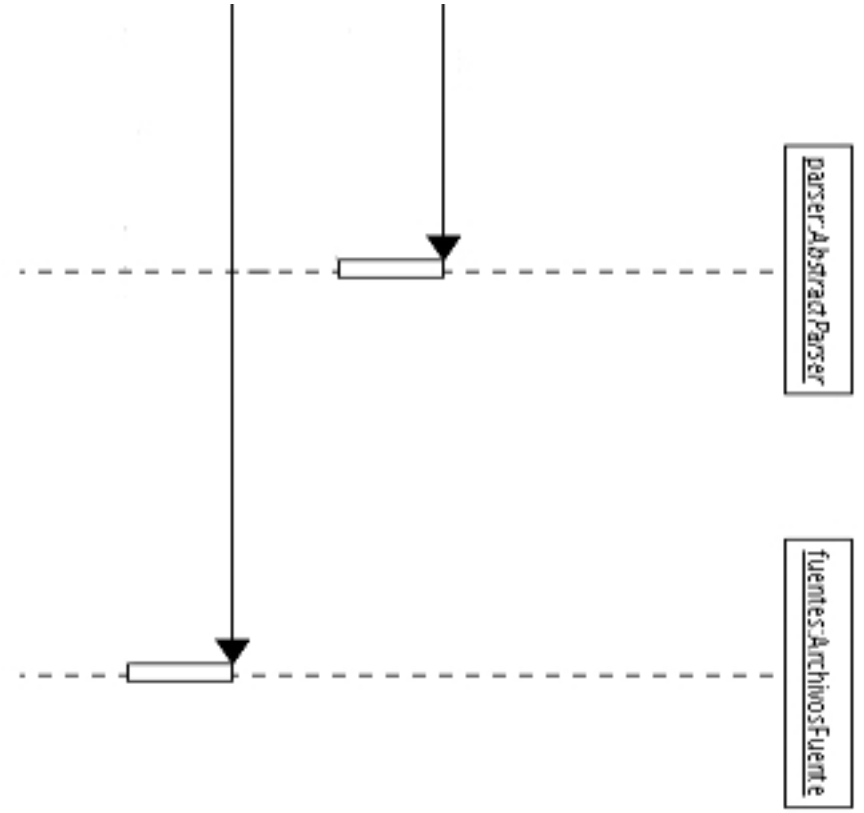

Figura 7-37 *Inicializar parser (2 de 2)*

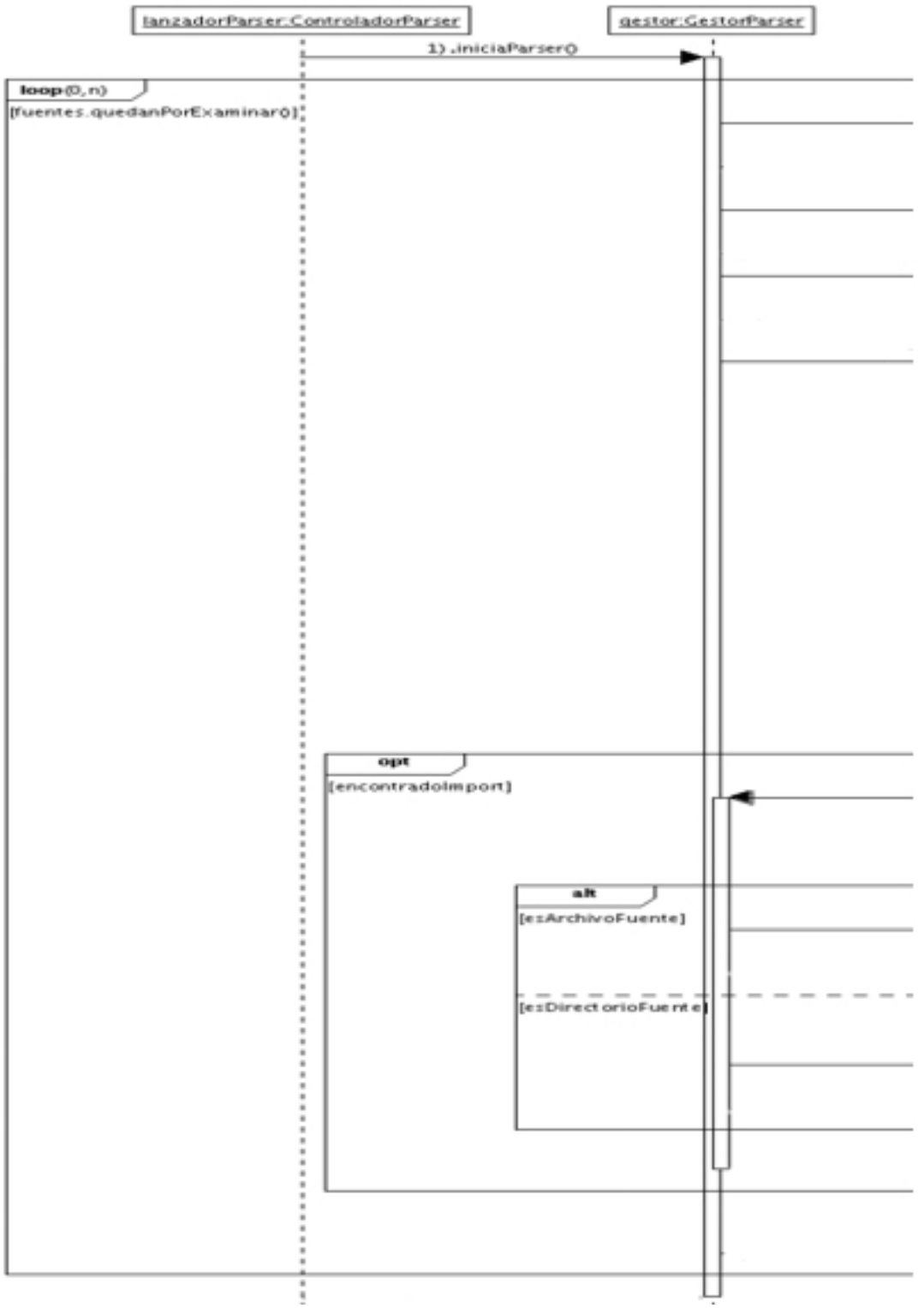

Figura 7-38 *Ejecutar parser (1 de 3).*

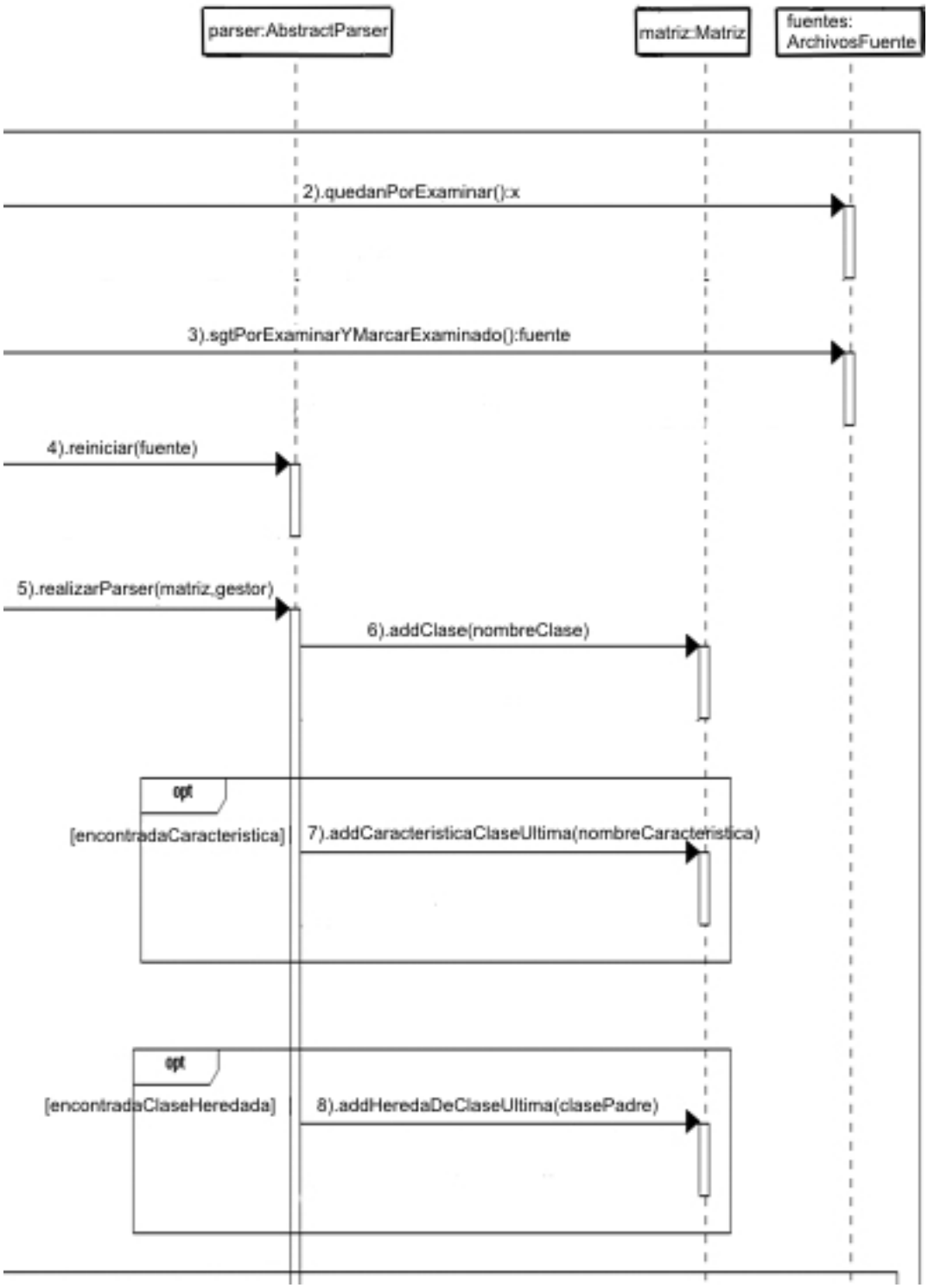

Figura 7-39 *Ejecutar parser (2 de 3).*

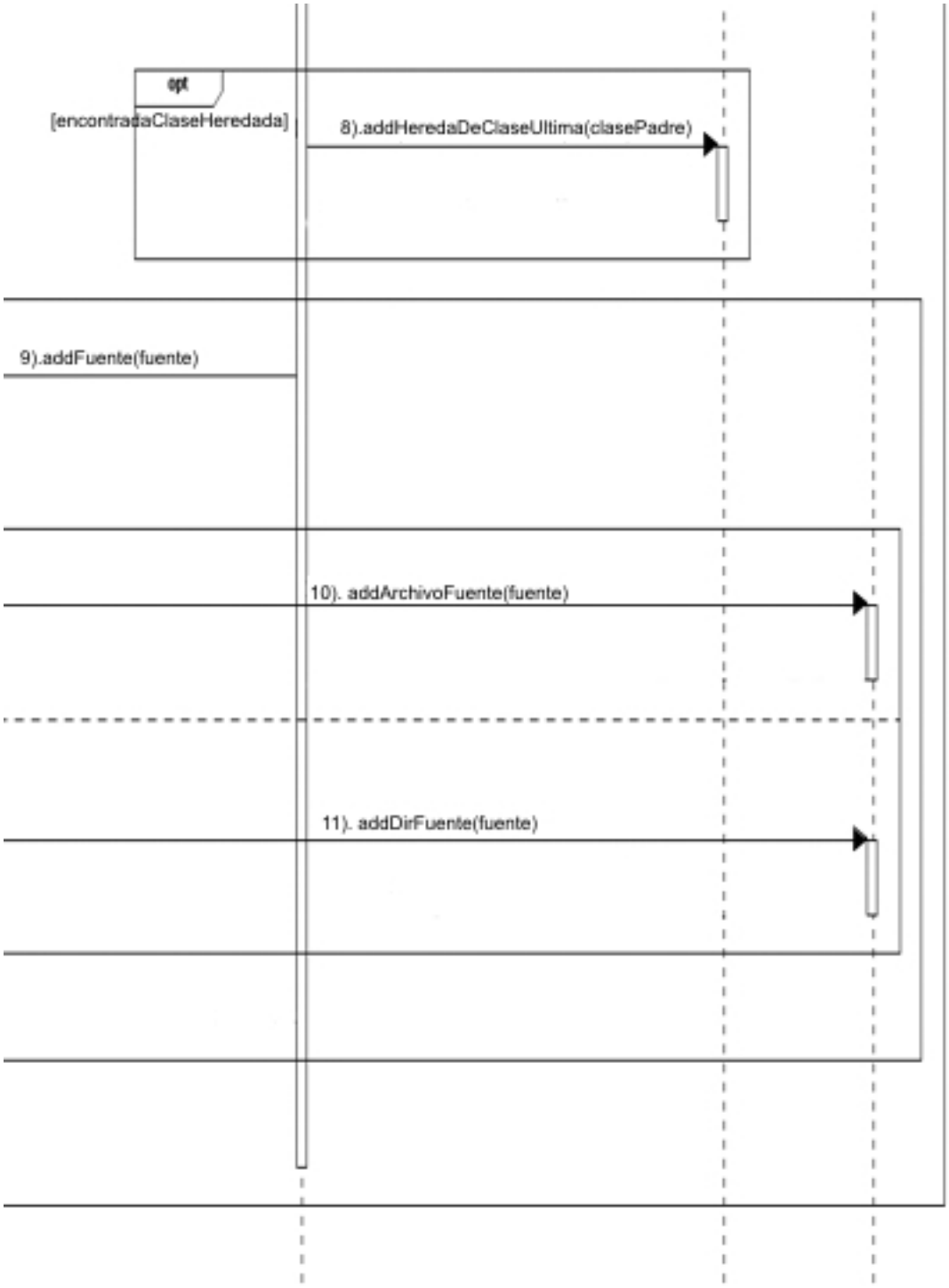

Figura 7-40 *Ejecutar parser (3 de 3).*

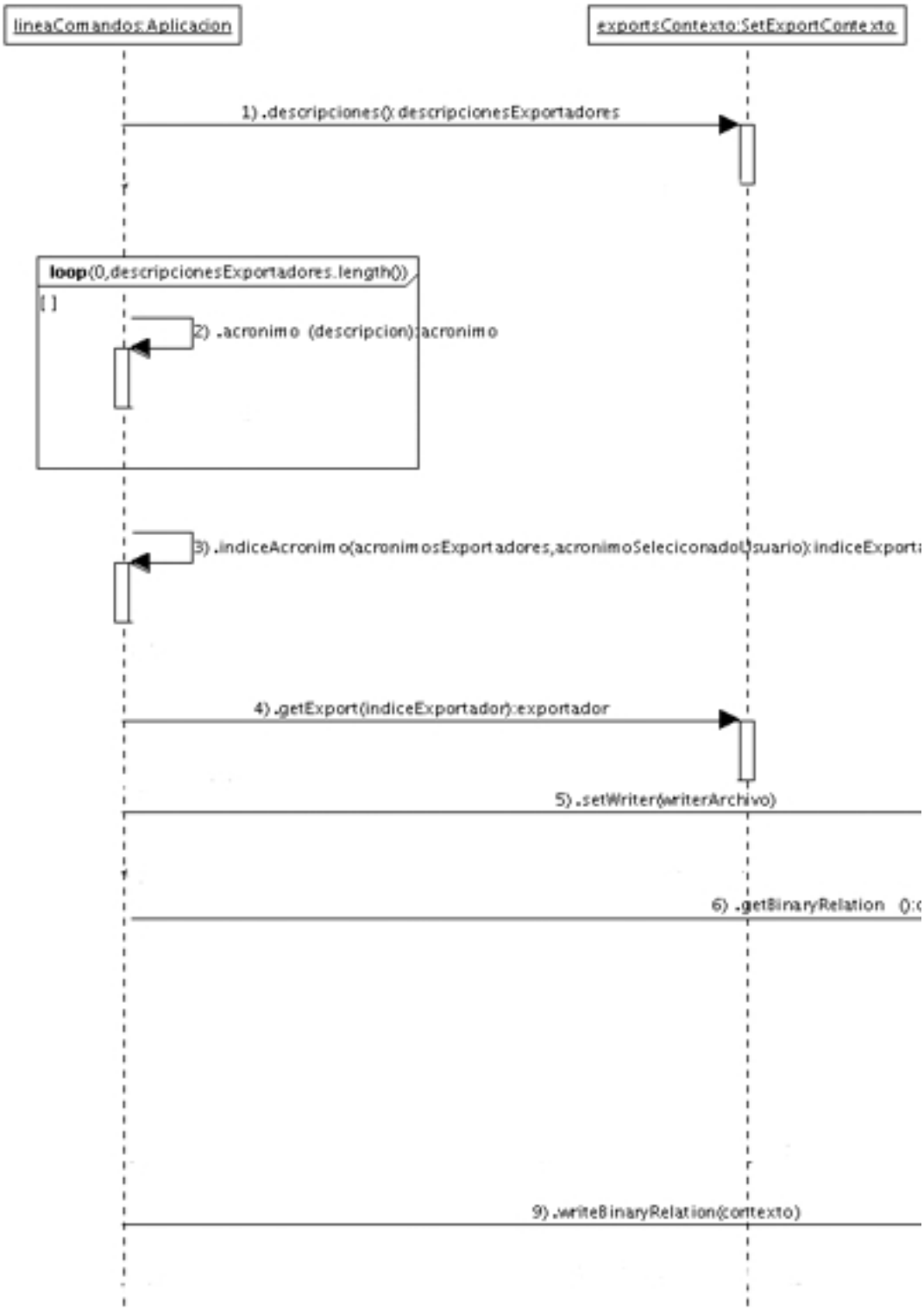

Figura 7-41 *Exportar contexto formal (1 de 2)*

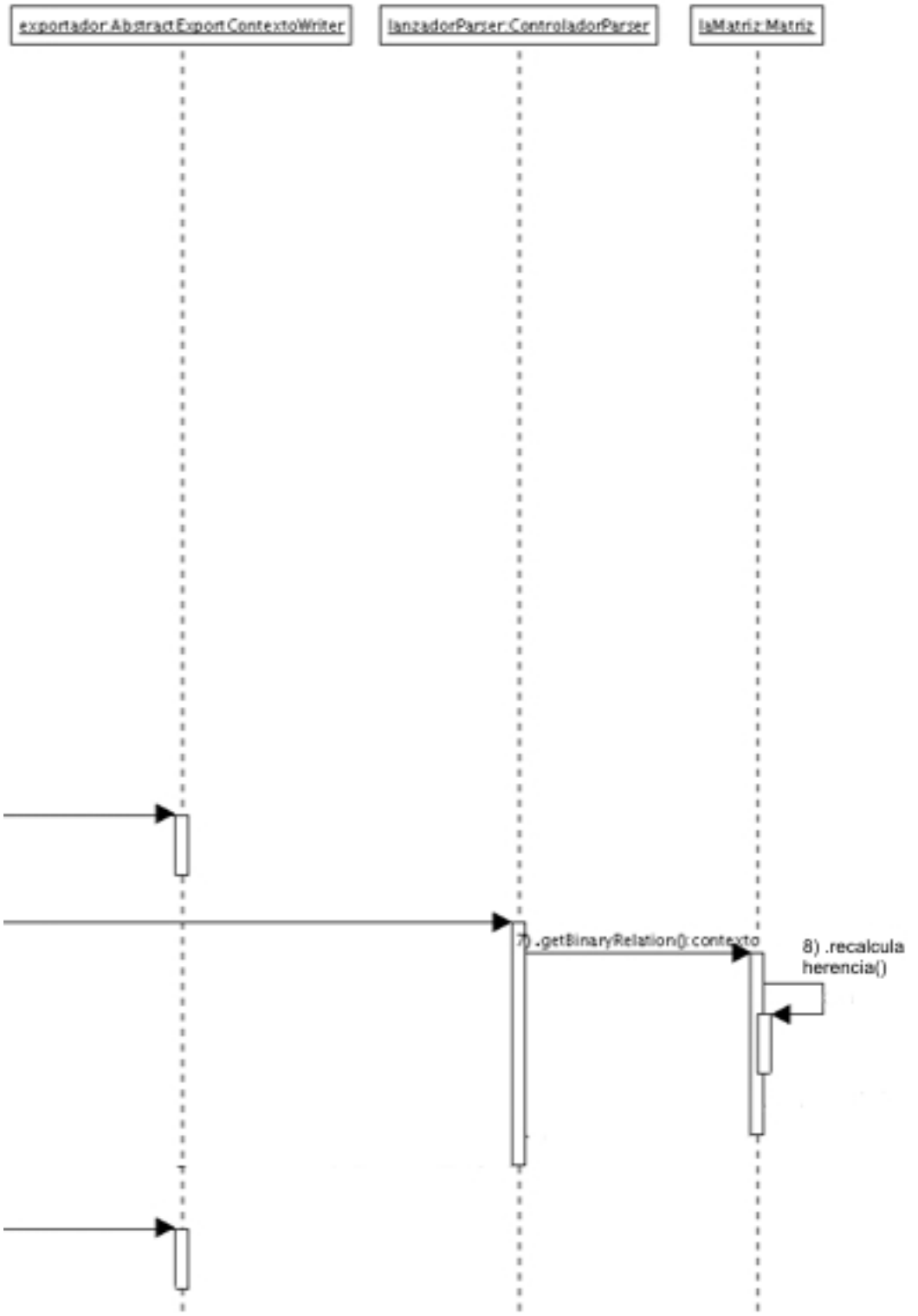

Figura 7-42 *Exportar contexto formal (2 de 2)*

