Desarrollo de Guías de Búsqueda de Metodologías del Proceso de Reingeniería de Aplicaciones Orientadas a Objetos

Magdalena Cerviño y Gimena Moro

Universidad Austral
Avda. Juan de Garay 125
(1054), Buenos Aires, Argentina

Resumen En la actualidad, las organizaciones no sólo se enfrentan con el problema de reemplazar sus sistemas de información, sino que deben mantener un control de sus sistemas *legacy*. El mantenimiento de estos sistemas puede implicar realizar cambios para adaptarlo a nuevos requerimientos o nuevas tecnologías o bien, mejorar la calidad de los mismos. Para un adecuado mantenimiento de un sistema, es necesario aplicar *reingeniería*, que es definida por Chikofsky como la *"recuperación de información sobre el diseño de un programa existente y el uso de esta información para reestructurar o reconstruir el programa existente"*. Este proceso tiene como objetivo el de conseguir una mayor facilidad de mantenimiento en el futuro (*mantenimiento preventivo*). En un proceso de reingeniería completo identificaremos dos etapas: *Reverse Enginnering* y *Forward Engineering*.

Debido a la complejidad de cada una de las etapas, durante los últimos 20 años han surgido diferentes metodologías que pueden aplicarse para obtener un proceso de reingeniería con calidad. Entonces se hace necesario realizar una evaluación del estado del arte actual y desarrollar una guía que ayude a un desarrollador a elegir una metodología adecuada para mantener su sistema, dependiendo en qué etapa de la reingeniería se encuentre.

El aporte de este trabajo es el desarrollo de una guía de lectura con estrategias de búsqueda para asistir a los desarrolladores en la elección de las metodologías de reingeniería adecuadas, dependiendo en qué etapa de la reingeniería se halla el sistema. Específicamente en nuestro caso de estudio, esta guía está basada en el estado del arte actual de metodologías de reingeniería que utilicen el Análisis de Conceptos Formales (ACF), como herramienta de base. A su vez, utilizaremos también el ACF como una técnica de clasificación para construir la guía mencionada. Se elige esta metodología debido a las ventajas que la misma provee para agrupar elementos basados en sus propiedades. Esto resulta vital para identificar metodologías con características similares y comprender cómo funciona cada propuesta de las publicaciones analizadas en el estado del arte.

1. Introducción

En la actualidad, la evolución (natural) de los sistemas de software hacen que los mismos se vean afectados por constantes cambios. Generalmente, la mayoría de los sistemas pueden lograr adaptarse a estos cambios, mientras que otros no logran hacerlo sino es a un costo muy alto. En este contexto, la *reingeniería* juega un papel crucial en el ciclo de vida de un sistema de software. El objetivo de la reingeniería es reducir la complejidad de un sistema lo suficiente como para que pueda continuar siendo usado y adaptado a un costo relativamente aceptable.

Existen muchas razones por las cuales se puede aplicar reingeniería, por ejemplo:

- La modularización de un sistema: Las partes individuales pueden ser más fácilmente comercializados por separado o combinadas de diferentes maneras.
- La mejora de la performance: La experiencia demuestra que la secuencia correcta es *primero hacer, luego hacerlo bien y, por último hacerlo rápido*, por lo que se necesita de la reingeniería para refactorizar el código antes de pensar en términos de *performance*.
- La migración de un sistema a una nueva plataforma: Antes de hacerlo, se debe revisar la arquitectura de un sistema para separar claramente el código que depende de la plataforma.
- Incorporación de nuevas tecnologías: Los sistemas deben adaptarse a las nuevas tecnologías, tales como las nuevas normas o las bibliotecas, como un paso hacia la reducción de los costos de mantenimiento.

1.1. La reeingeniería como una necesidad.

Es muy importante reconocer cuando es necesario un proceso de reingeniería. Identificar esta necesidad, es en definitiva conocer si es posible el mantenimiento del propio sistema. Los sistemas que tienden a *romperse* en un lapso de tiempo corto, son aquellos que *no son mantenibles* en el tiempo y que presentan *síntomas* que muestran la existencia de algunos de los siguientes problemas:

- Obsoletos o sin la documentación.
- Falta de *Tests*.
- Desarrolladores o usuarios que han dejado el proyecto.
- Comprensión limitada de todo el sistema.
- Demasiado tiempo para hacer cambios sencillos.
- Necesidad de una constante corrección de errores.

1.2. El ciclo de vida de la reingeniería.

La Figura 1 muestra el ciclo de vida completo del proceso de reingeniería. El mismo está compuesto por dos procesos importantes llamados: *reverse engineering*, el cual podemos identificar en la primera mitad de la curva de la figura y *forward engineering*, el cual reconocemos en la segunda mitad de la curva.

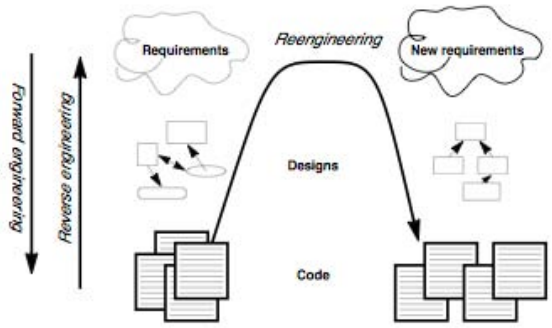

Figura 1. Ciclo de vida del proceso de reingeniería

Chikofsky y Cross [CCI90] propone las siguientes definiciones:

Reverse Engineering: Es el proceso de análisis del código fuente de un sistema, para luego poder identificar sus componentes y relaciones, con el objetivo de crear una representacón del mismo en otro formato o en un nivel de abstracción de más alto.

Reingeniería: Es la revisión y alteración de un sistema para reconstruirlo en una nueva forma.

La introducción del término *reverse engineering* genera también el concepto de **Forward Engineering**, que es el proceso tradicional de pasar de abstracciones de alto nivel y diseño lógico e independiente del lenguaje a una implementación física del sistema.

1.3. Solución concreta al problema planteado

Debido a la complejidad de cada una de las etapas, durante los últimos 20 años han surgido diferentes metodologías que pueden aplicarse para obtener un proceso de reingeniería con calidad. Cabe mencionar que cada metodología funciona dentro de un contexto y bajo ciertas condiciones. Por ejemplo, el trabajo documentado en “Revealing Java Class Structure with Concept Lattices”[DG03] analiza aplicaciones implementadas en Java y el objetivo es el estuido de archivos de clases para comprender cómo están

estructuradas las mismas usando como información los métodos y cómo interactúan entre ellos. Toda esta información queda ímplicitas dentro de las publicaciones que los documentan, y hacen que las metodologías no puedan aplicarse o usarse en forma genérica dentro de otros sistemas.

Entonces se hace necesario generar un *estado del arte completo* y *guía de lectura con estrategias de búsqueda* que ayude a un desarrollador a *elegir* una metodología adecuada para mantener su sistema, dependiendo en qué etapa de la reingeniería se encuentre. La *guía de lectura y las estrategias de búsqueda* le permitirá al desarrollador elegir la metodología que necesita a partir de temas propuestos por el mismo, o bien propuestos por el sistema diseñado. Una vez que el desarrollador encuentra la metodología puede consultar los detalles de la metodología utilizando el estado del arte propuesto.

En este trabajo de grado nos concentraremos en el análisis de metodologías de reingeniería que utilicen el Análisis de Conceptos Formales, como herramienta de base, y construiremos la guía de lectura usando también la técnica mencionada. La elección del Análisis de Conceptos Formales se justifica en que la misma puede agrupar elementos basados en sus propiedades. Esto resulta vital para identificar metodologías con características similares, y así construir una jerarquía de publicaciones que formarán las diferentes partes de la guía de lectura.

