UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

Aspect Mining Mediante Sistemas Expertos

por
Lucía Masola
Nahuel Sliba

Dra. Claudia Marcos
Directora

Ing. Esteban Abait
Co-Director

Tandil, Diciembre 2009

Índice de Contenidos

Índice de Ilustraciones

Índice de Tablas

Índice de Gráficos

Agradecimientos

A mi familia, a mi novio y a mis amigos…

A Timpa, Pablo, Kevin y Juan P. Oregioni…

A mis directores, Claudia Marcos y Esteban Abait…

Lucía Masola.

A mi familia por ser amigos y a mis amigos por ser familia.

A mis directores por su infinita paciencia, Claudia Marcos y Esteban Abait.

Nahuel Sliba.

CAPÍTULO I

Introducción

1. Mantenimiento de Sistemas de Software

Los sistemas de software exitosos están destinados a sufrir incesantes modificaciones con el objetivo de mantenerse vigentes en el mercado. Los diseños de estos sistemas se ven desbordados por los nuevos requisitos funcionales surgidos en la etapa de mantenimiento, deteriorándose con el transcurso del tiempo. Incluir nuevas funcionalidades o realizar modificaciones a las ya existentes presupone un alto costo cuando estos eventuales – y asiduos escenarios en sistemas exitosos – no han sido contemplados en las etapas tempranas del proceso de desarrollo. Es así que resulta de suma importancia que el atributo de calidad de modificabilidad se vea reflejado claramente en estos sistemas [3].

2. Separación de Concerns

Se define a un concern como cualquier cuestión de interés en un sistema de software. La separación de estos desde la etapa de diseño es una forma de obtener sistemas más flexibles, que no sufran en demasía el impacto generado por requerimientos cambiantes. Realizar una separación exitosa de los concerns de importancia en un sistema de software aumenta la calidad del mismo, reduciendo los costos de producción y mantenimiento y facilitando su evolución, asegurando una competitividad prolongada en el mercado [2].

Conseguir aislar los concerns de un sistema de software no es una tarea sencilla. En la práctica resulta casi imposible llevar a cabo en forma completa esta separación. Este

fenómeno es conocido como la “tiranía de la descomposición dominante” [1]. No importa cuán bien una aplicación se descomponga en unidades modulares, siempre existirán concerns que atraviesan dicha descomposición. Estos se denominan crosscutting concerns y se encuentran diseminados por diferentes módulos del sistema, generando un impacto negativo en la calidad del mismo: la comprensibilidad del código se ofusca, la adaptabilidad disminuye, y la evolución del sistema se torna ardua y costosa.

3. Programación Orientada a Aspectos (AOP, Aspect Oriented Programming)

Con el objetivo de resolver la problemática del aislamiento exitoso de concerns surge un nuevo paradigma, la programación orientada a aspectos (AOP) [2]. AOP provee mecanismos para separar la funcionalidad central de un sistema de software de los concerns que atraviesan sus módulos. El código correspondiente a los crosscutting concerns es encapsulado en un nuevo constructor llamado aspecto. Esta nueva estructuración permite liberar a los módulos centrales del sistema de código que no corresponden a la lógica del negocio, y centralizar este código en los aspectos.

Dado que el paradigma introducido es bastante joven, el principal desafío reside en la migración de sistemas legados. Es decir, en tomar sistemas orientados a objetos – ya consolidados – y encontrar su equivalente orientado a aspectos. Esta migración permitirá mejorar la comprensión de los sistemas y la mantenibilidad y extensibilidad de los mismos.

En el proceso de migración se definen dos actividades principales, aspect mining y aspect refactoring [6]:

- **Aspect mining**: es la actividad de descubrir aquellos crosscutting concerns desde el código fuente o las trazas de ejecución de una aplicación que podrían ser encapsulados como aspectos del nuevo sistema.

- **Aspect refactoring**: es la actividad de transformar los aspectos candidatos identificados en el código orientado a objetos en aspectos reales en el código fuente.

Como resultado del estudio de estas actividades, surge un conjunto de técnicas que facilitan su puesta en práctica. Debido a la dimensión y complejidad de los sistemas legados y a la documentación incompleta sobre los mismos, la ejecución manual de estas técnicas de migración se torna un proceso propenso a errores, dificultoso, y muy costoso. Es así que surge la necesidad de contar con herramientas que asistan al programador en las etapas del proceso de migración aspectual.

4. Synergy Analysis Tool (SAT)

El objetivo del presente trabajo es el desarrollo de una herramienta capaz de asistir en la actividad de aspect mining. Para la implementación de la misma se utiliza un motor de inferencia que permite la identificación de crosscutting concerns de manera semi-automática. La herramienta se sustenta de la implementación de algoritmos estáticos de aspect mining para conseguir los resultados deseados. La combinación entre el motor de inferencia y el conocimiento sobre las técnicas de aspect mining constituyen un sistema experto en el dominio en cuestión. [7]

La herramienta desarrollada puede analizar proyectos codificados en lenguaje Java, examinando sus elementos en busca de seeds candidatos. En el proceso de identificación de seeds se pueden distinguir tres etapas fundamentales: la traducción de la información necesaria del proyecto a hechos lógicos, el procesamiento de estos para la obtención de resultados, y la presentación de estos últimos al usuario de la herramienta (Fig. I - 1).

Al ejecutar un análisis sobre un proyecto, el código de este es recorrido por un parser. El parser representa la estructura sintáctica del mismo en forma de árbol, haciendo uso de la noción de AST (Abstract Sintax Tree o árbol sintáctico abstracto) [5]. El propósito de este recorrido es generar un conjunto de hechos, que representan información estática

del proyecto, que sirvan como entrada para los algoritmos de aspect mining conocidos por el sistema experto. Una vez procesada esta información, se obtienen como resultado los seeds candidatos del sistema. Con la posterior intervención del programador se pueden extraer del código del proyecto los crosscutting concerns a refactorizar.

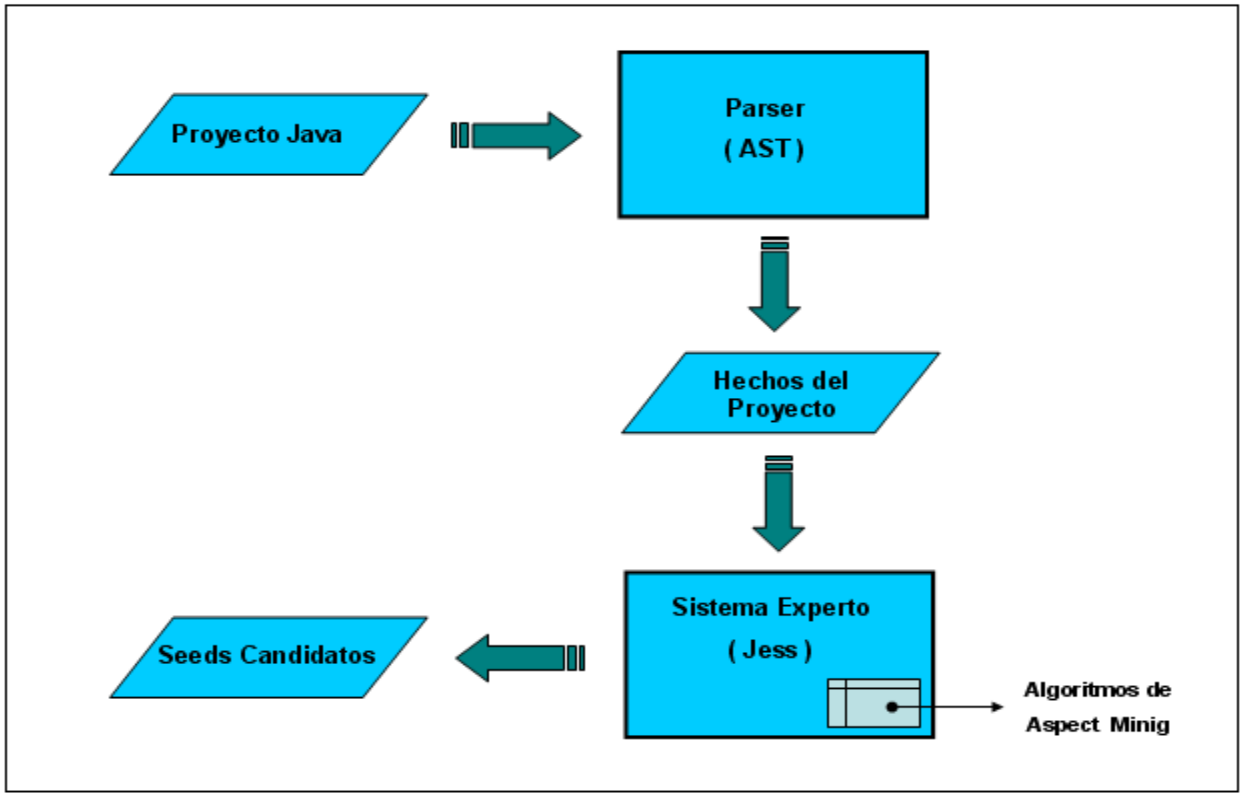

Fig. I - 1. Proceso de extracción de crosscutting concerns.

Los enfoques que se desarrollan en el trabajo analizan la información estática del sistema, es decir, la proveniente del código fuente. Se implementan cinco algoritmos: análisis de Fan-in, detección de Métodos Únicos, análisis de Relaciones de Ejecución, análisis de Métodos Redireccionadores y un último algoritmo, denominado Sinergia, que une a los 3 primeros.