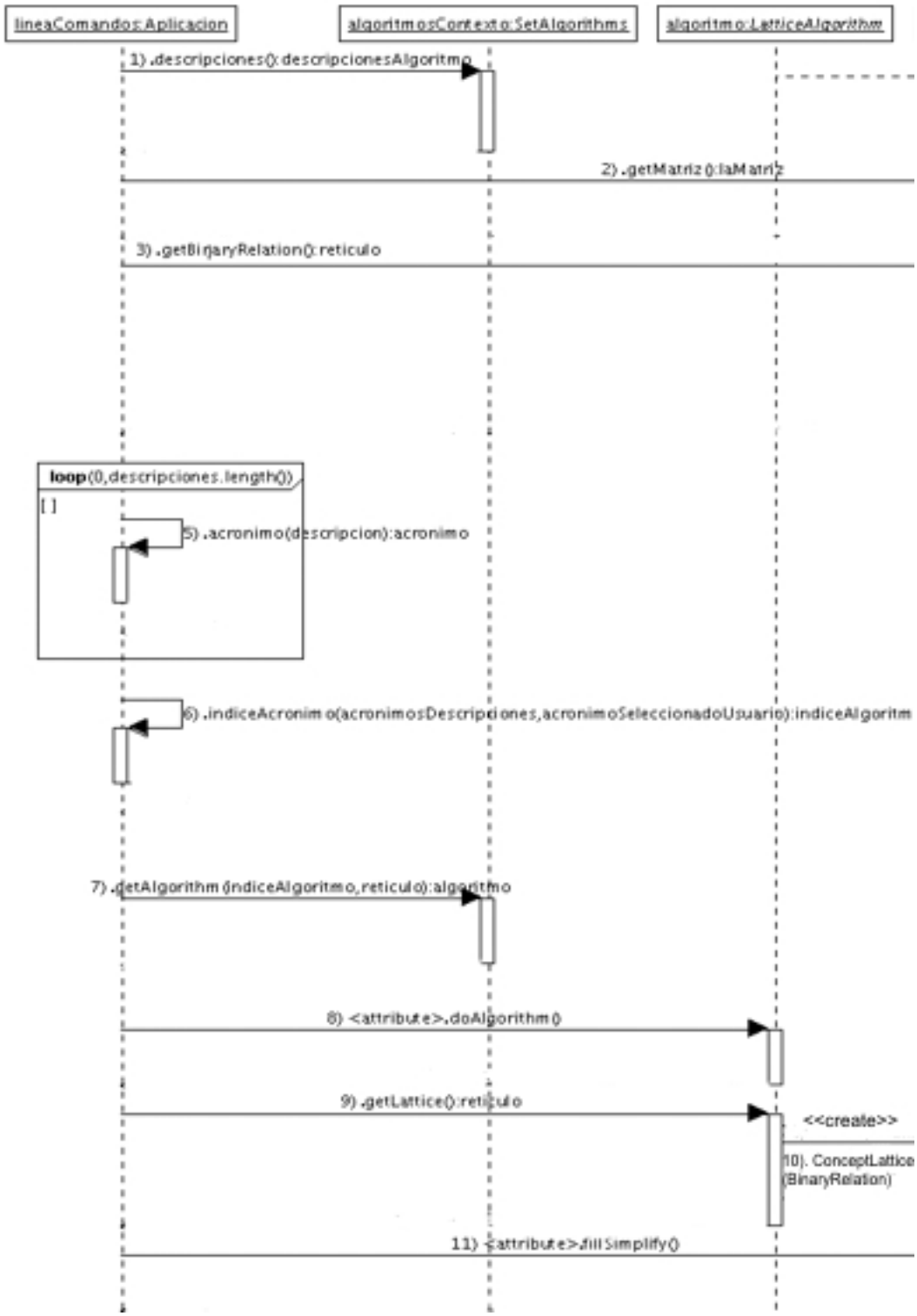

Figura 7-43 *Generar retículo de Galois (1 de 2)*

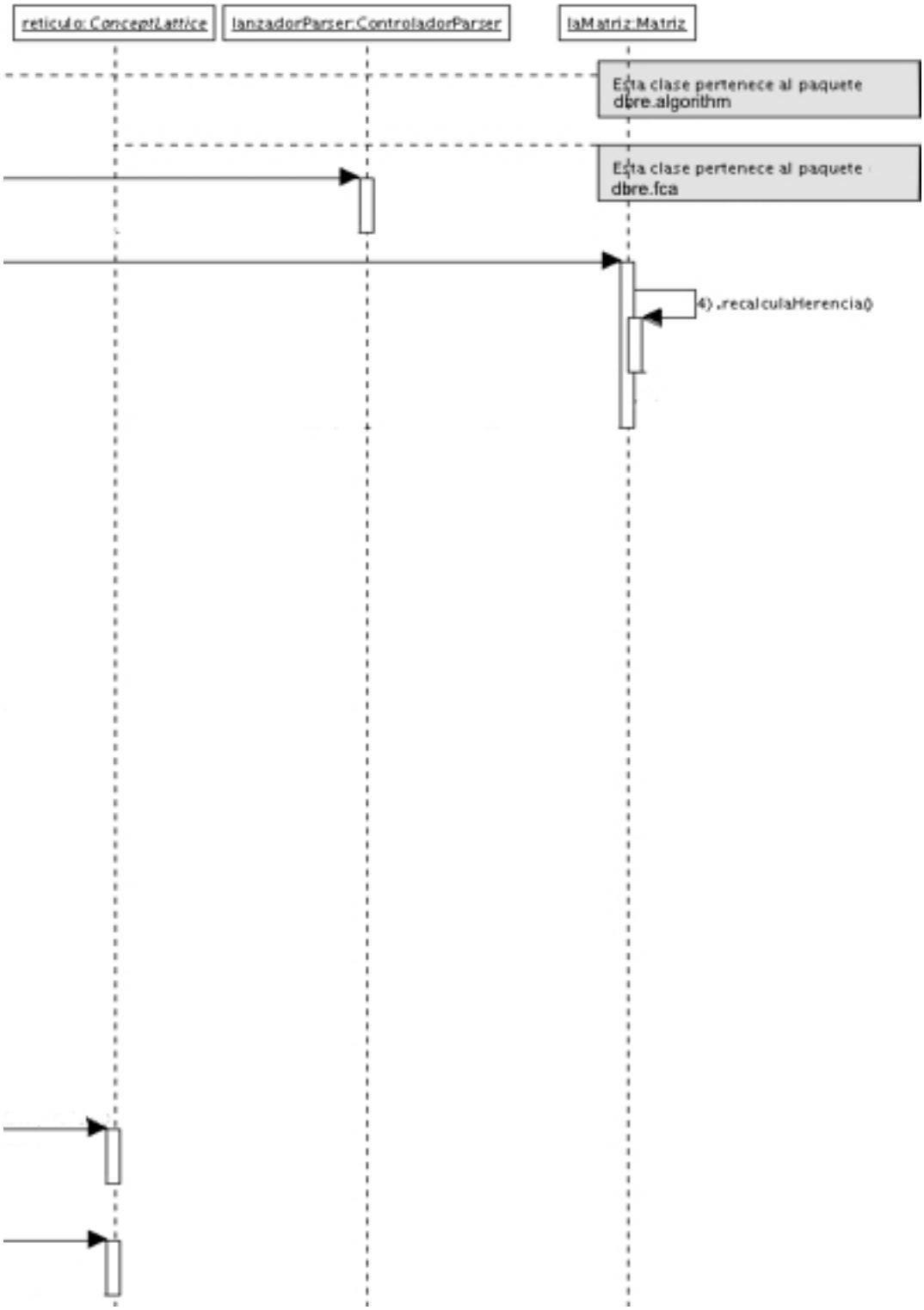

Figura 7-44 *Generar retículo de Galois (2 de 2)*

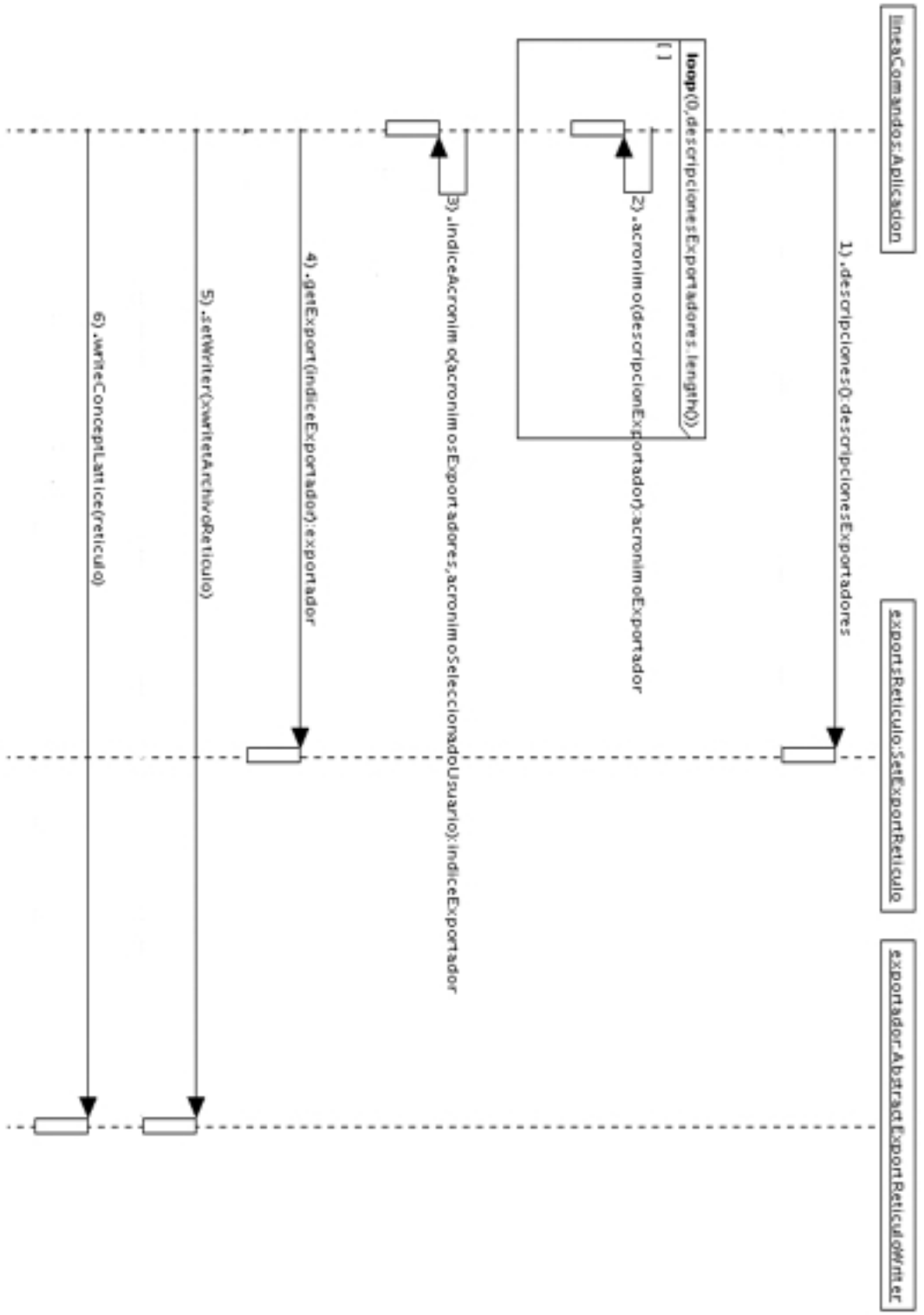

Figura 7-45 *Exportar retículo de Galois.*

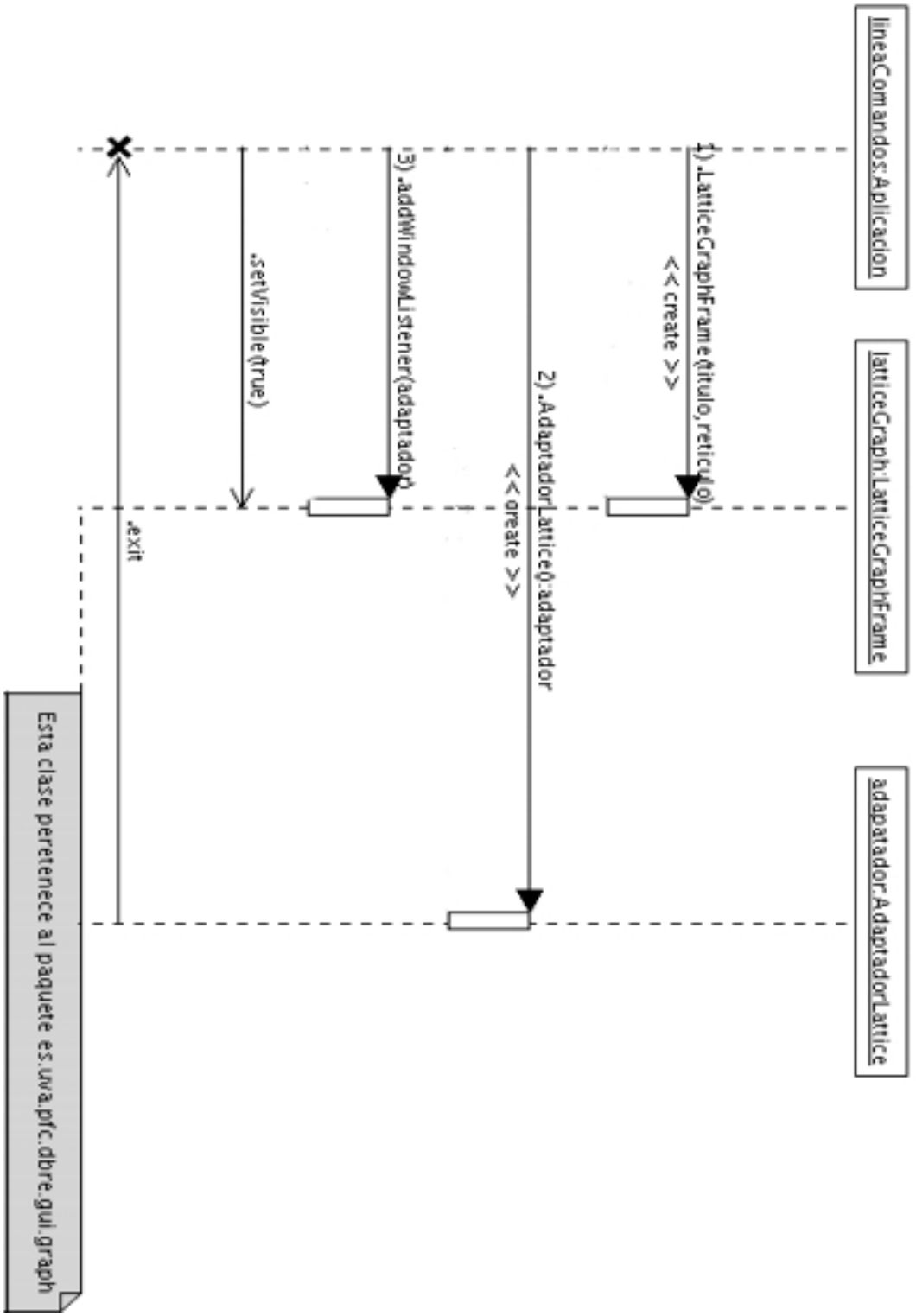

Figura 7-46 *Mostrar retículo de Galois.*

7.7 Conclusiones

Junto con el análisis efectuado, se han descrito los Diagramas de Clases para cada subsistema y los Diagramas de Secuencia más importantes dentro del diseño de nuestro sistema. Todos estos diagramas reflejan la realización de una colaboración, que a su vez cubren los Casos de Uso descritos en el análisis de requisitos del sistema. Siguiendo las ideas expuestas por UML, no se han introducido todos los posibles diagramas sino los más importantes puesto que la explosión combinatoria a partir de los Casos de Uso expuesta nos llevaría a un número de Diagramas de Secuencia excesivo tanto en número como en tamaño de los diagramas.