Este trabajo se organiza de la siguiente manera: la Sección 2 introduce brevemente la técnica del Análisis de Conceptos Formales. La Sección 3 presenta el desarrollo de la guía de uso propuesta. La Sección 4 presenta las conclusiones de este trabajo de grado y, finalmente detalla ideas interesantes que podrían enriquecer a este trabajo.

2. Análisis de Conceptos Formales

El Análsis de Conceptos Formales es una rama de la teoría de *Lattices* que permite identificar grupos significativos de elementos que tienen propiedades comunes [GW99]. Si consideramos un conjunto de elementos y propiedades de esos elementos, ACF determinará grupos maximales de elementos y propiedades. Estos grupos maximales son llamados *conceptos*. Cada uno de estos conceptos está formado por un conjunto de elementos que tienen una o más propiedades en común, tal que ningún otro elemento tenga esas propiedades.

Veamos un ejemplo. Consideremos como elementos animales que son famosos en distintas partes del mundo: los personajes animados Garfield y Snoopy; la famosa ballena Willy, y por último la collie más famosa del mundo Lassie. Así nuestro conjunto de elementos queda formado de la siguiente manera {*Garfield, Snoopy, Willy, Lassie*}. Las propiedades describen si estos personajes son cartoons, o animales “reales”, si son perros, gatos, mamíferos o ballenas. De esta manera, los atributos quedaron definidos con el siguiente conjunto: {*cartoon, real, perro, gato, ballena*}.

Los elementos y las propiedades se ordenan en una *tabla de incidencia*, que muestra la relación entre los elementos y sus propiedades. El Cuadro 1 muestra la tabla de incidencia correspondiente al ejemplo.

	Cartoon	Real	Perro	Gato	Ballena
Garfield	x			x	
Snoopy	x		x		
Willy		x			x
Lassie		x	x		

Cuadro 1. Tabla de incidencia

El conjunto de elementos y propiedades junto con sus respectivas relaciones forman lo que se llama *contexto formal*. Informalmente llamaremos O al conjunto de elementos, P al de propiedades y por último I al que representa la relación binaria entre O y P, mostrada en la tabla de incidencia (Cuadro 1). En nuestro ejemplo, los elementos (conjunto O) son los animales, las propiedades (conjunto P) son las características, y la relación binaria (I) es "es-un". Por ejemplo, la tupla {Garfield, Cartoon} está en I, pero {Snoopy, Real} no. Llamamos entonces contexto formal C a la terna (O, A, I) con $I \subseteq OxA$, indicando de esta forma las características que posee cada ítem, es decir que la relación binaria I nos da la incidencia del conjunto de propiedades sobre el conjunto de elementos.

Para un conjunto de elementos $E \subseteq O$, podemos identificar las *propiedades comunes*, escritas como $\sigma(E)$, de la siguiente manera: $\sigma(E) = \{a \in P \mid \forall e \in E : (e, a) \in I\}$. Por ejemplo, $\sigma(\{Garfield, Snoopy\}) = \{Cartoon\}$.

De la misma manera, definimos para $A \subseteq P$ el conjunto de los *elementos comunes*, escritos como $\tau(A)$, así: $\tau(A) = \{e \in O \mid \forall a \in P : (e, a) \in I\}$. Por ejemplo, $\tau(\{Real, Ballena\}) = \{Willy\}$.

Un *concepto* es el par (E, A) de elementos y propiedades tal que $A = \sigma(E)$ y $E = \tau(A)$. Es decir, un concepto es una *colección maximal* de elementos que comparten un conjunto de propiedades en común. El conjunto de elementos de un concepto (el *conjunto* E) es llamado la *extensión*, y las propiedades de un concepto (el *conjunto* A), la *intención*. Debemos mencionar que la extensión de un concepto determina la intención, y la intención determina la extensión. Es ésta, la razón por la que la noción de concepto formal contiene información redundante. Sin embargo, esto es muy útil ya que uno puede elegir cuál de las partes de un concepto debería usar en cada situación. El Cuadro 2 muestra la extensión e intención de cada uno de los conceptos de nuestro ejemplo.

Los conceptos de un contexto dado forman un *orden parcial*. Si definimos un concepto (E_1, A_1), *subconcepto* de (E_2, A_2), tal que $(E_1, A_1) \leq (E_2, A_2)$, si $E_1 \subseteq E_2$, o lo que es lo mismo $A_2 \subseteq A_1$. Un concepto C es un *subconcepto* de D si la extensión de C es un subconjunto de la extensión de D (o lo que es lo mismo si la intención de C es un superconjunto de la intención de

c9	({Garfield, Snoopy, Willy, Lassie}, {∅})
c8	({Garfield, Snoopy}, {Cartoon})
c7	({Willy, Lassie}, {Real})
c6	({Snoopy, Lassie}, {Perro})
c5	({Snoopy}, {Cartoon, Perro})
c4	({Garfield}, {Cartoon, Gato})
c3	({Lassie}, {Real, Perro})
c2	({Willy}, {Real, Ballena})
c1	({∅}, {Cartoon, Real, Perro, Gato, Ballena})

Cuadro 2. Conceptos

D). Si analizamos nuestro ejemplo, el concepto ({Snoopy}, {Cartoon, Perro}) conforma un subconcepto del concepto ({Snoopy, Lassie}, {Perro}). Así, el conjunto de conceptos constituye un *Lattice de conceptos* $\mathcal{L}(\mathcal{T})$. El *lattice* de conceptos de nuestro ejemplo se muestra en la Figura 2.

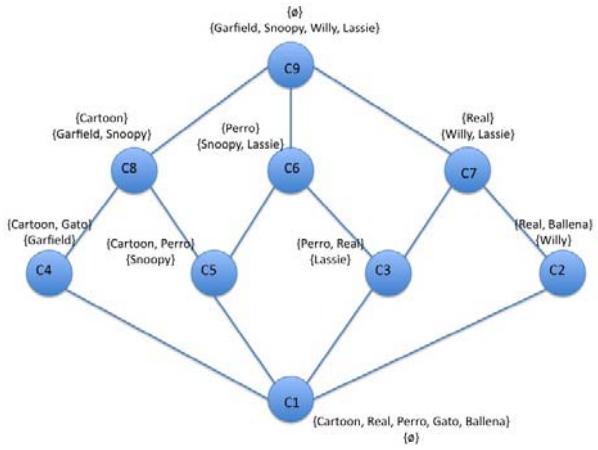

Figura 2. Lattice para el contexto del Ejemplo

En determinadas aplicaciones, el *Lattice* representado en la Figura 2 no es adecuado para llevar a cabo un análisis. Esto se debe a la redundancia de información presente en cada concepto: para cada concepto $C = (A, B)$, los componentes de A estarán presentes en todos los ancestros de C y, simétricamente, los componentes de B estarán contenidos en todos sus descendientes. Por esta razón, frecuentemente se trabaja con una versión reducida que representa la misma información. Para un concepto $C = (A, B)$, sea $N(A)$ los componentes no redundantes en A, y $N(B)$ los componentes no redundantes en B, un elemento e estará en $N(A)$ si el correspondiente concepto C es el mayor límite inferior de los conceptos que contienen e. Simétricamen-

te, una propiedad p se encontrará en $N(B)$ si el concepto C correspondiente es el menor límite superior de los conceptos que contienen p. La propiedad del *Lattice* garantiza la existencia del mayor límite inferior y el menor límite superior [GW99]. Reemplazando los pares (A, B) por $(N(A), N(B))$ en el *Lattice* anterior y eliminando aquellos pares $(N(A), N(B))$ vacíos, obtenemos el *Lattice* reducido que se muestra en la Figura 3. Una notable característica del mismo es que cada elemento y cada propiedad se encuentran presentes en sólo un concepto.