Para la implementación y ejecución de los algoritmos de aspect mining se utiliza Jess [4, 7], un motor de reglas para el lenguaje Java, desarrollado en este mismo lenguaje en los laboratorios Sandia. La elección de este motor se fundamenta básicamente en la naturaleza

del problema. La facilidad de representar los algoritmos como un sistema experto proviene de la posibilidad de escribir conjuntos de reglas de manera independiente. Esta característica también permite el enriquecimiento del sistema sin realizar mayores modificaciones. Si se desea agregar conocimiento al sistema experto – agregado de algoritmos de aspect mining o refinamiento de los ya implementados – basta con escribir nuevas reglas o modificar las existentes. A su vez, la velocidad de procesamiento constituye un punto muy importante a causa de la magnitud de la información entrante. Jess usa una versión optimizada del algoritmo Rete [6]. El mismo sacrifica espacio para ganar velocidad.

Health Watcher (HW) [8] fue el sistema elegido para realizar las pruebas de la herramienta. HW es una aplicación desarrollada acorde a una arquitectura por capas utilizando tecnología J2EE. El mismo es seleccionado debido a que es un sistema real y suficientemente complejo e involucra un gran número de concerns clásicos, como por ejemplo sistemas usables, concurrentes, persistentes y distribuidos.

Los enfoques seleccionados para realizar las pruebas fueron Sinergia y Métodos Redireccionadores. Se realizaron dos corridas del primer enfoque variando sus parámetros de entrada y se logró identificar un 50% y 66.66% de los concerns existentes en el sistema con una precisión de 75% y 33.33% respectivamente. El análisis de Métodos Redireccionadores identificó el 16.66% de los concerns con una precisión del 100%, y adicionó al conjunto de concerns identificados clases adapters diseminadas en la capa de negocio del código.

5. Organización del Documento

El presente documento se organiza de la siguiente manera:

- En el capítulo 2, se introduce la problemática relacionada al mantenimiento de aplicaciones. Se presenta el paradigma de Programación Orientada a Aspectos (AOP) y el proceso de migración de sistemas legados.

- En el capítulo 3, se presentan y explican las técnicas de aspect mining de interés para este trabajo.

- El capitulo 4 constituye un marco teórico sobre los Sistemas Expertos Basados en Reglas.

- La propuesta del trabajo de tesis es presentada y desarrollada en el capítulo 5, donde se explican los algoritmos desarrollados como sistemas expertos y la herramienta en cuestión.

- El capitulo 6 muestra la experiencia del uso de la herramienta sobre un caso de estudio, analizando los resultados obtenidos.

- El 7mo y último capítulo muestra las conclusiones y trabajos futuros obtenidos del desarrollo del proyecto.

CAPÍTULO II

Mantenimiento de Aplicaciones

1. Concerns en un Sistema de Software

Un sistema de software es la realización de un conjunto de "concerns" [36]. Se define a un concern como todo lo que un stakeholder quiera considerar como una unidad conceptual, incluyendo características, requerimientos no funcionales y decisiones de diseño e inclusive *programming idioms* [1] [37]. Dichos concerns deben estar implementados en el sistema a fin de satisfacer su objetivo general y pueden ser clasificados en dos categorías [2]:

- **Core concerns**: son aquellos que capturan la funcionalidad central de un módulo.
- **Crosscutting concerns**: son aquellos que capturan requerimientos a nivel del sistema que atraviesan múltiples módulos. Ejemplos de estos son la autenticación, logging, seguridad, integridad en las transacciones, etc.

Esta clasificación se realiza con el fin de reducir la complejidad del diseño y la implementación de un sistema. Para realizar esta separación se descompone el conjunto de requerimientos en concerns. Independientemente de la metodología que se use, dicha separación e identificación es un ejercicio importante en el desarrollo del software. El problema surge cuando los concerns identificados no pueden implementarse en módulos

[1] Los programming idioms son patrones de bajo nivel específicos a un lenguaje de programación. Estos patrones describen cómo solucionar ciertos problemas específicos a la implementación en un lenguaje en particular [37].

independientes [2]. El programa derivado de dicha implementación presenta dificultades de mantenimiento debido a que un simple cambio en uno de ellos puede impactar en muchas partes del sistema [3]. A pesar de que la separación mencionada pueda ser natural, las metodologías de programación actuales carecen de mecanismos para realizarlo en la fase de implementación. [2]

2. Evolución de las Metodologías de Programación

La ingeniería de software ha atravesado un largo camino comenzando en los lenguajes a nivel máquina, pasando por la programación procedural y llegando a la programación orientada a objetos (OOP, Object-oriented programming). Esta evolución de las metodologías de programación permite a los ingenieros enfrentarse con problemas de mayor complejidad y nivel de abstracción que en décadas anteriores. Si bien todos los lenguajes de programación proveen un conjunto limitado de abstracciones, estos mecanismos no permiten mantener la separación de concerns en el código de una aplicación [3].

Actualmente, la programación orientada a objetos (OOP) es la metodología elegida en los nuevos proyectos de desarrollo de software. La mayor ventaja de este paradigma reside en el modelado del comportamiento común, por lo que los sistemas orientados a objetos son desarrollados mapeando las entidades del mundo real del dominio de la aplicación en una jerarquía de clases [25]. Sin embargo, no todos los requerimientos de la aplicación pueden mapearse a una única unidad modular (clase). La OOP no cumple un buen papel en abordar estos requerimientos ya que quedan dispersos en varios módulos, generalmente no relacionados entre sí [2]. Como consecuencia, se observan dificultades en el mantenimiento de estos sistemas.

El mantenimiento es la parte central del ciclo de vida del software y comúnmente representa más de la mitad del costo del desarrollo del sistema. Es por esto que no es

sorprendente que la capacidad de mantenimiento para los programas haya sido un punto clave en el diseño de lenguajes de programación [2].

Para modificar una aplicación, los desarrolladores deben identificar la idea de alto nivel, o concepto a ser transformado, y luego localizar, comprender y modificar el concern que representa a dicho concepto en el código [26]. Es por esto que probablemente el factor más importante que determina la mantenibilidad de un programa es la estructuración del mismo [3]. Está comúnmente aceptada la premisa de que la mejor manera de solucionar la complejidad es simplificándola. En diseño de software, la mejor manera de simplificar un sistema complejo es identificar cada concern y luego asignarlo a un único módulo que lo realice.

La programación orientada a objetos fue desarrollada en respuesta a la necesidad de dicha modularización, en particular, este paradigma permite realizar una buena modularización de los core concerns, aunque falla cuando se trata de modularizar crosscutting concerns.

Los módulos centrales en las aplicaciones orientadas a objetos, por lo general, se encuentran débilmente acoplados gracias al uso de interfaces. No sucede lo mismo para los crosscutting concerns, debido a que la implementación se lleva a cabo en dos partes: la pieza perteneciente al lado del servidor y la pieza perteneciente a la de los clientes. Los términos “cliente” y “servidor” son usados en el sentido clásico de OOP, los cuales definen objetos que proveen un conjunto de servicios y objetos que usan estos servicios, respectivamente. En este tipo de sistemas se modularizan en clases e interfaces los servicios provistos. Sin embargo, el pedido del servicio se encuentra disperso en todos los clientes, provocando un fuerte acoplamiento entre los módulos que necesitan los servicios y el módulo que lo provee.

La Fig. II -1 muestra cómo un sistema bancario implementa el registro de eventos de la información utilizando el paradigma orientado a objetos. El módulo de logging representa

el servicio provisto (servidor) y los módulos de “accounting”, “ATM” y “database” utilizan este servicio. El módulo de logging puede implementarse utilizando una interface y así proveer los siguientes beneficios:

- Disminuir el acoplamiento entre los clientes y la implementación del logging. Cualquier cambio en la implementación del servicio no afectará a los clientes.
- Permitir el reemplazo de la implementación del servicio con solo instanciar la interface del logging.

A pesar del buen diseño del módulo de logging, los módulos clientes necesitan el código para invocar la API del servicio. En color gris se denota el acoplamiento de cada módulo.