La idea principal es mostrar el comportamiento del sistema, y su reflejo en comportamientos cooperativos por parte de las clases de diseño que posteriormente serán implementadas. A través del estudio de dicho comportamiento, el lector debe adoptar una idea general del comportamiento esperado del sistema y de la solución adoptada para llevarlo a cabo.

7.8 Pruebas

En los subsiguientes apartados se incluye la información correspondiente a los diferentes tipos de pruebas que se deben realizar lo largo del desarrollo del presente proyecto. No se expondrán las listas concretas de las diferentes pruebas que se realicen puesto que es una información que no tiene cabida dentro del presente documento, que trata de reflejar una idea global hacia el lector de los objetivos, funcionalidad y del desarrollo efectuado para la construcción del producto final.

Para la realización de las pruebas del sistema, deben tenerse en cuenta en cuenta tres niveles de pruebas: Pruebas de unidad, Pruebas de integración, y Pruebas de sistema.

7.8.1 Pruebas de unidad

Al ser una aplicación desarrollada en el contexto orientado a objetos se deben realizar las pruebas de clases entendiendo cada clase como la unidad mínima de prueba. A diferencia de la prueba de unidad del software convencional, el cual tiende a centrarse en el detalle algorítmico de un módulo y los datos que fluyen a lo largo de la interfaz de este, la prueba de clases está dirigido por las operaciones encapsuladas por la clase y el estado del comportamiento de la clase.

Las pruebas de clase se realizarán comprobando para cada clase de los diferentes subsistemas desarrollados, que los métodos son accesibles y que realizan la funcionalidad para lo que fueron diseñados.

7.8.2 Pruebas de integración

La prueba de integración es aquella que se realiza para identificar cualquier fallo cuando dos o más elementos desarrollados independientemente se combinan para ejecutar sus funcionalidades. Entre los fallos típicos se tienen los debidos a defectos internos y externos en las interfaces, fallos de sincronización y de rendimiento.

Las pruebas de integración se realizarán para probar la comunicación entre los diferentes subsistemas una vez haya acabado su desarrollo.

7.8.3 Pruebas de sistema

Las pruebas se realizarán como pruebas de caja negra. El objetivo planteado consiste en comprobar la funcionalidad establecida para cada una de las opciones viables que proporciona la aplicación, sin considerar la estructura interna de cada una de ellas. Por su naturaleza, son similares en cualquier tipo de software, pues se prueban elementos visibles al usuario final, donde no importa tanto la forma de implementación.

.

Para realizar dichas pruebas se comprobarán los requisitos funcionales y globales planteados en el documento de análisis.

7.9 Interfaz de Usuario

7.9.1 Interfaz gráfica de usuario

El objetivo final de cara al usuario es facilitar la construcción de contextos formales con variados objetivos de extracción de conocimiento, aunque en nuestro caso nos centramos en la extracción de conocimiento sobre las relaciones de herencia en una colección de clases o interfaces. Por este motivo la interfaz gráfica se centra en representar de una forma intuitiva un contexto formal a través de su matriz de incidencia como una tabla, donde es posible visualizar con un simple golpe de vista cualquier relación entre objetos y atributos, resaltando dicha relación con un color claramente distinguible de las celdas en las que no existe relación alguna entre objetos y atributos.

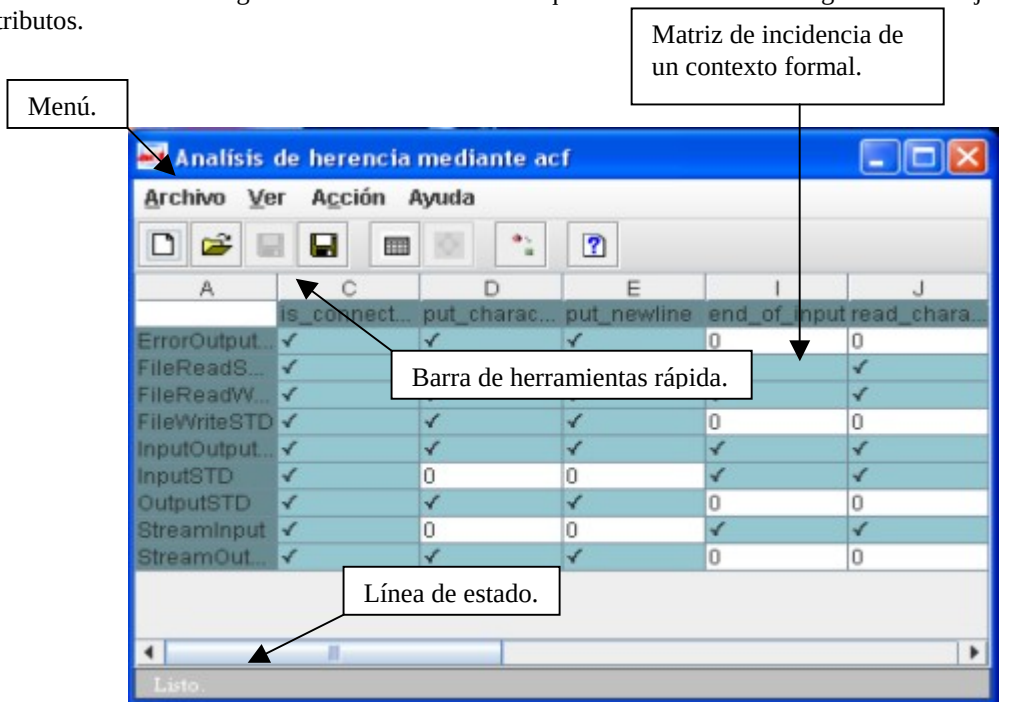

Figura 7-47 *Ventana de presentación de contextos formales.*

Los menús contendrán las siguientes opciones:

- **Archivo**
    - *Nuevo*: generará un nuevo contexto formal a partir del análisis de un conjunto de clases o interfaces. Será necesario indicarle donde están localizadas dichas clases y elegir un parser adecuado para el análisis léxico-sintáctico del código objeto de estudio
    - *Abrir*: abrirá cualquier fichero con uno de los formatos de entrada con los que es compatible la aplicación. Pudiendo ser el contenido del archivo tanto un conjunto de contextos (pudiendo ser uno) como de retículos.
    - *Guardar retículo*: guardará el retículo calculado a partir del contexto con el que estemos trabajando en un formato XML elegido por el usuario (DBRE).
    - *Guardar contexto*: almacenará el contexto formal con el que estemos trabajando en ese momento en un archivo de datos XML con un formato elegido (DBRE o Galicia). Si no se ha elegido ningún formato, por defecto lo guardará conservando la compatibilidad con DBRE.
    - *Modificar parser*: permitirá modificar las opciones del parser elegido para el análisis de jerarquías de herencia de un nuevo conjunto de clases o interfaces introducidas por el usuario; por tanto la matriz de contexto formal obtenida podrá ser modificada a través del análisis de las nuevas clases o interfaces junto con el reanálisis de las antiguas.
    - *Salir*: abandonar el programa.
- **Ver**
    - *Matriz de contexto*: muestra la matriz de contexto, aunque ya se dibuja por si misma cuando analizamos un conjunto de clases o interfaces, al seleccionar dicha opción se actualiza volviéndose a dibujar.
    - *Retículo de Galois:* visualiza el retículo de *Galois* construido. Aunque una vez seleccionada la opción de construir retículo ya se visualiza en una ventana aparte, si cerramos la ventana, a través de esta opción podemos volverla a ver.
    - *Opciones:* permite al usuario elegir el formato de exportación de los archivos, pudiendo elegir entre DBRE o Galicia para la exportación de la matriz de contexto formal o DBRE para la exportación del retículo.
- **Acción**
    - *Construir Retículo ->Algoritmo de Bordat*: una vez obtenida la matriz de contexto, seleccionando esta opción construiremos el retículo de *Galois*

a través del algoritmo de *Bordat*. Si se añadieran nuevos algoritmos, aparecerían en este menú.

- **Ayuda**
    - Contenidos: muestra los contenidos de ayuda sobre la aplicación.
    - Acerca de: muestra la información acerca de los autores de la herramienta así como del título del proyecto, la universidad y el año.

Sobre estos contextos se aplicarán diversos algoritmos cuyos resultados serán representados en ventanas gráficas adicionales a la ventana de edición de contextos.

El objetivo básico de desarrollar inicialmente una interfaz gráfica es dar una idea al cliente de la funcionalidad y manejo de la misma que se va a obtener con el producto.

7.9.2 Línea de comandos

El sistema, además de proporcionar un interfaz de usuario gráfico, ofrece la posibilidad de interactuar con la herramienta a través de la línea de comandos, pudiendo realizar la mayoría de las acciones que se facilitan en modo gráfico. Sin embargo debido a las limitaciones que plantea un entorno de consola, algunas funcionalidades que aporta el trabajar en un entorno gráfico no serán contempladas, como se puso de manifiesto en el análisis de requisitos. Tales como la apertura de un archivo de datos donde este almacenado un retículo. Ya que carece de sentido cargar un retículo ya construido si no se está trabajando en un entorno de ventanas. En definitiva, las principales funciones si se contemplaran, construir una matriz de contexto a partir del análisis léxico-sintáctico de un conjunto de clases o interfaces objeto de estudio, permitir la construcción del retículo *Galois* únicamente si se trabaja en la consola de un entorno de ventanas, así como la exportación tanto del retículo como de la matriz de contexto en un archivo de datos XML con el objetivo de posteriormente permitir trabajar con dichos archivos ya sea en la misma herramienta pero ya en modo gráfico o en futuras aplicaciones que tomen de flujo de entrada dichos archivos.

```
C:\Documents and Settings\Alberto\Escritorio>java -jar hacf-pfc-1.0-bin.jar -nog
ui
Error en parametros:
No se ha dado ningun nombre de archivo donde comenzar el parser
Sintaxis:
[-nogui
        -nivel_salida nivel
        -fuentes archivoCodigo1 [archivoCodigo2 [...] ]
        -classpath directorio1 [directorio2 [...] ]
        -parser parserAUsar
        [(-out_contexto nombreArchivoSalida
                -out_format_contexto formatoSalida) ]
        [-overwrite_contexto ]
        [-algoritmo algoritmoGeneradorReticulo
                [-overwrite_reticulo ]
                [-mostrar_reticulo ]
                [(-out_reticulo nombreArchivoSalida
                        -out_format_reticulo formatoSalida)]
        ]
]
NOTA: con el modificador -nogui, hay que indicar al menos una de las siguientes:
 (-mostrar_reticulo, -out_reticulo, -out_contexto)

Donde los posibles parsers son (Nombre / descripcion):
java.incompleto.1.4             java (incompleto) 1.4
java.clases.1.5         java (clases) 1.5
java.interfaces.1.5             java (interfaces) 1.5

Los formatos de archivos de salida para el contexto son:
Galicia3XML             Galicia 3 XML
DBREXML         DBRE XML

Los algoritmo de generacion de reticulos son:
AlgoritmodeBordat               Algoritmo de Bordat

Los formatos de archivos de salida para el reticulo son:
DBREXML         DBRE XML

Los posibles niveles de salida son (Numero / Descripcion):
0               Mensajes de error
1               Mensajes esenciales
2               Mensajes informativos
3               Mensajes de depuracion
Un nivel de salida implica que tambien se mostraran los niveles inferiores
```

Figura 7-48 *Sintaxis y opciones del modo línea de comandos de la aplicación.*

En dicho modo de interactuar con al aplicación están contempladas todas las opciones que se contemplan en el *GUI*, conforme a la elección de algoritmo, *parser*, exportadores... De forma que se conserve la genericidad de diseño para las futuras ampliaciones. Cargándose dinámicamente cualquier nueva incorporación, ya sea de exportador de contexto, de retículo, *parser* o algoritmo de construcción de retículo.