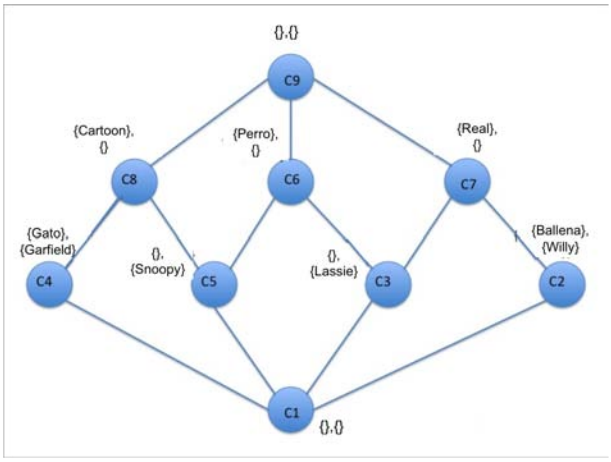

Figura 3. Lattice para el contexto del Ejemplo

La notación utilizada en la Figura 2 se denomina *normal* y la notación utilizada en la Figura 3 se denomina *reducida*.

3. Generación de la guía de lectura

En esta sección, explicaremos cómo generaremos la guía de lectura usando Análisis de Conceptos Formales clasificando las publicaciones enumeradas en el Apéndice A. Hemos elegido esta técnica debido a las ventajas que la misma provee para agrupar elementos basados en sus propiedades. Esto resulta vital para identificar publicaciones con características similares.

Una vez, diseñada la guía, explicaremos cuáles son las estrategias de búsqueda sugeridas para el desarrollador.

Debemos destacar que las publicaciones enumeradas en el Apéndice A son metodologías del proceso de reingeniería de aplicaciones orientadas a objetos que usan el Análisis de Conceptos Formales como herramienta base para el anásis de código.

Para construir la guía de lectura, se realizarán los siguientes pasos:

1. Construcción del contexto binario y *lattice*
2. Extracción de las jerarquías
3. Construcción de la guía a partir de las jerarquías
4. Algoritmo de búsqueda

3.1. Construcción del contexto binario y *lattice*

Como estamos interesados en construir una guía de lectura, nuestro contexto binario está compuesto por elementos, O, que son las publicaciones, y las propiedades P, son las palabras claves identificadas en cada una de ellas. La relación binaria R resulta del tipo "contiene". Por ejemplo nuestra publicación: [CM09] "contiene" Análisis de Conceptos Formales.

A continuación listamos las publicaciones analizadas para el estado del arte. El tag enumerado entre paréntesis es el utilizado como elementos de la tabla de incidencia.

- Reverse Engineering Aspectual Views using Formal Concept Analysis (314769) [TM04]
- Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach (Arevalo2006) [AFHN06]
- Reverse Engineering of use case realizations in UML (BV00) [BV00]
- Clustering and Concept Analysis for Software Evolution (davey) [DB01]
- ARES, Adding a class and REStructuring Inheritance Hierarchy (DDHL95) [DDHL95]
- Revealing Java Class Structure with Concept Lattices (DG03) [DG03]
- Aiding Program Comprenhension by Static and Dynamic Feature Analysis (EKS01a) [EKS01]
- Feature-Driven Program Understanding Using Concept Analysis of Execution (EKS01b) [EEKS01]
- Locating Features in Source Code (EKS03) [EKS03]
- Conceptual Code Mining: Mining for Source-Code Regularities with Formal Concept Analysis (esug2004) [MT04]
- Types and Concept Analysis for Legacy Systems (KM00) [KM01]
- On the Inference of Configuration Structures from Source Code (KS94) [KS94]
- Using FCA to Suggest Refactorings to Correct Design Defects (Moha06-CLA) [MRG+06]
- Applying Concept Formation Methods to Object Identification In Procedural Code (SMLD97) [SMLD97]
- A Scalable Approach to User-session based Testing of Web Applications through Concept Analysis (SMSP04) [SMSP04]
- Aspect Mining throught the Formal Concept Analysis of Execution Traces (TC04) [TC04]
- Concept Analysis for Module Restructuring (TON01) [Ton01]
- Multi-Dimensional Concerns Mining for Web Applications via Concept-Analysis (v4-34) [BMT05]

- Identifying Object using Cluster and Concept Analysis (vdk99) [vDK98]

Las *palabras claves* utilizadas en la tabla de incidencia son los siguientes (en orden alfabético): abstract data type, aspect mining, aspect restructuring, aspect reverse engineering, aspectual views from object-oriented source code, class hierarchies, cluster analysis, component retrieval, concept analysis., concept formation, concept lattice, concerns mining, configuration structures, crosscutting concerns, database schema, dendogram, design defects, dynamic analysis, encapsulation, execution traces, factorizing, feature location, feature-component correspondence, framework, galois lattice, impact analysis, inheritance hierarchy, Java, lattice, legacy systems, model-driven, modularization, multi-dimensional separation of concerns, object identification, Object-oriented systems, procedural code, program analysis, program comprehension, program understandig, re-modularization, reengineering, refactoring, restructuring, reverse engineering, schema design, schema evolution, software architecture recovery, software classification, software clustering, software evolution, software maintenance, software migration., source code, source-code mining, testing, transformation, type inference, UML, UML modeling, use cases, web applications.

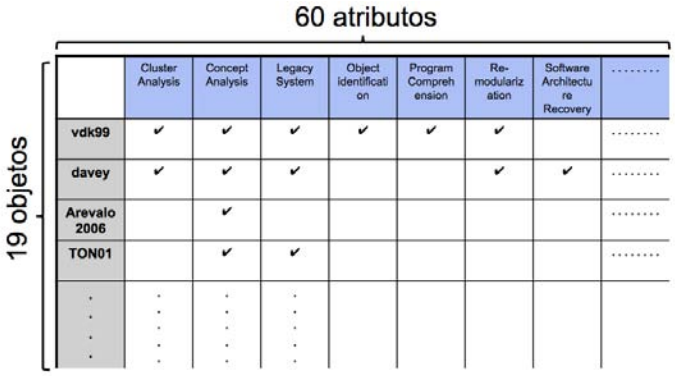

Figura 4. Tabla Incidencia

La Figura 4 muestra la Tabla de incidencia del mismo (entorno parcial) y la Figura 5, el *lattice* resultante a partir de la tabla de incidencia. Para la generación del *lattice* hemos utilizado la herramienta *Galicia 3.0* [Gal].