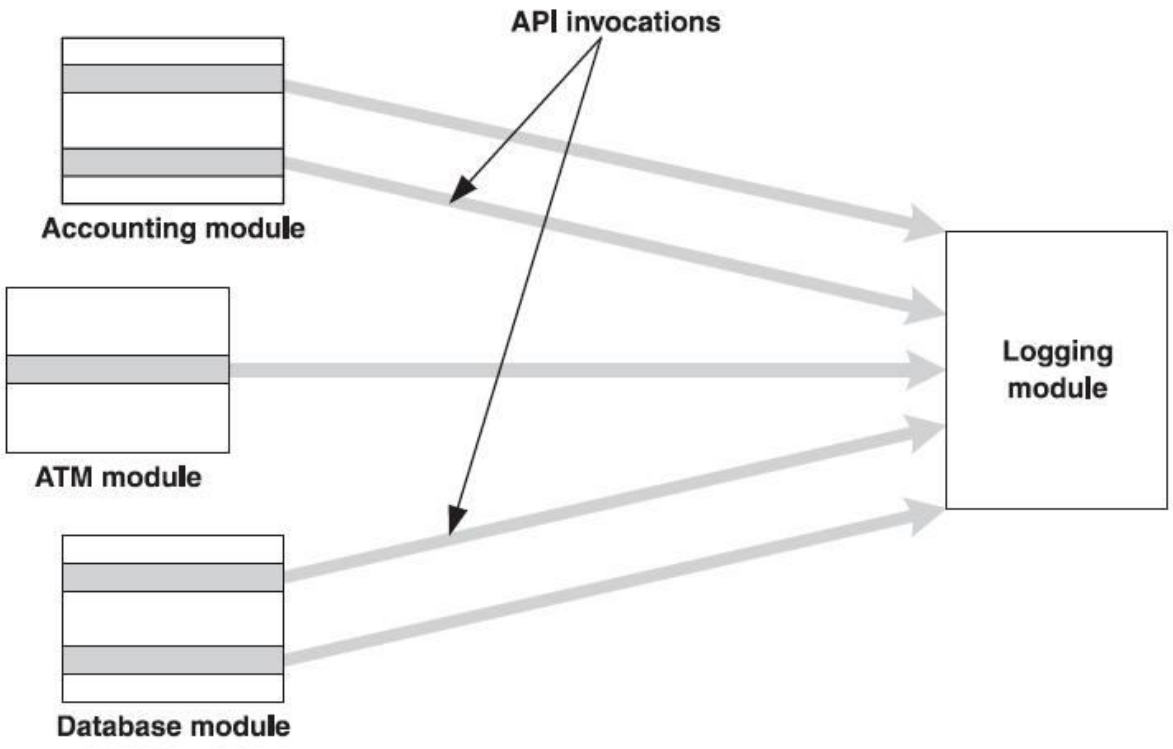

Fig. II - 1. Implementación del concern logging utilizando las técnicas convencionales.

En consecuencia, se observa que la implementación de los crosscutting concerns en OOP no es la más adecuada si se piensa en relación a la mantenibilidad del sistema. La modularización de los crosscutting concerns no es lo suficientemente independiente y el código que invocan a los servicios se encuentra entremezclado con el código de la lógica de los clientes. Por esta razón, la mantenibilidad del sistema en referencia a los crosscutting concerns es más dificultosa y costosa, conllevando a problemas a la hora de modificar, agregar o reutilizar este tipo de concerns [3]. Estos problemas tienen su origen en la llamada "tiranía de la descomposición dominante", la cual determina que no importa cuán bien una aplicación se descompone en unidades modulares, siempre existirán concerns que atraviesen dicha descomposición [27].

El Desarrollo de Software Orientado a Aspectos (AOSD, Aspect-Oriented Software Development) [2] es un paradigma que permite dar solución al problema de la separación de la funcionalidad central de un sistema de software de los concerns que atraviesan la descomposición del mismo. Para esto, el paradigma provee de un nuevo constructor denominado aspecto, cuyo objetivo es encapsular un crosscutting concern. La combinación de los aspectos con los módulos centrales de la aplicación se denomina weaving la cuál es llevada a cabo con el fin de formar la versión final del sistema final.

La Fig. II - 2 muestra la implementación del mismo ejemplo de logging expuesto anteriormente (Fig. II-1) utilizando la orientación a aspectos. La lógica del logging reside dentro del aspecto "logging" y los clientes no tienen código que haga referencia al mismo. Con esta modularización, cualquier cambio al requerimiento de logging afecta solo al aspecto logging aislando completamente a los clientes. En consecuencia, la implementación presenta los siguientes beneficios en comparación con el ejemplo citado previamente: mayor modularización y claridad en las responsabilidades de cada módulo, mayor reusabilidad y flexibilidad y menor acoplamiento de los módulos.

3. Implementación Orientada a Aspectos

Un lenguaje de programación orientado a aspectos debe proveer 3 componentes básicos. Estos componentes son:

- **Lenguaje funcional**: implementa la funcionalidad básica de la aplicación con el cual se programan los componentes (por ejemplo Java).
- **Lenguaje de aspectos**: implementa los crosscutting concerns.
- **Weaver o tejedor**: permite unir el lenguaje funcional y el lenguaje de aspectos utilizando las reglas de weaving. Las reglas especifican cómo integrar los concerns implementados a fin de formar el sistema final.

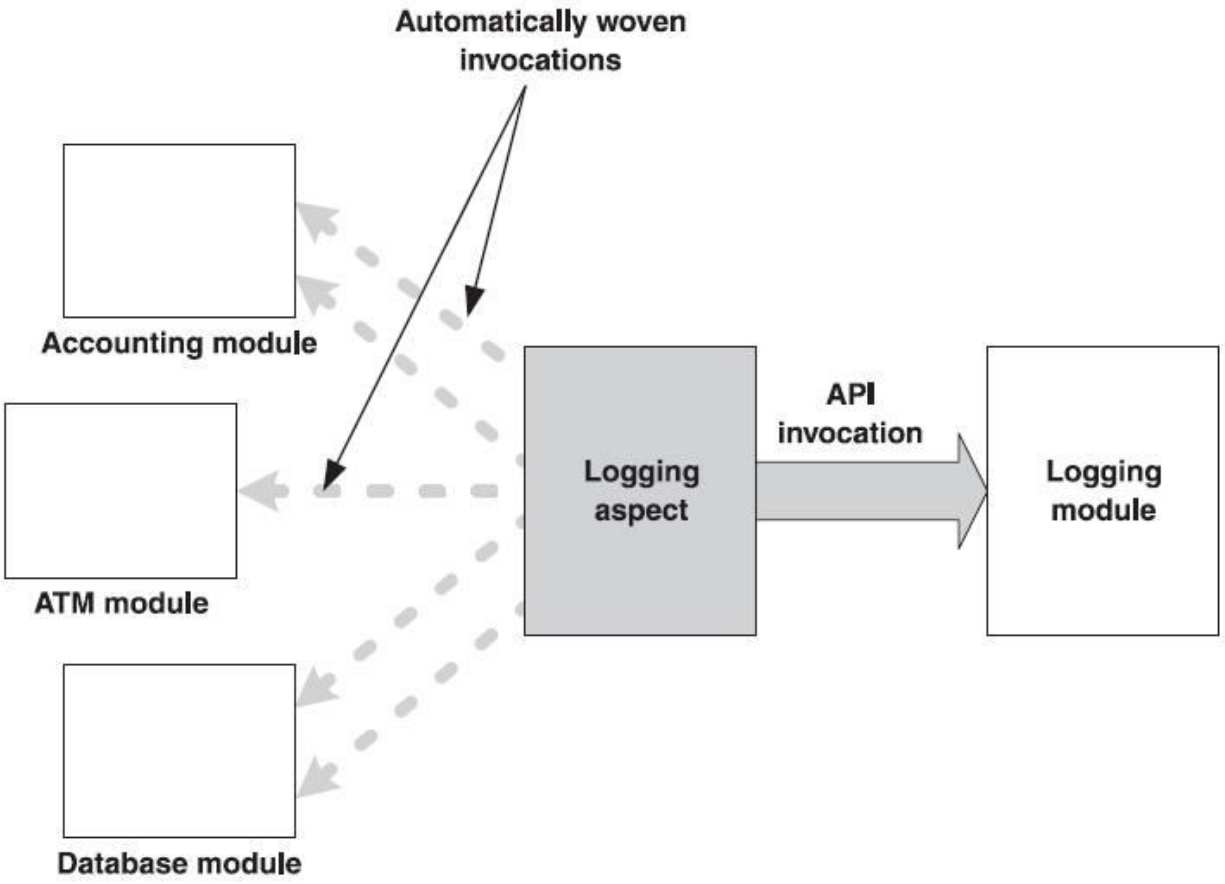

Fig. II - 2. Implementación del concern logging utilizando las técnicas de AOP.

El poder de la programación orientada a aspecto (AOP sus siglas en inglés) proviene de la forma en que las reglas de weaving pueden ser expresadas ya que se pueden especificar unas pocas líneas de código.

Las plataformas más conocidas para el desarrollo orientado a aspectos son las siguientes:

- **AspectJ**: es una extensión de propósito general orientada a aspectos del lenguaje de programación *Java*. El primer paso en la compilación, weaving, adhiere los aspectos y las clases como si el código de los aspectos estuviera dispersado a través de las clases centrales. El segundo paso realiza la compilación normal de Java utilizando el comando *javac* [33]. AspectJ utiliza aspectos, join-point, pointcuts y advices como constructores del lenguaje. Los aspectos son unidades modulares que encapsulan los crosscutting concerns. Los joint-points representan cualquier punto de ejecución del programa en el que su comportamiento puede ser modificado por un aspecto. Los pointcut se utilizan dentro de los aspectos para especificar uno o más join-points en los módulos principales o incluso en otros aspectos. Un advice define un tipo de método que se utiliza para encapsular código de crosscuttings [3].

- **SpringAOP**: Un componente clave en Spring es el framework de AOP. Si bien no existe la restricción de utilizar aspectos, esta funcionalidad está soportada en el framework. SpringAOP provee solamente joinpoints a nivel de métodos y los aspectos son configurados utilizando la sintáxis de definición normal de los beans [35].

- **JBossAOP**: es un framework orientado a aspectos construido completamente en Java. Puede ser usado en cualquier ambiente de programación e integrado en el application server. JBossAOP no es

simplemente un framework, sino que provee un conjunto de aspectos que pueden ser aplicados mediante anotaciones, pointcuts o dinámicamente en tiempo de ejecución. Esto último, denominado weaving dinámico, es una de las principales ventajas del mismo [34].