En el modo de actuar el usuario con la herramienta a través del modo consola, se opto por una política parecida a la gran mayoría de programas que permiten su actuación en línea de comandos. Esto es, permitir una acción sobre el programa a través del uso de parámetros en la línea de ejecución, de forma que según los argumentos o parámetros introducidos hará una cosa u otra. Como ejemplo diremos que si el usuario introduce el argumento `–nogui` inmediatamente después del nombre del paquete (`hacf-pfc-1.0-bin.jar`) le estará indicando a la aplicación que se va a operar en modo consola.

La elección de dicha decisión fue debida a que un usuario acostumbrado a utilizar programas en modo consola, estará más habituado a este formato que a uno distinto que pudiéramos haber introducido. Facilitando de esa la comprensión y utilización en la utilización de los diferentes parámetros que se pueden emplear, así como poder realizar un uso de la misma de una manera intuitiva.

8 Documentación de Usuario

8.1 Introducción

El presente documento trata de ser una referencia rápida para el usuario acerca de la instalación y manejo de la aplicación obtenida como producto final tras el desarrollo del proyecto presentado en los capítulos anteriores. Se ofrecerá una guía de instalación y los requisitos que el sistema debe ofrecer para la ejecución de la aplicación y una introducción al manejo de la misma desarrollada como una guía de usuario destinada a las personas que pretendan realizar un análisis de las jerarquías de herencia en *Java* para el desarrollo de *frameworks de dominio* mediante técnicas de *Análisis de Conceptos Formales* (ACF). La guía de usuario será esquemática, en la que la mayor parte de los conceptos han sido desarrollados y explicados a lo largo de los capítulos del presente trabajo. Por ello, es preferible una lectura ordenada de los capítulos finalizando con la guía de usuario para una mayor comprensión de todas las funciones presentadas de la aplicación.

8.2 Objetivo de la Aplicación

El objetivo fundamental del producto obtenido será servir de soporte al desarrollo de *frameworks de dominio* mediante técnicas ACF, concretamente dentro de la primera etapa, análisis de herencia. A través del análisis de un conjunto de clases o interfaces desarrolladas en Java, el usuario debe detectar todas las construcciones tanto explícitas (relaciones de herencia se dan en el código) como implícitas (posibles relaciones de herencia que podrían darse) que están definidas en el código objeto de estudio. Para ello se construirá el retículo de *Galois* a través del algoritmo de *Bordat* que junto a una interpretación adecuada de dicho retículo permitirá sacar dichas relaciones de herencia tanto implícitas como explicitas.

8.3 Manual de Instalación

8.3.1 Requisitos del Sistema

Como la aplicación está totalmente escrita en *Java*, funciona de manera independiente de la plataforma usada para su ejecución. Sin embargo, para un mejor rendimiento con conjuntos de datos relativamente grandes, se recomienda utilizar un sistema con las siguientes características:

- **Sistema Operativo**: *Linux, MacOs, Microsoft Windows 9X/Me/2000/XP* (en general cualquier plataforma en la que pueda instalarse una máquina virtual java, JVM).
- **Hardware:**
    - Un *Intel Pentium, AMD Athlon, y* procesadores similares.
    - Cualquier tamaño de memoria que permita la ejecución de la máquina virtual según los requisitos de la misma. El uso de conjuntos de datos grandes requerirá de una mayor velocidad de CPU y de un mayor tamaño de memoria. Se puede usar el parámetro de la JVM "-Xmx<size>m" para permitir que la plataforma use más memoria de la que aproveche por defecto del sistema.

- Entorno a 1Mb de espacio en disco duro para la instalación de la aplicación.

- **Software:** JDK 1.5.0 o más alto. Si se utiliza una versión más baja, la aplicación avisará de ello al usuario y éste podrá decidir si continuar la ejecución o no. Si continúa con la ejecución, es posible que ciertas funciones de la aplicación no funcionen correctamente.

8.3.2 Instalando la Aplicación

En el CD-ROM adjunto se encuentra disponible la versión 1.0 de la aplicación dentro del directorio "programa". El directorio incluye el fichero `hacf-pfc-1.0-bin.jar` con el conjunto de clases compiladas y los ficheros gráficos que utiliza la aplicación, un fichero con el código fuente de la aplicación, `hacf-pfc-1.0-src.zip`, y los ficheros de lotes que permitirán ejecutar la aplicación en las plataformas de Windows y Linux.

Para realizar la instalación se deben seguir las siguientes instrucciones:

Paso 1 – Copie el fichero del CD-ROM "X:\programa\hacf-pfc-1.0-bin.jar", donde X es su unidad de CD-ROM, a un directorio de su elección en el disco duro de su ordenador.

Paso 2 – Si su Sistema Operativo es tipo Windows copie el fichero del CD-ROM "X:\programa\run.bat", donde X es su unidad de CD-ROM, al mismo directorio que eligió para el paso 1. Si su Sistema Operativo es tipo Linux copie el fichero del CD-ROM "X:\programa\run.sh", donde X es su unidad de CD-ROM, al mismo directorio que eligió para el paso 1.

Paso 3 – Asegúrese de que *Sun* JDK 1.5.0 está instalado en su ordenador y que el comando *Java* forma parte del *path* de su sistema. Existen tres formas de ejecutar la aplicación:

- Use el siguiente comando dentro del directorio de instalación

java –jar hacf-pfc-1.0-bin.jar

- Alternativamente, el entorno de ejecución de *Java* (JRE), cuando se instala asocia los ficheros con extensión jar con la máquina virtual de java (JVM) por lo que haciendo doble *clic* en el fichero, hacf-pfc-1.0-bin.jar se iniciará la ejecución de la aplicación.

- Haciendo doble *clic* sobre el fichero run.bat para Windows y run.sh para Linux.

8.4 Manual de Usuario

Hay dos formas de interactuar con la aplicación, una es a través de modo consola, mediante la línea de comandos y otra en un entorno gráfico. Explicaremos primer la esta última y luego detallaremos la línea de comandos.

8.4.1 Ventana Principal

La primera ventana que aparece tras la ejecución de la aplicación, es la ventana principal que nos permitirá crear un contexto formal a partir de un conjunto de clases o interfaces que le suministremos. Inicialmente los tres elementos principales de esta ventana son el menú principal localizado en lo alto de la ventana, la barra de herramientas que mostrará las operaciones más típicas dentro de la aplicación y que también pueden encontrarse en el menú principal y la barra de estado que ira donde se irán manifestando las operaciones que vayamos haciendo. A través del menú principal podremos acceder a todas las funciones que proporciona la aplicación.

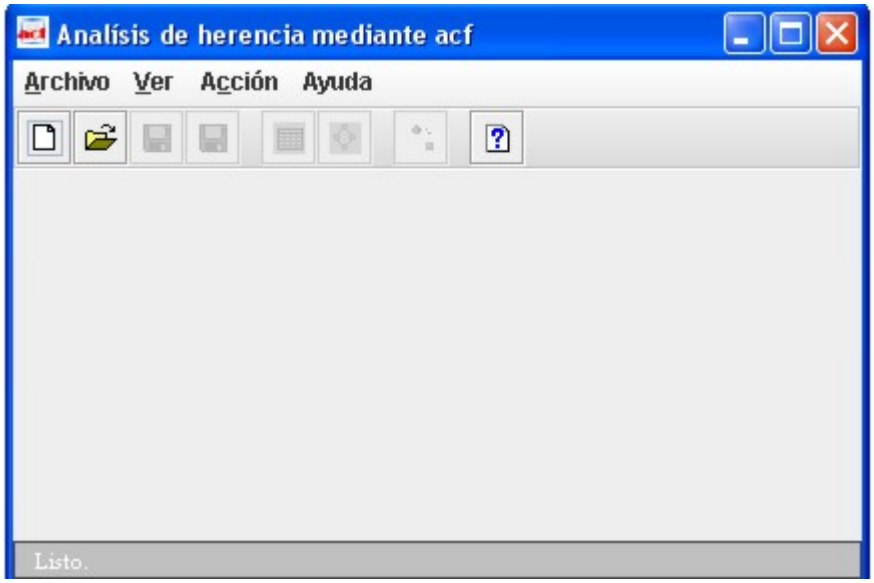

Figura 8-1 *Ventana principal de la aplicación.*

A continuación explicaremos cada opción elegible dentro del menú principal.

8.4.2 Menú Archivo

Las opciones del menú **Archivo** son las siguientes:

Figura 8-2 *Menú Archivo de la ventana principal.*

- ***Abrir***: Abrirá cualquier fichero con uno de los formatos de entrada/salida que ofrece la aplicación.

- ***Nuevo.*** Crea un nuevo proceso para la obtención de un contexto formal, donde introduciremos el código fuente analizar.

- ***Guardar Retículo***: Guardará el retículo de *Galois* que hayamos creado en un momento dado.

- ***Guardar Contexto***: Guardará el contexto con el que estemos trabajando en la ventana gráfica, en un fichero XML con una extensión que por defecto será DBRE, pero eso lo aclararemos más adelante.

- ***Modificar Parser***: Permite modificar el contexto formal que ya tenemos creado, mediante la introducción de nuevas clases o interfaces analizar, permitiendo también modificar el parser.

- ***Salir***: Abandonara el programa.

Hablemos más detalladamente de cada uno de estos elementos y las posibilidades que ofrecen.

Abrir

En general es posible abrir dos tipos de ficheros: un grupo de contextos formales (*.setBin.xml con formato compatible con el proyecto [DBRE, 2005], y *.bin.xml con formato compatible con Galicia versión 3) y un grupo de retículos de conceptos (*.setLat.xml con formato compatible con proyecto [DBRE, 2005]).

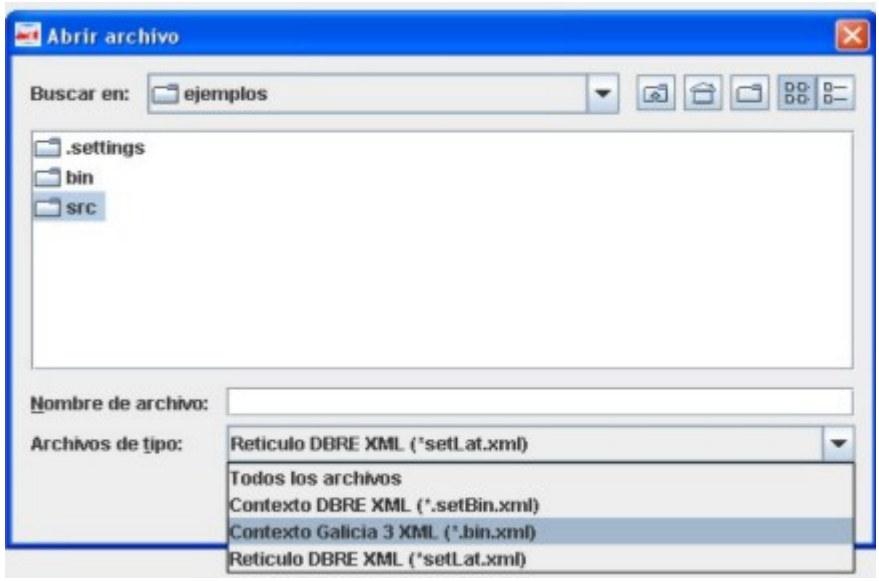

Figura 8-3 *Ventana de elección de archivo a cargar.*

Como nuestra aplicación únicamente trabaja con un contexto o un retículo, al cargar archivos con formato compatible DBRE, se deberá elegir del grupo de contextos o de retículos,

cual de ellos será el que cargaremos en la aplicación. Ya que en [DBRE, 2005] se trabaja en un entorno de múltiples contextos y retículos.