3.2. Extracción de Jerarquías

El *lattice* generará un conjunto de jerarquías, que luego se usarán para construir la guía de lectura. Las jerarquías se obtienen a partir de los nodos que son hijos del nodo *top*. De este modo, los nodos hijos representan las

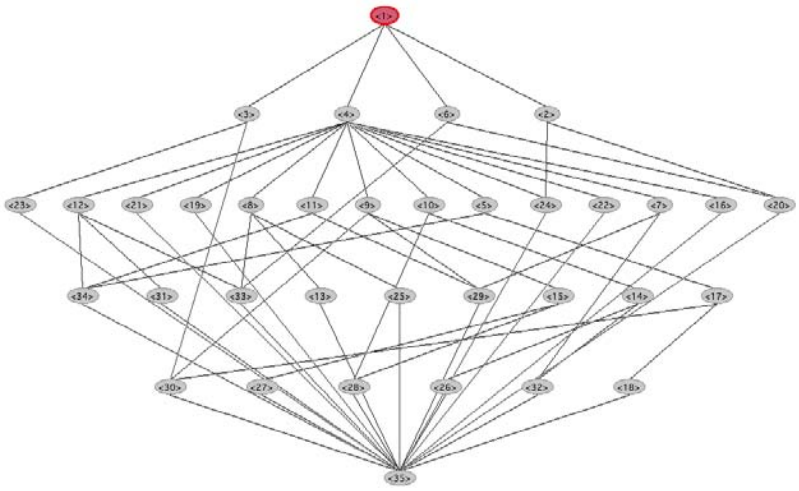

Figura 5. Lattice resultante

raíces de las jerarquías resultantes. A su vez, los nodos que tienen como hijo el nodo *bottom* pasan a representar los nodos terminales de las respectivas jerarquías. Las Figuras 7 a 17 muestran las estructuras de las jerarquías obtenidas a partir del *lattice* (Figura 5).

Para analizar las jerarquías obtenidas en el contexto del *lattice* resultante, hemos considerado los siguientes puntos de partida:

1. El nodo *top* y el nodo *bottom* no participan en el análisis de las jerarquías
2. La lectura de las jerarquías se hace en sentido *top-down*
3. Usaremos un etiquetado combinado en los conceptos: la notación normal para la extensión (conjunto de publicaciones) y la notación reducida para la intención (palabras claves de las publicaciones).

De esta manera podemos distinguir en la Figura 5, que los nodos candidatos son 3, 4, 6 y 2. Como se puede observar, en la Figura 6, el nodo 4 tiene un número considerable de hijos, por esta razón decimos dividir la grafica en partes para poder apreciar con más detalle el contenido de los mismos (Figuras 8 a 15). El conjunto reducido de propiedades del nodo 4 está formado por $A = \{ConceptAnalysis\}$, mientras que el conjunto de elementos de este nodo contiene todas las publicaciones relacionadas al conjunto de atributos mencionado anteriormente. Los nodos hijos contienen mayor presición, es decir tienen información más específica con respecto al nodo 4. La lectura del mismo, se formula de la siguiente manera: no existe ninguna publicación que desarrolle sólo el tema de Análisis de Conceptos Formales, sin embargo, existen varias que lo contemplan dentro su análisis.

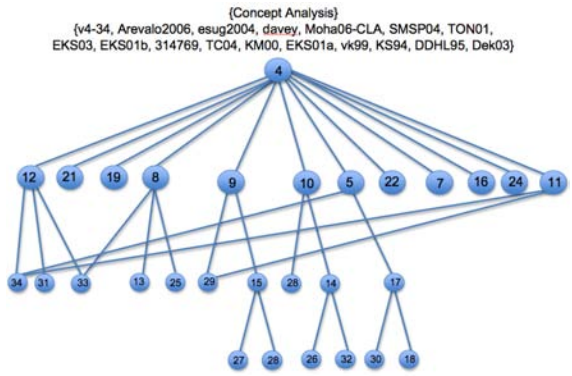

Figura 6. Nodo 4, con todos sus hijos

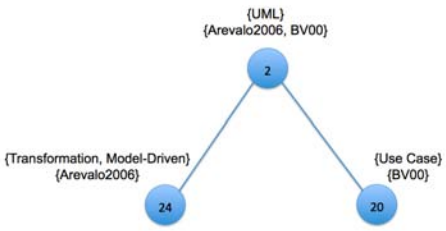

Figura 7. Jerarquía del Nodo 2

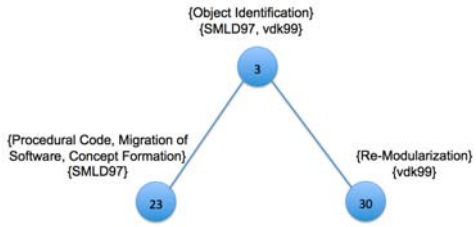

Figura 8. Jerarquía del Nodo 3

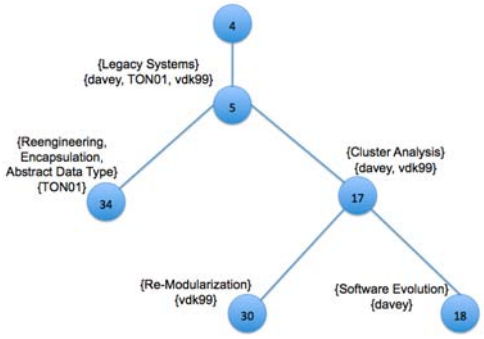

Figura 9. Jerarquía del Nodo 4

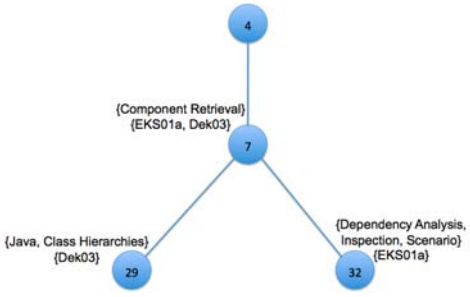

Figura 10. Jerarquía del Nodo 4

A continuación podemos ver otro ejemplo en la Figura 15, donde observamos aquellos nodos que tienen como padre el nodo 4 pero no poseen hijos, es decir, en el *lattice* original (Figura 5) terminaban en el nodo *bottom*.

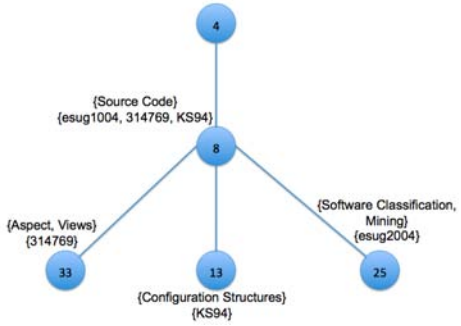

Figura 11. Jerarquía del Nodo 4

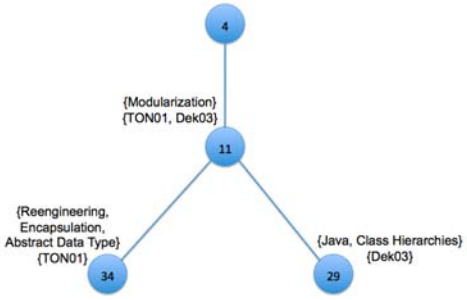

Figura 12. Jerarquía del Nodo 4

Los nodos terminales (es decir, nodos que tiene como hijo único el nodo *botton*) y en general cualquier otro nodo por debajo de la raíz, no sólo poseen la información contenida en ellos, sino que además tienen la información de todos sus antecesores.

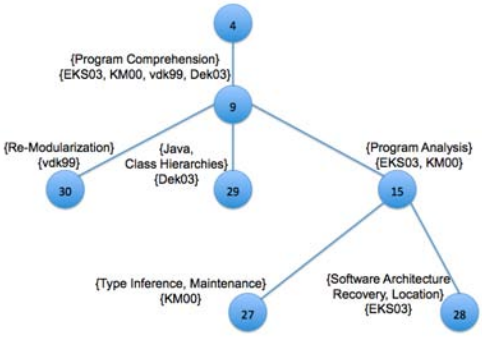

Figura 13. Jerarquía del Nodo 4

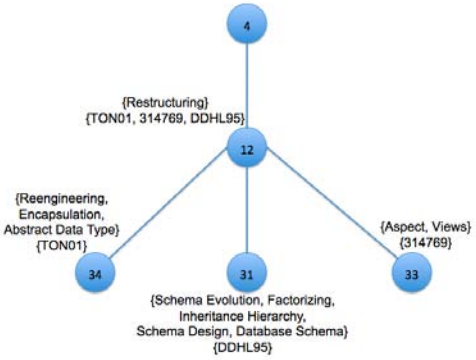

Figura 14. Jerarquía del Nodo 4

3.3. Construcción de la guía de lectura con las estrategias de búsqueda

La guía de lectura que se propone a partir del *lattice* resultante tiene la metáfora de "sigue tu propia aventura" [1] (la idea básica del hipertexto), donde se propone como raíz un tema general, y luego en base a la elección del usuario se guía la lectura a temas más específicos.