4. Migración de Sistemas Orientado a Objetos hacia Sistemas Orientado a Aspectos

La adopción de un nuevo paradigma de programación conduce a la pregunta de cómo migrar los sistemas existentes al nuevo paradigma, pregunta que en la actualidad se aplica al paradigma orientado a aspectos [28]. Si bien, el encapsulamiento de los crosscutting concerns de un sistema legado en aspectos es potencialmente beneficioso, decidir qué partes del código corresponden a un crosscutting concern es muy dificultoso [25].

Se pueden distinguir tres fases diferentes para realizar y evolucionar la migración a aspectos: “aspect exploration”, “aspect extraction” y “aspect evolution”’ [46].

- **Aspect exploration**: Previo a la introducción de aspectos en el código de un software existente, se debe poder explorar si el sistema exhibe o no algún crosscutting concern al cual valga la pena ser extraído a aspectos. La tiranía de la descomposición dominante [27] implica que es probable que grandes sistemas los contengan. Durante esta etapa se trata de descubrir aspectos candidatos, se intenta discernir qué representan los crosscutting concerns, dónde y cómo están implementados y cuál es su impacto en la calidad del programa.

- **Aspect extraction**: Una vez que los crosscutting concerns han sido identificados en un sistema de software y su impacto ha sido evaluado, se puede considerar migrar el mismo hacia una versión orientada a aspectos. En el caso de decidir realizar dicha migración, se necesita una forma de convertir

los aspectos candidatos, esto es los crosscutting concerns identificados en la fase de exploración, en aspectos reales. Al mismo tiempo, hay una necesidad de técnicas para testear el software refactorizado con el fin de asegurarse de que la nueva versión continúa trabajando como se esperaba y para manejar los pasos de las modificaciones durante la fase de transición.

- **Aspect evolution**: De acuerdo a la primera ley de evolución de software de Belady y Lehman [47], todo sistema de software que está siendo usado se someterá continuamente a cambios o se convertirá en obsoleto luego de un período de tiempo. No hay razones para creer que esta ley no se aplica a los sistemas de software orientado a aspectos, por lo que surgen preguntas referidas a su evolución. Esta fase intenta responder preguntas tales como: ¿cuán diferente es la evolución de un sistema orientado a aspectos de uno tradicional?, ¿pueden las mismas técnicas de evolución de software tradicional ser aplicadas en el nuevo paradigma?, ¿los nuevos mecanismos de abstracción de AOP pueden incurrir en nuevos problemas de evolución de software que requieren nuevas soluciones?

La Fig. II - 3 visualiza el proceso de migración y evolución de un sistema legado a un sistema que utiliza aspectos.

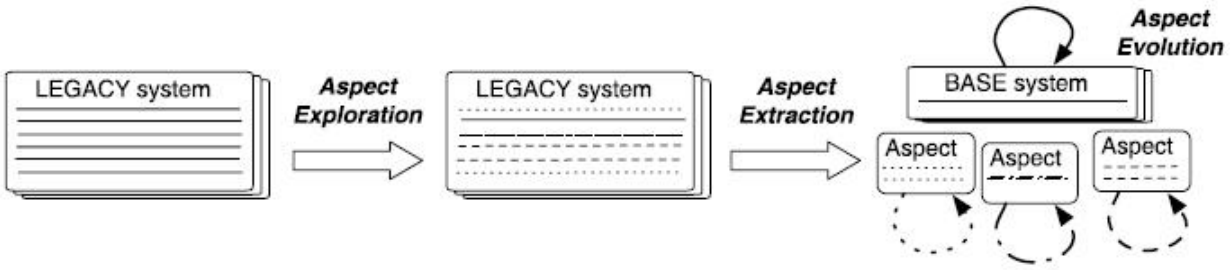

Fig. II - 3. Migración y evolución de un sistema legado a un sistema orientado a aspectos.

Debido al tamaño y complejidad de los sistemas orientados a objetos, existe la necesidad de técnicas y herramientas que automaticen la migración de estos a la

orientación a aspectos [6]. La mayoría de las herramientas distinguen dos fases en el proceso de migración: aspect exploration and aspect extraction [46]. Se define a aspect exploration como la actividad de identificar y analizar crosscutting concerns en sistemas no orientado a aspectos. Aspect extraction es la actividad de separar el código de los crosscutting concerns del sistema moviéndolo en uno o más aspectos y removiéndolo del código original [46].

4.1. Aspect Exploration

Kellens et Al. [38] distinguen tres categorías principales de técnicas que pueden ayudar a localizar los crosscutting concerns en un sistema de software. Estas técnicas son:

- **Early aspect discovery techniques:** la investigación de esta técnica trata de descubrir aspectos en las fases tempranas del ciclo de vida de un software [39] tales como requerimientos y análisis de dominio [27, 28, 29] y diseño de la arquitectura [40]. A pesar de que la técnica mencionada puede ayudar a identificar ciertos crosscutting concerns en un sistema de software, se la considera menos prometedora que aquellas en las cuales el enfoque se centra en el código fuente. Esto se debe a que los documentos de requerimientos y la arquitectura generalmente se encuentran desactualizados u obsoletos.

- **Dedicated browsers:** Una segunda clase de enfoque son los browsers de código avanzado de propósito especial, los cuales ayudan al desarrollador a navegar manualmente el código fuente de un sistema para explorarlo en busca de crosscutting concerns. Esta técnica comienza típicamente desde una ubicación en el código llamada “seed” como punto de entrada para guiar a los usuarios mediante la sugerencia de otros lugares en el que el mismo concern pueda aparecer. De esta manera, se construye de forma interactiva un modelo de las diferentes zonas del código que constituyen un crosscutting

concern. Se pueden listar los siguientes ejemplos clasificados bajo dedicated browsers: Concern Graphs [41], Intensional Views [42], Aspect Browser [43], (Extended) Aspect Mining Tool [34, 35], SoQueT [44] y Prism [45].

- **Técnicas de aspect mining:** Las técnicas de aspect mining automatizan el proceso de descubrimiento de crosscutting concerns y proponen al usuario uno o más aspectos candidatos. Para este fin, las técnicas evalúan el código fuente o datos adquiridos ejecutando o manipulando el sistema. Estas técnicas tienen en común, al menos, que buscan síntomas de crosscutting concerns tales como código disperso y código entremezclado mediante la aplicación de técnicas de data mining, comprensión de software o de análisis de programas [6]. El código disperso corresponde a concerns cuya implementación abarca diferentes módulos del sistema. Por otra parte, el código entremezclado corresponde a módulos que manejan múltiples concerns simultáneamente [2]. Las técnicas de aspect mining pueden clasificarse en dos grupos diferentes: técnicas de análisis estático y técnicas de análisis dinámico. Las técnicas basadas en análisis estático analizan la frecuencia de los elementos del programa y se basan en la homogeneidad sintáctica de los crosscutting concerns. Por otra parte, las técnicas basadas en análisis dinámico buscan patrones de ejecución durante la ejecución del programa.

4.2. Aspect Extraction

Debido al alto costo de mantenimiento de los sistemas de software, existe la necesidad de técnicas que reduzcan la complejidad e incrementen la calidad interna de los mismos. Se conoce al dominio de investigación que comprende a este problema como reestructuración [48]. En el caso específico del desarrollo de software orientado a objetos se denomina refactorización [30], el cual se define al refactoring como el proceso de

cambiar un sistema de software orientado a objetos para mejorar la estructura interna del código de manera de no alterar el comportamiento externo del mismo [31].

Entre los diversos refactorings existentes, se encuentra el denominado aspect refactoring, el cuál, como se mencionó anteriormente, define la migración de código orientado a objetos hacia código orientado a aspectos. Los refactorings son organizados sistemáticamente en catálogos. En [32] Monteiro define un catálogo de 28 refactorings de aspectos, para todos ellos especifica su nombre, situación típica, descripción de la acción recomendada, motivación, mecanismos y códigos de ejemplo. En motivación describe cuándo debería usarse el refactoring, en mecanismos se describen una serie de pasos a seguir para poder aplicar el refactoring. Finalmente, en ejemplos de código se plasma en concreto lo descrito en los puntos anteriores, ilustrando de esta manera el refactoring. Luego, el catálogo divide los refactorings en cuatro grupos:

- **Código java a aspectos**: Comprende el encapsulamiento de diferentes elementos del código en un aspecto.
- **Estructura interna de los refactorings de aspectos**: implica mejorar la estructura interna de un aspecto
- **Generalización de los aspectos**: transformaciones en la jerarquía de aspectos.
- **Código legado**: se utiliza cuando existen interfaces de código legado que no pueden ser modificadas.

5. Automatización de la Fase de Identificación de Crosscutting Concerns

En el proceso de migración de sistemas legados orientados a objetos a sus análogas versiones orientadas a aspectos, el conjunto de técnicas de aspect mining disponible

constituye un recurso muy valioso en la identificación de crosscutting concerns. La herramienta desarrollada en este trabajo utiliza un subconjunto de estas técnicas con el objetivo de automatizar la etapa de aspect exploration. En el siguiente capítulo se presentan y explican estas técnicas, así como también los conceptos fundamentales encontrados en la actividad de aspect mining.

