Centro de Investigación en Matemáticas, A.C.

CIMAT

Evaluando el uso de técnicas de procesamiento de lenguaje natural y métricas de centralidad para la detección de patrones de diseño de software

TESIS

Que para obtener el grado de

Maestro en Ingeniería de Software

P r e s e n t a

Juan Manuel Mauricio Zamarrón

Director(a) de Tesis

Dra. Perla Velasco Elizondo

Dra. Alejandra García Hernandez

Zacatecas, Zacatecas., 4 de diciembre de 2015.

Resumen

El diseño de un sistema está formado por un conjunto elementos organizados en diversas estructuras. Muchas de estas estructuras corresponden a patrones de diseño. La identificación de patrones en un sistema legado proporciona a los ingenieros de software una visión sobre su estructura y comportamiento aun cuando ninguna documentación del diseño exista.

Existen diversas herramientas que pueden ser utilizadas para soportar la identificación de patrones de diseño. Sin embargo éstas en muchos casos solo funcionan para sistemas codificados en un lenguaje de programación específico, solo detectan un subconjunto de patrones y no extensibles a la inclusión de nuevos.

En este trabajo se evalúa la pertinencia de combinar el uso de técnicas procesamiento de lenguaje natural y métricas de centralidad para desarrollar un método, y herramienta de soporte, para la detección de patrones de diseño de sistemas orientados a objetos codificados en Java a partir de su código fuente minimizando las limitaciones anteriores. Los resultados obtenidos no solo muestran que esta combinación es adecuada en el sentido de requerir poco esfuerzo para incluir nuevos patrones diseño y lenguajes de programación, sino también que el método y herramienta exhiben en la mayoría de los análisis una buena precisión de identificación de patrones.

Agradecimientos

Agradezco el apoyo recibido a mis directoras de tesis las doctoras Perla Inés Velasco Elizondo y Alejandra García Hernández. También al MATI José Guadalupe Hernández Reveles director de la maestría.

Agradezco igualmente a CONACyT y COZCyT, por sus programas de becas, que soportan la formación de estudiantes e investigadores.

Tabla de Contenido

Índice de Figuras

Índice de Tablas

Índice de Ilustraciones

Anexos

1 Introducción

En éste capítulo se hace una introducción a este trabajo. Inicialmente se discute su contexto, posteriormente se describen la problemática a resolver, objetivos y estrategia de solución. Finalmente, se describe la organización de los capítulos subsecuentes de este documento.

1.1 Contexto

El diseño de un sistema de software es un artefacto fundamental en el contexto de su desarrollo y mantenimiento ya que no sólo permite guiar y restringir varias actividades durante su construcción sino también porque, una vez construido, permite realizar tareas de mantenimiento y evolución del sistema de una manera más sistemática y localizada.

Este diseño está formado por un conjunto de artefactos que describen elementos de software. Estos elementos están organizados en diversas estructuras. Muchas de estas estructuras corresponden a *patrones de diseño*. En términos generales, un patrón es una solución reutilizable que se emplea para resolver un problema de diseño recurrente.

Existen patrones en diferentes niveles de abstracción tales como estilos arquitectónicos [1], patrones arquitectónicos [2], o patrones de diseño [3]. Independientemente de su nivel de abstracción, la identificación de patrones en un sistema de software proporciona a los arquitectos y desarrolladores de software una visión sobre su estructura y podría ayudarles a comprender el comportamiento detrás de su diseño, aun cuando ninguna documentación de este tipo se encuentre disponible [4]. Esto es relevante porque la falta de documentación de un sistema de software es un problema común que afecta el mantenimiento del software. Dentro de la comunidad de Ingeniería de Software (IS), es bien conocido que del 50% al 90% de la actividad de mantenimiento se invierte en estudiar el software para entenderlo y determinar cómo se puede implementar la modificación requerida [5].

1.2 Problemática

Existen escenarios en los que los sistemas no cuentan con una documentación que describa su diseño o, si existe, muchas veces no corresponde a la implementación actual. Varias razones pueden ser la causa de esto, por ejemplo la falta de cultura de documentación durante el diseño del sistema, la no actualización de la documentación del diseño cuando el sistema cambió y la erosión del diseño. Adicionalmente, los sistemas legados suelen no poseer documentación alguna y es muy común que nuevos sistemas de software tengan que integrarse con estos. En estas situaciones es que cobra importancia la detección de patrones de diseño.

Durante años, se han realizado varios esfuerzos para soportar el proceso de detección de patrones de diseño de forma sistemática y (semi-) automática. Actualmente existen diversas herramientas que pueden ser utilizadas por los diseñadores e ingenieros de software con este propósito. Sin embargo, estas herramientas presentan las siguientes limitaciones:

a) Son orientadas a un lenguaje de programación específico.
b) Son cerradas y no extensibles a la inclusión de nuevos patrones.

A continuación explicamos cada una de estas limitaciones.

a) Orientadas a un lenguaje de programación específico.
La mayoría de las herramientas actuales de detección de patrones de diseño sólo funcionan para código fuente que esté escrito en un lenguaje de programación específico. Esto es, si la herramienta funciona para código fuente escrito en Java es muy probable de que no funcione para código fuente en C, por poner un ejemplo.

b) Cerradas y no escalables a la inclusión de nuevos patrones.

En muchas de las herramientas actuales no es posible acceder a su código fuente (son cerradas), por lo que no se pueden realizar adecuaciones que permitan escalar la herramienta a otros contextos, por ejemplo detectar nuevos patrones de diseño.

1.3 Objetivos

El objetivo general de este trabajo es:

Definir un método y herramienta de soporte, que permita determinar la pertinencia de combinar el uso de técnicas procesamiento de lenguaje natural y métricas de centralidad para la detección de patrones de diseño de sistemas orientados a objetos codificados en Java a partir de su código fuente.

Específicamente:

1. Evaluar el nivel de precisión con respecto a la detección de patrones de diseño. Esto porque consideramos que para soportar mejor la comprensión de un sistema es importante el disminuir el número de falsos negativos detectados.
2. Evaluar la extensibilidad a otros lenguajes de programación orientados a objetos.
3. Evaluar la extensibilidad a otros patrones.

1.4 Estrategia de solución

En la Figura 1 se describe de forma general el proceso para definir el método y la herramienta de soporte para la detección de los patrones de diseño:

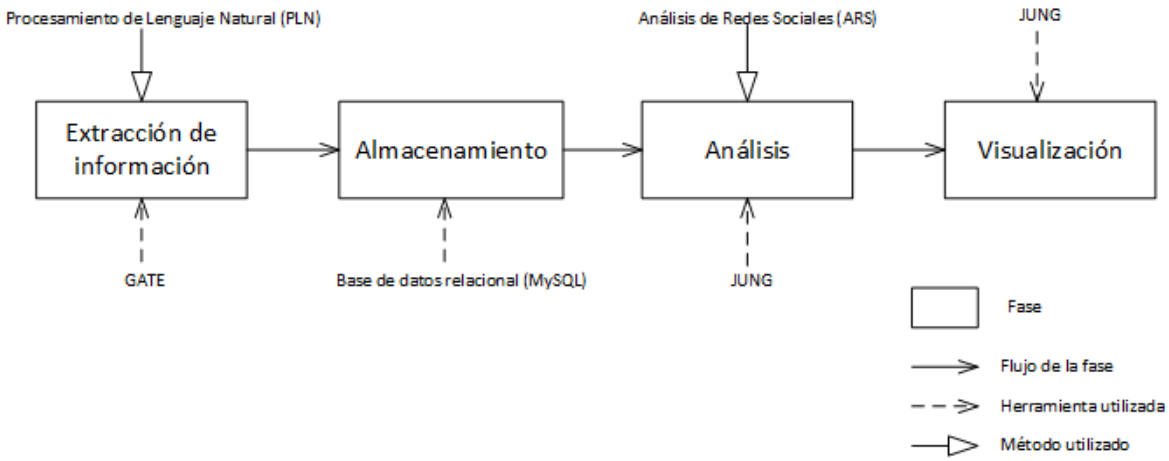


Figura 1. Estrategia de la solución

Las fases del proceso están inspiradas en la guía descrita por Kazman [6] para la recuperación de arquitectura de software. Sin embargo, se adaptó para recuperar elementos de más bajo nivel, es decir, para recuperar patrones de diseño.

La primera fase, extracción de información, tiene como propósito analizar los artefactos de implementación de un sistema, es decir, el código fuente para obtener diversos elementos como clases y las relaciones entre ellas. Para ello, en este trabajo se utiliza el Procesamiento de Lenguaje Natural (PLN). Creemos que esta decisión contribuye a lograr que el método y la herramienta sean adaptables con poco esfuerzo, para permitir la detección de patrones en código fuente escrito en otros lenguajes de programación.

La segunda fase, almacenamiento, tiene como propósito convertir la información extraída en la forma estándar Rigi (un formato de datos basada en tuplas de la forma: “relación <entidad 1> <entidad 2>”). Esto permite identificar la relación que existe entre dos clases diferentes de una manera estándar, contribuyendo así con el objetivo de la adaptabilidad a diversos lenguajes de programación.

La tercer fase, análisis, tiene como propósito analizar la información guardada en la base de datos. Para ello, en este trabajo se utiliza el Análisis de Redes Sociales (ARS) con la finalidad de exhibir una precisión por lo menos igual a la de las herramientas de detección de patrones de diseño actuales.

Finalmente la cuarta fase, visualización, tiene como propósito que el usuario pueda distinguir los patrones detectados, así como observar la red de software formada.

1.5 Organización del documento

Este documento está organizado de la siguiente manera. En el Capítulo 2 se describen los conceptos principales que son necesarios para entender la naturaleza de este trabajo. A continuación, en el Capítulo 3, se revisan algunos trabajos relacionados relevantes. En el Capítulo 4, se describe el enfoque para el diseño de detección de patrones. Los resultados de la aplicación del enfoque para detectar algunos patrones de diseño se discuten en el Capítulo 5. Y finalmente, se afirman las conclusiones y el trabajo futuro en el Capítulo 6.

2 Marco teórico

En este capítulo se presenta una descripción de los principales conceptos relacionados con éste trabajo: (i) las métricas de análisis de redes sociales, (ii) los sistemas de software orientados a objetos, (iii) los patrones de diseño y (iv) el procesamiento de lenguaje natural.

2.1 Métricas de Análisis de Redes Sociales

El ARS tiene que ver con un conjunto de teorías y técnicas que permiten una mejor comprensión de los aspectos relacionados con la estructura y el comportamiento de las redes sociales. Por ejemplo, durante los últimos años, el ARS se ha aplicado para explicar el desempeño organizacional (Luo [7]; Reagan y Zuckerman [8]; Sparrowe et. al [9]) y para estudiar el proceder de los usuarios en las redes sociales en línea (Malhotra et. al. [10]). En el campo de la IS, un gran número de trabajos de investigación se centran en la aplicación del ARS para entender los aspectos organizacionales de los equipos de desarrollo de software (Wolf et al., [11], Zachor y Gunes [12], Serrano et al. [13] y Jermakovics et al. [14]) y para comprender la estructura y el comportamiento de los sistemas que estos producen (Subelj y Bajec [4]).

Para ello, y entre otras cosas, el ARS define un conjunto de métricas relacionadas con la red, que se organizan generalmente en dos categorías principales. La primera considera métricas que se centran en la naturaleza de los vínculos de una red (por ejemplo, la intensidad de las relaciones o la reciprocidad). La segunda categoría se refiere a las métricas que miden aspectos relacionados con las características estructurales de un conjunto de nodos conectados (por ejemplo, la centralidad de la red y la densidad o agrupamiento) [15] y [16]. En esta sección sólo se describirán métricas de la segunda categoría pues son las que se usan en este trabajo. En concreto nos enfocamos en medidas de *centralidad*.

La centralidad de la red es una propiedad estructural que ha sido ampliamente utilizada con diferentes propósitos [17]. Ésta puede ser calculada para evaluar la importancia de un elemento de la red y por lo tanto ayuda a entender la influencia y el poder de dicho elemento dentro de la misma. La centralidad puede estimarse tanto a nivel individual como a nivel de red. A nivel individual, mide la proporción de relaciones que una persona tiene en relación a otros actores que están incrustados en la estructura de la red. A nivel de red, la centralidad mide el grado en que los enlaces de red se concentran en uno, pocos o varios actores [18]. Hay cuatro fórmulas diferentes para medir la centralidad, éstas son, la *centralidad de grado*, la *intermediación*, la *cercanía* y la *centralidad del vector propio*. A su vez, la centralidad de grado puede medirse como centralidad de grado de entrada y como centralidad de grado de salida. Aunque la aplicación de alguna de estas fórmulas depende de los aspectos a resolver, la centralidad de grado es la medida más simple y la más utilizada.

La centralidad de grado de un nodo es el número de relaciones que inciden con él. La centralidad de grado de un nodo es un recuento que va de 0, si no hay nodos adyacentes a un nodo dado, a un máximo de *g*-1 si un nodo dado es adyacente a todos los otros nodos en el grafo, donde g es el total de los nodos presentes en la red.

Con relación a la centralidad de grado están las métricas de *centralidad de grado de entrada* y *de salida*. La centralidad de grado de entrada denota el número de relaciones referidas hacia un nodo específico por otros. La centralidad de grado de salida denota el número de relaciones que un nodo en específico tiene con el resto.

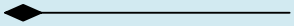
2.2 Sistemas orientados a objetos

El software que hoy se construye, en su mayor parte utiliza el paradigma orientado a objetos (OO). Esto habla de las ventajas de éste sobre otros paradigmas. Entre las ventajas más discutidas encontramos: mejor representación de aspectos del mundo real, mejoras en la cohesión, mejor protección a las propiedades de un objeto, e incluso mejora la capacidad de compartición y extensión de su comportamiento.

El paradigma (OO) es un enfoque de diseño e implementación de sistemas de software en el cual la descomposición de un sistema está basado en el concepto de objetos. Esto es, un sistema orientado a objetos se diseña e implementa como un grupo de objetos interactuando. Un *objeto* representa una entidad del mundo real que es representativa dentro del contexto del sistema que se está construyendo. Una *clase* es una plantilla para definir instancias de objetos en términos de dos elementos de información principales: el *estado* (es decir, un conjunto de marcadores de posición de datos y sus valores) y *operaciones* (es decir, un conjunto de acciones que cambian los valores que definen el estado del objeto). Cada objeto se construye a partir de una clase y muchas instancias de objetos pueden ser creadas a partir de la misma clase.

Las clases y los objetos interactúan a través de *relaciones* que denotan formas específicas de interacciones. La Tabla 1 describe algunas relaciones comunes definidas en el contexto de sistemas orientados a objetos. También se incluye una representación visual de cada relación en el Lenguaje de Modelado Unificado (UML) [21], el cual es un lenguaje de diseño utilizado para describir sistemas orientados a objetos.

Tabla 1. Relaciones comunes en el contexto de sistemas orientados a objetos

Nombre	Semántica	Sintaxis UML
Asociación	Denota que una clase puede causar que otra lleve a cabo una operación en su nombre.	
Agregación	Denota una relación todo-parte entre una clase (el todo) y sus componentes (las partes).	
Generalización (Herencia)	Denota que una de las dos clases relacionadas (la subclase) se considera que es una forma especializada de la otra (la superclase).	
Implementación (Realización)	Denota que una de las clases relacionadas realiza las operaciones especificadas por la otra.	
Composición	Denota una relación todo-parte entre una clase (el todo) y sus componentes (las partes)	

Existen diversos lenguajes de programación que se pueden utilizar para la implementación de sistemas de software orientados a objetos. En el presente trabajo, se considera que los sistemas están codificados en el lenguaje de programación Java [22].

Seguramente, se preguntará, ¿por qué utilizar Java? Según TIOBE [23], que es un indicador de popularidad de los lenguajes de programación, Java es el segundo más popular, por debajo de C, pero hay que recordar que C es un lenguaje estructurado, así que al descontarlo, Java queda como el lenguaje orientado a objetos más utilizado a nivel mundial. Sin embargo, hay que mencionar que el enfoque propuesto en este trabajo puede ser extendido para ser utilizado en otros lenguajes orientados a objetos como por ejemplo C#, .Net, PHP5 o algún otro siempre que soporten las abstracciones del paradigma OO.

2.3 Patrones de diseño

Un patrón es una solución reutilizable que se usa para resolver un problema recurrente en el diseño de software. Los patrones son estructuras de elementos de software y relaciones específicas entre ellos. Aunque es difícil clasificar los patrones, son generalmente aceptados los estilos arquitectónicos [1], patrones arquitectónicos [2] y patrones de diseño [3] y denotan estructuras de diseño de software en diferente nivel de abstracción.

Uno de los catálogos más importantes de patrones es el propuesto por Gamma et al., [3], el cual describe 23 patrones de diseño para sistemas de software orientados a objetos. Los patrones, en éste catálogo, son estructuras de diseño de software en términos de clases u objetos y relaciones específicas entre ellos. Las siguientes figuras (Figura 2, Figura 3 y Figura 4) muestran, en sintaxis UML, tres patrones contenidos en éste catálogo: Façade, Adapter y Composite, respectivamente. Estos patrones son los que se usarán para ilustrar el enfoque de detección de patrones presentado en este trabajo.

El patrón de diseño Façade (Figura 2), se define como una estructura de diseño útil para proveer un punto único y simple de acceso a un conjunto de elementos de un sistema. Esto es, la Façade es una clase que provee un punto de acceso a una colección de clases, haciéndolo más atractivo y fácil de usar. Por ejemplo, en un sistema orientado a objetos el “menú principal” podría ser diseñado usando este patrón de forma que una clase Façade podría proveer el punto de acceso a diferentes funcionalidades que podrían ser implementadas en clases diferentes. Como se puede observar en la Figura 2, la forma básica del patrón de diseño está definida en términos de una clase Façade, un conjunto de clases del subsistema y un conjunto de clases cliente, las cuales son las únicas que están requiriendo acceso a las operaciones definidas por las clases en el subsistema. Todas las clases en el patrón Façade son conectadas por medio de relaciones de asociación.

El patrón de diseño Adapter, mostrado en la Figura 3, provee una estructura de diseño útil cuando es necesario adaptar la interfaz de una clase en una compatible para otra. Una interfaz define un protocolo que las clases deberían utilizar para comunicarse unas con otras correctamente. En este patrón un adapter es también una clase que hace algún tipo de adecuación, que permite que otras clases trabajen juntas cuando no podrían hacerlo debido a sus interfaces incompatibles. Este patrón se define en términos de un conjunto de clases cliente, una interfaz que el cliente utiliza, comúnmente llamada interfaz target, una interfaz que necesita ser adaptada con el objetivo de que el cliente pueda comunicarse con ella, y una clase adapter que conecte a esas dos interfaces para que puedan

trabajar juntas. Como puede observarse en la Figura 3, todos estos elementos se interconectan unos con otros por medio de relaciones de asociación, a excepción de la interfaz target y la clase adapter que se relacionan a través de una relación de implementación.

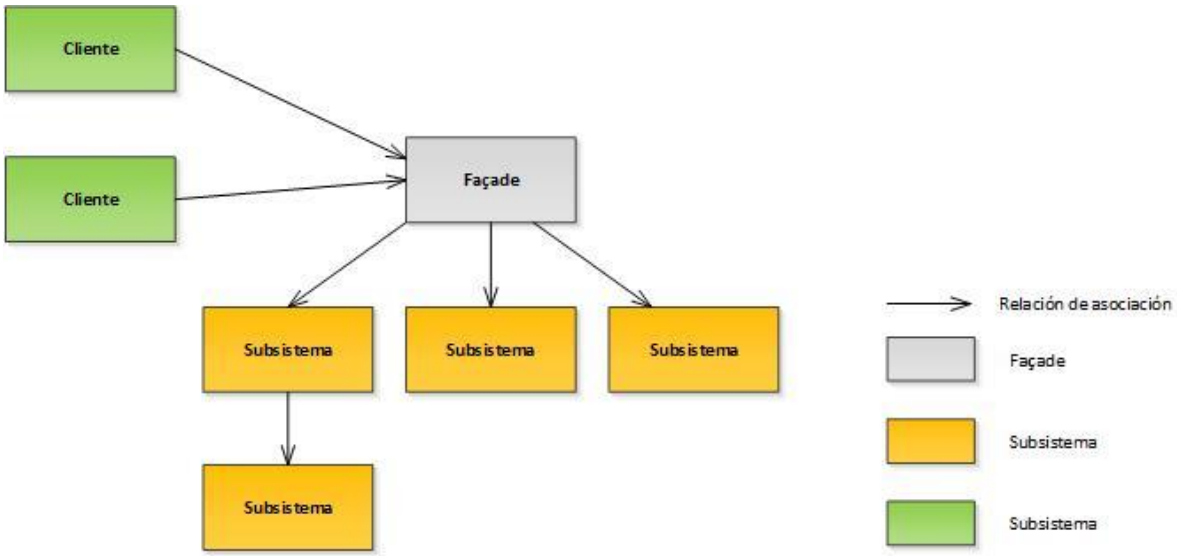


Figura 2. Patrón de diseño Façade.

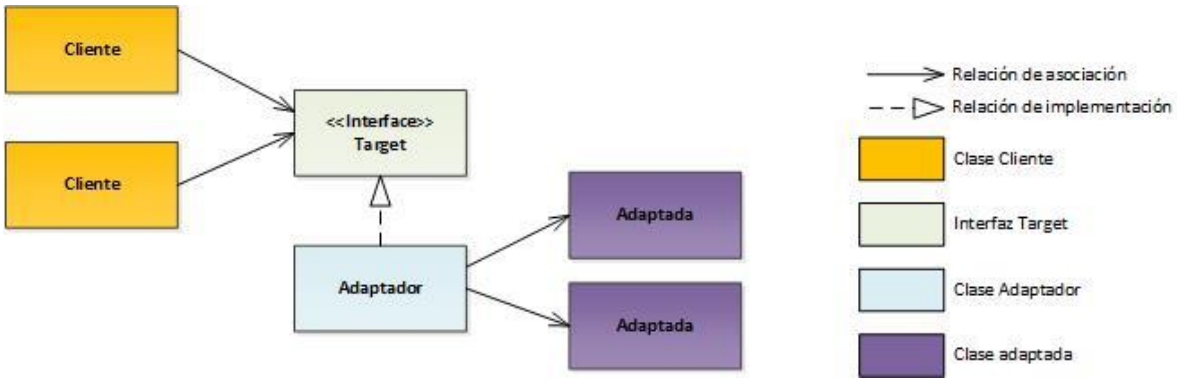


Figura 3. Patrón de diseño Adapter

Finalmente, el patrón de diseño Composite, presentado en la Figura 4, provee una estructura de diseño para agrupar un conjunto de clases para que sean tratadas como una sola. Por lo tanto, el patrón Composite es útil cuando hay una jerarquía todo-parte de clases y una clase cliente necesita tratar con ellos de manera uniforme, independientemente de su posición en la jerarquía de clases.

Por ejemplo, en un sistema de archivos un cliente puede ejecutar las operaciones de copiar, renombrar y eliminar tanto para los archivos (nodos hoja), como para los directorios (nodos no hoja). Como se muestra en la Figura 4, el patrón de diseño Composite se define en términos de: un conjunto de clases hoja, un conjunto de clases Composite que permiten organizar las clases hoja en jerarquías, y una interfaz Component que provee una interfaz funcional para acceder a las operaciones provistas por ambas, es decir, por las hojas y por las clases Composite. Como se aprecia en la Figura 4, las relaciones de agregación, asociación y generalización son usadas para conectar los elementos en éste patrón.