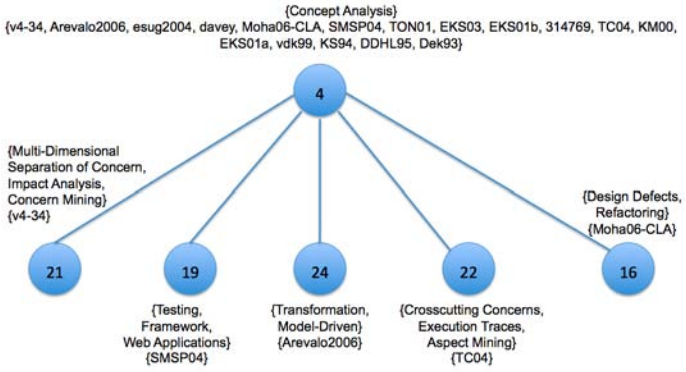

Figura 15. Nodo 4, con nodos terminales

Lo que le proponemos es la lectura de un conjunto de publicaciones desde dos puntos de vista:

1. sugerirle al usuario un conjunto de palabras claves desde las cuales puede empezar la búsqueda
2. que el usuario proponga la palabra clave sobre la cual quiere buscar.

Explicaremos como se desarrollan ambas búsquedas.

(1) Se le sugiere al usuario un conjunto de palabras claves desde las que se puede iniciar la búsqueda. Este conjunto se desprende de las intenciones con notación reducida de las propiedades de los nodos hijos del conjunto *top* (nodos 3, 4, 6 y 2). Estas palabras claves son: {*Object Identification, Concept Analysis, Object-Oriented, Reverse Engineering, UML*}

A través de un ejemplo mostraremos la navegación por una jerarquía. Tomamos como punto de partida el nodo 6, que tiene como intención reducida las palabras claves {*Object Oriented-Reverse Engineering*}, la extensión de este nodo muestra las publicaciones que analizan las palabras claves mencionadas anteriormente.

[1] La metáfora relacionada con los libros para niños y adolescentes, especialmente de detectives, donde al final de la hoja se debe decidir por dónde continuar la lectura de la aventura entre diversas opciones y que según lo elegido se debe ir a una página u otra.

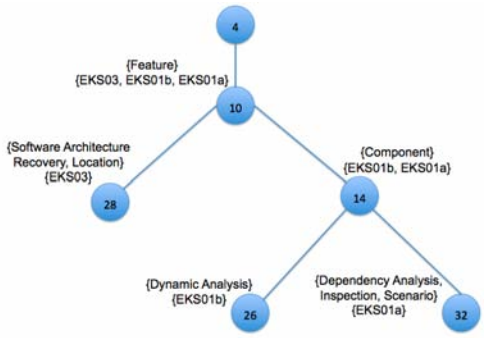

Figura 16. Una jerarquía que nace del Nodo 4

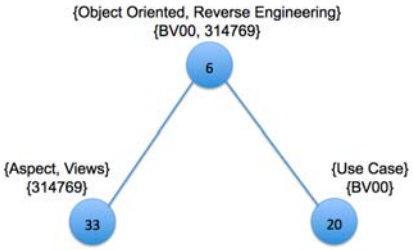

Figura 17. Jerarquía del Nodo 6

Se le pregunta al usuario si le interesa profundizar más en estos temas. Si su respuesta es negativa, se le ofrece un conjunto de publicaciones {*[BV00],[314769]*}, que tratan los temas relacionados con las palabras claves {*Object-Oriented, Reverse Engineering*}, es decir la extensión con notación completa del concepto 6.

Si su respuesta es afirmativa, se le ofrecen otros temas que están contenidos en la intensión (con notación reducida) de los nodos hijos. En el ejemplo, las opciones para continuar la navegación es a través del nodo 33 o 20. Los temas que se proponen, para seguir profundizando, a partir del nodo 33 son {*Aspect, Views*}, por otro lado la propuesta a partir del nodo 20 es {*Use Case*}.

Dado que esta jerarquía no tiene otro nivel de búsqueda, el usuario está condicionado a elegir algunos de los temas mencionados o iniciar nuevamente la búsqueda.

En cualquiera de los casos, ya sea que el usuario eliga los temas del nodo 33 o 20, las publicaciones asociadas a esos nodos incluyen el tratamiento de todos los temas contenidos en sus antecesores, es decir cada publicación trata o bien el conjunto de temas de {*Object-Oriented, Reverse Engineering, Aspect, Views*} o el conjunto de temas {*Object-Oriented, Reverse Engineering, Use Case*}. Esta característica en los conceptos se obtiene por la forma en que calculan para formar el lattice.

Para automatizar esta búsqueda proponemos un algoritmo, la Figura 18 muestra la implementación del mismo desarrollada en *Java*.

(2) El usuario propone una palabra clave sobre la que desea realizar una búsqueda. Se utiliza un algoritmo que recibe como parámetro de entrada la palabra clave solicitada por el usuario. Si encuentra la misma retorna las publicaciones asociadas a este tema. Si no la encuentra alerta al usuario que la palabra buscada no está contenida en el *lattice*.

Para automatizar la búsqueda de publicaciones en el lattice resultante, proponemos un algortimo basado en la teoría de *Búsqueda en Anchura*.

```
public static String firstSearchAlgorithm (Lattice lattice, int positionNode){

    Queue <Node> queue = new LinkedList<Node>();

    List <Integer> childList = new ArrayList<Integer>();

    Node nodeActive = lattice.getNode(positionNode);

    queue.insertIfPosible(nodeActive);

    while(! queue.isEmpty()){

        childList = lattice.getAdjacentList(queue.peek());

        if(childList.size()!=0){

            for (int i = 0; i < childList.size(); i++) {

                Node node = lattice.getNode(childList.get(i));
                HashMap <Integer, Keywords> keywordList = keywordList.add(node.getPosition, node.getKeyword());

            }
        } else{ // No tiene hijos. Es un nodo hoja

            return "La busqueda alcanzo un nodo terminal (hoja), los papers asociados a este nodo son: "
                    + (queue.poll()).getPapers();
        }

        HashMap <Integer, boolean> decisionMap = takeDecision(keywordList); // Metodo takeDecision() resuelve si desea seguir la busqueda, o quedarse y por qué palabra seguir

        Node nodeDecision = lattice.getNode(decisionMap.getkey(0)); // El integer que me devuelve el HashMap es la posicion del nodo

        if(! decisionMap.getValue(0)){      // Si decision=false (no quiere seguir adelante con la busqueda)

            if(decisionMap.getKey(0) != -1){  //y  si eligió una palabra en la que quedarse


                return  "Papers asociados a la keyword elegida: " + nodeDecision.getPapers();   // devuelvo las publicaciones de esa palabra

            } else { // decidió no seguir adelante y no quería ninguna de las opciones

                return "Papers asociados al antecesor: " + (queue.poll()).getPapers();  // Entonces se le devuelven las publicaciones del padre, que fue su ultima elección
            }

        } else{ //si la decision = true (quiere seguir con la busqueda)


            queue.poll();
            queue.add(nodeDecision);            // Con la palabra clave elegida pongo en la cola el nodo asociado
        }
    }
    return "";
}
```