CAPÍTULO III

Aspect Mining: Trabajos Previos

1. Introducción

Aspect mining es la actividad de descubrir potenciales crosscutting concerns a partir del análisis del código fuente de un sistema o desde un conjunto de trazas derivadas de su ejecución [58]. En este capítulo, se presentan los trabajos previos desarrollados en él área. Para cada una de las técnicas propuestas se describe su algoritmo, se exhibe un ejemplo y se muestra la herramienta que la implementa.

2. Conceptos de Aspect Mining

En esta sección se detallan conceptos comunes a todas las técnicas propuestas.

- **Seed**: elemento del código fuente que pertenece a la implementación concreta de un crosscutting concern. Una vez encontrado, se puede expandir la búsqueda del resto de los elementos (clases y métodos) correspondientes al concern.

- **Seed candidata o aspecto candidato**: salida obtenida de las herramientas de aspect mining. Se denominan de esta manera, ya que se requiere de la interacción humana para determinar si se corresponde a un crosscutting concern o no.

- **Seed confirmada**: seed candidata que fue confirmada por el desarrollador.

- **Falso positivo**: seed candidata rechazada por el desarrollador.

- **Falso negativo**: parte de un crosscutting concern conocido que no fue detectado por la herramienta.

3. Procesamiento del Lenguaje Natural sobre el Código Fuente

Shepherd, Pollock y Tourwé [49] propusieron una técnica de aspect mining que intenta descubrir aspectos candidatos en el código fuente de un sistema por medio de técnicas de Procesamiento del Lenguaje Natural (PLN). Particularmente, se utiliza la técnica de lexical chaining [51] para identificar grupos de entidades de código fuente relacionadas semánticamente y evaluar cuándo estos representan crosscutting concerns.

El enfoque se basa en la hipótesis de que los crosscutting concerns generalmente están implementados mediante una rigurosa convención de nombres. Los desarrolladores recurren a estas convenciones con el fin de vincular entidades del código fuente relacionadas, y mejorar el entendimiento del software. Por ejemplo, implementaciones de patrones de diseño [50] abarcan múltiples clases y métodos que se relacionan mediante nombres bien específicos, se pueden citar nombres como “acceptVisitor” o “addObserver”. El punto clave es aplicar PLN para explotar relaciones semánticas entre palabras y entre nombres predefinidos.

La ventaja que presenta este enfoque sobre trabajos previos que utilizan convenciones de nombres [52] es que se reconocen relaciones semánticas más complejas entre palabras analizando identificadores y comentarios del código fuente para entender y comprender la semántica del mismo.

3.1. Algoritmo

El algoritmo propuesto consiste en dos pasos:

1. Obtener todas las palabras del código fuente de un programa.

2. Construir las cadenas léxicas procesando cada palabra: por cada palabra busca la cadena, por medio de lexical chaining, que esté relacionada semánticamente y agrega la palabra a la cadena. Si no encuentra tal cadena comienza una nueva con dicha palabra. Luego de procesar cada palabra se obtiene una lista de cadenas de palabras de distintas longitudes.

3.2. Lexical Chaining

Lexical chaining es una técnica de PLN que agrupa en cadenas léxicas palabras semánticamente relacionadas de un documento [52]. El proceso de encadenamiento toma como entrada un texto y procede agrupando cada palabra en una cadena con otras palabras del mismo texto que están semánticamente relacionadas.

Con el fin de obtener estas cadenas léxicas, se debe calcular la distancia semántica o la fuerza de la relación entre dos palabras [5]. Por ejemplo, existe una relación muy fuerte entre ***novela*** y ***poema***, ambos siendo trabajos literarios. En cambio, existe una relación más débil entre ***novela*** y ***tesis***, ambos siendo escritos, lo cual es una relación menos específica.

En función de automatizar el cálculo de distancia, se utiliza una base de datos de relaciones conocidas de palabras, como por ejemplo WordNet [53]. La distancia semántica se calcula midiendo la longitud de los caminos de relaciones entre dos palabras desde la base, donde a mayor distancia las palabras se encuentran menor es su relación semántica. En adición a la distancia obtenida, se debe tener en cuenta la naturaleza de la palabra, el tipo de relación puede variar en caso de que las palabras se utilicen como adjetivos, sustantivos, verbos, etc.

3.3. Utilización de Lexical Chaining para Identificar Concerns

Los autores suponen que algunas de las cadenas léxicas que se pueden derivar del código fuente se corresponden con concerns de alto nivel. Esto se debe a que los desarrolladores están forzados a utilizar pistas del lenguaje debido a la falta de estructuras

adecuadas para modularizar correctamente estos crosscutting concerns. Estas pistas proveen información sobre la funcionalidad del código, por lo que si dos regiones de código fuente contienen palabras similares, es probable que ambas regiones implementen funcionalidades relacionadas.

Sin embargo, esta suposición no es suficiente para determinar la presencia de crosscutting concerns. En adición a lo mencionado, se buscan cadenas en las que sus miembros presenten gran cantidad de dispersión (por ejemplo, las palabras provienen de distintos archivos fuentes).

Debido a la gran cantidad de palabras que aparecen en el código de un sistema la propuesta se focaliza en subconjuntos específicos de strings con el objetivo de reducir asociaciones sin significado, se tienen en cuenta los comentarios y campos, y el tipo y nombre de los métodos.

Posteriormente, cada uno de estos strings es procesado acorde a su tipo:

- **Comentarios:** usualmente se escriben en forma de sentencias o frases. Por esta razón, se utiliza un speech tagger, el cual etiqueta las palabras de un comentario y elimina la ambigüedad entre ellas. Por ejemplo, las palabras “dirección” y “destino” podrían relacionarse si ambas se utilizan como sustantivos y no debieran hacerlo si se utilizaran como verbos.

- **Nombre de métodos:** se separan los nombres de los métodos que contengan palabras en mayúscula en palabras separadas. Por ejemplo goAndDoThatThing consiste en 5 palabras: go, and, do, that y thing. Luego de la separación se utiliza speech tagger para la clasificación.

- **Nombre de clases y campos:** se asume que los campos y las clases son sustantivos. Se separan los identificadores de la misma manera que los nombres de los métodos.

3.4. Ejemplo

La Fig. III-1 presenta un ejemplo de lexical chaining en donde los concerns ***auction*** y ***money*** están representadas con superíndice (todos los miembros del concern auction están marcados con el superíndice [1], y los concerns de Money con [2]).

A 9-year-old boy successfully underwent surgery Wednesday to remove most of a brain tumor he nicknamed "Frank", and which was the subject of an **online auction**[1] to help raise *money*[2] for *medical bills*[2].

Cells from the tumor, which had been treated with chemotherapy and radiation, will now be studied to determine if it is malignant. David Dingman-Grover, of Sterling, Virginia., went into surgery around 10 a.m. at Cedars-Sinai Medical Center, said Frank Groff . . .

David named his tumor after Frankenstein's monster, who scared him until he dressed up as the fictional character for Halloween. His parents **sold**[1] a bumper sticker reading "Frank Must Die" on eBay to raise *funds*[2] for his treatment.

Fig. III - 1. Párrafo con cadenas léxicas.

Se reconocen dos concerns en el texto, uno correspondiente a las cadenas léxicas de la palabra **auction** (*online auction* y *sold*) y el segundo a la palabra money (*money*, *medical bills* y *funds*).

3.5. Herramienta

Los autores implementaron la herramienta de lexical chaining como un Plug-in de Eclipse [54]. Tiene como propósito automatizar el algoritmo propuesto, incluyendo la selección de palabras y el lexical chaining. Luego de correr la aplicación, los usuarios navegan las cadenas obtenidas.

4. Detección de Métodos Únicos

Gybels y Kellens [55] utilizaron heurísticas para descubrir crosscutting concerns mediante la identificación de métodos únicos. Este enfoque intenta identificar los concerns

que fueron implementados centralizados en un único método, al cual se lo invoca desde varios lugares del código fuente.

En los lenguajes no orientados a aspectos los desarrolladores implementan los crosscutting concerns de tal forma que el código resultante se encuentra disperso y entremezclado [71]. Dichos concerns son implementados de manera que el weaving que se realiza en los lenguajes orientados a aspectos es manual. Una forma de hacerlo es tener métodos centralizados para realizar tareas específicas e invocarlos de diversos lugares del código, dando lugar así a código disperso.

Un ejemplo típico es el caso del logging, en donde la funcionalidad se encuentra bien implementada mediante los mecanismos provistos por la programación orientada a objetos: una clase Singleton [50] encapsula el manejo del archivo del logging y representa con el método log el servicio mencionado. A pesar de que no se evidencia código entrelazado, el método log será llamado de diversos lugares donde se requiera registrar información (Fig. III - 2).

Otro ejemplo de este tipo de concerns es la implementación del mecanismo de notificación y actualización del patrón Observer [50] en Smalltalk. La Fig. III-3 muestra como un método único “changed” es definido en la raíz de la clase para ser llamado cuando se realiza una notificación de actualización.

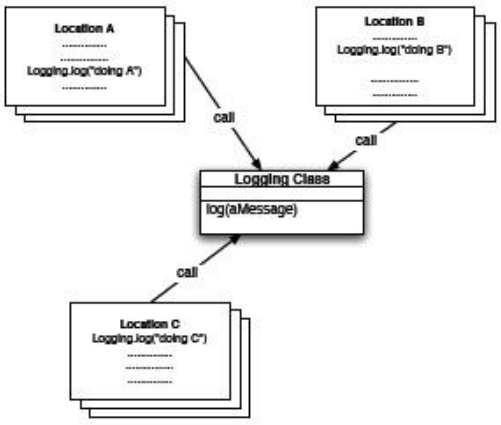

Fig. III - 2. Logging como clase central que provee la funcionalidad de loggeo.

```
moveTo: newx and: newy
  x := newx.
  y := newy.
  self changed: #x.
  self changed: #y.

shiftSidewaysTo: newx
  x := newx.
  self changed: #x.
```

Fig. III - 3. Implementación del concern de actualización en Smalltalk.