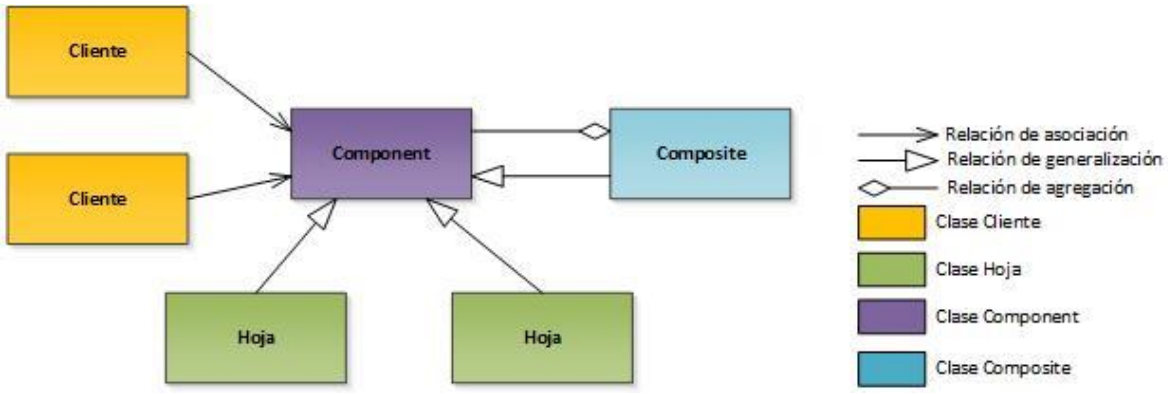


Figura 4. Patrón de diseño Composite

2.4 Procesamiento de Lenguaje Natural

El Procesamiento de Lenguaje Natural (PLN) es un área de ciencias de la computación cuyo objetivo es estudiar las interacciones entre las computadoras y el lenguaje humano. Esto es, se persigue que las computadoras puedan entender (a cierto grado) el texto o el habla en lenguaje humano (por ejemplo, inglés o español) y para soportar tareas que requieren de tal comprensión. Entre estas tareas están, la búsqueda y recuperación de información, la traducción automática de un lenguaje a otro (por ejemplo, de inglés a español), etcétera [24].

El número de similitudes entre el lenguaje natural (humano) y los lenguajes de programación puede ser bastante sorprendente [25]. Por una parte los lenguajes naturales y los lenguajes de programación sirven para el propósito de la comunicación: el primero como medio de comunicación entre humanos y el segundo como un medio de comunicación entre humano y máquina. Además, los lenguajes de programación son un puente entre los lenguajes naturales y el código

binario que la computadora lee y comprende [26]. Esto se debe a que muchos códigos en los lenguajes de programación tienen comandos que son parecidos a sustantivos o verbos en los lenguajes naturales, facilitando la codificación en los lenguajes de programación.

La mayor diferencia entre los lenguajes naturales y los lenguajes de programación es la sintaxis. Mientras los lenguajes naturales son flexibles y evolucionan en el tiempo, los lenguajes de programación son estáticos y permanecen consistentes una vez que son creados [25].

Puede observarse en Figura 5, que la estructura gramática de cada ejemplo (escrito en lenguaje de programación Java) se conserva de manera muy aproximada a como se escribiría el mismo enunciado en lenguaje natural.

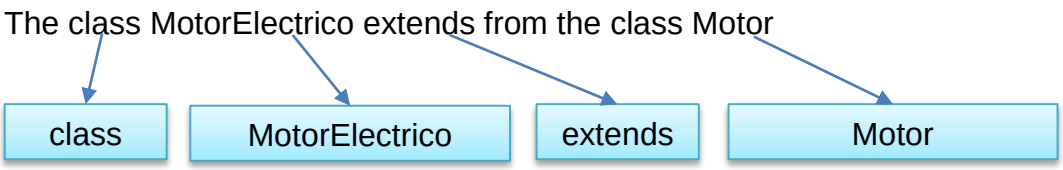


Figura 5. Ejemplo 1, correspondencia entre una oración escrita en lenguaje natural y código fuente escrito en Java

2.5 Resumen

En este capítulo se definieron los siguientes conceptos y comunicaron las siguientes ideas:

- La centralidad es una métrica que forma parte de las características estructurales de la red. La centralidad suele ser calculada para reconocer la influencia o importancia de un nodo dentro de una red.
- El paradigma orientado a objetos (OO) es un enfoque de diseño e implementación de sistemas de software en el cual la descomposición de un sistema está basado bajo el concepto de objetos.
- Un *objeto* representa una entidad del mundo real que es representativa dentro del contexto del sistema que se está construyendo.
- Una *clase* es una plantilla para definir instancias de objetos.
- Un *patrón* es una solución que resuelve un problema recurrente. Al aplicar éste tipo de soluciones en el ámbito de software se está hablando de patrones de diseño.
- Los patrones de diseño estudiados en ésta tesis son: Façade, Adapter y Composite.

- El procesamiento de lenguaje natural permite que las computadoras tengan cierto grado de comprensión del lenguaje humano.
- Los lenguajes naturales son flexibles y evolucionan en el tiempo.
- Los lenguajes de programación son estáticos y permanecen consistentes en el tiempo.
- El lenguaje natural y el lenguaje de programación son muy parecidos en cuanto a léxico, sintaxis y semántica.

3 Trabajos relacionados

En años anteriores se propusieron diversos trabajos de detección de patrones de diseño. Este capítulo se centra en describir métodos similares al de ésta tesis, donde la detección de patrones de diseño se realiza de forma automática y se analizan de la estructura estática del sistema. Esto es, su código fuente. Para ello, en las siguientes secciones se muestran trabajos relevantes considerando las fases que se describieron en la estrategia de la solución (ver Sección 1.4, del Capítulo 1). Las fases que serán analizadas son: extracción de información, almacenamiento, análisis y visualización.

3.1 Extracción de información

La extracción de información tiene como propósito analizar los artefactos de implementación de un sistema, es decir, el código fuente para obtener diversos elementos de interés. En la Figura 6, se muestra un diagrama en el que se clasifican los métodos y herramientas de extracción que han sido estudiados en diferentes investigaciones.

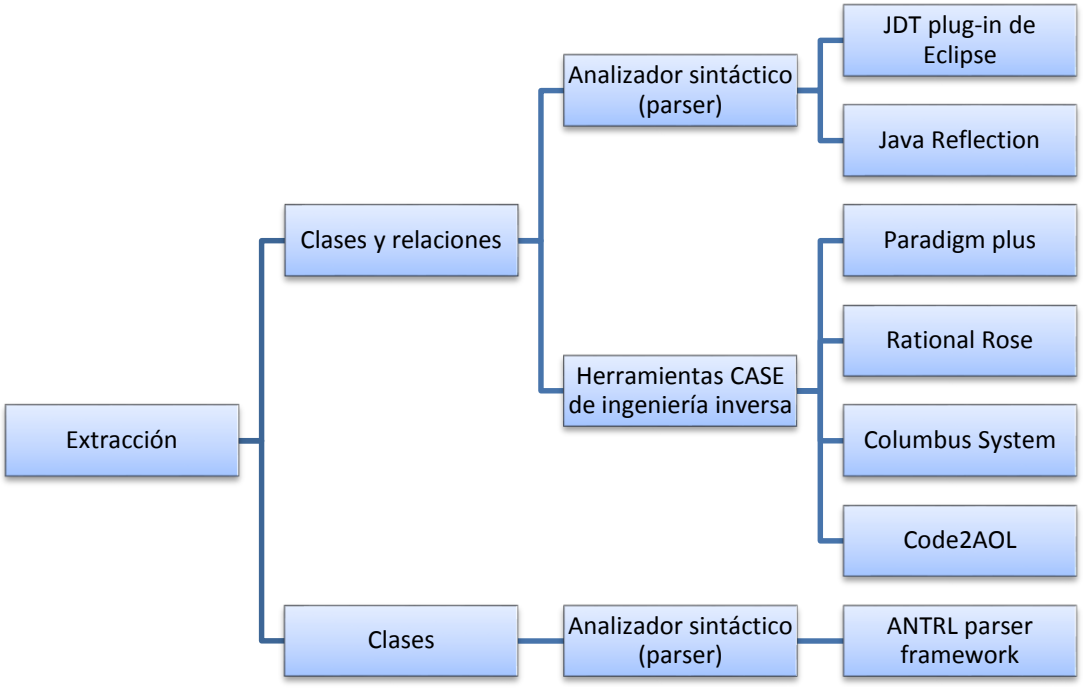


Figura 6. Métodos y herramientas usados en la fase de extracción de información.

Como se observa en la Figura 6, la mayoría de los trabajos extraen dos tipos de elementos: clases y relaciones entre clases.

En la Figura 6 se observa que la extracción de clases y relaciones es comúnmente posible a través de dos herramientas de software, que son analizadores sintácticos, también conocidos como “*parsers*”, y herramientas CASE (*Computer Aided Software Engineering*, por sus siglas en inglés) de ingeniería inversa.

Un analizador sintáctico convierte el código fuente en otras estructuras, frecuentemente árboles de sintaxis abstracta. Las herramientas CASE de ingeniería inversa toman el código fuente como entrada y generan representaciones de un nivel más alto de abstracción, tales como gráficas, diagramas u otros datos. Ambas herramientas tienen como propósito facilitar el análisis de la estructura del software y a través de ella entender su funcionamiento. Aun así, el costo del entendimiento del software se manifiesta en el tiempo requerido para comprenderlo [27].

Las herramientas utilizadas para realizar el análisis sintáctico del código fuente son: JDT (*Java Development Tools*, por sus siglas en inglés) [28], Java Reflection [29], y ANTRL parser [30]. A continuación, estos se describen a mayor detalle.

Hayashi y Katada [31] utilizaron JDT, particularmente el componente *Core* para obtener el modelo de sintaxis abstracta de un proyecto Java y posteriormente extraer sus conexiones jerárquicas. Cada conexión posee detalles acerca de las relaciones que posee cada clase, estas son: herencia, asociación y agregación. Su principal desventaja es que su componente extractor de datos está limitado al lenguaje de programación Java.

Java Reflection fue utilizado por Huang et al. [32] para extraer clases y relaciones a partir de código fuente escrito en Java. Las relaciones que fueron capaces de extraer son las relaciones de dependencia, generalización, asociación y realización. Además, Java Reflection también permite descubrir información acerca de campos, métodos y constructores de las clases cargadas. Sin embargo, el uso de este enfoque es limitado porque no todos los lenguajes orientados a objetos soportan reflexión [32].

Por otro lado, se dispone de varias herramientas CASE que automatizan los procesos de ingeniería inversa. Algunas son de código abierto, y otras son privativas. Dentro de las privativas están Paradigm Plus y Rational Rose; dentro de las de código abierto están Columbus System y Code2AOL.

Paradigm Plus 2.01, es una herramienta CASE orientada a objetos para soportar diversas actividades de las fases del ciclo de vida del software. El aspecto de interés de Paradigm Plus es un componente que facilita la extracción de información acerca de las clases directamente de los archivos de cabecera de

C++. Una de sus limitaciones es que no puede extraer relaciones de asociación y de agregación [33].

Al igual que Paradigm Plus 2.01, Rational Rose es una herramienta de modelado de software que posee características de ingeniería inversa, pero a diferencia de éste, Rational Rose tiene la capacidad de extraer relaciones de asociación, agregación, dependencia y generalización (herencia) [34].

Dentro de las herramientas libres se encuentra Columbus System. Columbus System es un framework de ingeniería inversa que es capaz de analizar y extraer información a partir del código fuente del lenguaje de programación C/C++. Uno de sus componentes es Columbus Schema [35]. Columbus Schema es un componente que es capaz de dar forma al código fuente, en términos de un conjunto de clases con atributos y relaciones. La descripción del esquema se establece usando diagramas de clases de UML estándar. Columbus System puede obtener información como el tamaño del código fuente, el número de subclases y algunos otros datos invariantes [36]. Además, es capaz de detectar las relaciones de agregación, asociación y composición.

Otra de las herramientas libres, es la conocida como Code2AOL. Code2AOL fue utilizada por Antoniol et al. [37], para extraer una representación AOL (Abstract Object Language, por sus siglas en inglés) a partir del código fuente. Tal representación contiene información acerca de las clases, sus métodos y atributos, así como también las relaciones entre clases [37] a partir de código fuente escrito en lenguaje de programación C++. Las relaciones que puede obtener Code2AOL son agregación, asociación y herencia.

A diferencia de los trabajos anteriores, en su propuesta, Kirasic y Basch [38] no obtienen información acerca de las relaciones existentes en el código fuente, sino que se centran en extraer información acerca de las clases existentes en el código fuente a un formato XML que detalla sus características. Kirasic y Basch construyeron dos analizadores sintácticos, uno para soportar el lenguaje de programación C# y otro para soportar Java. Estos analizadores fueron construídos utilizando ANTRL. ANTLR es un framework para construir analizadores sintácticos, intérpretes y programas similares desde la gramática del lenguaje.

3.2 Almacenamiento

El almacenamiento consiste en guardar la información extraída en la fase anterior en algún tipo de estructura. En la Figura 7 se muestra que en los trabajos y herramientas estudiados, la mayoría utiliza estructuras no persistentes como matrices, árboles y grafos.

Unos pocos trabajos dividen el almacenamiento en dos partes, una parte se encarga de guardar los datos en un medio de almacenamiento persistente, mientras que la otra parte crea una estructura de datos en un almacenamiento no persistente a partir de los datos guardados en el almacenamiento persistente. Esto permite que el tiempo de procesamiento sea menor debido a que la fase de extracción de información puede ser omitida en diversas ejecuciones. En este caso la información se guarda como archivos de texto plano o en formato XML.

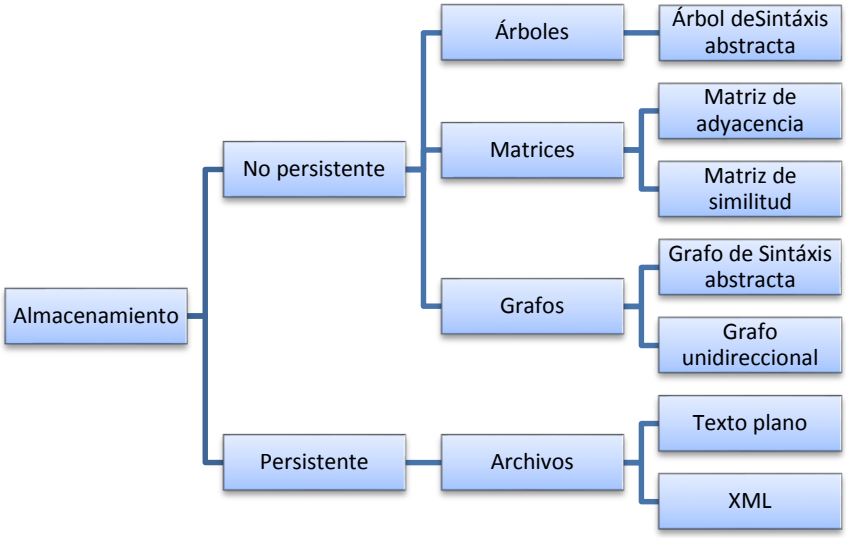


Figura 7. Métodos y herramientas utilizadas en la fase de almacenamiento.

3.3 Análisis

El análisis consiste en examinar las estructuras creadas en la fase anterior para determinar la existencia o la ausencia de un patrón de diseño. En la Figura 8 se muestran los métodos y herramientas utilizadas en la fase de análisis.

El análisis puede ser realizado utilizando principalmente dos métodos, la verificación y la medición. La verificación consiste en confirmar una lista de requisitos que se deben cumplir para que alguna parte del código pueda ser considerada como un patrón de diseño. La medición consiste en un conjunto de cálculos matemáticos que dan como resultado un valor o un conjunto de valores que determinan la existencia o ausencia de un patrón de diseño.

La verificación ha sido implementada utilizando herramientas de lógica y análisis estructural. La utilización de lógica ha sido implementada utilizando herramientas como Prolog y Ontologías.

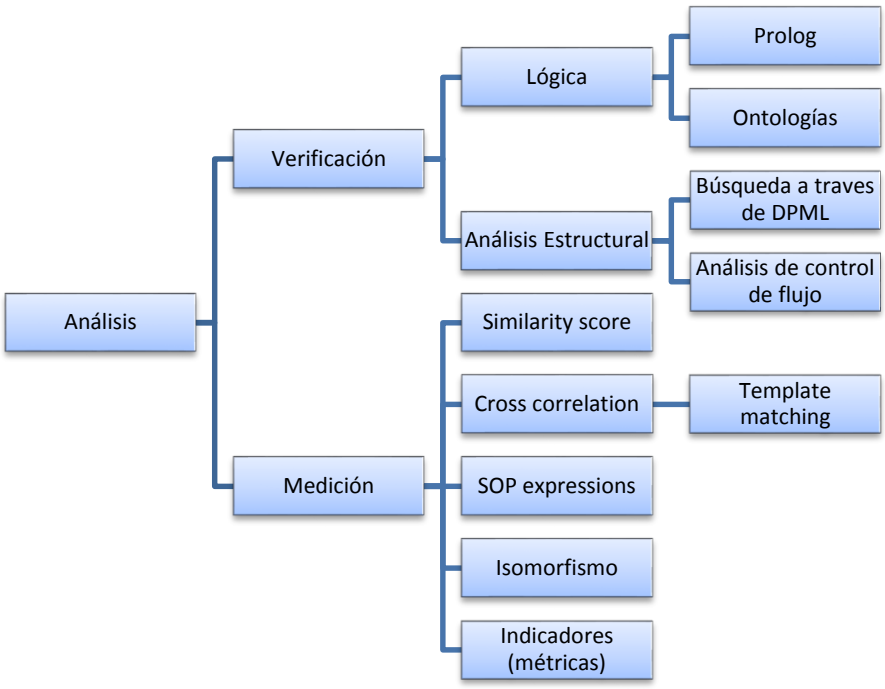


Figura 8. Métodos, herramientas y técnicas usadas en la fase de análisis.

Los trabajos de Kramer [33], Hayashi [31], y Huang [32] crearon un componente que consume dos archivos de texto plano que son creados en la etapa de almacenamiento; el primero contiene la representación del código fuente en un formato de lógica de predicados; el segundo contiene patrones de búsqueda en un formato de reglas de inferencia, las cuales son aplicadas sobre los enunciados que representan el código fuente y de esa manera el solucionador detecta patrones de diseño.

Existen algunas diferencias entre ellos, por una parte, el analizador de Kramer está enfocado a sistemas de software escritos en lenguaje C++, el de Hayashi se enfoca a Java y .Net. Mientras tanto el de Huang solamente consideró software escrito en Java.

Otra de sus diferencias se centra en la manera de utilizar Prolog. Kramer utiliza el solucionador de lógica de predicados de Prolog, mientras tanto Hayashi utiliza tuProlog, una versión reducida de Prolog que se basa en el *minimal Core* de Prolog. Finalmente, Huang creó un motor de reconocimiento de patrones de diseño extendiendo el solucionador de lógica de predicados de Prolog.

La principal desventaja de los métodos que utilizan lógica es que solamente pueden detectar patrones de diseño sin variaciones en su implementación.

La otra rama que utiliza lógica es la de ontologías. Según Castells [39] una ontología es una jerarquía de conceptos con atributos y relaciones que proporciona un vocabulario de clases y relaciones para describir un dominio. El análisis a través de herramientas ontológicas fue propuesto por Kirasić [38]. Kirasić utilizó dos ontologías, una de ellas posee el conocimiento sobre los patrones de diseño a detectar, y la otra posee conocimiento del paradigma de programación orientada a objetos. La ontología toma un AST (Abstract Syntax Tree) como entrada, generado en las etapas anteriores, y prueba hacer la correspondencia con un patrón específico. Éste enfoque detecta patrones de diseño con variaciones en su implementación y solamente está disponible para el análisis de software codificado en lenguaje de programación C#.

Dentro de los trabajos de verificación que utilizan herramientas de análisis estructural se encuentra el propuesto por Balanyi [35], quien propone un conjunto de reglas para realizar una búsqueda dentro de la estructura de un ASG (Grafo de Sintáxis Abstracta), que es obtenido en las fases anteriores. Estas reglas están definidas en un formato llamado DPML (Design Pattern Markup Language) para representar las características de un patrón de diseño. Su principal desventaja es que el tiempo de búsqueda es largo en casos con muchas operaciones y atributos en las clases del código fuente.

Por otra parte, Shi [40] realiza el análisis estructural a través de un análisis de control de flujo en sentido *backward* (hacia atrás, en idioma inglés), que según Lamas [41] uno de sus principales objetivos es intentar predecir el comportamiento futuro de un sistema. Para hacerlo fue necesario realizar una nueva categorización de los patrones definidos por Gamma [3] de acuerdo a la dificultad de su detección vía ingeniería inversa. De ésta manera, es posible utilizar un enfoque distinto de acuerdo a cada caso para así mejorar la precisión y la rapidez de la detección.

El trabajo de Shi toma como entrada un grafo de control de flujo, creado en fases anteriores, en el cual se busca determinar el comportamiento de una variable o de algún método. Para el siguiente ejemplo tenga en cuenta que el patrón *Singlenton* [3] crea una instancia de un objeto, o si ésta ya está creada, solamente la retorna. La Figura 9 demuestra el grafo de control de flujo de un método llamado *getTheSpoon* que implementa el patrón *Singlenton*:

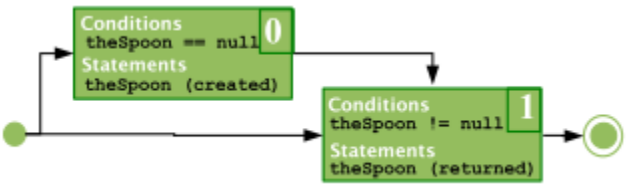


Figura 9. CFG *getTheSpoon* [45]

Es necesario contar con el grafo de control de flujo tanto del patrón como del código fuente. De esa manera es posible recorrer el grafo de control de flujo del patrón a lo largo del grafo de control de flujo del código fuente para así obtener los patrones detectados.

Sin embargo, hay que tomar en cuenta que si se realiza un análisis de esa manera, solamente se obtendrían patrones que hayan sido implementados sin variaciones. La herramienta de soporte de Shi, solamente puede extraer código fuente escrito en lenguaje de programación Java.

Por el lado de la medición, se encuentran métodos basados principalmente en el cálculo matricial, es decir, utilizando el modelo matemático de las matrices y sus herramientas asociadas [42]. Entre este tipo de cálculos se encuentran, el cálculo de *similarity score*, el cálculo de la correlación cruzada, el cálculo de SOP expressions, el cálculo de isomorfismo de grafos, y el álgebra matricial. Otros trabajos se basan en la medición de diversos indicadores, como por ejemplo, la correspondencia entre el número de relaciones o número de métodos.

Similarity score fue utilizado por Tsantalis en [43] y [44]. Similarity score es una medida que tiene como fin determinar de una manera cuantitativa qué tanto se parecen dos matrices adyacentes. Las matrices adyacentes, representan grafos en los que cada vértice representa una clase y cada arista una relación.

Tsantalis toma como entrada un conjunto de matrices que representan diversos segmentos de un sistema de software y los compara con otro conjunto de matrices que representan los patrones de diseño. Como resultado, se obtiene una matriz que contiene un puntaje que indica la similitud entre cada vértice.

Además, Tsantalis compara su trabajo con otros que calculan la distancia de edición entre dos grafos; es decir, el número de modificaciones necesarias para convertir un grafo en otro. La Figura 10 ilustra dos segmentos de un sistema en los que se trata de identificar cuál es la implementación más cercana a un patrón de diseño ilustrativo llamado *Elemental Design Pattern*:

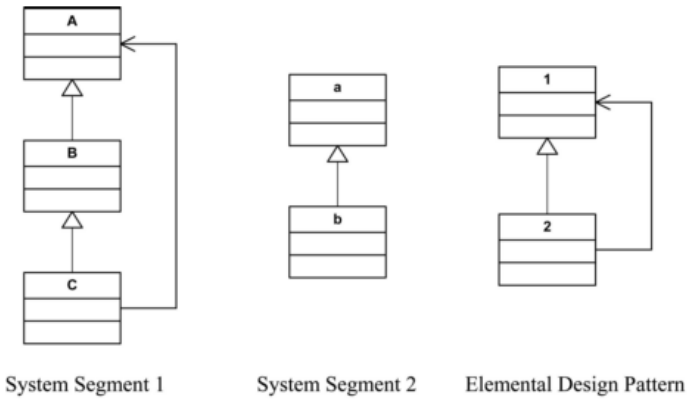


Figura 10. Diagramas de clase UML de dos segmentos de un sistema y un patrón de diseño [15]

Según la definición de los algoritmos de la distancia de edición entre dos grafos, "*System Segment 2"* sería el candidato más fuerte dado que solamente se tendría que ejecutar un solo paso de edición que consistiría en agregar la relación de asociación entre las clases *b* y *a*. Mientras que para "*System Segment 1"* se tendrían que realizar los siguientes pasos:

1. Eliminar las relaciones de generalización (B, A)
2. Eliminar las relaciones de generalización (C, B)
3. Eliminar la clase B
4. Agregar la relación de generalización (C, A).