Figura 18. Algoritmo de búsqueda para el primer caso

Formalmente, la *búsqueda en anchura* es un algoritmo de búsqueda sin información, que examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística. Basándonos en esta teoría formalizamos un algoritmo de búsqueda en anchura que permite localizar una o más publicaciones a través de palabras claves. La Figura 19 muestra la implementación del mismo en *Java*.

```
public static String secondSearchAlgorithm (Lattice lattice, int positionNode, KeyWord keyToSearch){
    boolean [ ] visit = new boolean[ lattice.size() ];
    Queue <Node> queue = new LinkedList<Node>();
    List <Integer> childList;
    visit[positionNode] = true;
    Node nodeActive = lattice.getNode(positionNode);
    queue.insertIfPosible(nodeActive);
    while(!queue.isEmpty()){
        System.out.println(" The first is: "+ queue.peek());
        if(nodeActive.hasKeyWord(keyToSearch)){
                return nodeActive.getId() + nodeActive.getPapers();
        }
         childList = lattice.getAdjacentList(queue.poll());
        if(childList.size()!=0){
                for (int i = 0; i < childList.size(); i++) {
                 if(!visit[childList.get(i)]){
                        queue.insetIfPosible(lattice.getNode(childList.get(i)));
                         visit[childList.get(i)] = true;
                 }
                }
        }
    }
    return "The keyword not found";
}
```

Figura 19. Algoritmo de búsqueda para el segundo caso

Para comprender el funcionamiento de este algoritmo en la Figura 20 se muestra el desarrollo paso a paso del mismo.

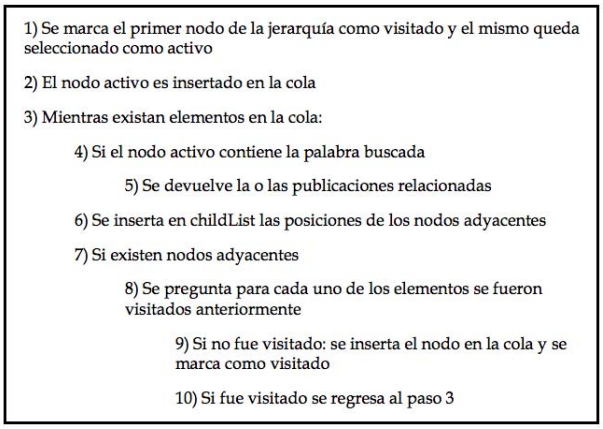

Figura 20. Funcionamiento del algortimo

4. Conclusiones y Trabajos Futuros

En este trabajo presentamos el desarrollo de una guía de uso (con estrategias de búsqueda) de publicaciones en el estado del arte de metodologías basadas en el Análisis de Conceptos Formales, el cual permite guiar la investigación de un tema a elección.

Es importante remarcar las dos propuestas, que hacemos, sobre la lectura de un conjunto de publicaciones. El primer punto de vista que se propone a partir de *lattice* resultante tiene la metáfora de *"sigue tu propia aventura"*. El segundo punto de vista consiste en que el usuario proponga la palabra clave sobre la cual quiere buscar. Para este propósito fue desarrollado un algoritmo de búsqueda.

Una dificultad que encontramos en el desarrollo de este trabajo fue que las metodologías analizadas abarcan en su gran mayoria el proceso de *Reverse Engineering*, dejando de lado el proceso de *Forward Engineering*. Con esto concluimos que el Análisis de Conceptos Formales es más apropiado de aplicar en el proceso de *Reverse engineering* que para el de *Forward*.

Una ventaja obtenida fue la facilidad de la construcción de jerarquías a partir del *lattice*. Una desventaja encontrada fue que las jerarquías están limitadas al dominio elegido.

Durante el desarrollo de este trabajo de grado surgieron muchas nuevas ideas que podrían ser implementadas e investigadas. A continuación se detallan aquellas que resultan más interesantes para su desarrollo en trabajos futuros:

- Desarrollar una herramienta visual que nos permita la navegación.
- Clasificar las publicaciones analizadas por conferencia en la que publicada, por año, o por autores, y evaluar los respectivos resultados.

- Proponer nuevas guías de lectura

Referencias

AFHN06. Gabriela Arévalo, Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Building abstractions in class models: Formal concept analysis in a model-driven approach. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2006.

BMT05. Carlo Bellettini, Alessandro Marchetto, and Andrea Trentini. Multidimensional concerns mining for web applications via concept-analysis. In Cemal Ardil, editor, *WEC (2)*, pages 129–132. Enformatika, Çanakkale, Turkey, 2005.

BV00. Dragan Bojic and Dusan Velasevic. Reverse engineering of use case realizations in uml. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 741–747, New York, NY, USA, 2000. ACM.

CCI90. Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

DB01. John Davey and Elizabeth Burd. Clustering and concept analysis for software evolution. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 146–149, New York, NY, USA, 2001. ACM.

DDHL95. H. Dicky, C. Dony, M. Huchard, and T. Libourel. Ares, adding a class and restructuring inheritance hierarchy, 1995.

DG03. Uri Dekel and Yossi Gil. Revealing class structure with concept lattices. *Reverse Engineering, Working Conference on*, 0:353, 2003.

EEKS01. Traces Thomas Eisenbarth, Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Feature-driven program understanding using concept analysis of execution. In *In 9th International Workshop on Program Comprehension*, pages 300–309. IEEE Computer Society Press, 2001.

EKS01. Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis, 2001.

EKS03. Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code, 2003.

Gal. Galicia 3.0. `http://sourceforge.net/projects/galicia/`.

GW99. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

KM01. T. Kuipers and L. Moonen. Types and concept analysis for legacy systems, 2001.

KS94. Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *In Proceedings of the 16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society Press, 1994.

MRG+06. Naouel Moha, Jihene Rezgui, Yann-Gaël Guéhéneuc, Petko Valtchev, and Ghizlane El-Boussaidi. Using fca to suggest refactorings to correct design defects. In Sadok Ben Yahia, Engelbert Mephu Nguifo, and Radim Belohlávek, editors, *CLA*, volume 4923 of *Lecture Notes in Computer Science*, pages 269–275. Springer, 2006.

MT04. Kim Mens and Tom TourwŽ. Conceptual code mining - mining for source-code regularities with formal concept analysis. *Journal*, 2004, 2004.

SMLD97. H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying concept formation methods to object identification in procedural code. *Automated Software Engineering, International Conference on*, 0:210, 1997.

SMSP04. Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society.

TC04. Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society.

TM04. Tom Tourwé and Kim Mens. Reverse engineering mining aspectual views using formal concept analysis. In *SCAM*, pages 97–106, 2004.

Ton01. Paolo Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, 2001.

vDK98. Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *234*, page 19. Centrum voor Wiskunde en Informatica (CWI), 30 1998.

A. Publicaciones Analizadas

En este apéndice presentaremos las metodologías analizadas (con una breve descripción) para poder desarrollar la contribución de nuestro trabajo. El identificador listado en cada publicación son los utilizados como el conjunto de elementos y las palabras claves son las propiedades de la tabla de incidencia para construir las guías de lectura.