4.1. Algoritmo

La heurística intenta encontrar aquellos concerns que fueron implementados utilizando un método central, particularmente un método único. Se define a un método único como: "Un método sin valor de retorno el cual implementa un mensaje que no es implementado por ningún otro método".

4.2. Ejemplo

Este enfoque fue probado en una imagen de Smalltalk, la cual contenía 3.400 clases con 66.000 métodos. Se reportaron 6248 métodos únicos a los cuales se les aplicaron un conjunto de filtros. El primer filtro eliminó del análisis los métodos de acceso, se seleccionaron los métodos que son llamados más de 5 veces, dejando un total de 228 métodos finales. El conjunto final de métodos es inspeccionado con el fin de identificar los aspectos candidatos. La Tabla lll-1 muestra algunos de los métodos únicos identificados y la Tabla lll-2 lista alguno de los aspectos identificados junto con el número de llamada de cada método.

Clase	Método Único
CodeComponent	#startLoad
Locale	#currentPolicy:

Menu	#addItem:value:
ScheduledWindow	#updateEvent:
UIFinderVW2	#showClasses:
ComposedText	#centered
UIBuilder	#wrapWith:
Text	#emphasizeAllWith:
Cursor	#show
Image	#pixelsDo:

Tabla III - 1. Ejemplo de métodos únicos imagen de Smalltalk.

Clase	Método único (Signatura)	Invocaciones
Parcel	#markAsDirty	23
ParagraphEditor	#resetTypeIn	19
UIPainterController	#broadcast PendingSelectionChange	18
CodeRegenerator	#pushPC	15
AbstractChangeList	#updateSelection:	15
PundleModel	#updateAfterDo:	10

Tabla III - 2. Clases, métodos únicos y cantidad de veces que los métodos son llamados.

5. Clustering Jerárquico de Métodos Relacionados

Shepherd y Pollock [56] reportaron un experimento en el cual utilizaron Clustering Jerárquico Aglomerativo (CJO) [77] para agrupar métodos relacionados.

Algunos trabajos previos en aspect mining [57] han utilizado análisis conceptual para agrupar código relacionado a un concern particular a partir de los nombres de métodos y

clases. Los resultados de estos enfoques presentan al usuario una lista de nodos conceptuales, donde cada uno contiene una lista de hijos y cada hijo es una clase o un método. En estos casos no se reporta el lattice de conceptos y tampoco se permite la visualización de cuerpos de métodos simultáneamente. En consecuencia muchas relaciones entre vecinos, por mínima distancia que haya entre ellos, se pierden de vista.

A diferencia de trabajos basados en análisis conceptual [62, 64], este enfoque le facilita al usuario la visualización de los crosscutting concerns del sistema ya que relaciona el código de los mismos. Para ello, los autores utilizan una función de distancia semántica basada en (PLN), la cual permite agrupar en clusters métodos semánticamente relacionados del sistema. En consecuencia, cada cluster contendrá métodos que colaboren para satisfacer funcionalidad relacionada o correspondiente a un único concern. Luego, mediante una inspección de estos clusters es posible identificar cuáles de ellos se corresponden a crosscutting concerns. Cada cluster contiene métodos relacionados semánticamente, representando, generalmente, un crosscutting concerns.

5.1. Algoritmo

Se aplica CJO con el fin de agrupar métodos relacionados [25] y se ubica cada método en un cluster. A continuación, se detallan los pasos de este enfoque:

1. Ubicar todos los métodos en su propio cluster.
2. Comparar todos los pares de clusters usando una función de distancia y marcar el par con la menor distancia.
3. Si la distancia de los pares marcados es menor a un cierto valor de umbral, se unen ambos grupos. En caso contrario se detiene el algoritmo.

Por consiguiente, CJO primero ubica todos los métodos en su propio grupo.

Posteriormente, se repiten los pasos 1 y 2 hasta que no existan grupos que estén lo suficientemente cerca del valor de umbral. El algoritmo devuelve como resultado los grupos con membrecía mayor a 1.

La función de distancia utilizada se describe a continuación. Para dos métodos *m* y *n* la distancia se define como 1 / longitudSubcadena(m.name, n.nane). Para el caso en que se deban comparar dos clusters, se aplica el mismo procedimiento pero utilizando la subcadena que todos los métodos del cluster comparten.

5.2. Ejemplo

Los autores proponen un caso de estudio basado en el sistema JHotDraw 5.4b2. El enfoque identifica 3 casos representativos de crosscutting concerns:

1. Crosscutting concerns representados como una interface y sus métodos implementados de manera consistente.

2. Crosscutting concerns representados como una interface con sus métodos implementados que contengan código duplicado y lógica dispersa y que se encuentran implementados de forma inconsistente

3. Crosscutting concerns representados como métodos con nombres similares entre las clases, sin interfaces explícitas, con gran cantidad de código duplicado pero implementados de forma consistente.

Según los autores, la categoría uno es la menos común de las tres, ya que esta categoría incluye aquellos métodos que implementan interfaces y cuya implementación es extremadamente específica. El segundo de los casos agrupa crosscutting concerns en los cuales la implementación de los métodos no es uniforme, cada uno implementa la lógica siguiendo distintos patrones. Por último, la tercera categoría identifica métodos que el usuario debería haber implementado como una interface o un aspecto y no lo hizo.

5.3. Herramienta

Los autores implementan la técnica como parte de un IDE orientado a aspectos llamado AMAV (Aspect Miner and Viewer). La herramienta permite la fácil adaptación de la medida de distancia y el algoritmo es usado en combinación con las herramientas de visualización del IDE que no solo lista los clusters que fueron encontrados, sino que también muestra un panel de crosscutting concerns.

6. Análisis de Fan-in

La técnica de análisis de Fan-in fue propuesta por Marin, van Deursen y Moonen [58]. Este enfoque se basa en la aplicación de la métrica de fan-in con el objetivo de descubrir aquellos métodos del sistema que presenten una mayor dispersión (scattering). Los autores argumentan que es muy común que el código duplicado en un sistema legado sea refactorizado en un único método cuyo cuerpo está formado por dicho código duplicado. En consecuencia, estos métodos serán llamados desde diversos lugares, obteniendo así un alto valor alto de fan-in. En una reestructuración orientada a aspectos de un código legado, los concerns identificados con un valor alto de fan-in constituirán parte de un advice y el sitio de llamado corresponderá al contexto que necesita ser capturado usando un pointcut.

6.1. Algoritmo

El análisis de Fan-in consiste en tres pasos bien diferenciados: calculo, filtrado y análisis.

6.1.1. Cálculo de Métrica de Fan-in

Se define a la métrica de fan-in como la medida del número de métodos que llaman a otro método [59]. Para el cálculo de la métrica se reúnen el conjunto de potenciales llamadores de un método y se toma la cardinalidad de este conjunto. Sin embargo, el valor

exacto de fan-in depende de la manera en que se interprete el polimorfismo de los métodos (tanto métodos llamadores como métodos llamados). Por esta razón, se plantean refinamientos para el cálculo de este valor.

El primer refinamiento toma en cuenta el número de cuerpos de métodos distintos que llaman a otro método. De esta manera, si un método abstracto es implementado en dos subclases concretas, se consideran a estas dos implementaciones como llamadores separados.

Teniendo en cuenta que se intenta encontrar métodos que son llamados desde diferentes lugares, siendo estos potenciales crosscutting concerns, se plantea un segundo refinamiento que comprende los llamados a métodos polimórficos. En el caso de que se encuentre un método *m* que pertenece a cierto concern, es muy probable que tanto el método redefinido de las superclases como de las subclases pertenezcan al mismo concern. Por esta razón, si un método *m'* llama a un método *m* de la clase *C,* se agrega también a *m'* como método llamador de cada método *m* declarado en las superclases y subclases de *C.* Con esta definición, los métodos abstractos actúan como acumuladores: cuando una implementación específica de una de sus subclases es invocada, no solo se aumenta el fan-in del método específico, sino que también aumenta el valor de fan-in del método padre.

El tercer y último refinamiento concierne a las superclases. Para este caso en particular se sabe qué método está siendo invocado, en consecuencia, solo se extiende el conjunto de llamadores de este método.

6.1.2. Filtrado de Resultados

Luego de completar el cálculo del valor de fan-in para todos los métodos, se aplican una serie de filtros con el fin de obtener un conjunto más pequeño de métodos que tengan mayor chance de implementar un crosscutting concern. Los autores plantean 3 filtros que se listan a continuación:

- Se restringe el conjunto de métodos a aquellos que presentan un fan-in superior a cierto umbral. Este puede ser un valor absoluto (por ejemplo 10) o un porcentaje relativo (por ejemplo el 5% de los métodos con mayor valor de fan-in). El valor absoluto de umbral puede interpretarse como un indicador del nivel de dispersión del método analizado.

- Se eliminan los métodos *getters* y *setters* de la lista de métodos. Este filtrado puede estar basado en convenciones de nombres (métodos que coincidan con el patrón "get*" y "set*") o en un análisis de la implementación de los métodos.