Basándose en el enfoque propuesto por Tsantalis se obtienen como resultado las siguientes matrices de similitud normalizada para ambos segmentos (Figura 11 y Figura 12):

$$\begin{array}{c} \\ \mathrm{A} \\ \mathrm{B} \\ \mathrm{C} \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{bmatrix} 0.75 & 0 \\ 0.25 & 0.25 \\ 0 & 0.75 \end{bmatrix} \end{array}.$$

Figura 11. Matriz de similitud normalizada para "*System Segment 1*" [49]

$$\begin{array}{c} \\ \mathrm{a} \\ \mathrm{b} \end{array} \begin{array}{c} \begin{array}{cc} 1 & 2 \end{array} \\ \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \end{array}$$

Figura 12. Matriz de similitud normalizada para *"System Segment 2"* [49]

Los dos puntajes más altos en la matriz indican una fuerte similitud entre la clase A y la clase 1, pertenecientes a "*System Segment 1*" y "*Elemental Design Pattern*" consecutivamente. Lo mismo sucede con la clase C y la clase 2. En contraste con el resultado del algoritmo de las distancias de edición entre dos grafos se obtiene que "*System Segment 1*" posee una estructura más parecida a *"Elemental Design Pattern*".

La limitación de éste enfoque es que el algoritmo solamente calcula la similitud entre dos vértices, en lugar de dos grafos. Una alta puntuación de similitud entre

dos vértices no garantiza una correspondencia entre dos conjuntos de vértices [45]. Además, su herramienta de apoyo solamente puede extraer información de sistemas de software codificados con el lenguaje de programación Java.

Con el fin de resolver las limitaciones del enfoque de Tsantalis, Dong [45] propone la utilización de Cross Correlation Matrix a través de Template Matching. Template Matching es una técnica utilizada en el área de visión computacional para encontrar áreas de una imagen que coincidan (sean similares) a una imagen plantilla [46]. Para identificar el área coincidente se compara la imagen plantilla a través de la imagen fuente por medio de un recorrido.

Ilustración 1. Recorrido de una imagen plantilla a través de fuente [46]

El recorrido consiste en mover la plantilla un pixel a la vez (izquierda a derecha, arriba hacia abajo). En cada movimiento se calcula una métrica que representa qué tan buena o qué tan mala es la coincidencia en esa área en específico.

Por su parte, el trabajo de Dong consume dos matrices, que son obtenidas en las fases anteriores. La primera, representa el patrón que desea buscarse y que funge como la imagen plantilla; y la segunda representa las clases y relaciones del sistema de software y que funge como la imagen fuente o el área de búsqueda.

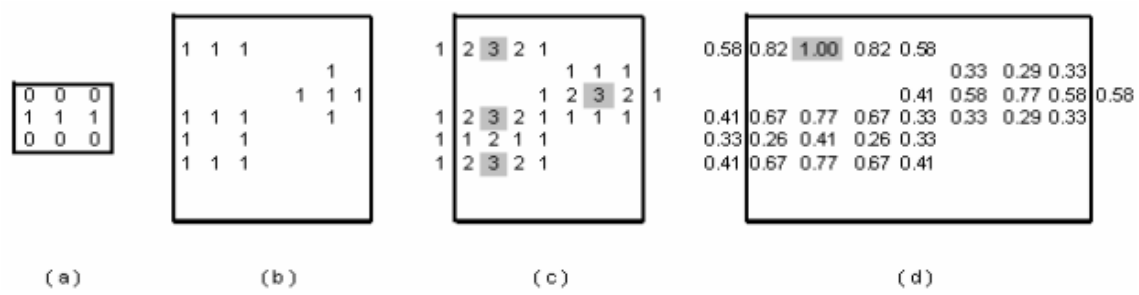


Figura 13. a) Plantilla del gráfico b) Gráfico objetivo c) Correlación cruzada d) Correlación cruzada normalizada

Las matrices de la Figura 13 (a) y (b) corresponden a los valores de los pixeles de una imagen plantilla y una imagen fuente, respectivamente. La Figura 13 (c) muestra los valores de la correlación cruzada donde existen cuatro posibles correspondencias, es posible saberlo mediante el máximo valor de correlación, es decir, 3. Sin embargo al aplicar la normalización se obtiene que existe solamente una correspondencia exacta, con valor 1, mostrado en la Figura 13 (d).

En el método de Tsantalis se utiliza una matriz por cada tipo de relación, pero el gasto de memoria y de procesamiento es mucho mayor que si solamente se utiliza una sola matriz que represente todas las relaciones de un sistema.

Así, Dong propuso, la identificación a cada tipo de relación con un número. Este número tenía que ser único, para que al realizar una operación algebraica se pudieran obtener los tipos de relaciones. De esa manera fue como Dong optó por identificar a cada tipo de relación con un número primo.

Por ejemplo, considere una clase llamada A, y otra llamada B. Entre ellas existen dos tipos de relaciones, una de agregación y una de generalización. En la Tabla 2 se representa una matriz por cada tipo de relación entre A y B:

Tabla 2. Matrices de agregación y asociación

Asociación

	A	B
A	0	1
B	0	0

Agregación

	A	B
A	0	1
B	0	0

Si se busca la existencia de ambas relaciones entre A y B, primero tendría que buscar en una tabla y luego en la otra. Para evitar esto, en la Tabla 3 se identifica a la agregación con el número primo 2, y a la generalización con el número primo 3:

Tabla 3. Matriz general

	A	B
A	$2^0 * 3^0$	$2^1 * 3^1$
B	$2^0 * 3^0$	$2^0 * 3^0$

En las celdas se multiplican todos los números que identifican a cada tipo de relación elevados a una potencia. Ésta se define por una regla simple, si existe alguna relación entre las celdas que se intersectan, el número que corresponde al tipo de relación se eleva a la potencia 1. En el ejemplo, se había mencionado que existen sólo dos relaciones en total y que ambas tienen dirección de A hacia B por esa razón el contenido de la celda (A, B) es $2^1 * 3^1$, es decir 6.

Si se desea saber cuáles son las relaciones existentes entre A y B solamente será necesario aplicar una división. Por ejemplo, si se divide el valor de la celda (A, B) entre el número primo 2 que corresponde a la relación de agregación, el resultado será exacto y esto indicará su presencia. Luego habrá que realizar el mismo tipo de comprobación con el número identificador de la relación de generalización.

El que éste enfoque considere la representación del sistema en una sola matriz tiene ventaja contra el enfoque de Tsantalis, ya que evita el trabajo con muchas matrices adyacentes.

Una desventaja es que no detecta todos los patrones de diseño debido a que Dong asegura que solamente se reconocen ocho características diferentes, entre ellas, las relaciones y las clases abstractas. Los cálculos que se realizan por celda para reconocer cuáles son las relaciones que están involucradas pueden llegar a ser demasiado grandes en un momento dado y a causa de ello se podría gastar mucho tiempo de procesamiento. Además, su herramienta de soporte solamente puede trabajar con sistemas de software codificados en lenguaje de programación Java.

Otra de sus desventajas es que las relaciones de dependencia o agregación no pueden ser detectadas de manera automática. Para resolver ese problema, Dong propuso diversas fases de análisis para establecer una mejor precisión. Pero algunas de ellas, como el llamado análisis semántico, deberían ser realizadas de acuerdo al criterio de una persona de manera manual.

Por otra parte, Gupta [47] implementa un enfoque susceptible a variaciones a la implementación de patrones. Gupta explica que su método de recuperación de los patrones de diseño consiste en consumir una matriz de adyacencia por cada una de las relaciones existentes. Estas matrices son creadas en fases anteriores, para después ser traducidas en cadenas con formato de SOP expressions (Sum Of Product, por sus siglas en inglés), mediante un algoritmo definido por él mismo. Las SOP expressions son populares en el ámbito del álgebra booleana ya que representan expresiones lógicas en un formato especial donde destaca la ausencia de paréntesis. Gupta describe el siguiente ejemplo en donde existe una correspondencia exacta entre el patrón y el sistema:

1. Gupta consume los grafos de los distintos tipos de relación que son generados en fases anteriores.

Por ejemplo, suponga que los grafos unidireccionales correspondientes a cada tipo de relación son los siguientes:

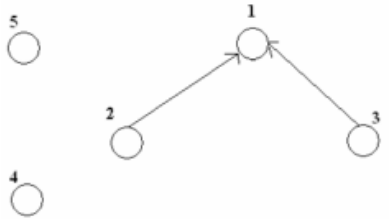


Figura 14. Grafo de la relación de generalización [47]

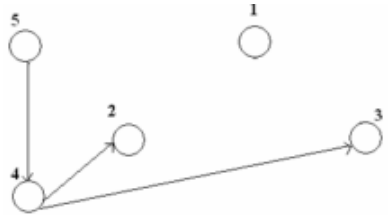


Figura 15. Grafo de la relación de asociación directa [47]

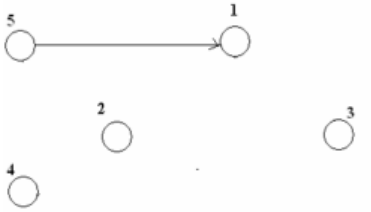


Figura 16. Grafo de la relación de agregación [47]

2. Después los grafos son convertidos a matrices de adyacencia.
3. Luego cada matriz es convertida a una expresión SOP utilizando el algoritmo definido por Gupta [47], éstas son detalladas en la Tabla 4:

Tabla 4. Expresiones SOP de cada relación de un sistema

Figura	Relación	SOP
11	Generalización	2.1+3.1
12	Asociación directa	5.4.2+5.4.3
13	Agregación	5.1

Figura 15 existen dos caminos, uno de ellos comienza en el vértice 5, llega al vértice 4 y termina en el 2 (5.4.2). El segundo recorre de la misma manera los primeros dos vértices pero su último vértice es el 3 (5.4.3). Por su parte el operador “+” solo indica que ambos caminos forman parte del mismo grafo.

La complejidad de búsqueda es muy sencilla por lo que no se gastaría mucho tiempo de procesamiento. Gupta afirma que la memoria requerida fue menor en comparación con otros enfoques similares pero no indica cuáles.

Una de las limitaciones de éste enfoque, es que no puede identificar aquellos patrones que tienen relación consigo mismos debido a que se crearía un tipo de ciclo que no podría ser representado por una expresión SOP.

Otro de los métodos que utilizan matrices para encontrar patrones de diseño, es el propuesto por Akshara [48], en el que se hace uso del isomorfismo. El isomorfismo es una técnica de correspondencia de grafos. Si existe una correspondencia uno a uno entre los nodos y las aristas de dos grafos, puede decirse que son isomorfos el uno al otro. El método de Akshara toma como entrada dos grafos que se obtienen en las fases anteriores, el primero representa las clases y relaciones del sistema de software y el segundo representa el patrón de diseño. Tales grafos son posteriormente convertidos a matrices de adyacencia para luego hacer el cálculo de la siguiente fórmula, que se aplica para conocer el isomorfismo entre dos matrices:

$$M2 = P\ M1\ P^T$$

Donde M1 corresponde a la matriz de adyacencia del sistema de software, M2 corresponde al patrón de diseño, P corresponde a la matriz de permutación del sistema de software y P^T corresponde a la matriz de permutación transpuesta.

Si se satisface la ecuación, entonces se determina la existencia del patrón de diseño.

En este trabajo no es posible detectar patrones de diseño con variaciones en su implementación. Su herramienta de soporte solo funciona con código de SmallTalk.

Finalmente, Antoniol et al. [37] propuso un método que toma como entrada un AOL, el cual se obtiene en fases anteriores, para luego extraer en ésta fase un conjunto de métricas, tales como el número de métodos privados, públicos y protegidos; número de atributos públicos, privados y protegidos; y número de relaciones de asociación, agregación y herencia. Todas estas métricas son sometidas a un conjunto de reglas que deben cumplir para que un componente llamado Constraints Evaluator determine la existencia de un patrón de diseño.

Este trabajo solamente puede detectar patrones de diseño que utilicen las relaciones de asociación, agregación y herencia. Su herramienta de soporte solamente trabaja con sistemas de software codificados en lenguaje de programación Java pero que es accesible vía web.

La Tabla 5 muestra los patrones de diseño detectados por cada uno de los trabajos estudiados.

Tabla 5. Patrones de diseño detectados por cada uno de los trabajos estudiados

	Kramer	Hayashi	Huang	Kirasić	Balanyi	Chen	Shi	Gupta	Tsantalis	Dong
Adapter	Sí		Sí		Sí		Sí		Sí	Sí
Bridge	Sí		Sí				Sí			
Decorator	Sí		Sí			Sí	Sí		Sí	Sí
Proxy	Sí	Sí	Sí		Sí					
Abstract Factory		Sí					Sí	Sí		
Builder		Sí			Sí					
Chain of responsability		Sí								
Composite		Sí	Sí			Sí	Sí		Sí	Sí
Factory method		Sí					Sí		Sí	
Observer		Sí							Sí	
State		Sí							Sí	Sí
Strategy		Sí			Sí				Sí	
Template method		Sí			Sí	Sí			Sí	
Visitor		Sí							Sí	
Singlenton		Sí		Sí			Sí	Sí	Sí	
Prototype					Sí				Sí	
Visitor					Sí					
Façade							Sí	Sí		
Flyweight							Sí			
Command									Sí	

3.4 Visualización

Ésta fase se refiere a la manera en que los resultados son presentados al usuario. Todos los trabajos estudiados presentan al usuario una contabilidad de los patrones detectados de manera textual. Algunos ejemplos son los trabajos de Hayashi (Tabla 6), Dong (Ilustración 2) y Tsantalis (Tabla 7).

Tabla 6. Resultados de Hayashi

		Hierarchical conditions		Non-hierarchical conditions	
		Program #1	Program #2	Program #1	Program #2
Common conditions	for meta patterns	234	22078		
	for connection	218	12532		
Meta patterns	1:1 Connection	141	2015		
	1:N Connection	94	2719		
	1:1 Recursive Connection	156	1906		
	1:N Recursive Connection	62	2000		
	Unification	47	3678		
	1:1 Recursive Unification	62	1828		
	1:N Recursive Unification	94	5187		
Design patterns	Abstract Factory	62	2203	1031	57219
	Builder	203	2828	984	56844
	Chain of Responsibility	63	2234	204	27672
	Composite	156	1906	1250	55891
	Factory Method	47	2203	297	24359
	Observer	204	3297	1078	59578
	Proxy	156	3843	1109	60828
	State	187	2156	1016	56266
	Strategy	187	2125	1032	55859
	Template Method	62	2203	265	24141
	Visitor	156	2047	1203	56031
	Singleton	109	1969	110	1875
Total		2700	82957	9579	536563

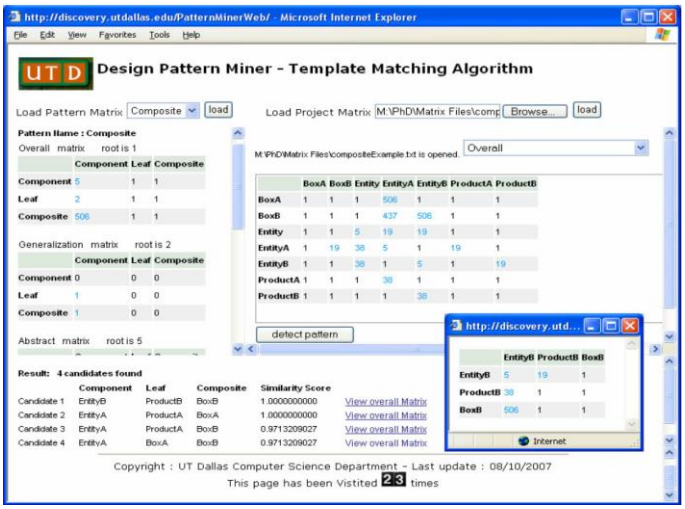


Ilustración 2. Aplicación y resultados de Dong

Tabla 7. Resultados de Tsantalis.

	JHotDraw v5.1			JRefactory v2.6.24			JUnit v3.7		
Design Patterns	***TP***	***FN***	***Recall***	***TP***	***FN***	***Recall***	***TP***	***FN***	***Recall***
Adapter/Command*	18	0	100%	7	0	100%	1	0	100%
Composite	1	0	100%	0	0	100%	1	0	100%
Decorator	3	0	100%	1	0	100%	1	0	100%
Factory Method	2	1	66.7%	1	3	25%	0	0	100%
Observer	5	0	100%	0	0	100%	4	0	100%
Prototype	1	0	100%	0	0	100%	0	0	100%
Singleton	2	0	100%	12	0	100%	0	0	100%
State/Strategy	22	1	95.6%	11	1	91.6%	3	0	100%
Template Method	5	0	100%	17	0	100%	1	0	100%
Visitor	1	0	100%	2	0	100%	0	0	100%

3.5 Resumen

En las siguientes secciones se resumirán las ideas principales de cada una de las fases estudiadas.

3.5.1 Extracción de información

- Es posible extraer dos tipos de información: de clases y relaciones o solamente de clases.
- Las clases y las relaciones pueden dar la información necesaria para describir cualquier estructura de un sistema y por lo tanto son los artefactos primordiales a ser extraídos.
- Existen dos tipos de herramientas para extraer información acerca de clases y relaciones: analizadores sintácticos y herramientas CASE de ingeniería inversa.
- Los analizadores sintácticos están basados en JDT y Java Reflection.
- JDT y Java Reflection son herramientas que solamente trabajan con código fuente en lenguaje Java.
- Existen herramientas CASE de ingeniería inversa privativas y de código abierto.
- ANTRL parser es un framework para generar analizadores sintácticos.
- La mayoría de las herramientas de extracción están diseñadas para funcionar con un determinado lenguaje de programación.

3.5.2 Almacenamiento

- Existen dos tipos de almacenamiento: el persistente y el no persistente.
- Los trabajos que emplean el almacenamiento no persistente comúnmente utilizan árboles, matrices y grafos.
- Los trabajos que emplean el almacenamiento persistente comúnmente utilizan archivos en formato de texto plano o bien en XML.
- La principal ventaja del uso del almacenamiento persistente es que con la misma fuente de datos es posible exportar la información hacia los artefactos de análisis.

3.5.3 Análisis

- Existen dos tipos de análisis: el basado en verificación y el basado en medición.
- Los métodos de verificación se basan en el cumplimiento de un conjunto de rubros y de esa manera se logra establecer la probable existencia de un patrón.
- Los métodos basados en lógica sólo pueden extraer patrones sin variaciones en su implementación.

- El análisis estructural busca una correspondencia exacta de las características de un patrón en el código fuente.
- Similarity score mide la similitud entre los vértices de un grafo.
- Template matching es utilizado en el campo de visión computacional.
- A través de la correlación cruzada es posible saber la correspondencia de un patrón en una matriz que representa las clases y relaciones de un sistema de software.
- Las expresiones SOP son utilizadas comúnmente en el campo de álgebra booleana.
- El isomorfismo es una técnica de correspondencia entre grafos.
- El isomorfismo sólo puede aplicarse a correspondencias uno a uno, es decir, a correspondencias exactas.
- Las métricas más comúnmente utilizadas para obtener patrones de diseño son: el número de atributos privados, públicos y protegidos; el número de métodos privados, públicos y protegidos; y el número de relaciones de asociación, agregación y herencia.
- En teoría, una verificación combinada con mediciones puede conducir a la detección de una gran parte de los patrones de diseño catalogados por Gamma [3] de una manera automatizada, y con un grado de exactitud mayor que los enfoques manuales anteriormente descritos.
- Ningún método automático promete la detección de todos los patrones de diseño catalogados por Gamma [3].
- La mayoría de los trabajos detecta los patrones Adapter, Bridge, Decorator y Composite.

3.5.4 Visualización

- Hasta el momento, ninguno de los métodos estudiados representa sus resultados de manera gráfica.

4 Enfoque

En éste capítulo se describe el proceso llevado a cabo para lograr la detección de patrones de diseño desde código fuente en el lenguaje de programación Java.

4.1 Proceso para detectar patrones de diseño

En el Capítulo 1, Sección 1.4 se describió la estrategia de la solución propuesta y el proceso a seguir. En éste capítulo se profundiza en ese proceso. Particularmente se detallan los métodos y herramientas que fueron utilizadas en cada fase del proceso. Esto está ilustrado en la Figura 17.

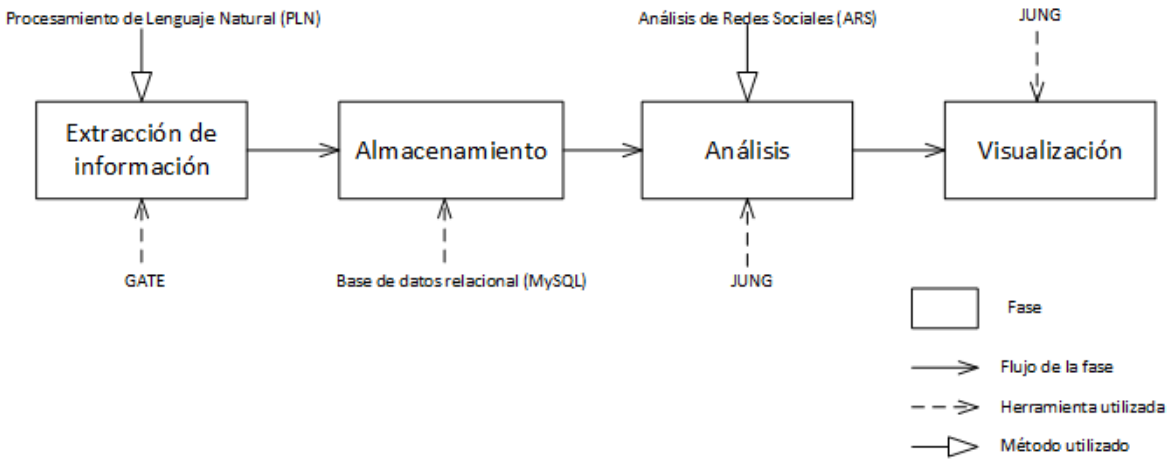


Figura 17. Estrategia de la solución

4.1.1 Extracción de la información

Hemos explicado que la extracción de información tiene como propósito analizar el código fuente para obtener diversos elementos de interés que permitan la identificación de patrones. De acuerdo a lo revisado en el Capítulo 3, Sección 3.1, los métodos y herramientas existentes sólo son útiles para determinados lenguajes de programación. Por lo tanto, ni los analizadores sintácticos ni las herramientas CASE de ingeniería inversa permiten la adaptación/extensión a diversos lenguajes de programación.

La fase de extracción de información es primordial ya que de ella dependen en gran medida los resultados de la fase de análisis, y por consecuencia la precisión del método.

En la Sección 3.1 de éste trabajo de investigación, pudimos observar que existen diferentes métodos y herramientas que han sido utilizados en la Fase de Extracción de la Información, en la Figura 18 se puede observar en color rojo, que en este trabajo extraemos clases y relaciones del código fuente utilizando PLN

mediante la herramienta GATE. A continuación describimos dicha herramienta y cómo fue utilizada.