A.1. Identifying Object using Cluster and Concept Analysis [vDK98]

Descripción: Este artículo propone una metodología para identificar objetos mediante una reestructuración semi-automática de la estructura de datos de un sistema *legacy*. Se aplica en un sistema desarrollado en COBOL, y utilizan los resultados para contrastar las técnicas de *Cluster y Concept Analysis*. En una aplicación escrita en COBOL, los datos almacenados y procesos representan la parte central del sistema, y ésta es la razón por la que en COBOL la estructura de registro de datos son los puntos para empezar con la identificación de objetos.
Etapa de reingeniería en la que participa: Reverse Engineering
Lenguaje de Aplicación: Sobre un sistema de 100,000 lineas de código en COBOL llamado Mortgage
Autor(es): Arie van Deursen, Tobias Kuipers - CWI, Amsterdam
Palabras Claves: legacy system, program comprehension, object identification, re-modularization, cluster analysis, concept analysis.

A.2. Reverse Engineering of use case realizations in UML [BV00]

Descripción: Es una técnica para recuperar elementos de modelos UML de un sistema de software. Esto incluye relaciones entre casos de uso, así como los roles de las clases en colaboración en cada caso de uso, identificando funcionalidades en común y estableciendo una vista jerárquica del modelo. Esta técnica está basada en el análisis dinámico de sistemas para seleccionar casos de usos que cubran casos de uso relevantes.
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Dragan Bojic, Dusan Velasevic.
Palabras Claves: Object-oriented systems, UML modeling, reverse engineering, use cases.

A.3. Aiding Program Comprenhension by Static and Dynamic Feature Analysis [EKS01]

Descripción: Es una técnica de construcción de mapeo entre el comportamiento visible externo de un sistema y las partes relevantes del código fuente. Combina análisis estático y dinámico para centrarse rápidamente

en las partes de un sistema que lo requiera y que tenga como objetivo un proceso dirigido a la comprensión del programa. Esta publicación describe un proceso y su aporte a la técnica para identificar un conjunto específico de características relacionadas. El proceso está automatizado en gran parte. Combina análisis estático y dinámico y usa Análisis de Conceptos Formales para obtener la correspondencia entre las características y los componentes.
Lenguaje de Aplicación: se aplica sobre dos navegadore: Mosaic y Chimera
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Thomas Eisenbarth, Rainer Koschke, Daniel Simon
Palabras Claves: feature, scenario, component, concept analysis, inspection, dependency analysis

A.4. Feature-Driven Program Understanding Using Concept Analysis of Execution [EEKS01]

Descripción: La primera tarea de un programador que quiere entender cómo se implementa una función, es localizar la implementación de un *feature* en el código. En el caso de grandes sistemas que están mal documentados, localizar funciones en el código, identificar aspectos comunes y variaciones de los componentes, así como también funciones, resulta difícil y un aumento en el consumo de tiempo. La función de componentes que describe la correspondencia de *feature-component* es necesaria para la ejecución de un conjunto de *features*, puntos comunes y variaciones de *features* y componentes. El artículo describe una técnica rápida para determinar la correspondecia entre *feature-component* basada en información dinámica y análisis de conceptos.
Lenguaje de Implementación: Prototipo para un ambiente en Unix, C , Perl script
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Thomas Eisenbarth, Rainer Koschke, Daniel Simon
Palabras Claves: feature-component correspondence, dynamic analysis, concept analysis.

A.5. Concept Analysis for Module Restructuring [Ton01]

Descripción: Este artículo propone un nuevo enfoque al usar Análisis de Conceptos Formales para la reconstrucción de módulos basandose en el cáculo de la extensión de subparticiones de conceptos. El bajo acoplamiento entre módulos y la alta cohesión dentro de cada módulo son las claves para un buen diseño de software. Esto se consigue encapsulando los detalles de la estructura interna de datos y exportando sólo las funciones públicas con una interfaz limpia. Si la modularización es inadecuada, puede provocar que el mantenimiento de *legacy systems* existentes sea caro y dificultoso. Mejorar la estructura modular de un programa es una forma de mantenimiento

preventivo ya que, por lo general, es necesario a medida que el sistema va incorporando nuevas versiones.

Lenguaje de Implementación: Lo aplican en un sistema hecho en C
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Paolo Tonella
Palabras Claves: concept analysis, modularization, encapsulation, abstract data type, legacy systems, reengineering, restructuring

A.6. A Scalable Approach to User-session based Testing of Web Applications through Concept Analysis [SMSP04]

Descripción: Un enfoque prometedor para las pruebas de funcionalidad de aplicaciones web, es aquel que aprovecha los datos de las sesiones de usuarios obtenidas por los servidores web. Este enfoque genera automáticamente casos de prueba basados en perfiles de usuarios reales. La contribución clave de este artículo es la aplicación de Análisis de Conceptos Formales para agrupar las sesiones de usuarios, para así reducir los casos de pruebas. Lograron automatizar completamente el proceso de recolección y reducción de sesiones a través de la repetición.

Estapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Sreedevi Sampath, Valentin Mihaylov, Amie Souter y Lori Pollock
Palabras Claves: concept analysis, web applications, testing, framework

A.7. Multi-Dimensional Concerns Mining for Web Applications via Concept-Analysis [BMT05]

Descripción: Como las aplicaciones web son cada vez más complejas, la comunidad científica ha enfocado su atención en el diseño, desarrollo, análsis y testing de las dichas aplicaciones, estudiando y proponiendo metodologías y herramientas. Este artículo propone un enfoque para automatizar *concern minig* multi-dimensional para aplicaciones web, badado en el Análisis de Conceptos Formales, Análisis de Impacto, e identificación de *concerns* basado en *tokens*. Este aporte permite a los usuarios analizar y generar la traziabilidad de software (web) importante a *concern* particulares, a través de la separación multi-dimensional de concerns. La técnica fue desarrollada en un contexto de *WAAT (Web Applications Analysis and Testing)*.

Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Carlo Belletini, Alessandro Marchetto, and Andrea Trentini
Palabras Claves: concept analysis, concerns mining, multi-dimensional separation of concerns, impact analysis.

A.8. Using FCA to Suggest Refactorings to Correct Design Defects [MRG+06]

Descripción: Los defectos en el diseño suelen resultar en un difícil mantenimiento del software. Esta es la razón por la que su detección y corrección son pasos claves en el proceso de lograr una alta calidad del software. En este artículo se propone un novedoso enfoque para la detección y eliminación de los mismos en programas Orientados a Objetos, en el que se combinan la efectividad de las métricas con el Análisis de Conceptos Formales.
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Nouel Moha, Jihene Rezgui, Yann-Gaél Guêhêneuc, Petko Valtchev, and Ghizlane El Boussaidi
Palabras Claves: design defects,concept analysis, refactoring

A.9. Applying Concept Formation Methods to Object Identification In Procedural Code [SMLD97]

Descripción: Los sistemas *legacy* suelen presentar un alto nivel de entropía combinada con una documentación imprecisa. Esta situación provoca que el mantenimiento sea bastante díficil, demande mucho tiempo, y a la vez, sea costoso. Este artículo describe un aporte computarizado enfocado a hacer de soporte en la migración de sistemas procedurales a sistemas Orientados a Objetos. El código procedural no contiene representaciones explícitas de objetos, contiene únicamente variables globales, estructuras de datos (*records*) y rutinas (funciones y procedimientos).
Lenguaje de Implementación: COBOL.
Lenguaje de Aplicación: En un sistema de larga-escala escrito en C.
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and Francois Dumont.
Palabras Claves: procedural code, concept formation, object identification, software migration.