- Se excluyen los métodos utilitarios, como por ejemplo el método *toString(),* clases del estilo *XMLDocumentUtils* que contengan la subcadena *utils* en su nombre, métodos que manipulan colecciones, etc.

6.1.3. Análisis de Resultados

El paso final de la propuesta consiste en el análisis manual del conjunto final de métodos. El razonamiento para la selección de los aspectos puede abordarse tanto en forma top-down como bottom-up.

El enfoque bottom-up consiste en descubrir los crosscutting concerns del sistema a partir del conjunto de seeds obtenidas. Esto implica buscar las invocaciones de un método que presente un alto valor de fan-in, en donde la regularidad de estos sitios hará posible que se capture a las llamadas en un mecanismo de pointcut, y al método con alto fan-in en un advice.

En el enfoque top-down, el desarrollador debe conocer en el dominio o las nociones típicas de crosscutting concerns presentes en el sistema para posteriormente buscar los seeds relacionados a dichos concerns.

6.2. Ejemplo

Se presenta un ejemplo del enfoque propuesto que permite observar el cálculo del valor de fan-in para el método *m*. La Fig. III - 4 muestra la jerarquía de clases del ejemplo, y la Tabla III-3 los resultados de distintas llamadas al método.

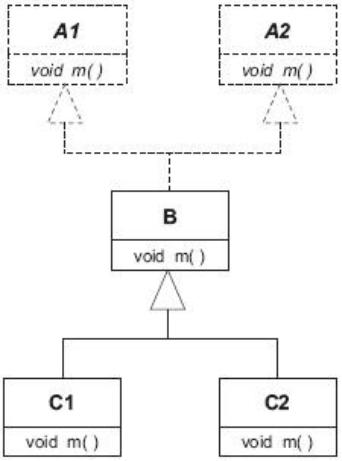

Fig. III - 4. Ejemplo de Jerarquía de Clases.

Contribución de Fan-in					
Llamadas	**A1.m**	**A2.m**	**B.m**	**C1.m**	**C2.m**
f1(A1 a1){a1.m();}	1	0	1	1	1
f 2(A2 a2){ a2.m();}	0	1	1	1	1
f 1(B b){ b.m();}	1	1	1	1	1
f 1(C1 c1){ c1.m();}	1	1	1	1	0
f 1(C2 c2){ c2.m();}	1	1	1	0	1
Valor Total	4	4	5	4	4

Tabla III - 3. Valores de Fan-in.

La Tabla III-3 muestra las llamadas directas a un método, y su respectivo impacto en el fan-in de cada uno de los métodos de la jerarquía de clases (Fig. III - 5). Por ejemplo, el llamado *f1(A1 a1){a1.m();}* suma 1 al fan-in de *A1.m* por ser un llamado directo, y acumula 1

al fan-in de *B.m*, *C1.m* y *C2.m* debido a la redefinición del método llamado por sus clases hijas. El valor total muestra el valor de fan-in de cada método luego de haber realiado todas las llamadas definidas en la tabla. Por ejemplo, el método *m* de la clase *B* presenta un valor de fan-in de 5. Esto se debe a que el método es llamado una vez directamente (fan-in = 1) y a que actúa como acumulador de sus clases hijas y padres (fan-in = 1 + 4).

6.3. Herramienta

Los autores desarrollaron FINT (Fan-in Tool), un plug-in para eclipse [54] que provee soporte automático para calcular la métrica y filtrar métodos, y provee ayuda para realizar el análisis de los aspectos candidatos.

7. Detección de Clones como Indicadores de Crosscutting Concerns

El código duplicado puede considerarse como un síntoma de la presencia de crosscutting concerns en el código fuente de un sistema [60]. Esto se debe a que los crosscutting concerns no se encuentran bien modularizados y ciertas partes de su implementación deben ser replicadas en diferentes partes del sistema.

Shepherd, E. Gibson, y L. Pollock [60] presentan un método basado en grafos de dependecia (PDG) con el fin de identificar aspectos candidatos en el código fuente de un sistema. Cada método es representado mediante un grafo, los cuáles pueden ser comparados de manera de encontrar código similar.

7.1. Algoritmo

El algoritmo planteado por los autores propone 4 fases y permite identificar aspectos candidatos cuyo refactoring hacia aspectos podría realizarse mediante un before advice de AspectJ. Esto se debe a que la técnica identifica código duplicado al comienzo de los métodos del sistema. A continuación, se listan los pasos del algoritmo:

1. Construir el grafo PDG a nivel código de cada método existente.

2. Identificar el conjunto de refactorings candidatos (control-based).

3. Filtrar refactorings candidatos no deseados (data-based).

4. Combinar conjuntos de candidatos relacionados en clases.

Cada grafo es construido representando cada sentencia en el código como un nodo en el grafo, donde las relaciones entre los nodos representan dependencias de control o datos entre las sentencias. El algoritmo comienza analizando las sentencias que se encuentran al principio de los métodos con el fin de reducir la complejidad del cálculo. Por esta razón, solo se pueden identificar advice del tipo "before.

La fase de identificación reporta, en numerosas ocasiones, seeds candidatos que no son deseados, por lo tanto deben aplicarse filtros, los cuáles descartan clones que poseen diferencias en dependencias de datos.

Los dos pasos previos dan como resultado pares de clones candidatos. Dado que la comparación es realizada método a método, es posible que candidatos similares o hasta idénticos hayan sido reportados en diferentes pares. La cuarta fase intenta eliminar estos duplicados combinando los conjuntos que presentan esta característica.

7.2. Ejemplo

Los autores reportan la aplicación del enfoque sobre el código fuente del contenedor Tomcat [61], el cual contiene 430 archivos, 7.704 métodos y 38.495 líneas de código. Se determinó que más del 90% de los aspectos candidatos reportados fueron extremadamente útiles. La Fig. III - 5 muestra un ejemplo de dos métodos específicos reportados para la aplicación Tomcat, los cuales son léxicamente diferentes, aunque de acuerdo a la representación PDG son equivalentes. Las líneas 10 y 11 son semánticamente idénticas a las líneas 19 y 20 aunque planteadas en contextos diferentes.

7.3. Herramienta

Los autores presentan el enfoque como una extensión del framework Ophir. El mismo está implementado como un plug-in de eclipse [54] y permite el análisis automático de los clones y el refactoring manual y automático de los aspectos.

8. Análisis de Patrones Recurrentes en Trazas de Ejecución

Breu y Krinke [62] propusieron un enfoque dinámico para la extracción de aspectos de un programa. El mismo, comienza con la extracción de un conjunto de trazas de ejecución, las cuales son generadas durante diferentes corridas del programa y usadas como pool de datos. Luego, estas trazas son investigadas con el fin de obtener patrones de ejecución recurrentes basados en diferentes restricciones que indican cuando un patrón se repite. Los autores consideran que este conjunto de patrones de ejecución recurrentes corresponde a potenciales crosscutting concerns que describen funcionalidad recurrente del sistema y por lo tanto posibles crosscutting concerns.

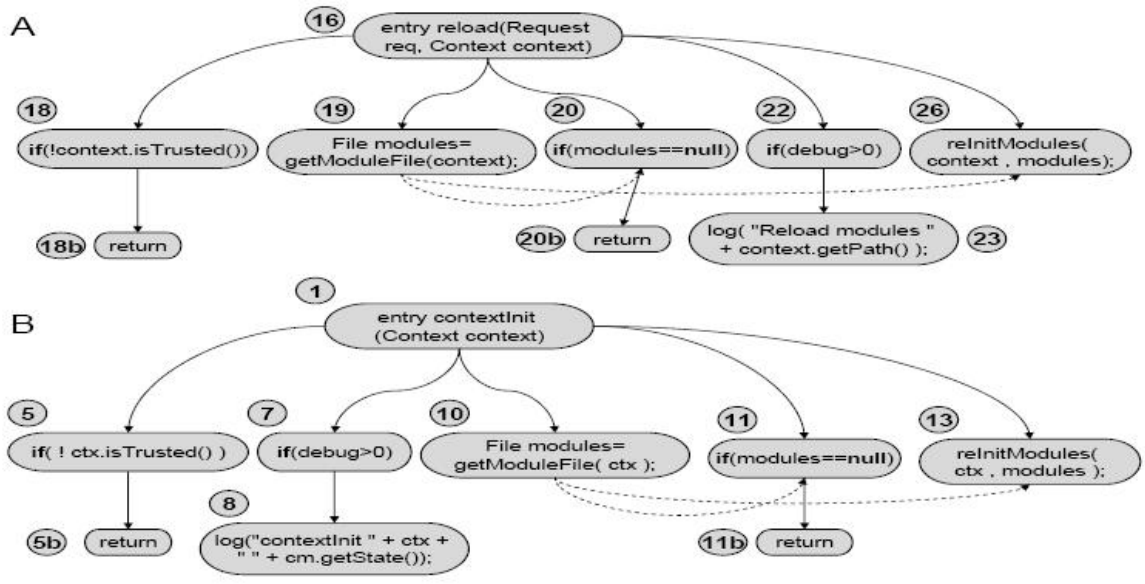

Fig. III - 5. Ejemplo de PDGs de dos métodos de la aplicación Tomcat.