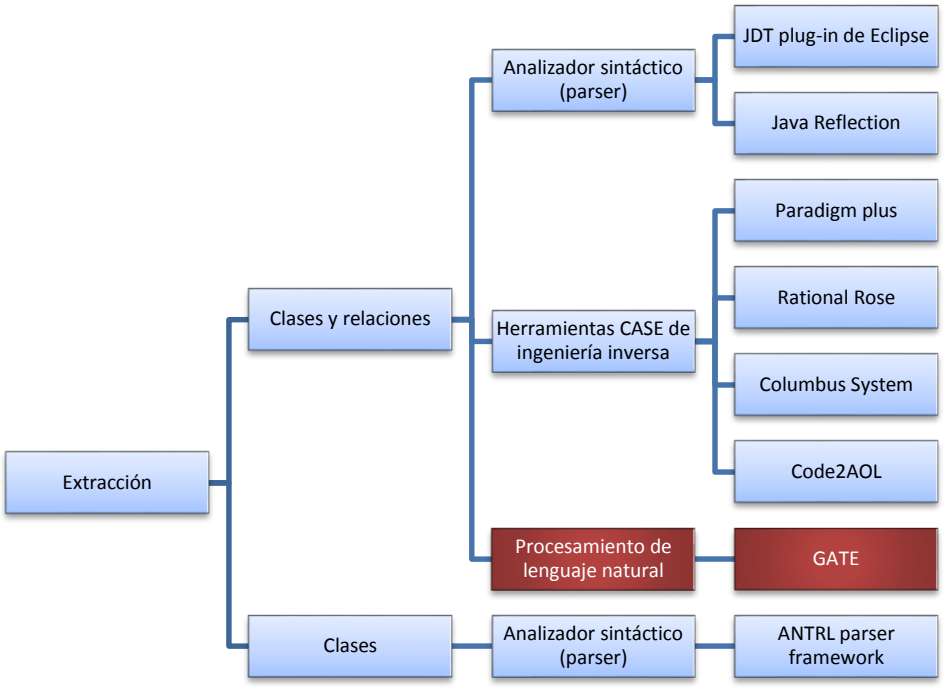


Figura 18. Métodos y herramientas de extracción de información

4.1.1.1 GATE

GATE (General Architecture for Text Engineering) [49] es una herramienta de código abierto, basada en Java, para resolver diversos problemas en el dominio del Procesamiento del Lenguaje Natural. GATE puede verse como una arquitectura, un framework y un ambiente de desarrollo. Como arquitectura, define la organización y la asignación de responsabilidades entre diferentes componentes. Como framework, GATE proporciona un conjunto de componentes de procesamiento que se pueden reusar, extender o adaptar. Por último, como ambiente de desarrollo, ayuda al usuario a disminuir el tiempo de desarrollo a través de una API y una GUI.

GATE se distribuye como una colección de componentes de procesamiento (“processing resources”) que actúan sobre documentos de texto no estructurado. Una de estas colecciones es denominada ANNIE (A Nearly-New Information Extraction System). Como se muestra en la Figura 19 los componentes de ANNIE están organizados en una arquitectura pipeline. La arquitectura pipeline permite a un conjunto de componentes ir transformando un flujo de datos de forma secuencial, siendo la entrada de cada componente la salida obtenida por el procesamiento realizado por el componente anterior. En ANNIE la salida del último componente es un documento anotado. Una anotación puede verse como un metadato, es decir, datos acerca de otros datos, y son utilizados en GATE para identificar cada elemento de un texto (token) a través de un ID y de un tipo (type).

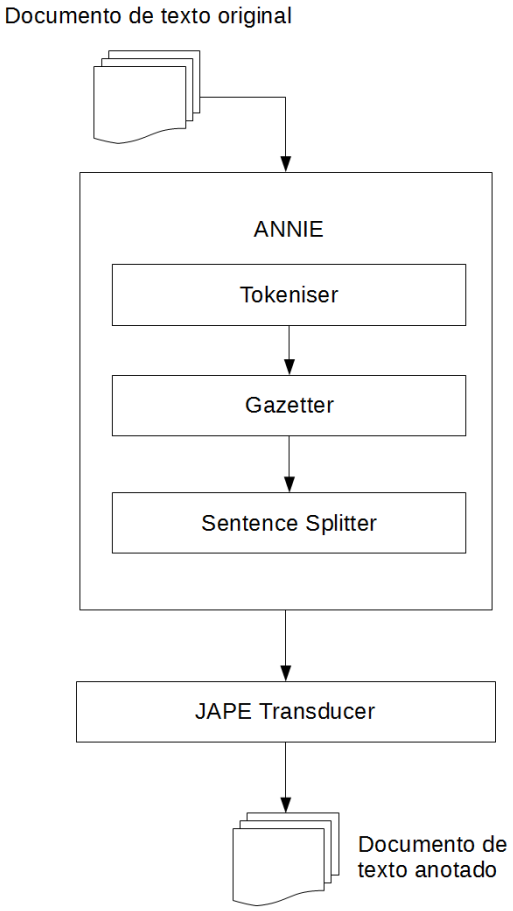


Figura 19. Componentes de procesamiento de GATE

Los diferentes componentes de procesamiento generan anotaciones de distintos tipos. Enseguida se explica qué anotaciones es capaz de realizar cada uno de los componentes de procesamiento de ANNIE y cómo fueron utilizados en este trabajo de tesis:

- **Tokeniser**. Éste componente tiene como finalidad separar el texto en tokens simples. Por default, Tokeniser es capaz de reconocer números, símbolos de puntuación, y palabras de diferentes tipos. En éste trabajo se realizan modificaciones con el objetivo de permitir el reconocimiento de nombres de variables en el código.

- **Gazetteer**. Este componente identifica a cada token de acuerdo a una serie de listas de palabras, que se encuentran almacenadas en archivos de texto. Cada lista representa un conjunto de nombres, por ejemplo nombres de ciudades, organizaciones, días de la semana, etcétera. Estas listas deben ser descritas cada una en un archivo diferente con la extensión ".lst". Además, debe existir otro archivo de texto nombrado como "lists.def", el cual es utilizado como un índice para acceder a cada una de las listas en particular. Para este trabajo de tesis y con la intención de detectar clases y relaciones se han agregado un conjunto de listas de palabras reservadas de la gramática del lenguaje de programación clasificadas de acuerdo a su tipo. Las listas agregadas en este trabajo y su contenido son mostrados en la Tabla 8.

- **Sentence Splitter**. Éste componente tiene como finalidad segmentar el texto en sentencias. En éste trabajo de tesis, se agregó una regla JAPE para segmentar el código fuente en clases.

Es necesario ejemplificar el proceso seguido por la arquitectura ANNIE mostrada en la Figura 19. Un documento de texto, en nuestro caso, código fuente, es consumido por el componente Tokeniser. El Tokeniser divide el texto en tokens y agregará anotaciones de tipo nombre de variable, símbolos, puntuación, etcétera y le asignará un ID a cada uno de los tokens según corresponda. El texto anotado resultante será consumido posteriormente por el componente Gazetter y éste le asignará un ID y un tipo a cada uno de los tokens que concuerden con las listas que han sido agregadas. Finalmente Sentence Splitter utiliza las anotaciones que ha realizado el componente Gazetter para segmentar el texto en clases y atributos de clase.

Tabla 8. Listas agregadas al componente FS Gazetteer Lookup.

Nombre de la lista	Elementos de la lista	Descripción de la lista
class.lst	class Class	Dependiendo del lenguaje de programación la definición de una clase puede empezar con la palabra reservada class o Class.
collections.lst	List Map Set Arraylist HashMap TreeMap Iterator HashSet TreeSet LinkedList ArrayList	Lista de palabras reservadas de tipos de datos genéricos que son capaces de guardar colecciones de datos.
datatype.lst	int float char boolean short long double byte	Lista de palabras reservadas que representan tipos de datos.
extends.lst	extends	Lista de palabras reservadas que representan la relación de herencia.
instance.lst	new getInstance getinstance	Lista de palabras que representan la instancia de un objeto.
interface.lst	interface implements	Lista de palabras reservadas que representan la relación de realización.
modifier.lst	static void main String	Lista de modificadores tanto de clases como de métodos y atributos.
tiposclase.lst	public private protected	Lista de tipos de alcance de una clase, método o atributo.

Otro componente de procesamiento de GATE es el JAPE Transducer (Java Annotations Patterns Engine) [50], también mostrado en la Figura 19. JAPE Transducer es un componente que permite efectuar reconocimiento de patrones de texto en un documento de texto previamente anotado. Además del texto anotado, JAPE utiliza como insumo una gramática, definida por el usuario, que está especificada como reglas en el lenguaje JAPE. De esta forma, si los patrones especificados en las reglas definidas en la gramática se satisfacen con las anotaciones generadas hasta el momento, el JAPE Transducer realiza una acción que también está especificada en la regla de la gramática correspondiente.

De esta forma, la gramática es un conjunto de reglas JAPE tipo “patrón – acción”, que ejecuta el JAPE Transducer en forma sucesiva sobre el texto anotado. Las reglas JAPE tienen especificado en el Lado Izquierdo (LI) un patrón en términos de anotaciones y en el Lado Derecho (LD) especifican una acción a ser ejecutada si es que el LI se satisface.

El LI de la regla es una expresión regular y, además de las anotaciones generadas hasta el momento por los componentes de procesamiento de ANNIE, puede contener los operadores (*", \?", \|", \+"). La regla es en sí una nueva anotación, de esta forma el LD de la regla contiene el nombre de la anotación que se debe agregar al conjunto de anotaciones existentes en caso de que se satisfaga el LI. También es posible que el LD contenga un bloque válido de código Java y es ésta propiedad que hace a JAPE una herramienta muy potente y flexible. El conjunto de archivos que contienen las reglas JAPE deben respetar la sintaxis del lenguaje JAPE y se sugiere disponer de un primer archivo de nombre “main.jape” a partir del cual se definen los archivos que componen la gramática.

En éste trabajo de tesis se emplearon algunas reglas JAPE que fueron definidas por el usuario para detectar los siguientes elementos en el código fuente.

- Clases.
- Interfaces.
- Relaciones de herencia, implementación, agregación, composición y asociación.

En las secciones subsecuentes se especificará cada una de ellas de manera textual y se darán algunos ejemplos de reglas JAPE.

4.1.1.2 Detección de elementos en el código fuente

Las reglas que se usaron para trabajar con código fuente fueron definidas tomando como referencia la manera en como Phillipow [51] extendió el enfoque llamado “Minimal Key Structures” en el que se definen tanto criterios de búsqueda positivos como negativos. Los criterios de búsqueda positivos (CP) son aquellas cualidades ideales que cierto fragmento de texto tiene que cumplir para que sea considerado como una coincidencia, pero también hay que tomar en cuenta aquellos aspectos con los que se debe tener cuidado en caer; es decir, aquellas cualidades no deseables que cierto fragmento de texto tiene que cumplir para ser descartado como posibilidad de coincidencia. Al identificar estos aspectos se elimina la mayor parte de resultados falsos positivos.

4.1.1.2.1 Detección de relaciones entre clases

Las clases e interfaces son detectadas durante la ejecución de las reglas JAPE en el componente JAPE Transducer para extraer relaciones entre clases.

Las relaciones son difíciles de extraer debido a que el léxico y la sintaxis del lenguaje de programación limitan los elementos que se puedan extraer de él. Una de esas limitaciones son los tipos de relaciones que puedan ser implementadas en un lenguaje de programación. Por ejemplo, en el lenguaje de programación Java existe la palabra reservada "*implements*" que define una relación de realización o de implementación -como también es conocida- entre dos clases. Sin embargo, existen muchos otros lenguajes en los que no existe esta palabra reservada, incluso ni siquiera el concepto de interfaces, tal es el caso de C++, Objective C y Python, por mencionar algunos.

Aunque sucede esto con diversos lenguajes de programación, se pueden dar diversas soluciones al problema, pero se entra en ambigüedades al haber diferentes maneras de implementar algún concepto y por lo tanto no es posible definir reglas en GATE de manera general, es decir, que funcionen para todos los lenguajes de programación.

Por otro lado, se estableció un proceso para elegir un conjunto de tipos de relaciones ya que existe una gran diversidad de ellas definidas en UML, por ejemplo, asociación, asociación unidireccional, asociación bidireccional, agregación, composición, etcétera. Este proceso consistió principalmente en contabilizar los tipos de relaciones que son usadas en los patrones de diseño de Gamma [3] con la finalidad de establecer cuáles eran las más comunes.

Como resultado de este proceso se obtuvieron las siguientes: generalización o herencia, realización o implementación, composición, agregación y asociación.

Enseguida se describen los criterios tanto positivos (CP) como negativos (CN) que debe cumplir cada una de ellas:

Generalización o herencia. Existe una palabra reservada en el lenguaje Java con la cual es posible captar el momento exacto en el que sucede este tipo de relación. Esta palabra es "*extends*", sin embargo también se tienen que cumplir los criterios mostrados en la Tabla 9.

Tabla 9. Criterios que deben considerarse para extraer la relación de Generalización (herencia).

Criterios positivos (CP)	Criterios negativos (CN)
CP1. El primer token debe de ser de tipo ALCANCE. ALCANCE comprende las palabras: public, private o protected. Éste token puede no existir (Es opcional).	CN1. El token anterior al CP1 no debe ser de tipo COMENTARIO_APERTURA. COMENTARIO_APERTURA comprende los símbolos: "//" y "/*".
CP2. El segundo token debe de ser de tipo TIPO_CLASE. TIPO_CLASE comprende las palabras: abstract, static, final, synchronizable. Éste token puede no existir (Es opcional)	
CP3. El tercer token debe ser de tipo DECLARACION_CLASE. DECLARACION_CLASE comprende la palabra class.	
CP4. El cuarto token pertenece a un nombre de clase. Éste token es etiquetado con el nombre *clase_hija*. Ésta clase se considera como la clase hija de la relación de generalización.	
CP5. El quinto token debe ser de tipo HERENCIA. HERENCIA comprende la palabra *extends*.	
CP6. El sexto token pertenece a un nombre de clase. Éste token es etiquetado con el nombre *clase_padre*. Ésta clase se considera como la clase padre de la relación.	

El LI de la expresión regular que especifica una de las reglas JAPE para detectar la relación de generalización en términos de los criterios definidos en la

Tabla 9, se muestra en la Figura 20. Hemos indicado los criterios especificados en la regla JAPE.

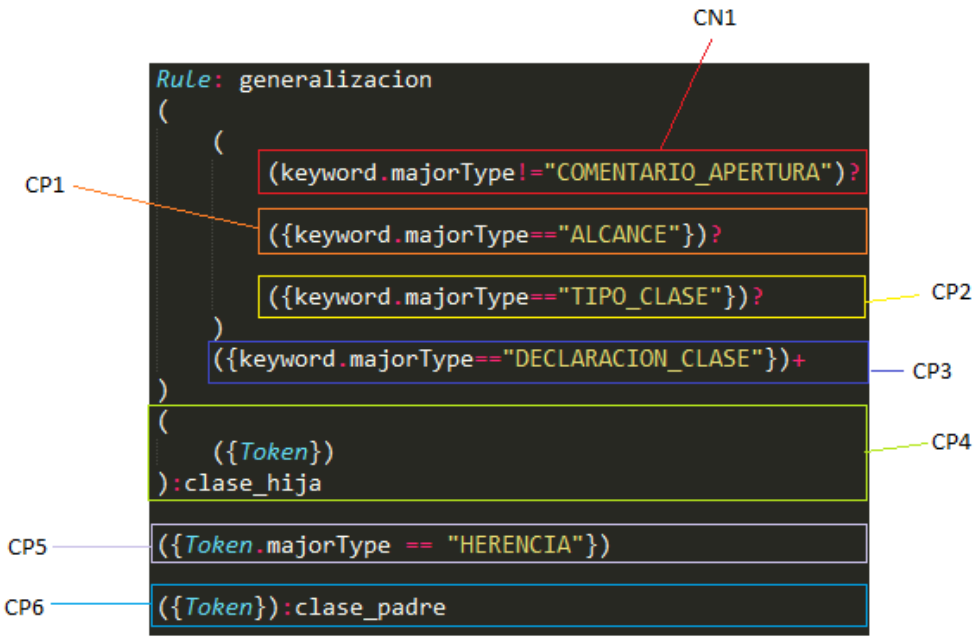


Figura 20. Lado izquierdo de la Regla JAPE para la detección de la relación de generalización.

En la Figura 21 se muestra un fragmento de código fuente escrito en Java y se destaca la correspondencia entre anotaciones y el orden que debe guardar con respecto a la expresión regular definida en el LI de la regla JAPE mostrada en la Figura 20.

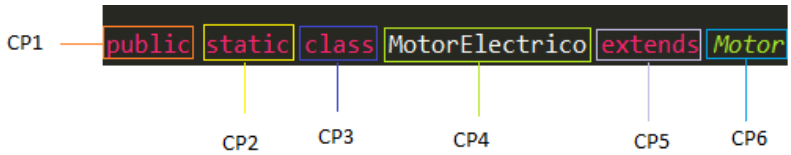


Figura 21. Correspondencia entre el LI de la Regla JAPE y el código fuente

Realización o implementación. Es una de las relaciones más fáciles de detectar debido a que existe una palabra reservada en el lenguaje Java con la cual es posible captar el momento exacto en el que sucede este tipo de relación. Esta palabra es “implements”, sin embargo también se tienen que cumplir los criterios mostradas en la Tabla 10.

Tabla 10. Criterios que deben considerarse para extraer la relación de Realización (implementación).

Criterios positivos	Criterios negativos
El segundo token debe ser de tipo ALCANCE. ALCANCE comprende las palabras: public, private o protected. Éste token puede no existir (Es opcional).	El primer token no debe ser de tipo COMENTARIO_APERTURA. COMENTARIO_APERTURA comprende los símbolos: "//" y "/*".
El tercer token debe ser de tipo TIPO_CLASE. TIPO_CLASE comprende las palabras: abstract, static, final, synchronizable. Éste token puede no existir (Es opcional)	
El cuarto token debe ser de tipo DECLARACION_CLASE. DECLARACION_CLASE comprende la palabra class.	
El quinto token pertenece a un nombre de clase. Éste token es etiquetado con el nombre *clase_hija*. Ésta clase se considera como la clase hija de la relación de generalización.	
Los siguientes tokens pueden no existir: El sexto token debe ser de tipo HERENCIA. HERENCIA comprende la palabra *extends*. El séptimo token pertenece a un nombre de clase. Ésta clase se considera como la clase padre de la relación.	
El siguiente token debe ser de tipo IMPLEMENTACION. IMPLEMENTACIÓN comprende la palabra *implements*.	
Los siguientes tokens pertenecen al nombre de interfaces. Los nombres de interfaces están separados por el signo de puntuación ','	

Composición. La implementación de las relaciones en el lenguaje Java de aquí en adelante ha sido extraída del libro de Deitel H. [52].

En el trabajo de Pallioto D. [53] se establece que en la relación de composición "el tiempo de vida del objeto está condicionado por el tiempo de vida del que lo incluye". El siguiente enunciado ejemplifica mejor éste concepto: una bicicleta está compuesta por un timón, pedales, asiento y ruedas, así que todos estos deben

existir antes de que se construya la bicicleta. Si la bicicleta desaparece, entonces también desaparecerán sus componentes.

El siguiente diagrama UML representa la relación que tiene la Bicicleta con cada uno de sus componentes:

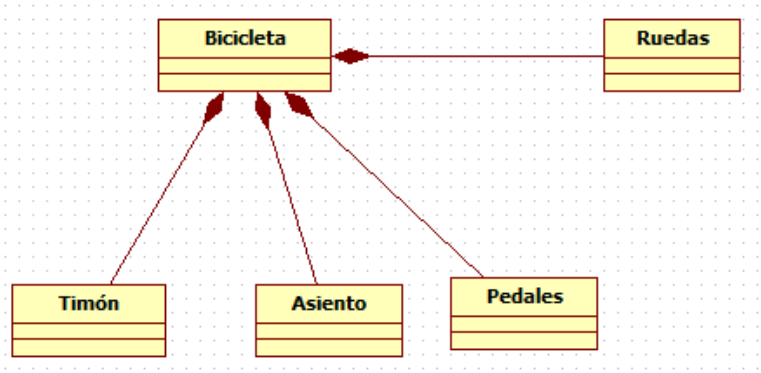


Figura 22. Ejemplo de la relación de composición en UML

El diagrama de la Figura 22 traducido a código fuente en lenguaje de programación Java quedaría de la siguiente manera:

```
class Bicicleta{
    public Timon timon;
    public Asiento asiento;
    public Pedales pedales;
    public Ruedas ruedas;

    public Bicicleta(){

        timon = new Timon();
        asiento = new Asiento();
        pedales = new Pedales();
        ruedas = new Ruedas();
    }
}
```

Figura 23. Ejemplo de la implementación de la relación de composición en Java

Sin embargo, la implementación mostrada en la Figura 23 no es la única que puede haber. Puede existir la siguiente variante:

```
class Bicicleta{
    public Timon timon = new Timon();
    public Asiento asiento = new Asiento();
    public Pedales pedales = new Pedales();
    public Ruedas ruedas = new Ruedas();

    public Bicicleta(){


    }
}
```

Figura 24. Ejemplo de la implementación de la relación de composición en Java

Si la clase Bicicleta es instanciada, Java primero creará los objetos Timon, Asiento, Pedales y Ruedas, y posteriormente creará el objeto Bicicleta. En ambas variantes se puede observar que si se destruye el objeto Bicicleta sus demás componentes también dejarán de existir.

También es posible crear objetos dentro de métodos. Sin embargo, el tiempo de vida de los componentes dependen del tiempo de vida del objeto compuesto, y si se crean dentro de un método nada asegura que ese método sea ejecutado y por lo tanto no se asegura la creación o destrucción de tales componentes y por consecuencia se rompería con la definición de la relación de composición.

Para detectar ésta relación fue necesario ejecutar dos fases. En la primera se detecta el nombre de la clase, tal como se exhibe en la Tabla 11.

La Tabla 11 se traduce a reglas JAPE tal como se muestra en la Figura 25.

```
(
    (
        (keyword.majorType=="COMENTARIO_APERTURA")?
        ({keyword.majorType=="ALCANCE"})?
        ({keyword.majorType=="TIPO_CLASE"})?
    )
    ({keyword.majorType=="DECLARACION_CLASE"})+
)
(
    ({Token})
):clase_fuente
```

Figura 25. LI de la Regla JAPE, para extraer el nombre de la clase fuente de la relación de composición

Tabla 11. Primera fase, extracción de nombres de clases

Criterios positivos	Criterios negativos
El segundo token debe ser de tipo ALCANCE. ALCANCE comprende las palabras: public, private o protected. Éste token puede no existir (Es opcional).	El primer token no debe ser de tipo COMENTARIO_APERTURA. COMENTARIO_APERTURA comprende los símbolos: "//" y "/*".
El tercer token debe ser de tipo TIPO_CLASE. TIPO_CLASE comprende las palabras: abstract, static, final, synchronizable. Éste token puede no existir (Es opcional)	
El cuarto token debe ser de tipo DECLARACION_CLASE. DECLARACION_CLASE comprende la palabra class.	
El quinto token pertenece a un nombre de clase. Este token es etiquetado con el nombre *clase_fuente*.	

En la Figura 26, se muestra, a manera de ejemplo, la correspondencia de los nombres de clases en la primera fase de detección de la relación de composición.

Figura 26. Primera fase, detección de nombres de clase

En la segunda fase, se extraen los nombres de los constructores y las instancias dentro de ésta, tal como puede observarse en la Tabla 12.

Tabla 12. Segunda fase, extracción de nombres de constructores e instancias

Criterios positivos	Criterios negativos
El segundo token debe ser de tipo ALCANCE. ALCANCE comprende las palabras: public, private o protected. Éste token puede no existir (Es opcional).	El primer token no debe ser de tipo COMENTARIO_APERTURA. COMENTARIO_APERTURA comprende los símbolos: "//" y "/*".
El tercer token es etiquetado como *nombre_constructor.*	
El cuarto token debe ser estrictamente un paréntesis de apertura "("	
El quinto token debe ser un paréntesis de cierre ")".	
El sexto token debe pertenecer a una llave de apertura "{"	
Los siguientes tokens deben pertenecer a la declaración de instancias. Una declaración de instancia se detecta de la siguiente manera: El primer token debe pertenecer a un NOMBRE_VARIABLE. Seguido de un token de '='. Luego de un token de tipo INSTANCIA. INSTANCIA está conformado por la palabra "new". El siguiente token es etiquetado con el nombre: clase_destino.	