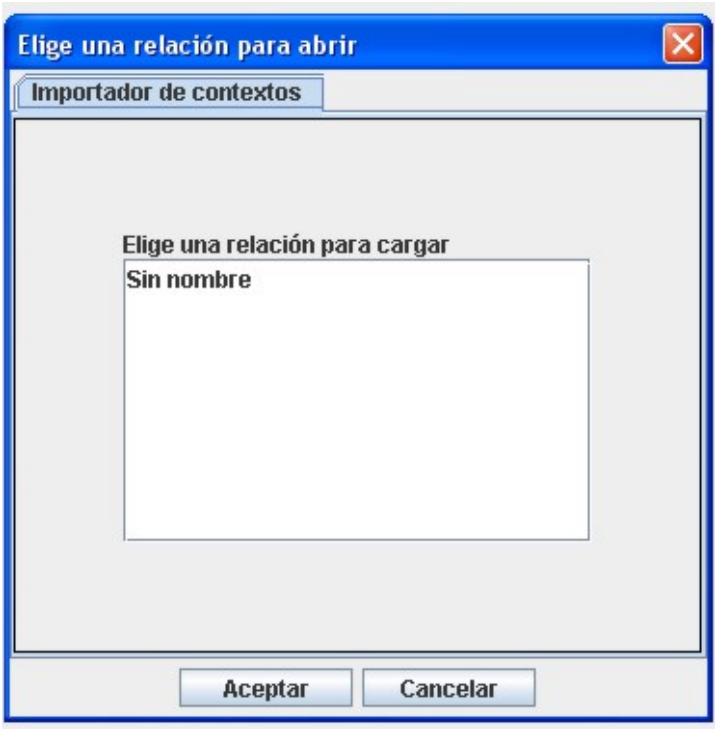

Figura 8-4 *Elige un contexto del grupo de contextos cargados.*

Al abrir el contexto formal seleccionado del grupo de contextos, las estructuras de datos codificadas en el fichero serán trasladadas a memoria creando una tabla de doble entrada donde se muestra la matriz de incidencia. (Ver **Figura 8-8**), con esa matriz podremos calcular su retículo, guardarla de nuevo…pero sin embargo no podremos modificarla a través de la modificación del parser, es decir analizando clases nuevas que se añadirán a las ya analizadas; ya que al seguir el formato compatible con el proyecto anterior, en él, obviamente, no se guardaba información referente a ningún parser, de forma que no se podrá modificar por la inexistencia de parser aplicado.

Al abrir un fichero que contenga un grupo de retículos, pasará lo mismo que con los contextos, el usuario deberá elegir cual de ellos cargar y el elegido se representara. A diferencia de los contextos, únicamente se mostrara sin poder realizar ninguna función con él, salvo ver su representación e interpretarlo. Las estructuras de datos codificadas en el fichero serán trasladadas a memoria creando una ventana de representación de retículo (Ver **Figura 8-9**).

Guardar retículo/Contexto

Como su propio nombre indica guardara el retículo o contexto con el que estemos trabajando, en un archivo de datos cuyo formato podremos elegir en la ventana de menú **Ver** > **Opciones**. Como precondición diremos que para guardar un retículo o un contexto deben existir previamente, sino la opciones de guardar estarán deshabilitadas. Lo mismo pasa cuando ya hemos guardado el retículo o contexto, ya que como no cambiaran se deshabilitan dichas posibilidades.

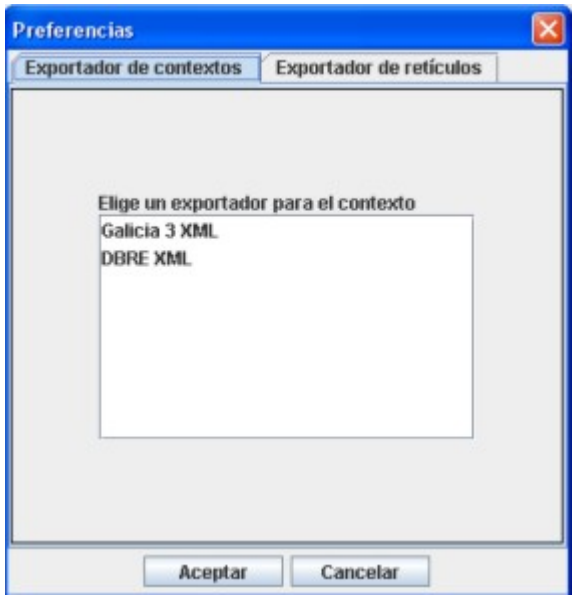

Figura 8-5 *Elección de preferencias para la exportación.*

Podremos elegir entre dos formatos para exportar contextos, por un lado DBRE, formato adoptado del anterior proyecto, y con una DTD definida en [DBRE, 2005] y Galicia, [Galicia, ref] con una DTD propia definida en el proyecto Galicia. Se han puesto estos 2 ya que uno era por requisitos obligatorio, DBRE, mientras que el otro proviene del proceso de pruebas de la herramienta como ya se detallo.

Sin embargo a la hora de exportar retículo únicamente se permite el formato DBRE, ya que proviene de los requisitos, mientras que Galicia no, y si bien permitimos la existencia de ese formato de exportador como derivado de la fase de pruebas en el salvaguardado de contexto, en el caso del retículo no fue tal cosa necesaria por lo que únicamente está DBRE.

Nuevo

Esta opción permite la construcción de una nueva matriz de contexto a través del análisis léxico-sintáctico de una colección de clases o interfaces (no amabas a la vez) suministradas por el usuario. Para ello, tendremos que informar al sistema del lugar donde están las clases que queremos analizar, añadiéndoselas, así como el analizador léxico-sintáctico que queramos emplear y el directorio de búsqueda o classpath donde estarán las clases objeto de estudio así como las clases a las que se haga referencia en los import.

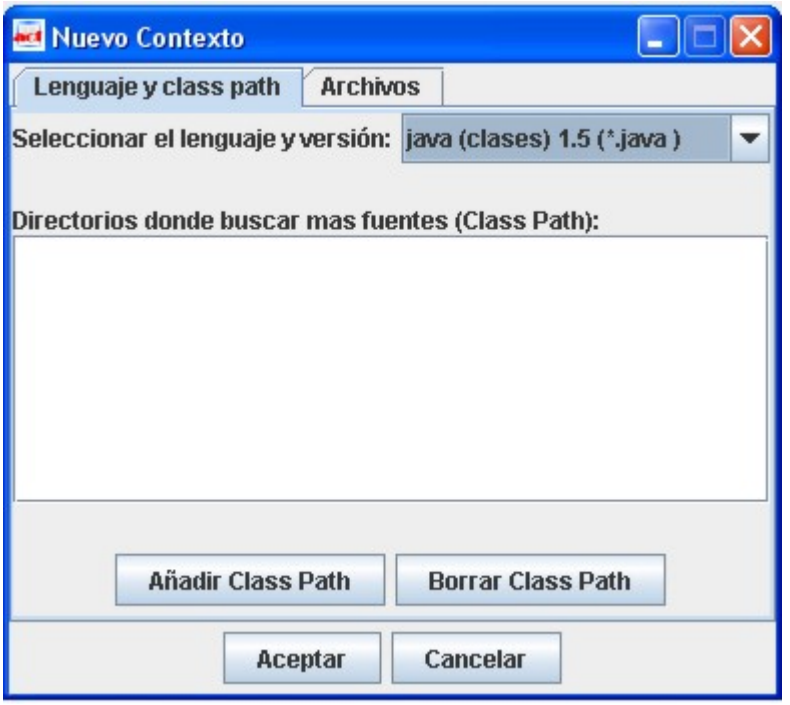

Figura 8-6 *Creación de nuevo contexto formal.*

Modificar Parser

Dicha opción nos permite modificar la matriz de contexto con la que estemos trabajando mediante el análisis de nuevas clases o interfaces junto con el reanálisis de las ya existentes. Cabe destacar que para poder utilizarla hemos tenido previamente que construir una matriz de contexto desde cero, no pudiendo realizar ninguna modificación sobre una matriz de contexto que hayamos recuperado de un archivo de datos, pues como hemos aclarado anteriormente, cuando realizamos dicha operación no dispondremos de ninguna información sobre el lugar donde estarán las clases que están en dicho archivo, ni de si siguen existiendo o no.

Al pinchar en modificar parser se abrirá una ventana como la que aparece al crear un nuevo contexto salvo por la diferencia de que aparecen ya cargadas las clases analizadas, y los classpath. De forma que únicamente tendremos que añadir los nuevos classpath y los nuevos archivos, así como seleccionar el parser que queramos emplear.

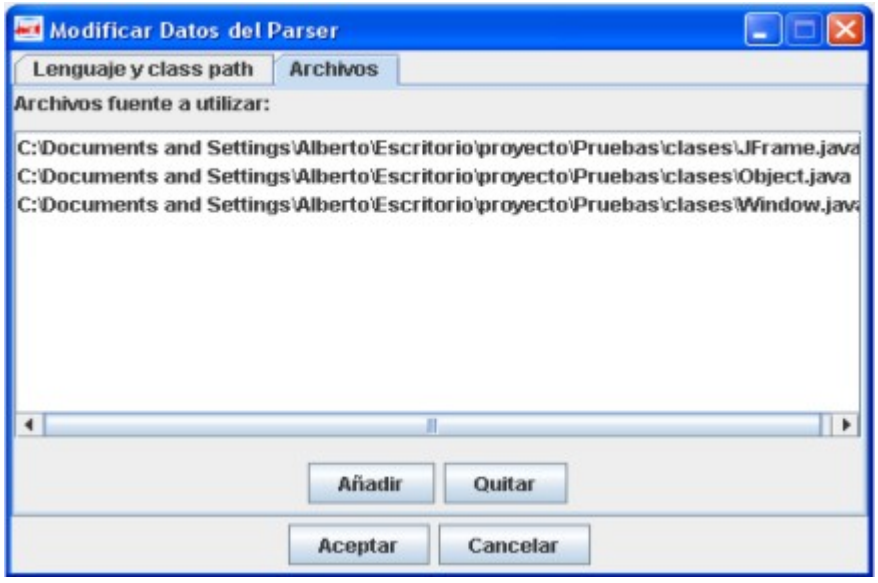

Figura 8-7 *Modificar la matriz de contexto a través de una modificación del parser.*

8.4.3 Menú Ver

Las opciones del menú **Ver** son las siguientes:

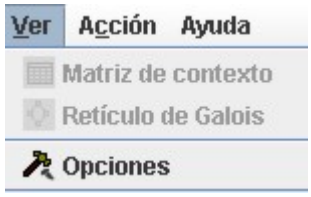

Figura 8-8 *Menú Ver de la ventana principal.*

- ***Matriz de Contexto***: muestra la matriz de contexto que ha sido creada.
- ***Retículo de Galois***: Muestra el reticulo de Galois que ha sido creado previamente, en el caso de que haya sido cerrada la ventana que lo mostraba.
- ***Opciones***: muestra las opciones para la exportación de contexto y reticulos, lo hemos visto anteriormente.

Matriz de Contexto

Lo único que señalaremos es que la matriz una vez creada se muestra por si sola, aquí lo que se hace es que cuando pinchas se vuelve a mostrar. Cabe señalar que no se puede modificar las celdas de la matriz, pero si el orden y ancho de las columnas, de forma que se podrán recolocar a nuestro antojo para facilitar una mejor comprensión del contexto.

A	B	C	D	E	F
	atr0	atr1	atr2	atr3	atr4
Clase1	✓	✓	0	0	0
Clase2	✓	✓	✓	✓	0
Clase3	✓	✓	0	0	✓
Clase4	✓	✓	✓	✓	✓

Figura 8-8 *Matriz de contexto o de incidencia.*

Retículo de *Galois*

Aquí, al igual que la funcionalidad anterior, volverá a dibujar la ventana del retículo. Cabe señalar que en dicha ventana podremos mover los nodos así como el dibujo del grafo a nuestra disposición para una representación más clara, aparte si pinchamos con el botón derecho sobre cualquier nodo aparecerán los conjuntos de extensión e intención de dicho concepto. Si volvemos a pinchar sobre el, desaparecerán.

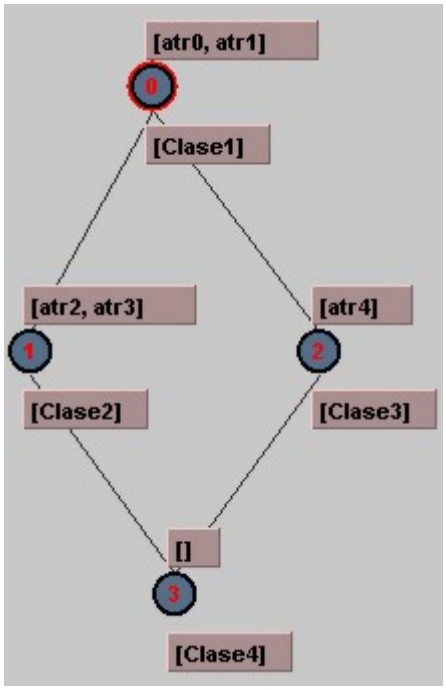

Figura 8-9 *Retículo de Galois.*

Opciones

Lo hemos visto anteriormente, aquí se escogerá el formato para la exportación, con lo que no nos extenderemos más sobre ello.