A.10. Types and Concept Analysis for Legacy Systems [KM01]

Descripción: En este artículo combinarán la inferencia de tipos y el Análisis de Conceptos Formales con el fin de conocer mejor los sistemas *legacy*. Se desarrolla una herramienta llamada *ConceptRefinery*, para poder manipular conceptos. Se demuestran como esta herramienta facilita experimentos con el Análisis de Conceptos Formales.
Lenguaje de Aplicación: sobre un sistema de 100,000 lineas de código en COBOL - El sistema estudiado se llama Mortgage
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Tobias Kuipers and Leon Moonen
Palabras Claves: software maintenance, program understandig, program analysis, type inference, concept analysis

A.11. Locating Features in Source Code [EKS03]

Descripción: La comprensión de la implementación de una determinado emphfeature de un sistema requiere la identificación de las unidades de cómputo del sistema que contribuyen a esta función. Se presenta una técnica semiautomática que reconstruye el mapeo de los *features* que se activan por el usuario y exhibe una conducta observable. La técnica combina análisis dinámico y estático que se centran en la rapidez de las partes del sistema, haciendo referencia a un conjunto específico de *features*.
Etapa de reingeniería en la que participa: Reverse Engineering
Lenguaje de Aplicación: dos navegadores web analizados: Mosaic y Chimera
Autor(es): Thomas Eisenbarth, Rainer Koschke, Daniel Simon
Palabras Claves: program comprehension, concept analysis, feature location, program analysis, software architecture recovery.

A.12. Reverse Engineering Aspectual Views using Formal Concept Analysis [TM04]

Descripción: La técnica se desarrolló con un experimento inicial usando la técnica de Análisis de Conceptos Formales para ingeniería inversa sobre vistas de aspectos de código orientado a objetos. Una vista de aspecto es un conjunto de entidades de código fuente, como por ejemplo jerarquía de clases, clases y métodos, que están estructuralmente y transversalmente relacionadas en una aplicación en particular. Inicialmente la técnica sólo consideró los nombres de las clases y métodos.
Etapa de reingeniería en la que participa: Reverse Engineering
Lenguaje de Aplicación: Smalltalk.
Autor(es): Kim Mens, Tom Tourwé
Palabras Claves: aspect reverse engineering, aspect restructuring, aspectual views from object-oriented source code, concept analysis

A.13. Revealing Java Class Structure with Concept Lattices [DG03]

Descripción: El trabajo de esta tesis sugiere, por primera vez, que la técnica de Análisis de Concepto Formales sea aplicada a las clases individuales. La técnica en un intento de clasificar automáticamente las características de estas clases, facilitando su reutilización, y revelando su estructura interna, con fines de garantizar calidad o reingeniería.
Etapa de reingeniería en la que participa: Reverse Engineering
Lenguaje de Aplicación: Java
Autor(es): Uri Dekel
Palabras Claves: java, lattice, concept analysis, reverse engineering, modularization, component retrieval, class hierarchies, program comprehension

A.14. Aspect Mining throught the Formal Concept Analysis of Execution Traces [TC04]

Descripción: La presencia de *crosscutting concerns* es uno de los mayores problemas en la comprensión y evolución del software. Son difíciles de localizar (dispersión - *scattering*) y pueden llevar a múltiples problemas (enredos - *tangling*). La Programación Orientada a Aspectos, ofrece mecanismos para separarlos en un módulo, llamado aspecto.
Lenguaje de Implementación: El caso de estudio es una implementación Java del algoritmmo de Dijkstra
Lenguaje de Aplicación: Desarrollaron una herramienta *Dynamo* en Java
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): Paolo Tonella and Mariano Ceccato
Palabras Claves: aspect mining, concept analysis, crosscutting concerns, execution traces

A.15. Clustering and Concept Analysis for Software Evolution [DB01]

Descripción: Tanto el Cluster de software como el Análisis de Conceptos son técnicas que se usan para determianr la estuctura de los sistemas de software. Este artículo propone el uso de estas técnicas para contribuir al estudio de la evolución del software. Si se aplican éstas técnicas en distintas versiones del sistema, posiblemente podamos descubrir sus tendencias evolutivas a lo largo de su vida útil. Estas dos técnicas comenzaron ser populares en los útlimos diez años gracias a la reeingeniería. En este artículo se describen técnicas básicas para la aplicación de *Cluster y Concept Analysis*.
Etapa de reingeniería en la que participa: Reverse Engineering.
Autor(es): John Davey and Elizabeth Burd
Palabras Claves: software clustering, concept analysis, software evolution, legacy systems, dendogram, concept lattice.

A.16. Conceptual Code Mining: Mining for Source-Code Regularities with Formal Concept Analysis [MT04]

Descripción: En particular, conociendo por dónde empezar, el proceso de comprensión es más difícil de lo que parece, sobre todo cuando es un sistema grande y complejo y el tiempo es escaso. Se propone un enfoque de un sistema de código fuente de forma automática y eficiente para los conceptos de interés. Lo que se denomina regularidades código fuente: a lo que se refiere direcciones en el código, qué patrones, idiomas de programación y convenciones adoptadas, además de dónde y cómo fueron implementadas.
Etapa de reingeniería en la que participa: Reverse Engineering
Lenguaje de Aplicación: SmallTalk
Autor(es): Kim Mens-Tom Tourwé

Palabras Claves: software classification, source-code mining, formal concept analysis.

A.17. Building Abstractions in Class Models: Formal Concept Analysis in a Model-Driven Approach [AFHN06]

Descripción: Diseñar modelos de clases es usualmente un proceso iterativo para detectar cómo expresar, para un domino específico, el concepto adecuado y sus relaciones. Durante estas iteraciones, la abtracción de conceptos y sus relaciones son un paso importante. Se propone automatizar este proceso de abtracción usando una técncias basadas en Análisis de Conceptos Formales en un modelo impulsado por un contexto. Usando modelos de clases UML 2.0 como lenguaje modelador de clases, se propone mostrar cómo el enfoque del modelo desarrollado permite parametrización, seguimiento y generalización para cualquier meta-modelo para expresar modelos de clases.
Etapa de reingeniería en la que participa: Reverse Engineering
Autor(es): Gabriela Arévalo, Jean-Rémi Falleri, Marianne Huchard, Clémentine Nebut
Palabras Claves: model-driven, concept analysis, UML, transformation

A.18. ARES, Adding a class and REStructuring Inheritance Hierarchy [DDHL95]

Descripción: En base de datos orientadas a objetos, los esquemas de diseño o evolución deben beneficiarse tanto como se posible de las herramientas automáticas. El enfoque se hace sobre la inserción automática de clases dentro jerarquías, preservando la *máxima factorización* de las propiedades de las clases. Se describe un algoritmo incremental cuyas propiedades de los factores pueden ser usadas para construir, reorganizar o mantener uan jerarquía.
Etapa de reingeniería en la que participa: Reverse Engineering
Autor(es): H. Dicky, C. Dony, M. Huchard, T Libourel
Palabras Claves: database schema, inheritance hierarchy, schema design, schema evolution, galois lattice, restructuring, factorizing

A.19. On the Inference of Configuration Structures from Source Code [KS94]

Descripción: Se aplica el Análisis de Conceptos Formales al problema de inferir estructuras de configuración a partir de código fuente existente. El algoritmo computa un *lattice* de conceptos que permite conocer la estructura y las propiedades de las posibles configuraciones. El *lattice* no sólo visualiza las dependencias entre los *threads* de configuración, sino que también visualiza la calidad general de la configuración de las estructuras de acuerdo con los principios de la ingeniería de software.

Etapa de reingeniería en la que participa: Reverse Engineering.
Lenguaje de Implementación: Aplicaron su herramienta sobre varios programas UNIX
Autor(es): Maren Krone and Gregor Snelting
Palabras Claves: formal concept analysis, source code, configuration structures, concept lattice