La Tabla 12 se traduce a reglas JAPE tal como se muestra en la Figura 27.

```
(

    (
        ({keyword.majorType=="ALCANCE"})
    )


    (
        ({Token})
    ):nombre_constructor

    (
        ({Token.string =="("})
        ({Token.string ==")"})
        ({Token.string =="{"})
        ({keyword.majorType=="NOMBRE_VARIABLE"})
        ({Token.string =="="})
        ({keyword.majorType=="INSTANCIA"})
    )

    (
        ({Token})
    ):clase_destino

)
```

Figura 27. LI de la regla JAPE para detectar nombres de constructores e instancias

En la Figura 28, se muestra, a manera de ejemplo, la correspondencia de los nombres de clases en la segunda fase de detección de la relación de composición.

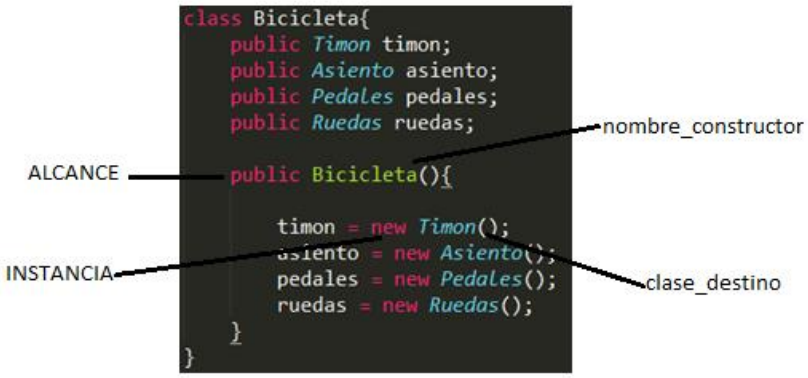


Figura 28. Segunda fase, detección de constructores e instancias

Mientras tanto, en el lado derecho de la regla JAPE para detectar una relación de composición se siguieron los siguientes pasos:

1. Se crea una variable de tipo anotación que almacenará las clases fuente detectadas, las clase destino detectadas y los constructores detectados.

```
AnnotationSet Fuente = bindings.get("clase_fuente");
AnnotationSet Destino =bindings.get("clase_destino");
AnnotationSet Clase =bindings.get("nombre_constructor");
```

2. Se declara una nueva variable para recorrer el conjunto de anotaciones.

```
Annotation destino = Destino.iterator().next();
Annotation fuente = Fuente.iterator().next();
Annotation clase = Clase.iterator().next();
```

3. Se obtiene cada anotación en tipo cadena (*string*).

```
String match=clase.getFeatures().get("string").toString();
```

4. Se realiza una comparación con todos los constructores para saber si tal constructor es parte de la clase detectada.

```
if(match.equals(source.getFeatures().get("string").toString())){
    db->guardar_relacion_composicion(fuente, destino)
}else{
}
```

Si la condición anterior es verdadera, entonces se ha detectado una relación de composición.

Las reglas de agregación y asociación serán descritas de manera más textual con la finalidad de que sea más comprensible.

Agregación. A diferencia de la relación de composición, en la agregación, según Deitel H. [52], “el tiempo de vida del objeto incluído es independiente del que lo incluye”. Por ejemplo, una empresa *“agrupa”* a varios clientes, pero no dependen uno del otro para existir. Este caso puede ser representado en UML de la siguiente manera:

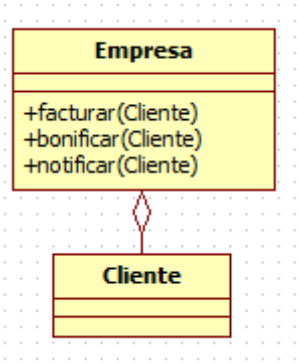


Figura 29. Ejemplo de la representación de la relación de agregación en UML

El diagrama de la Figura 29 puede ser representado en código fuente Java de la siguiente manera (Figura 30):

```
public class Empresa
{
    public Empresa(){

    }

    public void facturar(Cliente cliente){}

    public void bonificar(Cliente cliente){}

    public void notificar(Cliente cliente){}

    public static void main(String args[]){}
}

class Cliente
{

}
```

Figura 30. Ejemplo de la implementación de la agregación en Java

Como se puede observar no es necesario que el objeto “Cliente” exista antes que la clase “Empresa” para poder funcionar. Solamente aquellos métodos que necesiten información de un cliente harán uso de ellos. Un método no solamente es capaz de recibir objetos como parámetros sino también interfaces. Las interfaces aseguran que el objeto recibido como parámetro implemente ciertos métodos en particular.

Se pueden describir las siguientes reglas para detectar las relaciones de agregación dentro del código fuente:

Asociación. La única particularidad que existe en las asociaciones a diferencia de otras relaciones es la navegabilidad. En una relación de asociación entre dos clases es posible navegar de uno a otro.

Las asociaciones en UML suelen ser dibujadas cuando se desea representar una relación estructural en el modelo, por ejemplo: pedidos que contienen productos, empleados que trabajan en departamentos, estudiantes que asisten a cursos, etcétera. En todos estos casos es necesario poder navegar entre los elementos relacionados, por ejemplo: ¿Qué empleados trabajan en qué departamento? y viceversa, ¿Qué productos tienen un pedido?, etcétera.

Existen diversos tipos de asociación, y se clasifican en función a su multiplicidad. Enseguida se definen cada una de estas relaciones, se muestra su representación en UML, se traducen a código Java, y finalmente se describen los criterios positivos y negativos para su detección en GATE:

- **Asociaciones 1-1**. Son aquellas que relacionan exactamente a dos clases, por ejemplo, suponga que una persona dirige un departamento, en UML se representa de la siguiente manera:

Figura 31. Asociación 1-1 entre Persona y Departamento

La relación de asociación 1-1 entre Persona y Departamento puede ser representada en código fuente de Java de la siguiente manera:

```java
public class Persona
{
  private Departmento departamentoDirigido;

  public Departmento getDepartmento()
  {
    return departamentoDirigido;
  }

  public void setDepartmento(Departmento departmento)
  {
    this.departamentoDirigido= departmento;
  }
}

public class Departmento {
  private Persona ;

  public Persona getDirector()
  {
    return director;
  }

  public void setDirector(Persona director)
  {
    this.director = director;
  }
}
```

Figura 32. Ejemplo de la relación de asociación 1-1 en Java.

Se pueden describir las siguientes reglas para detectar las relaciones de asociación 1-1 dentro del código fuente:

Criterios positivos	Criterios negativos
Se deben considerar todas aquellas clases que tengan otra clase como Atributo.	No se deben considerar atributos de clase que sean de tipos de datos simples, por ejemplo: integer, string, float, double, etcétera.
Se debe explorar los atributos de la clase para saber si ésta contiene un atributo con el nombre de la primera clase coincidente, si es así, se ha encontrado una relación de asociación.	No se deben considerar atributos de clase que sean de tipo envoltura, por ejemplo: Integer, String, Float, Double, etcétera.
	Se deben descartar todas aquellas coincidencias que se encuentren entre cualquier tipo de comentarios.

- **Asociaciones 1-n.** En este tipo de relaciones existe una clase que relaciona a varias clases de otro tipo. Por ejemplo, un departamento tiene varios empleados.

Figura 33. Asociación 1-n entre Departamento y Persona

En términos de implementación en lenguaje Java, la única diferencia con la asociación 1-1 es que la clase de la izquierda (Departamento) contendrá como atributo una colección de elementos de la derecha (Persona).

```java
public class Persona{
    private Departamento lugar_trabajo;

    public Departamento getLugarTrabajo(){
        return lugar_trabajo;
    }
}

public class Departamento{
    private ArrayList<Persona> empleados;

    public void AddEmpleado(Persona empleado){
        empleados.add(empleado);
    }

    public void RemoveEmpleado(Persona empleado){
        empleados.remove(empleado);
    }
}
```

Figura 34. Asociación 1-n entre Departamento y Persona

Se tienen que considerar una gran cantidad de palabras reservadas del lenguaje Java para almacenar colecciones de datos. Por ejemplo, ArrayList, List, Set, HashSet, TreeSet y LinkedList. Existen muchos más tipos para almacenar colecciones de datos, pero estos son los más utilizados.

Sí alguno de estos tipos es encontrado en los atributos de la clase, entonces hay que buscar nombres de clases, estos se encuentran entre los símbolos “<” y “>”. En la Figura 34 puede observar la clase “Persona”, la cual cumple con las descripciones anteriores.

Luego se tiene que buscar la clase Persona y buscar en sus atributos uno que tenga el nombre “Departamento”, si esto ocurre, entonces se encuentra con una relación de asociación 1-n.

Se pueden describir las siguientes reglas para detectar las relaciones de asociación 1-n dentro del código fuente:

Criterios positivos	Criterios negativos
Se deben considerar todas aquellas clases que tengan colecciones de clases, tales como ArrayList, List, Set, HashSet, TreeSet y LinkedList. Las cuales se encuentran encerradas entre los símbolos “<” y “>”	No se deben considerar atributos de clase que sean de tipos de datos simples, por ejemplo: integer, string, float, double, etcétera.
Las clases encontradas entre los símbolos “<” y “>” deben ser exploradas para buscar el nombre de la clase fuente.	No se deben considerar atributos de clase que sean de tipo envoltura, por ejemplo: Integer, String, Float, Double, etcétera.
Los arreglos de clases que se encuentren en los atributos de una clase deben ser considerados como clases destino.	Se deben descartar todas aquellas coincidencias que se encuentren entre cualquier tipo de comentarios.
	Se deben descartar los atributos de clase ArrayList, List, Set, HashSet, TreeSet y LinkedList que no tengan algún tipo descrito entre los símbolos de “<” (menor que) y “>” (mayor que).
	Se deben descartar los atributos de clase ArrayList, List, Set, HashSet, TreeSet y LinkedList que tengan un tipo propio del lenguaje, tales como String, Integer, int, etc., entre los símbolos de “<” (menor que) y “>” (mayor que).

Es posible encontrar arreglos de clases dentro de los atributos de una clase, estos arreglos de clases deben ser considerados y almacenados como la clase destino de la relación de asociación.

- **Asociaciones n-n.** Se trata de relaciones de múltiples clases de un tipo que contienen múltiples clases de otro tipo. Por ejemplo, varios empleados pueden estar trabajando en varios proyectos al mismo tiempo, en un momento dado.

Figura 35. Asociación n-n entre Empleado y Proyecto

En éste caso se tienen ahora colecciones en ambas clases para asociar los elementos de la otra clase. En términos de implementación en lenguaje Java, la única diferencia con la asociación 1-n es que la clase de la izquierda (Empleado) contendrá como atributo una colección de elementos de la derecha (Proyecto).

4.1.2 Almacenamiento.

Una vez que se han extraído las clases y las relaciones a partir del código fuente hay que establecer la manera de guardar dicha información para después generar, en base a ésta, la red del sistema.

El almacenamiento se realiza en dos momentos; el primero después de haber extraído la información, y el segundo al formar el grafo. Esto puede observarse en la Figura 36.

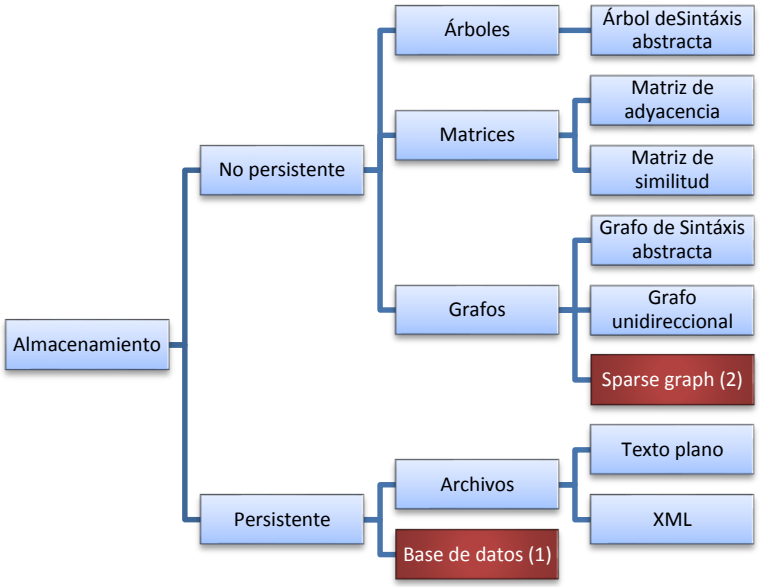


Figura 36. Métodos y herramientas de almacenamiento

Ninguno de los trabajos estudiados en el Capítulo 3, Sección 3.2, como puede observarse en la Figura 36, ha utilizado una base de datos relacional.

La intención del almacenamiento primero en una base de datos es que si se necesitara estudiar el código fuente de un sistema varias veces habría que ejecutar el proceso de extracción de información cada vez y se consumiría demasiado tiempo, de tal forma que se llegó a la conclusión de que la mejor opción era vaciar toda esa información a una base de datos donde la información pudiera estar disponible para posteriormente formar el grafo con la ayuda de estructuras de datos no persistentes, en éste caso JUNG lo hace internamente mediante listas enlazadas.

El método de almacenamiento fue basado en la manera en cómo Pajek lo realiza. Pajek es un programa para el análisis y visualización de redes extensas, usa archivos de texto plano donde se indica cuáles son los nodos y qué relaciones existen entre ellos, así que se decidió usar una estructura similar pero en una base de datos, que además tiene otras ventajas respecto a los archivos de texto plano, como la persistencia de datos, el control sobre la redundancia de datos, la búsqueda de información mediante consultas, entre muchas otras.

Para éste trabajo fue utilizado MySQL como servidor de bases de datos, no obstante pudo haber sido utilizado algún otro (tal como Informix, SQL Server, PostgreSQL, etcétera). Sin embargo se llegó a la conclusión de que MySQL era la mejor opción debido a las ventajas presentadas frente a estos en un informe de Oracle del 2012 [54].

Una de las ventajas del almacenamiento en una base de datos se da cuando el sistema analizado es muy grande, ya que probablemente la RAM se desborde al construir las diversas estructuras de datos sobre ella al realizar la extracción de datos. En cambio, al utilizar una base de datos esto no ocurrirá ya que los datos se guardan de manera persistente en el disco duro.

Enseguida se muestra el modelo Entidad – Relación utilizado para almacenar la información extraída en la fase anterior:

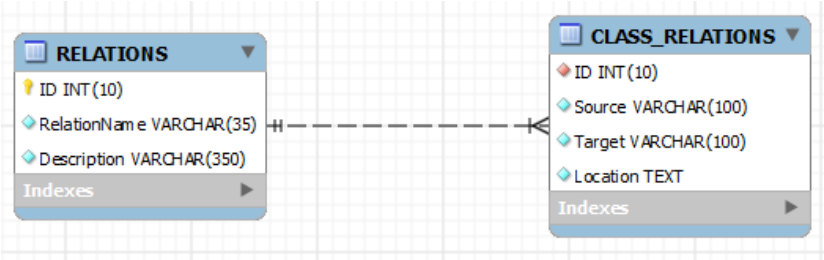


Figura 37. Modelo Entidad - Relación del enfoque

Se puede observar que el almacenamiento es muy simple, solo consta de dos Tablas.

En la Tabla izquierda (RELATIONS), se guardan las relaciones que pueden ser detectadas por el sistema. Mientras tanto, en la Tabla derecha (CLASS_RELATIONS) puede observarse que existen cuatro campos primordiales:

- El campo "*Source*" se refiere a la clase que funge como origen de la relación.
- El campo "*Target*" funge como el destino de la relación
- El campo "ID" es un campo foráneo proveniente de la Tabla RELATIONS.

- El campo “Location” es un campo que guarda la ruta absoluta donde se encuentra almacenada la relación.

Con la estructura de la Tabla derecha es posible almacenar el tipo de relación que ocurre y entre qué clases ocurre, pero además, se puede identificar el lugar físico en el que se encuentra (con el campo “*Location*”). Éste campo también es muy importante ya que tendrá la función de descartar relaciones que probablemente se repitan, ya que puede haber más de una clase tanto de origen como de destino con el mismo nombre, e incluso con el mismo tipo de relación, entonces se tomará el campo “Location” para descartarlas; de ésta manera se simplificará la estructura del sistema al momento de construir el grafo.

4.1.3 Análisis

Durante el Capítulo 3, Sección 3.3, pudimos observar que muchos de los trabajos existentes realizan el análisis de código fuente mediante métodos de verificación y medición. En éste trabajo de tesis se realiza un método basado en una mezcla de ambos, tal como se resalta con color rojo en la Figura 38. Es un método de medición porque se consideran medidas relacionadas a aristas y vértices; pero también es de verificación ya que estas medidas deben cumplirse de acuerdo a ciertos criterios que se establecen en algoritmos específicos para su detección.

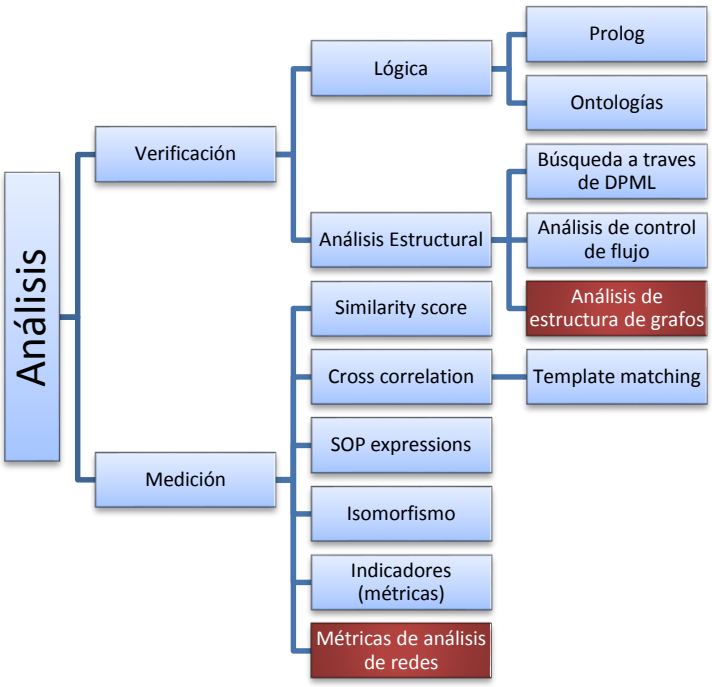


Figura 38. Métodos y herramientas de la fase de análisis

En las siguientes secciones se mostrará, paso a paso, cómo fueron generadas las redes de software a partir de la información extraída en la fase anterior. Se

describe cómo se creó un componente para exportar datos desde una base de datos relacional al framework JUNG [55]. También se describirán algunos conceptos de teoría de grafos y métricas que fueron utilizadas en éste trabajo. Finalmente se describirán los algoritmos para detectar los patrones de diseño Façade, Composite y Adapter usando dichos conceptos y métricas.

4.1.3.1 Generando redes de software

En ARS, una red consiste de un conjunto de *nodos* representando actores, los cuales pueden ser personas, grupos, empresas, etcétera, y las *aristas* (o enlaces) representan relaciones específicas entre los actores. En éste trabajo, una red consiste de nodos que representan clases del código fuente y aristas que representan relaciones válidas entre ellas, tales como, composición, realización, generalización, asociación o agregación.

En el Capítulo 4, Sección 4.1.1, se explicó cómo se ha automatizado el proceso de detectar clases y sus relaciones directamente del código fuente del sistema de software usando técnicas de extracción de información y la herramienta GATE. Luego en la sección 4.1.2 se describió cómo se almacenan las clases y sus relaciones en una base de datos relacional MySQL.

En ésta fase se importa la información de la base de datos relacional al software JUNG para generar y analizar una red en la que sus nodos representan clases y sus aristas representan sus respectivas relaciones.

4.1.3.2 JUNG

JUNG es una librería de software de código abierto que provee un lenguaje común para el modelado, el análisis y la visualización de datos que pueden ser representados como una red. La arquitectura de JUNG está diseñada para soportar representaciones de grafos, tales como grafos dirigidos o no dirigidos. En la Figura 39 se pueden observar los principales componentes de JUNG.

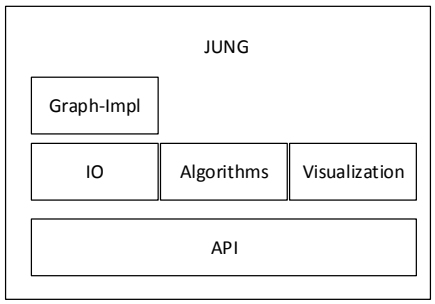


Figura 39. Arquitectura de JUNG

El componente API facilita el acceso a los componentes internos de JUNG. Los componentes de más importancia para éste trabajo son, el componente

Algorithms, y el componente Visualization. El componente Algorithms contiene diferentes métricas de análisis de grafos. El componente Visualization se encarga de presentar la red de datos en manera gráfica. El componente IO, se encarga de la entrada y salida de datos a los diferentes tipos de grafos que se encuentran implementados en el componente Graph-Impl. JUNG IO comúnmente puede leer diferentes formatos de archivos de texto que son utilizados por otras herramientas similares. Los formatos de archivos que puede leer son GraphML, Pajek y text matrix format. A través de JUNG es posible calcular una variedad de métricas de ARS y crear disposiciones personalizadas para visualizar las redes generadas y los resultados de las métricas.

JUNG fue seleccionado de entre otras herramientas similares porque reúne las siguientes características:

- Es posible contar con su código fuente, lo cual permitió agregar un componente para detectar patrones y modificar los componentes IO y Visualization. El componente IO fue modificado para que JUNG pudiera extraer datos de una base de datos y el componente Visualization fue modificado para agregar elementos gráficos que facilitaran el uso de la herramienta de detección de patrones.
- Es posible utilizar sus métricas de análisis de redes sociales en una aplicación propia.
- Es posible manejar la visualización de los grafos, como cambiar los colores de los nodos o de las aristas para que con ello se pueda dar un mejor entendimiento al usuario.

Las herramientas candidatas a ser utilizadas fueron JUNG, Gephi, GraphVis, IGraph, NetworkX, UCINET, Pajek y Tulip. Así que se realizó una serie de comparaciones entre cada una de éstas. Los resultados se muestran en la Tabla 13.

Tabla 13. Cuadro comparativo entre herramientas de ARS

Criterio	JUNG	Gephi	GraphViz	IGraph	NetworkX	UCINET	Pajek	Tulip
Código fuente	Sí	Sí	Sí	Sí	Sí	No	No	Sí
Extracción de métricas	Sí	Sí	No	Sí	Sí	No	Sí	Sí
Detección de comunidades	Sí	Sí	No	Sí	Sí	Sí	Sí	Sí
Visualización de grafos	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
IDE / lenguaje	Eclipse /Java	Netbeans/Java	C/C++	C, Python, R	Python	No disponible	No disponible	C/C++, Python
Métricas soportadas	Betweeness, Closeness, Clustering, Centrality Structural holes Scoring Shortest path Transformation, weight, etcétera	Betweenness, Closeness, Diameter, Clustering Coefficient, Average shortest path, PageRank, HITS Clustering	Ninguna	Shorter path, Component detection, Centrality, Similarity, Spanning Trees, K-Cores, Density, etcétera.	Approximation, Centrality, Clustering Communities, Cores, Distance Shortest path	Clustering, Betweness, Centrality, Structural holes, Scoring, Shortest path, Transformation, K-Core	Clustering, Betweness, Centrality, Structural holes, Scoring, Shortest path, Transformation	Centrality, Clustering, Degree Eccentricity, Size

La Tabla 13 muestra que iGraph, NetworkX y Tulip fueron las más competitivas ante JUNG, sin embargo fueron descartadas por las siguientes razones:

- Experiencia en el lenguaje. La experiencia o curva de aprendizaje en un framework o un lenguaje es un factor al momento de hacer una elección.
- Extensibilidad. iGraph, NetworkX, Tulip y JUNG son de código abierto y por lo tanto son modificables, pero esto no significa que sean extensibles. En éste ámbito se debe entender como extensibilidad a la capacidad que tiene un sistema para extender sus funcionalidades sin la necesidad de cambiar una sola línea en su código fuente original (infraestructura). En éste rubro JUNG fue mejor que las demás herramientas debido a que en ésta hay que cambiar 0 líneas de su código fuente para soportar nodos y aristas personalizadas a diferencia de iGraph, NetworkX y Tulip.