8.4.4 Menú Acción

La única opción del menú Acción es:

Figura 8-10 *Menú Acción.*

En dicho menú será donde se cargarán los futuros algoritmos que se podrían añadir.

La única opción de este menú, ahora mismo es construir retículo aplicando el algoritmo de *Bordat*, para lo cual es necesario disponer de una matriz de contexto, ya sea generada desde cero o cargada a través de un archivo de datos.

8.4.5 Menú Ayuda

Las opciones del menú Ayuda son las siguientes:

Figura 8-11 *Menú Ayuda.*

En dicho menú estarán las opciones referidas a la ayuda en línea, donde se cargara una copia de este manual de usuario en HTML y otra referida a la información del proyecto.

8.4.6 Modo línea de comandos

Es posible ejecutar la herramienta en modo línea de comandos, donde tras las comandos:

```
java –jar hacf-pfc-1.0-bin.jar
```

Le acompañamos del siguiente parámetro:

```
-nogui
```

Si no introducimos nada más, el sistema informara de un error de parámetros, pues se espera la introducción de más parámetros, así como de la posible sintaxis de de parámetros.

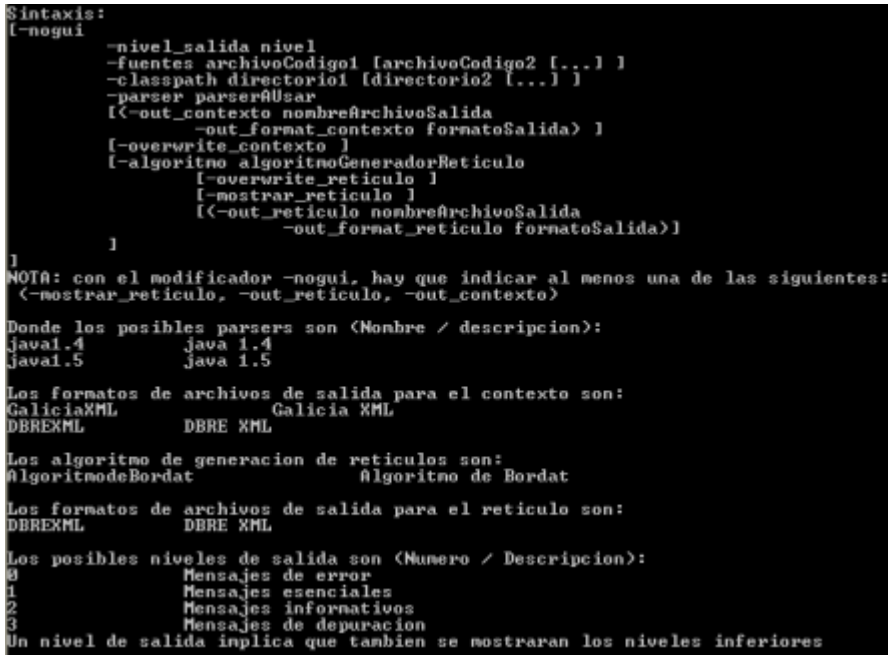

Figura 9-12. *Opciones del modo consola.*

Las posibles opciones que deberán preceder a `-nogui` son los siguientes:

```
-nivel_salida nivel

-fuentes archivoCodigo1 [archivoCodigo2 [...]]

-classpath directorio1 [directorio2 [...]]

-parser parserAUsar

-out_contexto nombreArchivoSalida -out_format_contexto
formatoSalida

-overwrite_contexto

-algoritmo algoritmoGeneradorRetículo

   [-overwrite_reticulo]
   [-mostrar_reticulo]
   [-out_reticulo   nombreArchivoSalida   -out_format_reticulo
formato]
```

Con el modificador `-nogui` hay que indicar al menos una de las siguientes:

`-mostrar_reticulo, -mostrar_reticulo, -out_contexto.`

Pasaremos a detallar cada una de las posibles opciones:

`-nivel_salida nivel` indicaremos con él, que nivel de detalle queremos en la salida pudiendo elegir entre:

0. Mensajes de Error
1. Mensajes esenciales
2. Mensajes informativos
3. Mensajes de depuración

`-fuentes archivoCodigo1 [archivoCodigo2 [...]]` introduciremos las clases o interfaces que queramos analizar, hay que reseñar que sólo se analizaran o clases o interfaces, no ambos a la vez.

`-classpath directorio1 [directorio2 [...]]` Se introducirá el directorio de búsqueda o classpath, donde se encontraran las referencias que se hacen en las clases analizar.

`-parser parserAUsar` Como su propio nombre indica se le dirá a la herramienta que analizador sintáctico utilizará, hay dos disponibles, Java1.4 y Java1.5, el primero es de ejemplo para futuros desarrollos de nuevos parsers y ampliaciones y no es del todo funcional, y Java1.5 que si que es enteramente funcional y que podrá ser capaz de analizar tanto clases como interfaces, ambas por separado, nunca juntas.

`-out_contexto nombreArchivoSalida –out_format_contexto formatoSalida` con esta opción extraeremos el archivo de contexto en un archivo de datos XML, cuyo nombre introduciremos al principio y cuyo formato introduciremos a continuación. Podemos optar por dos formatos distintos de salida, GaliciaXML y DBREXML, el primero compatible con Galicia [Galicia, ref] y el segundo con el proyecto en el que nos basamos [DBRE, 2005].

`-overwrite_contexto` que indicara que en el caso de encontrarse ya un archivo de datos XML con el nombre que hemos introducido para guardar, que le sobrescriba.

`-algoritmo algoritmoGeneradorRetículo` indica que algoritmo utilizar para la extracción del retículo, siendo la única opción por el momento AlgortimodeBordat, pudiéndose en un futuro incorporar nuevos algoritmos. Va acompañado de las siguientes opciones:

> `–overwrite_reticulo` indica que sobrescriba si encuentra un archivo de datos XML con el mismo nombre con el que vamos a guardar nuestro retículo.
>
> `–mostrar_reticulo` mostrara con una ventana gráfica el retículo construido. Si estamos en un modo no grafico, no se podrá construir dicho retículo y la aplicación avisará de ello.
>
> `-out_reticulo nombreArchivoSalida –out_format_reticulo formato` exportará el retículo a un archivo de datos XML con nombre, nombreArchivoSalida, y con formato, el indicado posteriormente. Disponemos de un único formato de exportación de archivos de datos XML para retículos, y es DBRE, guardando la retrocompatibilidad con el anterior proyecto.

9 Conclusiones y líneas futuras

9.1 Conclusiones

En primer lugar destacaremos que se han cumplido los objetivos previstos desde un principio: desarrollar una herramienta que implemente la propuesta planteada referente a la aplicación del *Análisis de Conceptos Formales* en el desarrollo de *frameworks de dominio* mediante técnicas ACF en relación a la primera etapa de dicho proceso, análisis de herencia. Además sirve como apoyo en el estudio de las jerarquías de herencia en un conjunto de clases o interfaces *Java* para una posterior refactorización, viendo las posibles relaciones de herencia que podrían darse, aparte de las ya existentes, en dicho código objeto de estudio. Pero, además de mencionar los objetivos cumplidos, es necesario realizar las siguientes consideraciones.

Además, se ha realizado **una revisión de la literatura existente** tanto sobre el desarrollo de *frameworks de dominio* como sobre el *Análisis de Conceptos Formales* aplicado a esa materia, y a otras para comprender mejor su uso y funcionalidad. Como resultado de esta revisión, se localizó una propuesta metodológica realizada por [Prieto, Crespo, Marques y Laguna, 2003] que se ha aplicado en la realización del caso de estudio, de manera que se han completado la fase inicial de dicho proceso, unas de forma manual y otras con el soporte de las herramientas que se han ido desarrollando a lo largo de este proyecto. Hacemos referencia en este punto a la interpretación que deberá hacer el usuario sobre el retículo de *Galois* extrayendo toda la información referente tanto a las posibles relaciones de herencia, como a las ya existentes en el código. Así como la discriminación que deberá hacer al principio de todo el proceso según quiera analizar clases o interfaces.

Importancia del desarrollo de *frameworks* de dominio como *asset* de reutilización. A lo largo de todo el proyecto ha habido una idea central que ha formado el núcleo del mismo, el desarrollo de f*rameworks de dominio mediante ACF*. En la primera etapa del proceso enunciado en [Prieto, Crespo, Marques y Laguna, 2003] se habla del análisis de herencia para la obtención de las características comunes del conjunto inicial de clases que sirven como punto de partida de dicho proceso. El enfoque adoptado permite la reutilización del conocimiento almacenado en las aplicaciones del dominio de una manera formal, a diferencia de otras propuestas que simplemente utilizan estas aplicaciones como punto de partida, o como un modo informal de adquirir conocimiento sobre el dominio. El proceso con todas sus etapas, no sólo el análisis de herencia, conduce a la idea de que se podría extender la aplicación a nuevos dominios, tales como la reingeniería y diagnóstico de software. Es por ello, que durante el desarrollo del proyecto se ha constatado la importancia del desarrollo de *frameworks de dominio mediante ACF* y con ello la necesidad de una infraestructura que facilite dicho proceso de una forma cómoda y no compleja.

Una solución orientada al *Análisis de Conceptos Formales*. El proyecto aplica técnicas de ACF como se habla en [Prieto, Crespo, Marques y Laguna, 2003], no solo para la representación de las relaciones de herencia entre clases, sino también para poder obtener información de las posibles relaciones que se podrían dar mediante la interpretación del retículo de *Galois*. Con el uso de estas técnicas basadas en el ACF permite automatizar el proceso de construcción de un *framework*, desde su versión inicial, y utilizando la información proporcionada por las aplicaciones del dominio que van siendo implementadas o instanciadas a partir del *framework*, lo que nos lleva a un enfoque más industrial y por ello más verificable y menos propenso a errores. Si bien, somos conscientes de que no es una solución definitiva, pues aun faltaría la realización de las otras etapas como es el estudio de clientela y el de dependencias funcionales, esperamos haber cubierto los requisitos de usuario establecidos desde un principio, que era el objetivo básico.

La necesidad de toma de decisiones. La realización de un proyecto de estas características obliga a los alumnos a tomar decisiones basadas en los conocimientos adquiridos en la carrera siempre en un espacio de tiempo muy acotado. El hecho de experimentar cómo estas decisiones, positivas o negativas, tienen un efecto posterior sobre todo el proceso ayuda al alumno a comprender la necesidad de argumentar las decisiones tomadas. Esto sirve como primera toma de contacto a la vida laboral a la que posteriormente nos tendremos que adaptar y como experiencia en el desarrollo de un proyecto completo en que no todas las soluciones son tomadas de forma trivial.

9.2 Líneas de Trabajo Futuras

La realización de un proyecto no debe quedar cerrada nunca a nuevas mejoras. Desde el principio del mismo se debe enfocar, no como un proyecto cerrado, sino como un módulo adicional al trabajo ya hecho. Este proyecto se ha intentado abordar desde el punto de vista del desarrollo de *frameworks de dominio mediante ACF en su primera etapa*, siempre enfocado a facilitar al usuario su labor, dándole una idea intuitiva y visual de las estructuras con las que se trabaja.

Una vez realizado el trabajo, se ha cumplido la totalidad de objetivos planteados, pero queda claro que el número de nuevos requisitos, tanto funcionales como técnicos pueden crecer de nuevo a lo largo del tiempo. En este apartado intentaremos enfocar aquellos puntos mejorables que pueden ser de interés en un futuro.

La incorporación de nuevas DTDs compatibles con el resto de proyectos parecidos existentes en este campo. Si bien nosotros adoptamos la ya utilizada en el proyecto [DBRE, 2005], e incorporamos la de Galicia [Galicia, ref] en la fase de pruebas para contrastar los resultados, no estaría demás la implementación con el resto de trabajos existentes que desempeñan funciones parecidas dentro del mismo campo. Si bien existe un gran numero de ellas si se podría dar compatibilidad con las más importantes y funcionales.