Tradicionalmente las redes son representadas por matrices de adyacencia. Sin embargo, una matriz no siempre es muy conveniente para redes de larga escala ya que es posible que exceda la capacidad de la memoria de la computadora. En éste caso la representación de una lista enlazada de una red es una mejor opción.

En JUNG, los nodos y las aristas pueden ser representados por cualquier tipo de dato existente en el lenguaje Java, tales como números enteros, cadenas, etcétera, pero no se limita a éstos ya que también pueden ser representados por clases. Esto es una ventaja frente a otras herramientas similares. Por ejemplo en Pajek sus nodos y aristas se limitan a estar representados por números enteros o cadenas.

En los patrones de diseño existen varios tipos de relaciones, por ejemplo generalización, realización, agregación y composición. Las herramientas listadas en la Tabla 13 están limitadas al uso de números enteros y cadenas de caracteres para identificarlas. Por lo tanto se debe asignar una cadena o un número entero diferente para identificar a cada tipo de relación. Esto significa que, no es posible asignar una cadena o un número único para representar a cada tipo de relación. Éste problema fue resuelto en JUNG ya que en ésta herramienta las relaciones son objetos, en lugar de números enteros o cadenas de caracteres.

4.1.4 Conceptos de ARS

Enseguida se describirán algunos conceptos propios del ARS con la finalidad de comunicar de una mejor manera los algoritmos de detección de patrones de diseño.

4.1.4.1 Grafo

En las herramientas de ARS, las redes pueden ser representadas como grafos. Un grafo es un conjunto de elementos llamados *nodos* o *vértices* que están unidos por

aristas o enlaces que permiten representar relaciones entre los elementos del conjunto. Existen grafos de diversos tipos. En éste trabajo nuestro grafo está compuesto por un conjunto de *vértices* que representan clases, un conjunto de *aristas dirigidas* que representan las relaciones entre clases y un conjunto de *etiquetas* representando los posibles tipos de relaciones que puede haber entre los vértices. Se permiten relaciones cíclicas en los nodos.

4.1.4.2 Ego network y Global network

Las redes pueden ser estudiadas a dos niveles, es decir, a nivel *Global Network* o a nivel *Ego Network* [56]. La Global Network comprende a la red en su totalidad. Las Ego Network cubren una subred de la Global Network. En ARS cada nodo de la red puede ser considerado a menudo como un *Ego*, cuando queremos estudiar las relaciones de un nodo en específico de la red.

En la Figura 40 se ilustra una Ego Network en la que se pueden distinguir varios tipos de nodos. El nodo Ego es marcado en rojo. En verde y morado se denotan los nodos vecinos. Los nodos marcados en azul, son nodos que están relacionados con el nodo Ego pero de manera indirecta, a través de otro nodo. Ese tipo de nodos es conocido como nodos vecinos de segundo grado.

En verde, se encuentran los nodos vecinos de entrada, y en morado los nodos vecinos de salida. Un nodo vecino es de *entrada* o de *salida* de acuerdo a la dirección de la arista, si la flecha de la arista apunta al nodo Ego entonces el nodo vecino denota un nodo vecino de entrada. Si las flechas salen del nodo Ego apuntando hacia otro nodo, el nodo al que apuntan se considera como un nodo de salida.

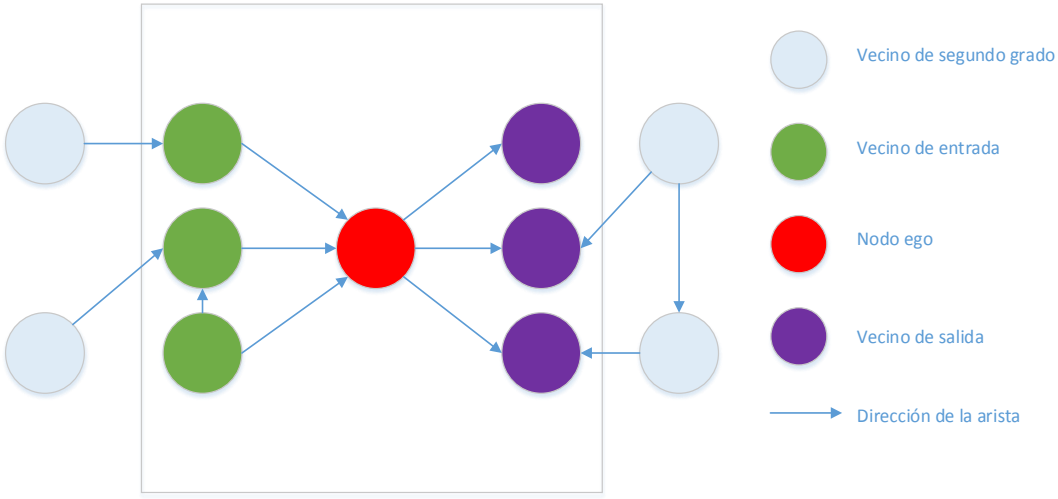


Figura 40. Representación de una EGO Network

De las Ego Networks es posible calcular métricas tales como, el número de enlaces de entrada (centralidad de grado de entrada) del nodo Ego, el número de enlaces de salida (centralidad de grado de salida) del nodo Ego, el número de enlaces totales (centralidad de grado) del nodo Ego, etcétera.

4.1.4.3 Propiedades y métricas de las redes

Tichy [16] clasificó diversas propiedades de las redes en tres categorías: *contenido de la relaci*ón, *naturaleza del vínculo* y *características estructurales*. Estas propiedades pueden ser medidas cuantitativamente a través de métricas.

En la primera categoría, contenido de la relación, Tichy agrupó diversas propiedades que están relacionadas a qué tipo de información se intercambian los nodos. En la segunda categoría, naturaleza del vínculo, él agrupó propiedades que están relacionadas a los enlaces existentes entre dos nodos. En la tercera, características estructurales, agrupó propiedades que están relacionadas a describir la red en su totalidad o a cierta parte de interés de la red.

En éste trabajo consideramos propiedades que tienen que ver con la naturaleza de los vínculos y con las características estructurales. Se analiza la Global Network partiéndola en Ego Networks. Es decir, de toda la red que se forma de un sistema de software, consideramos a cada nodo de la red como un nodo Ego que junto con sus nodos vecinos de entrada y salida, se considera como un Ego network. Se verifica que los enlaces hacia sus nodos vecinos de entrada y salida cumplan con ciertas características y de ésta manera se determina si la Ego Network es un patrón de diseño.

En lo que respecta a la naturaleza de los vínculos es posible considerar la propiedad de la *dirección*. La dirección permite comparar y contrastar el sentido de la información que fluye entre dos nodos. En éste trabajo de tesis ésta propiedad es muy importante para entender la semántica de las relaciones de asociación, agregación, generalización, realización o composición. Por ejemplo, no es lo mismo que una clase A tenga una relación de realización hacia B, a que una clase B tenga una relación de realización hacia A.

Por otra parte, de las características estructurales nos enfocamos en la *centralidad de grad*o que mide la contribución de un nodo, según su grado de relación, en la red. La centralidad de grado es el número de nodos a los cuales un nodo está directamente unido sin tener en cuenta la dirección de las aristas. Por ejemplo, en el nodo 6 de la Figura 41 la centralidad de grado es 6.

La *centralidad de grado de entrada* es la suma de las relaciones referidas hacia un nodo por otros. A través de ésta métrica es posible observar los nodos de entrada que están directamente relacionados al nodo Ego. En la Figura 41 se puede

observar que la centralidad de grado de entrada para el nodo 6 es 3, y sus nodos predecesores (marcados en verde) son {3, 4, 5}.

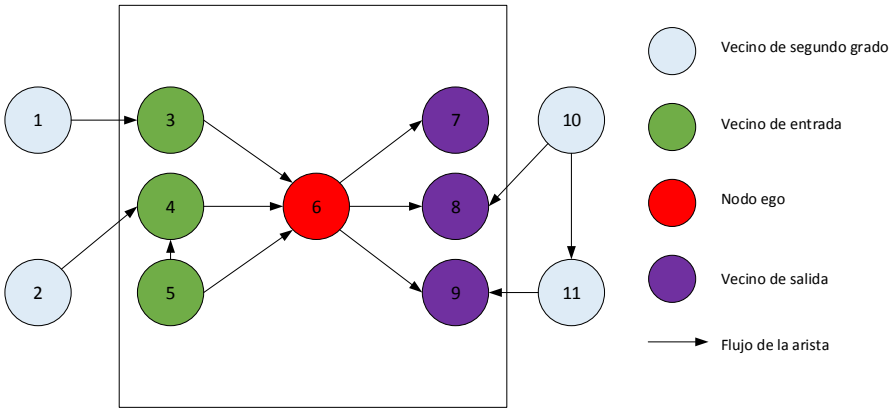


Figura 41. Ejemplo de centralidad de grado de entrada

La *centralidad de grado de salida* es la suma de las relaciones de salida que los nodos tienen con el resto. En la Figura 41 es posible observar que la centralidad de grado de salida del nodo 6 es 3. Los nodos sucesores del nodo 6 son {7, 8, 9}. Para la detección de patrones de diseño es necesario además de calcular la centralidad de grado de entrada y salida de los nodos, considerar también el tipo de enlaces entre nodos lo cual explicaremos en la siguiente sección.

4.1.4.4 Detección de patrones

Se ha analizado y caracterizado cada patrón de diseño utilizando métricas de centralidad de ARS. En las siguientes secciones se describe la caracterización y los algoritmos para detectar tres patrones de diseño. Es importante mencionar que las siguientes suposiciones se cumplen para los tres patrones de diseño que serán detectados:

Dados dos conjuntos $A\ y\ B$, una relación de $A\ a\ B$ es un subconjunto $\mathcal{R}$ del producto cartesiano $A\ x\ B$, simbólicamente $\mathcal{R} \subseteq A\ x\ B$. Los elementos de $\mathcal{R}$ son entonces pares ordenados de la forma $(a, b) = \{\{a\}, \{a, b\}\}$, donde $a \in A$ y $b \in B$. El conjunto A se llama dominio, en tanto que B es conocido como rango. También se usan los términos conjunto de salida y conjunto de llegada.

Las expresiones $A\ \mathcal{R}\ B$ y $(a, b) \in \mathcal{R}$ son equivalentes y son usados con el mismo significado, por ejemplo $a \leq b$ significa que $(a, b) \in \leq$, aunque esta segunda expresión es bastante menos familiar.

Una gráfica dirigida o digráfica es un par (V, A) donde $V\ y\ A$ son conjuntos, donde además cada arista dirigida o diarista, es decir, cada elemento de A es un par ordenado de vértices, es decir, de elementos de V. Se dice que el vértice $a \in V$ es el origen de la diarista $(a, b) \in A$ y que $b \in A$ es su extremo.

En términos de relaciones, una digráfica puede ser interpretada como una relación de $V\ en\ V$. Un lazo es una arista de la forma (x, x).

4.1.4.4.1 Análisis para detectar el patrón Façade

Como lo explicamos en el Capítulo 2, Sección 2.3, el patrón Façade se compone por tres tipos diferentes de nodos: el nodo Façade, los nodos cliente y los nodos del subsistema. Las características de las relaciones del patrón Façade se muestran en la Tabla 14.

Tabla 14. Características del patrón Façade

Dirección	Tipo	Grado
Cliente(Predecesor)->Façade(Ego)	Asociación	Entrada>=1
Façade(Ego)->Subsistema(Sucesor)	Asociación	Salida>=2

Como se puede observar en la Tabla 14, en el patrón Façade debe existir al menos una relación de entrada del nodo cliente a el nodo Façada y ésta relación debe de ser de tipo Asociación. También deben existir al menos dos relaciones de salida de tipo Asociación del Nodo Façada hacia al menos dos nodos Subsistema.

De esta forma, el algoritmo definido para detectar el Façade considerando las suposiciones descritas anteriormente, es el siguiente:

Una gráfica patrón es una terna (O, E, A), tal que (V, A) es una digráfica, donde $V = O \cup E$, y además:

1. $x \in O$ si y sólo si existe $y \in E$ tal que $(x, y) \in A$, en tanto que
2. $y \in E$ si y sólo si existe $x \in O$ tal que $(x, y) \in A$.

El conjunto O es el conjunto de orígenes y el conjunto E es el conjunto de extremos. Una gráfica dirigida bipartita es una gráfica patrón en la que $O \cap E \neq \emptyset$. Nótese que una gráfica patrón no es necesariamente bipartita.

Un patrón pre-façada es un par de gráficas patrón de la forma (C, F, A) y (F, S, B). Un patrón façada es una colección finita de patrones pre-façade.

$$(C, F, A_1), (F, S, B_1), \ldots, (C, F, A_n), (F, S, B_n)$$

con la propiedad de que los conjuntos $A_f = \{(x, y) \in A \,/ y = f\}$ y $B_f = \{(y, z) \in B \,/ y = f\}$ satisfacen $\#(A_f) \geq 1$ y $\#(B_f) \geq 2$, donde $A = A_1 \cup \ldots \cup A_n$ y $B = B_1 \cup \ldots \cup B_n$.

Si C es el conjunto de clientes, F es el conjunto de façades y S el de subsistemas, las condiciones anteriores indican que cada façada tiene al menos un cliente, y por lo menos dos subsistemas.

El grado de entrada de una façada $f \in F$ se define como $g_E(f) = \#(A_f)$ y el grado de salida de $f \in F$ mediante $g_s(f) = \#(B_f)$.

Proposición 1. Los grados de entrada y salida de un elemento $f \in F$ satisfacen $g_E(f) \leq \#(C)$ y $g_s(f) \leq \#(S)$.

Si denotamos $g_E(c,f) = 1$ siempre que $(c,f) \in A$ y $g_E(c,f) = 0$ en caso contrario, tenemos

$$g_E(f) = \sum_{c \in C} g_E(c,f)$$

Análogamente, $g_s(f,s) = 1$ siempre que $(f,s) \in B$ y $g_s(f,s) = 0$ en caso contrario, tenemos

$$g_s(f) = \sum_{s \in S} g_s(f,s)$$

De forma análoga, denotamos $g_{E,k}(c,f) = 1$ siempre que $(c,f) \in A_k$ y $g_{E,k}(c,f) = 0$ en caso contrario, al igual que $g_{S,k}(f,s) = 1$ siempre que $(f,s) \in B_k$ y $g_{E,k}(f,s) = 0$ en caso contrario. Tenemos entonces que:

$$g_E(c,f) = \sum_{k=1}^{n} g_{(E,k)}(c,f)$$

$$g_S(f,s) = \sum_{k=1}^{n} g_{(S,k)}(f,s)$$

4.1.4.4.2 Análisis para detectar el patrón Adapter

Como lo explicamos en el Capítulo 2, Sección 2.3, el patrón Adapter se compone de cuatro tipos diferentes de nodos: el nodo Adapter, los nodos Adaptee (Adaptado), los nodos Client (Cliente), y los nodos Target. Las características de cada nodo del patrón Adapter se muestran enseguida:

Tabla 15. Características del patrón Adapter

Dirección	Tipo	Grado
Adapter(Ego)->Target(Sucesor)	Realización	Salida>=1
Adapter(Ego)->Adaptee(Sucesor)	Asociación	Salida>=1
Cliente (Predecesor)->Target(Sucesor)	Asociación	Salida>=1

Como se puede observar en la Tabla 15, en el patrón Adapter debe existir al menos una relación de entrada del nodo Adapter a el nodo Target y ésta relación debe de ser de tipo Realización. También debe existir al menos una relación del nodo Adapter al nodo Adaptee de tipo Asociación y al menos debe de existir una relación de tipo Asociación del nodo Cliente al nodo Target.

De esta forma, se ha definido el algoritmo para detectar este patrón como se describe a continuación:

Un patrón pre-adapter está compuesto por tres gráficas patrón de la forma $(CLIENTE, TARGET, A)$, $(ADAPTER, TARGET, B)$ y $(ADAPTER, ADAPTEE, C)$. Un patrón Adapter es una colección finita de patrones pre-adapter.

$(CLIENT, TARGET, A_1)$,$(ADAPTER, TARGET, B_1)$, $(ADAPTER, ADAPTEE, C_1)$,….., $(CLIENT, TARGET, A_n)$, $(ADAPTER, TARGET, B_n)$, $(ADAPTER, ADAPTEE, C_n)$

con la propiedad de que los conjuntos $A_{target} = \{(x, y) \in A \,/ y = target\}$, $B_{target} = \{(w, y) \in B \,/ y = target\}$ y $C_{adaptee} = \{(w, z) \in A \,/ z = adaptee\}$ satisfacen $\#(A_{target}) \geq 1, \#(B_{target}) = 1$ y $\#(C_{adaptee}) \geq 1$ donde $A = A_1 \cup \ldots \cup A_n$, $B = B_1 \cup \ldots \cup B_n$, y $C = C_1 \cup \ldots \cup C_n$.

Si $CLIENT$ es el conjunto de clientes, $TARGET$ es el conjunto de targets, $ADAPTER$ es el conjunto de adapters y $ADAPTEE$ es el conjunto de adaptees, las condiciones anteriores indican que cada Target tiene al menos un Cliente, cada Target tiene un Adapter y cada Adaptee tiene al menos un Adapter. Note que A es una relación entre clientes y targets, B es una relación entre adapters y targets, y C es una relación entre adapters y adaptees.

El grado de entrada de un target $target \in TARGET$ se define como $g_E(target) = \#(A_{target})$ y el grado de salida de un target como $g_s(target) = 0$.

Proposición 1. Los grados de entrada y salida de un elemento $target \in TARGET$ satisfacen $g_E(target) \leq \#(C)$ y $g_E(target) = \#(Adapter) = 1$ $(Adapter)$ y $g_s(target) = 0$.

Si denotamos $g_E(client, target) = 1$ siempre que $(client, target) \in A$ y $g_E(cliente, target) = 0$ en caso contrario, y además que $g_E(adapter, target) = 1$ siempre que $(adapter, target) \in \boldsymbol{B}$ y $g_E(adapter, target) = 0$ en caso contrario. Por lo tanto podemos decir que:

$$g_E(target) = \sum_{client \in CLIENT} g_{\square}(client, target) + \sum_{adapter \in ADAPTER} g_E(adapter, target)$$

Análogamente, $g_s(adapter, target) = 1$ siempre que $(adapter, target) \in B$ y $g_s(adapter, target) = 0$ en caso contrario. Además que $g_s(adapter, adaptee) = 1$ siempre que $(adapter, adaptee) \in B$ y $g_s(adapter, adaptee) = 0$ en caso contrario. Por lo tanto, podemos decir que:

$$g_s(adapter) = \sum_{target \in TARGET} g_s(adapter, target) + \sum_{adaptee \in ADAPTEE} g_s(adapter, adaptee)$$

Hasta este momento se han tomado relaciones existentes en los conjuntos sin tener en cuenta el tipo de relación. Por lo tanto definimos un conjunto K:

$$K = \{generalización, realización, asociación, agregación, composición\}$$

De forma análoga, denotamos $g_{E,k}(client, target) = 1$ siempre que $(client, target) \in A_k$ y $g_{E,k}(client, target) = 0$ en caso contrario, al igual que $g_{S,k}(adapter, target) = 1$ siempre que $(adapter, target) \in B_k$ y $g_{E,k}(adapter, target) = 0$ en caso contrario. Tenemos entonces que:

$$g_{\square}(target) = \sum_{k=K} g_{(E,asociación)}(client, target) + \sum_{k=K} g_{(E,realización)}(adapter, target)$$

y

$$g_S(adapter) = \sum_{k=K} g_{(S,asociación)}(adapter, adaptee) + \sum_{k=K} g_{(S,realización)}(adapter, target)$$

4.1.4.4.3 Análisis para detectar el patrón Composite

El patrón Composite está compuesto por cuatro tipos diferentes de nodos: un nodo Composite, nodos Component, nodos Client y nodos Leaf.

Dirección	Tipo	Grado
Cliente(Predecesor)->Component(Ego)	Asociación	Entrada>=1
Hoja(Predecesor)->Component(Ego)	Generalización	Entrada>=1
Composite(Predecesor)->Component(Ego)	Generalización	Entrada>=1
	Agregación	Entrada>=1

De acuerdo a esto, se ha definido el algoritmo para detectar el patrón Composite como se describe a continuación:

Un patrón pre-composite está compuesto por tres gráficas patrón de la forma $(CLIENT,\ COMPONENT, A)$,$(HOJA, COMPONENT, B)$ y $(COMPOSITE, COMPONENT, C)$. Un patrón Composite es una colección finita de patrones pre-composite.

$(CLIENT, COMPONENT, A_1)$,$(HOJA, COMPONENT, \boldsymbol{B_1})$,
$(COMPOSITE, COMPONENT, C_1)$,….., $(CLIENT, COMPONENT, A_n)$,
$(HOJA, COMPONENT, \boldsymbol{B_n})$, $(COMPOSITE, COMPONENT, C_n)$

con la propiedad de que los conjuntos $A_{component} = \{(x, y) \in A\ / y = component\}$ y $B_{component} = \{(w, y) \in B\ / y = component\}$, $C_{component} = \{(w, z) \in A\ / z = component\}$ satisfacen $\#(A_{component}) \geq 1, \#(B_{component}) \geq 1$ y $\#(C_{comp\square nent}) \geq 1$ donde $A = A_1 \cup \ldots \cup\ A_n,\ B = B_1 \cup \ldots \cup\ B_n$, y $C = C_1 \cup \ldots \cup\ C_n$.

Si $CLIENT$ es el conjunto de clientes, $COMPONENT$ es el conjunto de components, $COMPOSITE$ es el conjunto de composites y HOJA es el conjunto de hojas, las condiciones anteriores indican que cada Component tiene al menos un Cliente, al menos una Hoja y al menos un Composite. Note que A es una relación entre clientes y components, B es una relación entre hojas y componentes, y C es una relación entre composites y components.

El grado de entrada de un $component \in COMPONENT$ se define como $g_E(component) = \#(A_{component})$ y el grado de salida de un component como $g_s(component) = 0$.

Si denotamos $g_E(client, component) = 1$ siempre que $(client, component)\ \in \boldsymbol{A}$ y $g_E(cliente, component) = 0$ en caso contrario, y que $g_E(hoja, component) = 1$ siempre que $(hoja, component)\ \in \boldsymbol{B}$ y $g_E(hoja, component) = 0$ en caso contrario. Además que $g_E(composite, component) = 1$ siempre que $(composite, component)\ \in \boldsymbol{C}$ y $g_E(composite, component) = 0$ en caso contrario

Por lo tanto podemos decir que:

$$g_E(component) = \sum_{client \in CLIENT} g_E(client, component) + \sum_{hoja \in HOJA} g_E(hoja, component) + \sum_{composite \in COMPOSITE} g_E(composite, component)$$

Así mismo, se puede decir que:

$$g_s(component) = 0$$

Hasta este momento se han tomado relaciones existentes en los conjuntos sin tener en cuenta el tipo de relación. Por lo tanto definimos un conjunto K:

$$K = \{generalización, realización, asociación, agregación, composición\}$$

De forma análoga, denotamos $g_{E,k}(client,\ component) = 1$ siempre que $(client, component) \in A_k$ y $g_{E,k}(client, component) = 0$ en caso contrario, al igual que $g_{E,k}(hoja, component) = 1$ siempre que $(hoja, component) \in B_k$ y $g_{E,k}(hoja, component) = 0$ en caso contrario. Además que: $g_{E,k}(composite, component) = 1$ siempre que $(composite, component) \in C_k$ y $g_{E,k}(composite, component) = 0$ en caso contrario

Tenemos entonces que:

$$g_E(composite) = \sum_{k=K} g_{(E,asociación)}(client, component) + \sum_{k=K} g_{(E,generalización)}(hoja, component) + \sum_{k=K} g_{(E,generalización)}(composite, component) + \sum_{k=K} g_{(E,agregación)}(composite, component)$$

4.2 Visualización

El propósito de la fase de visualización es comunicar la información del patrón clara y eficientemente a los usuarios de manera gráfica. En nuestro enfoque esto fue llevado a cabo utilizando la herramienta JUNG. Extendimos la funcionalidad de JUNG para construir un componente adicional de software para soportar la visualización de los patrones de diseño detectados. En nuestra implementación hay una representación para cada nodo (clase de software) y los diferentes tipos de relación entre ellos (por ejemplo, asociación, agregación, generalización o implementación). La Ilustración 3 es una captura de pantalla de la visualización de un patrón en JUNG.

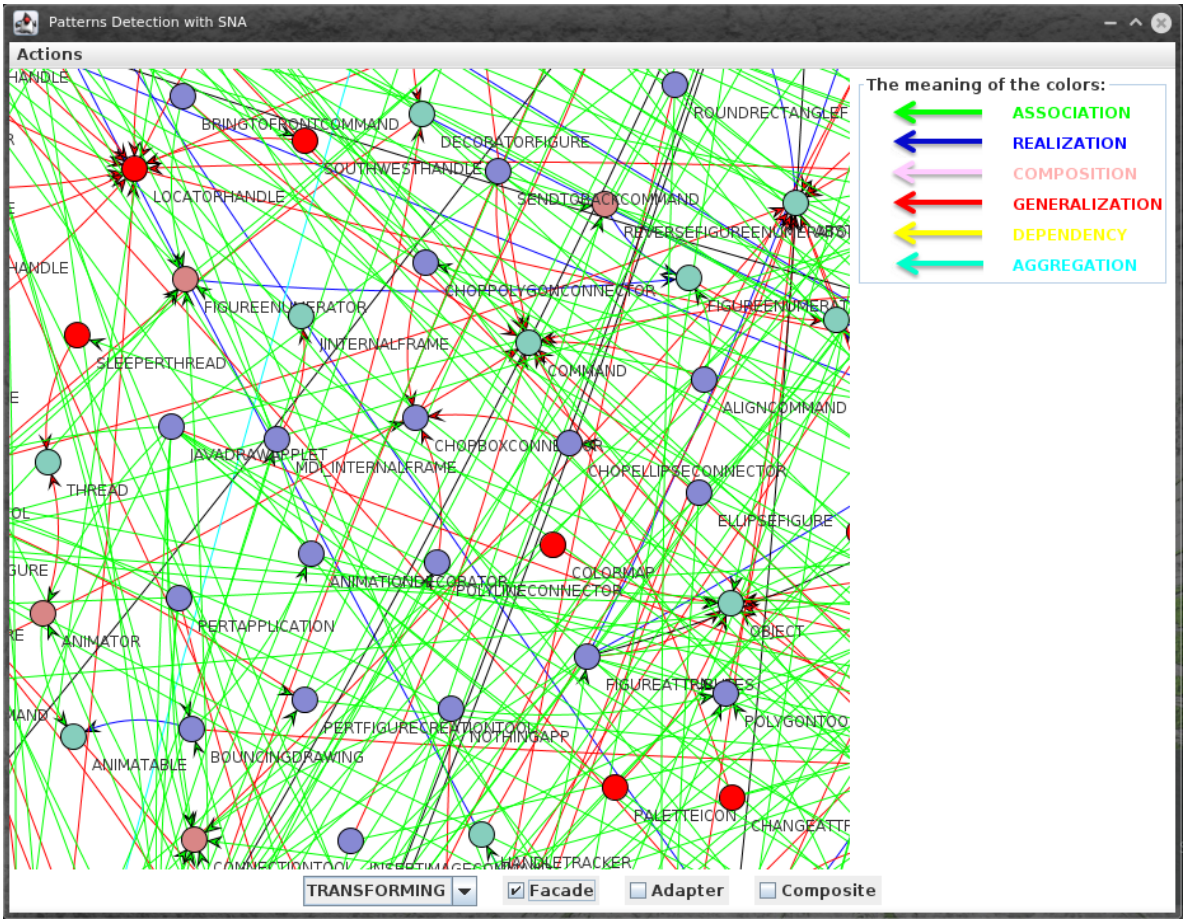


Ilustración 3. Captura de pantalla de la visualización de un patrón Façade

4.3 Resumen

- En las siguientes secciones se resumirán las ideas principales de cada una de las fases del enfoque de este trabajo de tesis.

4.3.1 Proceso para detectar patrones de diseño

- Seguimos un proceso secuencial, en el que la salida de cada fase es la entrada de la siguiente.

4.3.2 Extracción de información

- En la fase de la extracción de la información se utilizó como enfoque el procesamiento de lenguaje natural.
- En la fase de extracción de la información se utilizó como herramienta GATE.
- GATE es una arquitectura, un framework y un ambiente de desarrollo.
- Se utilizó la arquitectura ANNIE de GATE.
- Los componentes utilizados de ANNIE son Tokeniser, Gazetter, Sentence Spliter.
- Las reglas utilizan criterios de búsqueda positivos y negativos.
- Es posible extraer las relaciones de generalización, asociación, composición, realización y agregación.

4.3.3 Almacenamiento

- El almacenamiento toma como entrada la información extraída a partir del código fuente.
- El almacenamiento se realiza en dos momentos, el primero en una base de datos y el segundo en un grafo en la memoria principal (RAM).
- El método de almacenamiento fue basado en la manera en cómo Pajek lo realiza.
- No necesariamente se tiene que implementar una base de datos en MySQL.

4.3.4 Análisis

- El análisis se basa tanto en métodos de medición como en métodos de verificación.
- Se importa la información de la base de datos relacional al software JUNG para generar y analizar una red en la que sus nodos representan clases, y sus aristas representan sus respectivas relaciones.
- JUNG es una librería que provee un lenguaje común para el modelado, el análisis y la visualización de datos que pueden ser representados como una red.

- JUNG utiliza una estructura de datos conocida como lista enlazada para representar el grafo.
- Un grafo es un conjunto de elementos llamados nodos o vértices que están unidos por enlaces o aristas.
- En éste trabajo definimos nuestro grafo compuesto por un conjunto de vértices que representan clases, un conjunto de aristas dirigidas que representan las relaciones entre clases y un conjunto de etiquetas representando los posibles tipos de relaciones que puede haber entre los vértices.
- En éste trabajo se analiza la red nodo por nodo, es decir, a nivel local.
- La cardinalidad de los nodos obtenidos por medio de la función *Predecesores* es igual al grado de entrada de un nodo *d* respecto a una etiqueta *e*.
- La cardinalidad de los nodos obtenidos por medio de la función *Sucesores* es igual al grado de entrada de un nodo *f* respecto a una etiqueta *e*.
- Los patrones fueron caracterizados de acuerdo a la dirección, tipo y centralidad de grado de entrada y centralidad de grado de salida respecto a un nodo Ego.

4.3.5 Visualización

- El propósito de la fase de visualización es comunicar la información del patrón clara y eficientemente a los usuarios de manera gráfica.
- Se modificó el componente Visualization de JUNG para lograr la visualización de los patrones de diseño detectados.

5 Evaluación

En éste capítulo se discute y evalúa el trabajo realizado en tres sistemas de software existentes. Estos son JHotDraw [57], JRefactory [58] y JUnit [59]. JHotDraw es un sistema de dibujo, JRefactory es un sistema de refactorización para el lenguaje de programación Java y JUnit es un framework útil para escribir pruebas unitarias de código Java. Se escogieron estos sistemas debido a que son libres y de código abierto, y esto permite obtener y estudiar fácilmente el código fuente. Además, todos estos sistemas están codificados en Java y han sido utilizados por otros autores, tales como Tsantalis [43], Huang [32], Dong [45] [60], De Lucia [61] como caso de estudio para evaluar sus enfoques de detección de patrones.

De acuerdo a los objetivos específicos de éste trabajo de tesis, se medirá la precisión del método propuesto para la recuperación de patrones de diseño. Además se evaluará la extensibilidad del método a otros lenguajes de programación.

5.1 Evaluando Precisión

La precisión es una métrica comúnmente utilizada para verificar la efectividad de los métodos de búsqueda de patrones, comparando los resultados obtenidos con los resultados esperados. El punto de partida para entender esta métrica es la Tabla 16:

Tabla 16. Conceptos básicos para medir precisión y exhaustividad

	Correctos	Incorrectos
Encontrados	True Positive	False Positive
No encontrados	False Negative	True Negative

La Tabla 16 indica que existen cuatro posibilidades que describen la verdad acerca de los resultados. Los resultados pueden ser correctos o incorrectos. De los resultados correctos y que a su vez fueron encontrados por el método de búsqueda de patrones se les conoce como True Positive (TP). A los patrones que son correctos pero que no fueron encontrados por el método de búsqueda, se les conoce como False Negative (FN).

Por la parte de los resultados incorrectos, existen otras dos posibilidades. Aquellos patrones encontrados y contados como correctos por el método de búsqueda de patrones, pero que en realidad son incorrectos, también conocidos como False Positive (FP) y aquellos resultados incorrectos que de verdad son incorrectos, también conocidos como True Negative (TN).

Por una parte, la precisión mide la relación que existe entre el número de patrones correctos que han sido encontrados, también conocidos como true positive (TP), sobre el total de los patrones que han sido recuperados, es decir, la suma entre los patrones que han sido encontrados y son correctos (TP) más los patrones que han sido encontrados y que son incorrectos, también conocidos como false positive (FP). Se puede definir a la métrica de precisión como:

$$Precisión\ (Prec) = \frac{true\ positive\ (TP)}{true\ positve\ (TP) + false\ positive(FP)}$$

Por otro lado, la exhaustividad es una medida que expresa la relación que existe entre el número de patrones correctos y que han sido encontrados (TP) y el total de los patrones que deberían haber sido encontrados. Es decir, la suma de los patrones correctos y encontrados más los patrones que son correctos y que no han sido encontrados por el método de búsqueda de patrones. En términos matemáticos, se puede definir a la métrica de exhaustividad como:

$$Exhaustividad\ (Exh) = \frac{true\ positive\ (TP)}{true\ positive\ (TP) + false\ negative(FN)}$$

Tanto la precisión como la exhaustividad son utilizadas para medir la relevancia de los resultados de una búsqueda. La precisión mide qué tan buena fue la detección de patrones correctos en relación a los resultados obtenidos. Mientras que la exhaustividad mide qué tan buena fue la extracción de patrones correctos en relación a los que deberían haberse obtenido.

Cuando se mide la exhaustividad es necesario saber el número exacto de patrones que deben encontrarse, para así poder establecer el número de patrones obtenidos y correctos (True Positive), y el número de patrones correctos que no fueron obtenidos por el método de búsqueda de patrones (Falsos Negativos). En éste contexto no se cuenta con estos datos, ya que es difícil conocer el número de falsos negativos existentes. Por esta razón se prefirió evaluar la precisión pues consideramos que para soportar mejor la comprensión de un sistema es importante el disminuir el número de falsos negativos detectados que confundirían al desarrollador.

En las siguientes secciones se describirán y discutirán los resultados obtenidos, en cuanto a precisión para los patrones de diseño Façade, Composite y Adapter de acuerdo a los autores Tsantalis, Huang, Dong y De Lucia en comparación con nuestro método de detección de patrones.

5.1.1 JHotDraw

En la Tabla 17 se muestran los resultados obtenidos por los trabajos de los autores Tsantalis, Huang, Dong, De Lucía y los resultados de nuestro método en cuanto a precisión.

En ésta tabla comparativa puede observarse que la mayoría de los trabajos no obtuvieron resultados para el patrón Façade, solamente De Lucia lo consideró y obtuvo un total de nueve patrones al igual que en nuestro método pero, según se reporta, con un 33% de precisión. En nuestro trabajo se obtuvieron 2 falsos positivos al examinar cada uno de ellos, lo que provocó un 77% de precisión por lo que mejora con respecto al trabajo de De Lucia.

Tabla 17. Patrones de diseño encontrados en JHotDraw

Autor	**Façade**			**Composite**			**Adapter**		
	TP	FP	Prec	TP	FP	Prec	TP	FP	Prec
Tsantalis	-	-	-	1	0	1	18	0	1
Huang	-	-	-	1	0	1	1	1	.5
Dong	-	-	-	0	0	-	51	0	1
De Lucia	9	18	0.33	0	0	-	41	0	1
Nuestro método	7	2	0.77	15	0	1	20	0	1

Por otro lado, Dong y De Lucia no obtuvieron resultados en la búsqueda de los patrones Composite en JHotDraw. A diferencia de ellos, Tsantalis y Huang obtuvieron un solo patrón Composite. Nuestro enfoque obtuvo un total de 15 patrones Composite de los cuales se estudiaron el 100% de ellos manualmente y se dedujo que todos ellos son patrones Composite True Positive.

Finalmente, en la búsqueda del patrón Adapter, Dong y De Lucía encontraron una gran cantidad de este tipo de patrones de diseño, que fue notablemente superior a la cantidad que nosotros encontramos. Con excepción del trabajo de Huang, todos presentaron una precisión de 1.

5.1.2 JRefactory

En la siguiente tabla se muestran los resultados obtenidos por Tsantalis y Dong en comparación con el nuestro. Se han omitido los autores Huang y De Lucia debido

a que no fueron parte de sus experimentos y por lo tanto no tienen resultados sobre el sistema JRefactory.

Tabla 18. Patrones de diseño encontrados en JRefactory

Autor	Façade			Composite			Adapter		
	TP	FP	Prec.	TP	FP	Prec.	TP	FP	Prec.
Tsantalis	-	-	-	0	0	-	7	0	1
Dong	-	-	-	0	0	-	27	0	1
Nuestro método	14	2	0.875	33	0	1	11	0	1

Ni Tsantalis ni Dong estudiaron el patrón Façade y por lo tanto no existen resultados comparables sobre este patrón de diseño. En nuestro método se obtuvieron 14 resultados True Positive, y 2 False Positive, por lo que se obtuvo una precisión del 87.5%.

En el patrón Composite, Tsantalis y Dong obtuvieron como resultado que no existen patrones Composite en JRefactory. Sin embargo hay que recordar que en el trabajo de Dong no es posible recuperar relaciones de tipo Agregación, y es por esa razón por la que Dong no obtuvo resultados en la búsqueda del patrón Composite.

Finalmente en la búsqueda del patrón de diseño Adapter existe una pequeña similitud entre los resultados de Tsantalis y los resultados de este trabajo de Tesis. También es posible observar que existe una gran diferencia con los resultados encontrados por Dong, quien obtuvo como resultado que existen 27 patrones de diseño de tipo Adapter en el sistema JRefactory.

5.1.3 JUnit

La siguiente tabla muestra los resultados obtenidos por Tsantalis, Huang y Dong en comparación con los resultados arrojados por nuestro método propuesto. El autor De Lucia no realizó estudios sobre JUnit y por lo tanto fue omitido.

Autor	Façade			Composite			Adapter		
	TP	FP	Prec.	TP	FP	Prec.	TP	FP	Prec
Tsantalis	-	-	-	1	0	1	1	0	1
Huang	-	-	-	0	0	-	1	0	1
Dong	-	-	-	1	0	1	5	0	1
Nuestro método	1	0	1	4	0	1	2	0	1

Se puede observar a simple vista que los demás métodos no estudiaron el patrón Façade, así que no es posible hacer una comparación. Nuestro método obtuvo solo un resultado que fue comprobado y que por lo tanto, obtuvo una precisión del 100%.

En el caso del patrón Composite tanto Tsantalis como Dong obtuvieron un resultado de 1 patrón de diseño Composite. Nuestro método obtuvo un total de 4, de los cuales pudo comprobarse que ambos son patrones de diseño True Positive.

Finalmente en el patrón Adapter, Tsantalis y Huang coinciden en que encontraron un patrón de diseño, este patrón de diseño es auténtico debido a que midieron el número de falsos positivos, la precisión y la medida de recall. Mientras tanto Dong menciona en su trabajo que encontró 5 patrones Adapter. En nuestro trabajo se encontraron 2 patrones de diseño, uno de ellos corresponde incluso en sus nombres de clase. El segundo patrón que fue encontrado con nuestro trabajo no corresponde en nomenclatura, pero sí en cuanto a las relaciones que un patrón Adapter tiene que poseer.

5.2 Extensibilidad a otros lenguajes de programación

Para medir la extensibilidad a otros lenguajes de programación orientados a objetos, se han medido las líneas agregadas, modificadas y eliminadas con respecto al código de las reglas JAPE generadas anteriormente para detectar relaciones en el lenguaje de programación Java con el fin de poder medir qué tan

fácil es poder soportar la detección de relaciones en otros dos lenguajes de programación, que son VB .Net y C#.

La razón para seleccionar estos dos lenguajes de programación y no otros, es que según el ranking de lenguajes de programación más utilizados en el mundo, según TIOBE [23] y detallada en la Ilustración 4, C# y VB .Net son los lenguajes puramente orientados a objetos que más alto se encuentran en el ranking aparte de Java.

May 2015	May 2014	Change	Programming Language	Ratings	Change
1	2	↑	Java	16.869%	-0.04%
2	1	↓	C	16.847%	-0.08%
3	4	↑	C++	7.875%	+1.89%
4	3	↓	Objective-C	5.393%	-6.40%
5	6	↑	C#	5.264%	+1.52%
6	8	↑	Python	3.725%	+0.67%
7	9	↑	JavaScript	3.127%	+1.34%
8	11	↑	Visual Basic .NET	2.968%	+1.70%
9	7	↓	PHP	2.720%	-0.67%
10	-	⇈	Visual Basic	1.893%	+1.89%

Ilustración 4. 10 lenguajes de programación más utilizados según TIOBE

Nuestro método de detección de patrones utiliza un componente para extraer los tipos de relación, tales como, asociación, agregación, composición, generalización y realización. Éste componente extrae las relaciones de acuerdo a un lenguaje de programación en específico e introduce los resultados a una base de datos en MySQL. Una vez que los datos se encuentran ahí, es posible realizar la búsqueda de los patrones de diseño sin importar en esa instancia el lenguaje de programación, por lo tanto, no existirán modificaciones en la parte del análisis, pero sí en la parte de la extracción de las relaciones.

Enseguida se muestran los resultados de haber agregado, modificado y eliminado líneas para detectar cada tipo de relación para adaptarlas a los lenguajes de programación C# y VB .Net, tomando como base las escritas para extraer relaciones a partir de código fuente Java.

Asociación 1 – 1

Respecto a la Relación de Asociación 1-1 medimos el número de líneas agregadas, modificadas y eliminadas con respecto al código de las reglas JAPE para Java, VB.Net y C# tal y como se muestra en la Tabla 16:

En la Tabla 19 se muestra que la relación de asociación 1-1 fue escrita en 61 líneas de código para soportar el lenguaje de programación Java. Tomando estas 61 líneas como referencia, se han modificado solamente 5 líneas para soportar el lenguaje de programación C#, por lo tanto esto hace que 56 líneas se mantengan sin cambio alguno. El archivo para soportar C# se mantiene con 61 líneas de código.

Tabla 19. Líneas agregadas, modificadas y eliminadas para soportar la relación de asociación 1-1 en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	61	-	-	-	-	-
C#	61	0	5	0	56	91.8
VB .Net	50	0	13	11	37	60.6

Al tomar como base las líneas de código de las reglas JAPE para adaptarla al lenguaje de programación VB .Net, se obtuvo un archivo con un total de 50 líneas debido a que se eliminaron 11. Las líneas modificadas (13) no incrementan ni reducen el tamaño total de las líneas de código, pero sí afectan el número de líneas reusadas, es decir el número de líneas que se mantuvieron sin cambio alguno.

Para éste trabajo se define la siguiente fórmula para obtener el número de líneas reusadas:

$$R = T - (A + M) - E,$$

R = Líneas reusadas. Son las líneas que no sufrieron cambio alguno con respecto a las líneas base.

T = Líneas totales. Es el número total de líneas de cada tipo de relación con respecto al lenguaje de programación.

A = Líneas agregadas. Es el número de líneas nuevas que se escribieron para adaptar la relación al nuevo lenguaje de programación.

M = Líneas modificadas. Es el número de líneas que ya existían en el código base pero que tuvieron que ser alteradas para adaptar la relación al nuevo lenguaje.

E = Líneas Eliminadas. Es el número de líneas que tuvieron que ser descartadas del código base para adaptar la relación al nuevo lenguaje.

Además, es posible obtener la tasa de reutilización de la siguiente manera:

$TR = \frac{R}{B}$, donde

B = Líneas base. Son las líneas que fueron escritas para detectar una relación, en este caso Java, y que posteriormente fueron tomadas como apoyo para escribir código para detectar nuevas relaciones.

TR = Tasa de reutilización. Es la correlación existente entre las líneas reusadas con respecto a las líneas base.

La tasa de reutilización para la relación de asociación en lenguaje C# es de 91.8%, es decir, que se reutilizó el 91% del código original. Esto se debe a que C# y Java son muy parecidos en cuanto a su gramática y sintaxis.

Por otro lado hay que notar que para VB .Net se reutilizó el 60.6%. Lo cual indica que aunque los lenguajes no son muy parecidos en cuanto a gramática y sintaxis, más de la mitad del código se mantuvo sin cambio alguno.

Asociación 1-n

En la Tabla 20 se puede apreciar que las modificaciones para soportar la relación de asociación 1-n en lenguaje de programación C# no cambiaron mucho, ya que solamente se han modificado 5 líneas de código, por lo tanto el archivo de C# se mantuvo con 114 líneas de código. El número de líneas sin cambios es de 109 y por lo tanto la tasa de reutilización para adaptar la detección de este tipo de relación en lenguaje C# es del 95.6%. Mientras tanto para adaptar el código original a lenguaje VB .Net se han eliminado 10 líneas de código y se han modificado 15, el número de líneas totales de este archivo es de 104. Las líneas que se mantuvieron sin modificaciones son 89, por lo tanto la tasa de reutilización ha sido del 78.07%.

Tabla 20. Líneas agregadas, modificadas y eliminadas para soportar la relación de asociación 1-n en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	114	-	-	-	-	-
C#	114	0	5	0	109	95.6

VB .Net	104	0	15	10	89	78.07

Asociación n-n

La relación de asociación n-n fue realizada en un total de 65 líneas de código para el lenguaje Java. En la Tabla 21 puede apreciarse que de estas 65 líneas de código, 60 fueron reusadas en su totalidad, lo cual muestra que el 92% de las líneas de código originales quedaron intactas para detectar este tipo de relación en lenguaje C#. Mientras tanto para el mismo tipo de relación, pero en lenguaje VB .Net, se han modificado 13 líneas de código, y se han eliminado 11, por lo que el número de líneas totales de las reglas JAPE para detectar Asociación n-n es de 54. La tasa de reutilización es del 63%, y esto significa que más de la mitad del código escrito originalmente ha sido utilizado sin cambio alguno.

Tabla 21. Líneas agregadas, modificadas y eliminadas para soportar la relación de asociación n-n en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	65	-	-	-	-	-
C#	65	0	5	0	60	92.3
VB .Net	54	0	13	11	41	63.07

Si promediamos las tasas de reutilización para detectar los diferentes tipos de relación en lenguaje C#, se obtiene que la tasa de reutilización promedio es de 92%. Lo cual hace pensar que los lenguajes de programación C# y Java comparten la mayoría de su gramática y reglas sintácticas.

En el caso de VB .Net, la tasa de reutilización promedio es del 67%, que es un buen número teniendo en cuenta que las reglas sintácticas y gramaticales de éste lenguaje no comparten mucho en común con el lenguaje Java. Sin embargo, se ahorró más de la mitad del esfuerzo requerido para implementar la detección de los diferentes tipos de relación de asociación en los lenguajes de programación C# y VB.Net.