La incorporación de la DTD estándar en cuanto salga. Ya que si bien el criterio adoptado de dar compatibilidad con otras herramientas parecidas es interesante, no deja de ser una solución temporal hasta la aparición de dicha DTD. Ya se vio como la herramienta está diseñada de una forma genérica de forma que facilite la incorporación de nuevas DTDs así como de la DTD estándar, cuando ésta vea la luz.

Añadir módulos de diagnostico que no hagan necesario la interpretación del retículo de *Galois*, ya que éste precisa de conocimientos sobre teoría matemática por parte del usuario. Si se añadiesen dichos módulos en un futuro se facilitaría en gran medida la labor al usuario, estableciéndose en términos que cualquier desarrollador de software pudiera entender sin precisar de dichos conocimientos matemáticos. Sin embargo no habría que descartar la idea de interpretar dicho retículo pues aun con las herramientas de diagnostico siempre pueden inducir a error, de forma que entre las herramientas de diagnostico y una interpretación de dicho retículo se sacarían conclusiones acertadas.

La incorporación de nuevos algoritmos que permitan la obtención del retículo de *Galois*. Debido a la gran variedad de algoritmos que existen para estos fines, sería interesante incorporarlos a la herramienta con el fin de observar las diferencias entre ellos en referencia tiempos. La aplicación deja abierta esa posibilidad pudiendo incorporar nuevos algoritmos para la extracción de los datos de la matriz de contexto y la posterior construcción del retículo de *Galois*. Esta diseñado para ello de una forma genérica y suficientemente fácil para que no surjan complicaciones en dicha incorporación.

La incorporación de nuevos parsers ya sean para nuevas versiones futuras de *Java* o para distintos lenguajes, la herramienta esta diseñada lo suficientemente genérica para permitir la incorporación de nuevos *parsers* de una forma fácil y sencilla, como ya se vio anteriormente. De esa forma se podría actualizar la herramienta cada vez que saliera una nueva versión de *Java* o haciéndola multilenguaje de forma que no sólo reconozca *Java* sino que también muchos más.

La representación de grafos es un problema de difícil solución cuando se toma como objetivo el minimizar el número de cruces de arcos. Se deja abierto el desarrollo de un algoritmo eficaz que minimice dicho número de cruces a la hora de dibujar el grafo. Gracias a la solución arquitectónica decida en este proyecto, nos encontramos ante un módulo independiente, reutilizado del proyecto [DBRE, 2005], de fácil sustitución por lo que quedan abiertas todas aquellas mejoras sobre el mismo.

Sería interesante en un futuro la posibilidad de estandarizar la aplicación para soportar varios idiomas e incorporar posibilidades de impresión ya que eso haría más sencilla la visión del grafo. Tampoco estaría mal la inclusión de una consola donde se vaya siguiendo el desarrollo del proceso donde poder observar paso a paso todo el proceso. Así como el permitir la construcción de la matriz formal de contexto con un número de atributos y objetos definidos por el usuario en el que se permita la edición, con el fin de poder experimentar el usuario con ella, como si de una herramienta ACF genérica se tratase.

Apéndice A: Contenido del CD-ROM

Contenido

La siguiente tabla muestra el contenido del CD-ROM adjunto al trabajo presentado.

Nombre del Fichero	Descripción
memoria.pdf	Documento de la memoria presente (versión pdf).
Programa	Directorio.
hacf-pfc-1.0-bin.jar	Fichero *jar* con las clases compiladas y el resto de ficheros necesarios, listo para su ejecución.
hacf-pfc-1.0-src.zip	Fichero *zip* con el código fuente de la herramienta listo para su compilación.
run.bat	Fichero por lotes que permite ejecutar la aplicación en un entorno Windows.
run.sh	Fichero por lotes que permite ejecutar la aplicación en un entorno tipo Linux.
Documentación	Directorio.
JavaDoc	Directorio dentro del directorio documentación.
index.html	Este directorio contiene la documentación detallada de las clases obtenida con la herramienta *Javadoc* en formato *html*. *index.html* será el fichero raíz de la documentación.
Diccionario	Directorio dentro del directorio documentación.
Diccionario.html	Diccionario de datos.
Ingeniería	Directorio dentro del directorio documentación.
Análisis_de _jerarquias_de _herencia.uml	Documento acerca de la ingeniería del software de la herramienta en formato starUML.
Diagramas	Directorio dentro del directorio Ingeniería.
Clases	Directorio dentro del directorio Diagramas.
*.jpg	Diagramas de clase correspondientes a los descritos en la memoria pero más detallados.
Secuencia	Directorio dentro del directorio Diagramas.
*.jpg	Diagramas de secuencia correspondientes a los descritos en la memoria, detallados y en una dimensión adecuada.
DBRE	Directorio.
ProyectoDBRE.zip	Archivo que contiene el programa del proyecto DBRE, el código fuente así como la memoria y el manual de usuario.

Bibliografía

Artículos

[Bordat, 1986] J.P. Bordat, "Calcul Partique du Treillis de Galois d'une Correspondance", *Mathématiques et Sciences Humaines*, vol. 96, pp. 31–47, 1986.

[Brodie 1995] Brodie, M., Stonebraker, M. "Migrating Legacy Systems". Morgan Kaufmann 1995.

[DBRE, 2005] A. Nistal Calvo, Tutora C. Hernández Díez "Reingeniería de BBDD mediante el análisis de conceptos formales: un caso de estudio", *PFC-I.T.I.G*, Septiembre 2005.

[Fayad y Schmidt, 1997] M. Fayad y D. C. Schmidt, "Object-Oriented Application Frameworks", *Communications of theACM*, vol. 40, No. 10, pp. 32–38, Oct. 1997.

[Fayad, 2000] M.E. Fayad, "Introduction to the Computing Surveys' Electronic Symposium on Object-Oriented Application Frameworks", *ACM Computing Surveys*, vol. 32, No. 1, pp. 1–9, Mar. 2000.

[Godin y Chau, 2000] R. Godin y T. Chau. "Comparaison d'Algorithmes de Construction de Hiérarchies de Classes",aceptado en *L'Object*, 2000.

[Godin y Mili, 1993] R. Godin y H. Mili, "Building and Maintaining Analysis-level Class Hierarchies Using Galois Lattices", *Proceedings of OOPSLA'93*, pp. 349–410, 1993.

[Ganter y Wille, 1999] B. Ganter and R. Wille. "Formal concept analysis: mathematical foundations". Springer, 1999.

[Johnson y B. Foot, 1988] R.E. Johnson y B. Foote, "Designing Reusable Classes", *Journal of Object-OrientedProgramming*, vol. 1, No. 2, pp. 22–35, 1988.

[Johnson y Russo, 1991] R.E. Johnson y V.F. Russo, *Reusing Object-Oriented Designs*, technical report *UIUC DCS 91-1696*, University of Illinois, Mayo 1991.

[Marticorena y Crespo, 2003] R. Marticorena, Y. Crespo. "Refactorizaciones de Especialización sobre el Lenguaje Modelo MOON Versión 1.0". *Dep. inf.*, Universidad de Valladolid, España, Febrero 2003.

[**Pérez, 2002]** D. Pérez Durán, Fco. J. Cano Sandoval, tutor: F. Prieto, "Implementación de un sistema de diagnóstico software mediante Análisis de Conceptos Formales". *PFC- I.T.I.* Sistemas, Enero 2002.

[Pree, 1995] W. Pree, *Design Patterns for Object-Oriented Software Development.* Addison Wesley, Wokingham, 1995.

[Prieto, Crespo, Marques y Laguna, 2000] F. Prieto, Y. Crespo, J.M. Marqués y M. Laguna, "Mecanos y Análisis de Conceptos Formales como Soporte para la Construcción de Frameworks", *Actas de las V Jornadas de Ingeniería de Software y Bases de Datos (JISBD'2000)*, pp. 163–175, Nov. 2000.

[Prieto, Crespo, Marques y Laguna, 2003] F. Prieto, Y. Crespo, J.M. Marqués y M. Laguna, "Análisis de Conceptos Formales como Soporte para la Construcción de Frameworks de Dominio", *I+D Computación*, Vol. 2, No. 1, Marzo 2003

[Roberts y Johnson, 1997] D. Roberts y R. Johnson, "Patterns for Evolving Frameworks", eds. R.C. Martin, D. Riehle y F. Buschmann, *Pattern Languages of Program Design*, vol. 3, Addison–Wesley, pp. 471–486, 1997.

[Rodríguez y Gil, 1999] J. M. Rodríguez Párraga y M. Gil San Martín, tutor: V. Cardeñoso, "Herencia Múltiple de Clases para Java", *PFC-I.T.I.* junio 1999.

[Schmid y Johnson,1999] H.A. Schmid, "Framework Design by Systematic Generalization", eds. M. Fayad, D. Schmidt y R. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Desing*, capítulo 15, Wiley Computer Publishing, pp. 353–378, 1999.

[Schwarzweller, 2000] Schwarzweller, C. *Mizar Formalization of Concept Lattices*. En Mechanizae Mathematics and its Applications, vol. 1, 2000.

[Snelting y Tip,1998] G. Snelting y F. Tip, "Reengineering Class Hierarchies Using Concept Analysis", *ACM SIGSOFT Software Engineering Notes*, 23(6), pp. 99–110, Nov. 1998; *Proceedings of the ACM SIGSOFT Sixth Internatioal Symposium on the Foundations of Software Engineering*.

[Stumme, Wille y Wille, 1988] G. Stumme, R. Wille y U. Wille, *Conceptual Knowledge Discovery in Databases Using Formal Concept Analysis Methods*. En: J. M. Zytkow, M. Quafofou (eds.): Principles of Data Mining and Knowledge Discovery. Proc. 2nd European Symposium on PKDD '98, LNAI 1510, Springer, Heidelberg 1998, 450-458

[Vogt y Wille, 1995] Vogt, F. y Wille, R. *TOSCANA– a Graphical Tool for Analyzing and Exploring*
Data. Lecture Notes of Computer Science, vol 894, Heidelberg, 1995. Springer.

[Wille, 1982] R. Wille. "Restructuring lattice theory: An approach based on hierarchies
of concepts". In Ordered Se ts, pages 225-470. Reidel, Dordrecht-Boston, 1982.

[W-B y Johnson, 1990] R.J. Wirfs-Brock y R. E. Johnson, "Surveying Current Research in Object-Oriented Design", *CACM*, vol. 33, No. 9, pp. 105–124, Sep. 1990.

Referencias Web

[Tockit, ref] **http://sourceforge.net/projects/tockit**
Tockit tries to build a framework for Conceptual Knowledge Processing (CKP) and Formal Concept Analysis (FCA) in Java, using a component-based approach, XML formats and a three-tier architecture. **Project Admins**: jhereth, peterbecker
Operating System: OS Independent (Written in an interpreted language)
License: BSD License
Category: Front-Ends, OLAP, Visualization

[Toscana, ref] **http://sourceforge.net/projects/toscanaj**
ToscanaJ reimplements the classic interface for Formal Concept Analysis, a data analysis technique based on set theory.**Project Admins**: jhereth, peterbecker
Operating System: OS Independent (Written in an interpreted language)
License: BSD License
Category: Front-Ends, Indexing/Search, Information Analysis, Visualization

[Galicia, ref] **http://sourceforge.net/projects/galicia**
Galicia is intended as a multi-tool open source platform for creating, visualizing and storing concept/Galois lattices. It features a set of FCA algorithmic methods for lattice construction and layout and for association rule extraction.
Project Admins: cel8068, croume, octepix69, rouanehm.

Libros

[Boooch et al., 99] G. Booch, I. Jacobson, J. Rumbaugh. “El Lenguaje Unificado de Modelado”. Ed. Addison -Wesley Iberoamericana, Madrid. 1999.

[Larman, 2003] C. Larman, “UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado”, 2ª ed., Ed Pearson Prentice Hall, Madrid 2003.

[Horton, 2005] I. Horton, “Ivor Horton’s Beginning Java 2, JDK 5 Edition” Ed. Wiley Publishing Inc. Wronx. 2005.

[Richardson, 2005] W. Clay Richardson, D. Avondolio, J. Vitale, S. Schrager, M.W. Mitchell y J. Scanlon, “Professional Java, JDK 5 Edition”, Ed. Wiley Publishing Inc. Wronx. 2005.

[Daum, 2005] B. Daum, “Professional Eclipse 3 for Java Developers”, Ed. Wiley Publishing Inc. Wronx. 2005.