Agregación

Para detectar la relación de agregación en lenguaje Java, se llevó un esfuerzo de 70 líneas de código. En la Tabla 22 es posible ver las líneas agregadas, modificadas y eliminadas para adaptar la detección de este tipo de relación en el lenguaje C# y VB .Net.

Tabla 22. Líneas agregadas, modificadas y eliminadas para detectar la relación de Agregación en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	70	-	-	-	-	-
C#	70	0	3	0	67	95.71
VB .Net	47	0	13	23	34	48.5

Para detectar la agregación en C# solamente fue necesario modificar 3 líneas de código, por lo que 67 se mantuvieron sin cambios. La tasa de reutilización para el lenguaje C# vuelve a ser muy significativa, del 95.71%.

Para soportar la detección de la relación de agregación en VB .Net se modificaron 13 líneas de código y se eliminaron 23, por lo que el número de líneas de código se redujo a un total de 47. Solamente 34 líneas de código se mantuvieron sin modificaciones. En éste tipo de relación, la tasa de reutilización bajó al 48.5%, sin embargo se considera como una buena medida ya que esto significa que cerca de la mitad del código que fue tomado como base, ha sido reutilizado.

Composición

Para realizar la detección de la relación de composición en lenguaje Java se escribieron 62 líneas de código. Estas líneas de código, fueron tomadas como base para detectar este tipo de relación en los lenguajes C# y VB .Net, tal como se muestra en la Tabla 23.

Tabla 23. Líneas agregadas, modificadas y eliminadas para detectar la relación de Composición en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% reutilización
Java	62	-	-	-	-	-
C#	54	0	14	8	40	64.5%
VB .Net	61	15	22	16	24	38.7%

Los resultados muestran que para detectar la composición en C# no se agregaron líneas, se modificaron 14 y se eliminaron 8, por lo que el tamaño del archivo se redujo a un total de 54 líneas de código. El número de líneas que se mantuvo sin cambios es de 40, por lo que solamente se reutilizaron el 64.5% líneas del código base para detectar la relación de composición en C#. Esta tasa ha sido la más baja que se ha presentado con relación al lenguaje C#, y esto es debido a que en C# existen las clases genéricas que pueden estar parametrizadas con diversos

tipos, y éste es un concepto que no existe en el lenguaje de programación Java y que tuvo que ser tomado en cuenta para poderlo detectar en C#.

Para VB .Net, se agregaron 15 líneas que incrementaron el total de líneas, se modificaron 22 y se eliminaron 16 que volvieron a reducir el tamaño del archivo. Solamente 24 líneas se mantuvieron sin cambios, esto representa que solamente se reutilizó el 38.7% del código base. A su vez, esto significa que la mayoría del código base fue desechado.

Generalización

Para implementar la relación de generalización en Java se crearon 43 líneas de código, que luego sirvieron como base para implementarlas en C# y VB .Net.

Tabla 24. Líneas agregadas, modificadas y eliminadas para detectar la relación de Generalización en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	43	-	-	-	-	-
C#	43	0	1	0	42	97.67
VB .Net	33	23	0	33	10	23.25

En el lenguaje C# solamente fue necesario cambiar una sola línea, y esto representa el 97% de código de reutilización del código base. En éste caso en particular, solamente se cambió la palabra reservada *"extends"* por *":"*.

En VB .Net se eliminaron 33 líneas de código, lo cual redujo el número de líneas de código a 10, luego se agregaron 23, que incrementó el total de líneas de código a 33. Solamente 10 líneas de las tomadas como base fueron utilizadas sin cambios. Estas 10 líneas de código representan una tasa de reutilización del 23.25%, una tasa muy por debajo del promedio para éste lenguaje y que significa que la mayoría de las líneas de código tomadas como base fueron desechadas.

Realización

Para detectar la relación de realización en lenguaje Java, se llevaron a cabo 61 líneas de código. Éstas líneas de código fueron tomadas como base para detectar éste tipo de relación en los lenguajes C# y VB .Net. Los resultados se muestran en la Tabla 25.

Tabla 25. Líneas agregadas, modificadas y eliminadas para detectar la relación de Realización en C# y VB .Net

Lenguaje	Total	Agregadas	Modificadas	Eliminadas	Reutilizadas	% Reutilización
Java	61	-	-	-	-	-
C#	61	0	1	0	60	98.36
VB .Net	41	6	13	14	34	55.73

En el lenguaje C# solamente fue necesario cambiar una sola línea de código, y esto representa que el 98.36% de código ha sido reutilizado del código base. En éste caso en particular, solamente se cambió la palabra reservada *"implements"* por *":"*.

En el caso del lenguaje VB .Net se agregaron 6 líneas que incrementaron el número total de líneas en la regla JAPE, pero que disminuyó en 14, otras 13 fueron modificadas para terminar con solamente 34 líneas que se mantuvieron del código base, lo cual representa el 55.73% de reutilización que significa que no se modificó más de la mitad del código base. Se realizaron diversas modificaciones debido a la diferencia gramática y sintáctica entre Java y VB .Net.

Cabe mencionar que las líneas agregadas, modificadas y eliminadas para el motor de GATE son cero. Y esto significa que será posible agregar las reglas que se deseen para cualquier lenguaje de programación sin tener que mover una sola línea en el código fuente de alguno de los componentes de GATE. Solamente habrá que agregar la regla al componente JAPE Transducer.

5.3 Extensibilidad a otros patrones

Para esta evaluación no se hicieron pruebas formales. Sin embargo a este punto debería ser evidente que ya no se tienen que definir nuevas reglas JAPE para soportar la etapa de extracción de información puesto que todas las relaciones presentes en la programación orientada a objetos ya fueron definidas en el trabajo actual extendiendo el enfoque llamado "Minimal Key Structures".

Lo que si se requiere hacer para esto es la caracterización del nuevo patrón utilizando métricas de centralidad de ARS e implementar el algoritmo correspondiente.

5.4 Resumen

- Se evaluó cuantitativamente la precisión del método propuesto para la recuperación de patrones de diseño ésta mide qué tan buena fue la extracción de patrones correctos en relación a los resultados obtenidos.
- En 7 de 9 pruebas el método presentó una precisión de 1.
- Se evaluó cuantitativamente la extensibilidad del método a los lenguajes de programación C# y VB.Net. mediante la tomando como medida la reutilización de código.
- Para el lenguaje C# se reutilizó en promedio el 90.84% de las líneas creadas originalmente para el lenguaje de programación Java. De acuerdo a éste resultado, puede decirse que C# y Java son sintácticamente muy parecidos.
- Para el lenguaje VB .Net se reutilizó en promedio el 52.56% de las líneas creadas originalmente para el lenguaje de programación Java. De acuerdo a éste resultado, puede decirse que se ahorró un esfuerzo aproximado al 50% para adaptar el lenguaje de programación VB .Net.

6 Resumen, reflexiones y trabajo futuro

En este capítulo se muestra un resumen y reflexiones de este trabajo considerando los objetivos planteados en un inicio. Además se discutirán las limitaciones, el posible trabajo futuro del presente trabajo.

6.1 Resumen y reflexiones

En este trabajo se definió un método un método, y herramienta de soporte, para determinar la pertinencia de combinar el uso de técnicas de PLN y métricas de centralidad para la detección de patrones de diseño de sistemas orientados a objetos codificados en Java a partir de su código fuente.

La extracción de datos de código fuente fue realizada utilizando técnicas de PLN. Se comprobó que los lenguajes de programación pueden ser analizados con técnicas de PLN para extraer información relevante a la detección de patrones de diseño.

Se pudo notar que la robustez de los algoritmos para la extracción de los datos del código fuente influye mucho en la precisión de los patrones detectados. Por ejemplo, existen métodos en los cuáles no se podía extraer la relación de agregación y por lo tanto, no se pudieron extraer los patrones Composite.

El área del análisis de redes permitió el análisis estructural de la red de software que se forma luego de la extracción de información. La métrica de centralidad ayudó a encontrar patrones. Estos patrones tienen en común un nodo visiblemente central, el cuál es identificable por nuestro método.

Se puede comprender que la fase de almacenamiento permite tener una representación intermedia que no dependa del lenguaje de programación, y a la que se le pueda aplicar uno o más métodos específicos de detección de patrones.

Se describió el método para detectar tres patrones de diseño, que son, Façade, Adapter y Composite a partir del código fuente del sistema. El método presentado es una combinación entre medición y verificación estructural.

6.2 Acerca de los objetivos planteados

El objetivo general de este trabajo era el determinar la pertinencia de combinar el uso de técnicas procesamiento de lenguaje natural y métricas de centralidad para la detección de patrones de diseño de sistemas orientados a objetos codificados en Java a partir de su código fuente. A continuación discutimos los aspectos relevantes a los objetivos particulares asociados al objetivo general.

1. Evaluar el nivel de precisión con respecto a la detección de patrones de diseño. Como se mostró en el capítulo anterior en 7 de 9 pruebas el método presentó una precisión de 1. Sin embargo se observó en algunas pruebas, sobre todo del patrón Adapter, que el número de verdaderos positivos fue significativamente menor. La razón de esto es que en este primer esfuerzo solo se orientó la detección a patrones en su forma pura. Es bien sabido que muchos patrones pueden presentar algunas variantes de implantación que preservan la semántica del patrón.

2. Evaluar la extensibilidad a otros lenguajes de programación orientados a objetos. Para la detección de patrones se creó un componente para extraer los tipos de relación, tales como, asociación, agregación, composición, generalización y realización. Éste componente extrae las relaciones de acuerdo a un lenguaje de programación en específico e introduce los resultados a una base de datos en MySQL. Una vez que los datos se encuentran ahí, es posible realizar la búsqueda de los patrones de diseño sin importar en esa instancia el lenguaje de programación, por lo tanto, no existirán modificaciones en la parte del análisis, pero sí en la parte de la extracción de las relaciones de acuerdo a la sintaxis para un lenguaje de programación en específico. No es necesario modificar absolutamente nada en los demás componentes.

   Para permitir esto solamente habrá que escribir las reglas JAPE necesarias para ese lenguaje. Incluso se pueden tomar las ya existentes y modificarlas para adaptarlas a ese lenguaje de programación. Sin embargo, en la sección de limitaciones se hablará de los aspectos que se tienen que tomar en cuenta para escribir estas reglas.

   Se observó que, en general, la reutilización de la reglas JAPE es del 50 % en adelante. Observándose para C# niveles bastante superiores.

3. Evaluar la extensibilidad a otros patrones. Se discutió que lo que se requiere hacer para lograr esto es la caracterización del nuevo patrón utilizando métricas de centralidad de ARS e implementar el algoritmo correspondiente.

6.3 Trabajo futuro

En éste trabajo solamente se estudiaron patrones estructurales. De esta forma una posible línea de trabajo futuro incluye el detectar otros patrones. En este contexto, algunos casos requieren considerar de forma más cuidadosa aspectos de comportamiento de los patrones, ya que existen patrones, por ejemplo State y Strategy, que tienen una estructura similar, pero que pueden comportarse de diferente manera. Esto mismo puede ser comprobado en tiempo de ejecución mediante el estudio de trazas de ejecución.

En la práctica es posible que un sistema contenga clases que no sean utilizadas, por ser “código muerto”, pero que a pesar de ello en el sistema son parte de ciertas estructuras que corresponde a patrones patrones. Por lo tanto, sería necesario necesario comprobar que no sean “código muerto” mediante la observación de su comportamiento en tiempo de ejecución mediante el estudio de trazas de ejecución.

También, este trabajo en su versión actual está limitado a analizar código fuente de lenguajes de programación puramente orientados a objetos. Por ello, explorar en un futuro su extensión en algún otro lenguaje de programación multi paradigma debería ser objeto de estudio.

Aunque en los casos descritos antes el estudio de trazas de ejecución (análisis dinámico) en conjunto con el de código fuente (análisis estático) puede dar una mejor precisión. Sin embargo, el tiempo del análisis y visualización podría incrementar.

También se podría considerar el usar éste trabajo como uno de los primeros pasos para lograr la meta final de definir un método para reconstruir la arquitectura de un sistema de software. La arquitectura de un sistema de software es una representación muy abstracta del sistema en términos de estructuras de componentes y sus interconexiones (Bass L., Clements P., y Kazman R., 2012). La arquitectura de un sistema de software no está limitada a un patrón simple, a menudo es una combinación de muchos tipos de niveles de abstracción que componen el sistema completo.

7 Referencias

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentince Hall, Inc., 1996.

[2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. New York, New York, USA: John Wiley, 2013.

[3] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Iberoamericana, 1995.

[4] L. Šubelj and M. Bajec, “Community structure of complex software systems: Analysis and applications,” *Phys. A Stat. Mech. its Appl.*, pp. 2968–2975, May 2011.

[5] H. Müller, S. Tilley, and K. Wong, “Understanding software systems using reverse engineering technology perspectives from the Rigi project,” pp. 217–226, 1993.

[6] R. Kazman, L. O’Brien, and C. Verhoef, “Architecture Reconstruction Guidelines Third Edition,” no. November, 2003.

[7] J. Luo, “Social network structure and performance of improvement teams,” *Int. J. Bus. Perform. Manag.*, pp. 208–223, 2005.

[8] R. Reagan and E. W. Zuckerman, “Networks, Diversity, and Productivity: The Social Capital of Corporate R&D Team,” *Organ. Sci.*, vol. 12, p. 502, 2001.

[9] R. T. Sparrowe, R. C. Liden, S. J. Wayne, and M. L. Kraimer, “Social Networks and the Performance of Individuals and Groups,” *Acad. Manag. J.*, vol. 44, pp. 316–325, 2001.

[10] A. Malhotra, L. Totti, and W. M. Jr, “Studying user footprints in different online social networks,” in *Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012)*, 2012, pp. 1065–1070.

[11] T. Wolf, A. Schröter, D. Damian, and L. D. Panjer, “Mining Task-Based Social Networks to Explore Collaboration in Software Teams,” *IEEE Comput. Soc.*, pp. 58–66, 2009.

[12] C. Zachor and M. H. Gunes, “Software Collaboration Networks,” in *Springer-Verlab*, 2013, pp. 257–264.

[13] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, “Categorizing bugs with social networks: A case study on four open source software communities,” in *In Preceedings of the 35th International Conference on Software Engineering ICSE 2013*, 2013, no. Icse.

[14] A. Jermakovics, A. Sillitti, and G. Succi, “Mining and visualizing developer networks from version control systems,” in *In Preceeding of International Conference on Software Engineering ICSE 2011*, 2011, p. 24.

[15] P. Monge and N. Contractor, *Theories of communication networks*. Oxford New York: OXFORD University Press, Inc., 2003, p. 406.

[16] N. M. Tichy, M. L. Tushman, and C. Fombrun, “Social Network Analysis For Organizations,” *Acad. Manag. Rev.*, pp. 507–519, 1979.

[17] S. Wasserman and K. Faust, *Social Network Analysis, Methods and Applications*, 5th ed. Cambridge, United Kingdom: Cambridge University Press, 1999, p. 827.

[18] M. Newman, *Networks: An Introduction*. 2010.

[19] S. Fortunato, “Community detection in graphs,” in *Complex Networks and Systems Lagrage Laboratory*, 2010, p. 103.

[20] M. Girvan and M. E. J. Newman, “Community structure in social and biological networks,” *Proc. Natl. Acad. Sci. U. S. A.*, p. 8, Dec. 2001.

[21] “Unified Modeling Language (UML).” [Online]. Available: http://www.uml.org/.

[22] “Java Programming Language.” [Online]. Available: http://www.java.com/. [Accessed: 01-Jan-2014].

[23] “TIOBE Index for May 2014,” 2014. [Online]. Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. [Accessed: 15-Jul-2014].

[24] A. Gelbukh, “Tendencias recientes en el procesamiento de lenguaje natural,” *Centro de Investigación en Computación*, 2002. [Online]. Available: http://www.gelbukh.com/CV/Publications/2002/SICOM-2002-Trends.htm. [Accessed: 18-Nov-2014].

[25] R. Kumar, “Natural vs Programming Languages,” 2012. [Online]. Available: http://meetrajesh.com/archives/natural-vs-programming-languages.html.

[26] A. Chen, “Programming and natural languages,” *Stock/P6*, 2004.

[27] E. Chikofsky and J. Cross, “Reverse engineering and design recovery: A taxonomy,” *Software, IEEE*, no. January, pp. 13–17, 1990.

[28] “Java Development Tools.” [Online]. Available: https://eclipse.org/jdt/apt/index.php. [Accessed: 12-Dec-2014].

[29] “Java Reflection,” 2014. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/. [Accessed: 15-Dec-2014].

[30] “ANTRL parser.” [Online]. Available: http://www.antlr.org/. [Accessed: 23-Jan-2015].

[31] S. Hayashi and J. Katada, “Design pattern detection by using meta patterns,” *Trans. Inf. Syst.*, vol. E91-D, no. 4, pp. 933–944, Apr. 2008.

[32] H. Huang, S. Zhang, J. Cao, and Y. Duan, “A practical pattern recovery approach based on both structural and behavioral analysis,” *J. Syst. Softw.*, vol. 75, no. 1–2, pp. 69–87, Feb. 2005.

[33] C. Kramer and L. Prechelt, “Design recovery by automated search for structural design patterns in object-oriented software,” *Proc. 3rd Work. Conf. Reverse Eng.*, no. November, 1996.

[34] J. Dong, D. Lad, and Y. Zhao, “DP-Miner: Design pattern discovery using matrix,” *Proc. 14th Annu. IEEE Int. Conf. Work. Eng. Comput. Syst.*, pp. 371–380, Mar. 2007.

[35] Z. Balanyi and R. Ferenc, “Mining design patterns from C++ source code,” *Int. Conf. Softw. Maintenance, 2003. ICSM 2003. Proceedings.*, pp. 305–314.

[36] R. Ferenc, A. Beszedes, and T. Gyimothy, “Fact extraction and code auditing with columbus and sourceaudit,” in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 513–513.

[37] G. Antoniol, R. Fiutem, and L. Cristoforetti, “Using Metrics to Identify Desing Patterns in Object-Oriented Software,” *Softw. Metrics Symp. 1998. Metrics 1998. Proceedings. Fifth Int.*, pp. 23 – 34, 1998.

[38] D. Kirasić and D. Basch, “Ontology-based design pattern recognition,” *Knowledge-Based Intell. Inf. Eng. Syst.*, pp. 384–393, 2008.

[39] P. Castells, “La web semántica,” *Sist. interactivos y Colab. en la web*, pp. 195–212, 2005.

[40] N. Shi and R. Olsson, “Reverse engineering of design patterns from java source code,” *… Softw. Eng. 2006. ASE’06. 21st …*, pp. 123–134, 2006.

[41] I. Lamas, “Comparación de analizadores estáticos para código java,” *Univ. abierta Cataluña*, pp. 1–18, 2011.

[42] J. de la Horra, “Aplicaciones del cálculo matricial,” *Univ. Autónoma Madrid*.

[43] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, “Design pattern detection using similarity scoring,” *Softw. …*, vol. 32, no. 11, pp. 896–909, 2006.

[44] A. Chatzigeorgiou, “Application of graph theory to OO software engineering,” p. 29, 2006.

[45] J. Dong, Y. Sun, and Y. Zhao, “Design pattern detection by template matching,” *Proc. 2008 ACM Symp. Appl. Comput. - SAC ’08*, p. 765, 2008.

[46] “Template Matching,” 2014. [Online]. Available: http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. [Accessed: 28-Jan-2015].

[47] M. Gupta, A. Pande, and a. K. Tripathi, “Design patterns detection using SOP expressions for graphs,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, p. 1, Jan. 2011.

[48] P. Akshara, G. Manjari, and A. K. Tripathi, “A New Approach for Detecting Design Patterns by Graph Decomposition and Graph Isomorphism,” *Contemp. Comput. - Commun. Comput. Inf. Sci.*, pp. 108–119, 2010.

[49] “Definición de GATE.” [Online]. Available: https://gate.ac.uk/sale/tao/splitch1.html#x4-40001.1. [Accessed: 13-Nov-2014].

[50] D. Maynard, V. Tablan, and H. Cunningham, “Architectural Elements of Language Engineering Robustness,” *Cambridge Univ. Press*, vol. 1, no. 1, pp. 1–20, 2002.

[51] I. Philippow, “Design pattern recovery in architectures for supporting product line development and application,” *… Var. OO Prod. …*, pp. 42–57, 2003.

[52] H. Deitel and P. Deitel, *Cómo Programar en Java*. Prentince Hall, Inc., 2007.

[53] D. Palliotto and G. Romano, “¿Qué es UML?” [Online]. Available: http://www.docirs.cl/uml.htm. [Accessed: 11-Nov-2013].

[54] Oracle, “Las 10 razones principales para usar MySQL como base de datos integrada,” 2012.

[55] “Java Universal Network/Graph Framework.” [Online]. Available: https://code.google.com/p/jung/.

[56] O. A. Velázquez Álvarez and N. Aguilar Gallegos, “Manual introductorio al análisis de redes sociales. Medidas de centralidad,” *REDES-Revista Hisp. para el análisis redes Soc.*, p. 45, 2005.

[57] “JHotDraw,” 2015. [Online]. Available: http://www.jhotdraw.org/. [Accessed: 10-Mar-2015].

[58] “JRefactory,” 2015. [Online]. Available: http://jrefactory.sourceforge.net/. [Accessed: 10-Mar-2015].

[59] “JUnit,” 2014. [Online]. Available: http://junit.org/. [Accessed: 25-Mar-2015].

[60] J. Dong, Y. Zhao, and T. Peng, “Architecture and design pattern discovery techniques-a review,” in *Proceedings of International Conference on Software Engineering Research and Practice (SERP), USA*, 2007, p. 11.

[61] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, “Design pattern recovery through visual language parsing and source code analysis,” *J. Syst. Softw.*, vol. 82, no. 7, pp. 1177–1193, 2009.

CIMAT

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS, A.C.

BIBLIOTECA

AUTORIZACION
PUBLICACION EN FORMATO ELECTRONICO DE TESIS

El que suscribe

Autor(s) de la tesis: Juan Manuel Mauricio Zamarrón

Título de la tesis: Evaluando el uso de técnicas de procesamiento de lenguaje natural y métricas de centralidad para la detección de patrones de diseño de software

Institución y Lugar: Centro de Investigación en Matemáticas, Zacatecas

Grado Académico: Licenciatura ( ) Maestría (X) Doctorado ( ) Otro ( )

Año de presentación: 2015

Área de Especialidad: Maestría en Ingeniería de Software

Director(es) de Tesis: Dra. Perla Velasco Elizondo,

Correo electrónico: Dra. Alejandra García Hernández

Domicilio: jmauricio@cimat.mx

Segunda de la estación #47, Bellavista, 6ge. Zacatecas

Palabra(s) Clave(s): lenguaje natural, patrones de diseño, centralidad

Por medio del presente documento autorizo en forma gratuita a que la Tesis arriba citada sea divulgada y reproducida para publicarla mediante almacenamiento electrónico que permita acceso al público a leerla y conocerla visualmente, así como a comunicarla públicamente en la Página WEB del CIMAT.

La vigencia de la presente autorización es por un periodo de 3 años a partir de la firma de presente instrumento, quedando en el entendido de que dicho plazo podrá prorrogar automáticamente por periodos iguales, si durante dicho tiempo no se revoca la autorización por escrito con acuse de recibo de parte de alguna autoridad del CIMAT

La única contraprestación que condiciona la presente autorización es la del reconocimiento del nombre del autor en la publicación que se haga de la misma.

Atentamente

Juan Manuel Mauricio Zamarrón

Nombre y firma del tesista

CALLE JALISCO S/N MINERAL DE VALENCIANA APDO. POSTAL 402
C.P. 36240 GUANAJUATO, GTO., MÉXICO
TELÉFONOS (473) 732 7155, (473) 735 0800 EXT. 49609 FAX. (473) 732 5749
E-mail: biblioteca@cimat.mx