8.1. Algoritmo

El enfoque consiste en dos pasos: la clasificación de relaciones de ejecución y su posterior análisis basado en restricciones. Luego de realizados estos dos pasos, se realiza una exploración manual de los resultados para extraer los aspectos candidatos.

8.1.1. Clasificación de Relaciones de Ejecución

Con el fin de analizar las trazas de un programa se introduce la noción de relaciones de ejecución entre dos métodos de una traza. Intuitivamente, una traza de ejecución de un programa es una secuencia de invocaciones a métodos. Formalmente, una traza de ejecución *Tp* de un programa *P* con *Np* (signatura método) es definida como una lista [*t1*,..., *tn*] de pares *ti* pertenecientes a (*Np* x {*ent*, *ext*}), donde *ent* indica la entrada a la ejecución de un método y *ext* la salida. Se representa a los puntos de entrada y salida con los caracteres "{" y ""}" respectivamente. En la Fig. III- 6 muestra un ejemplo de una traza de ejecución.

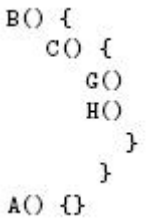

Fig. III - 6. Ejemplo de traza de ejecución.

Los autores proponen 4 diferentes tipos de relaciones de ejecución:

- **Outside-before-execution**: el método B es llamado antes que el método A. Formalmente se define como: u->v es una relación Outside-before-execution si *u*, *v* ε *Np,* donde [(u,ext),(v,ent)] es una sublista de *Tp.* Luego, $S^{->}(Tp)$ es el conjunto de relaciones Outside-before-execution en una traza de ejecución *Tp.*

- **Outside-after-execution**: El método A es llamado antes que el método B. Formalmente se puede definir como la reversa de Outside-before-execution. Formalmente, v <- u es una relación Outside-after-execution si u -> v ε $S^{->}$*(Tp)*. El conjunto de todas las relaciones Outside-after-execution en una traza de ejecución *Tp* se denota como $S^{<-}$*(Tp).*

- **Inside-first-execution**: El método G es el primero en ser invocado durante la ejecución del método C. Formalmente *u* ϵ_T *v* es una relación Inside-first-execution si *u, v* ϵ *Np* y si [(v,ext),(u,ent)] es una sublista de *Tp.* Luego, $S^{\epsilon T}$*(Tp)* es el conjunto de todas las relaciones de este tipo en una traza de ejecución.

- **Inside-last-execution**: El método H es el último en ser invocado durante la ejecución del método C. Formalmente *u* $\varepsilon_\perp$ *v es* llamada una relación Inside-last-execution si *u, v* ϵ *Np* y [(u,ext),(v,ent)] es una sublista de *Tp.* Luego, $S^{\epsilon\perp}$*(Tp)* es el conjunto de todas las relaciones de este tipo en una traza de ejecución.

8.1.2. Restricciones de Relaciones de Ejecución

Se deben definir ciertas restricciones con el fin de decidir bajo qué circunstancias las relaciones de ejecución serán consideradas como patrones recurrentes en las trazas de ejecución y en consecuencia potenciales crosscutting concerns en el sistema. El enfoque plantea dos tipos de restricciones:

- **Restricción uniforme:** se refiere a relaciones de ejecución que ocurren siempre de la misma manera. Siendo *u -> v* una relación outside-before-execution se define como una relación recurrente si cada ejecución de *v* es precedida por *u*. La argumentación para las relaciones outside-after-execution es análoga. Para las relaciones inside-execution, *u* ϵ_T *v* (o *u* $\varepsilon_\perp$ *v*) la

restricción de uniformidad exige que *v* nunca sea ejecutado en primer lugar (o último) por otro método que no sea *u*.

- **Restricción crosscutting:** refiere a relaciones que ocurren en más de un contexto de ejecución. Para las relaciones inside-execution el contexto se define como los métodos que rodean al método invocado. Para las relaciones outside-executions el contexto de llamada es el método que se ejecuta antes o después a un método específico.

8.1.3. Análisis de Relaciones de Ejecución

Luego de realizado el filtrado, se obtienen las relaciones de ejecución que se repiten a lo largo de la aplicación (en varios contextos y de manera uniforme), lo que representa un indicador de código entrelazado.

8.2. Ejemplo

La Fig. III - 7 ejemplifica una traza de ejecución de un código fuente.

```
1 B() {
2   C() {
3     G() {}
4     H() {}
5   }
6 }
7 A() {}
8 B() {
9   C() {}
10 }
11 A() {}
12 B() {
13   C() {
14     G() {}
15     H() {}
16   }
17   J() {}
18 }
19 F() {
20   K() {}
21   I() {}
22 }
23 J() {}
24 G() {}
25 H() {}
26 A() {}
27 B() {
28   C() {}
29   G() {}
30   F() {
31     K() {}
32     I() {}
33   }
34 }
35 D() {
36   C() {}
37   A() {}
38   B() {
39     C() {}
40   }
41   K() {}
42   I() {
43     J() {}
44   }
45   G() {}
46   E() {}
47 }
```

Fig. III - 7. Ejemplo de traza de ejecución.

La Fig. III - 8 muestra el conjunto $S^{->}$ de relaciones Outside-before-execution para el ejemplo mostrado en la Fig. III - 7. Luego, el conjunto $S^{<-}$ se obtiene directamente de las trazas o puede aplicarse la reversa de $S^{->}$.

$$
\begin{aligned}
S^{\rightarrow} = \{ & \mathtt{B}() \rightarrow \mathtt{A}(), \mathtt{G}() \rightarrow \mathtt{H}(), \mathtt{A}() \rightarrow \mathtt{B}(), \mathtt{C}() \rightarrow \mathtt{J}(), \\
& \mathtt{B}() \rightarrow \mathtt{F}(), \mathtt{K}() \rightarrow \mathtt{I}(), \mathtt{F}() \rightarrow \mathtt{J}(), \mathtt{J}() \rightarrow \mathtt{G}(), \\
& \mathtt{H}() \rightarrow \mathtt{A}(), \mathtt{B}() \rightarrow \mathtt{D}(), \mathtt{C}() \rightarrow \mathtt{G}(), \mathtt{G}() \rightarrow \mathtt{F}(), \\
& \mathtt{C}() \rightarrow \mathtt{A}(), \mathtt{B}() \rightarrow \mathtt{K}(), \mathtt{I}() \rightarrow \mathtt{G}(), \mathtt{G}() \rightarrow \mathtt{E}() \}
\end{aligned}
$$

Fig. III - 8. Conjunto $S^{->}$.

La Fig. III - 9 muestra los conjuntos de relaciones Inside-first-execution y Inside-last-execution respectivamente ($S^{\in T}$, $S^{\in \perp}$).

$$
\begin{aligned}
S^{\in_{\top}} = \{ & \mathtt{C}() \in_{\top} \mathtt{B}(), \mathtt{G}() \in_{\top} \mathtt{C}(), \mathtt{K}() \in_{\top} \mathtt{F}(), \mathtt{C}() \in_{\top} \mathtt{D}(), \\
& \mathtt{J}() \in_{\top} \mathtt{I}() \} \\
S^{\in_{\perp}} = \{ & \mathtt{H}() \in_{\perp} \mathtt{C}(), \mathtt{C}() \in_{\perp} \mathtt{B}(), \mathtt{J}() \in_{\perp} \mathtt{B}(), \mathtt{I}() \in_{\perp} \mathtt{F}(), \\
& \mathtt{F}() \in_{\perp} \mathtt{B}(), \mathtt{J}() \in_{\perp} \mathtt{I}(), \mathtt{E}() \in_{\perp} \mathtt{D}() \}
\end{aligned}
$$

Fig. III - 9 Conjunto $S^{\in T}$ *y* $S^{\in \perp} S^{->}$.

8.3. Herramienta

Los autores desarrollaron una herramienta llamada DynAMiT (Dynamic Aspect Mining Tool) que implementa el enfoque.

9. Análisis Formal de Trazas de Ejecución (FCA)

Tonella y Ceccato [64] proponen una técnica de aspect mining dinámica, la cual aplica algoritmos de formal concept analysis o análisis conceptual (FCA sus siglas en inglés, Formal Concept Analysis) a las trazas de ejecución con el fin de identificar posibles aspectos. FCA es una rama de lattice theory, el cual, dado un conjunto de objetos y atributos que los describen, genera conceptos (grupos máximos de objetos con atributos en común).

Esta técnica utiliza trazas de ejecución generadas mediante casos de uso correspondientes a la funcionalidad principal de la aplicación. A continuación, la relación

entre las trazas, la funcionalidad asociada y las unidades de compilación (método, clases) invocadas en cada ejecución son exploradas en función del lattice de conceptos producido por FCA. Los conceptos son clasificados como aspectos candidatos si presentan código entrelazado y disperso.

9.1. Concept Analysis para Feature Locations

Concept análisis [78] es una rama de lattice theory que provee una forma de agrupar objetos maximizando la cantidad de atributos que tienen en común. Dado un contexto (O, A, R), siendo O el conjunto de objetos, A el conjunto de atributos y R el conjunto de relaciones binarias entre O y A, un concepto específico *c* se define como el par (X,Y):

$$X = \{o \in O, Va \in Y: (o,a) \in R\}$$
$$Y = \{a \in A, Vo \in X: (o,a) \in R\}$$

Siendo X la extensión del concepto *c*, Ext[c] e Y la intensión de *c*, Int[c]. La Fig. III - 10 ejemplifica la noción de concept lattice.

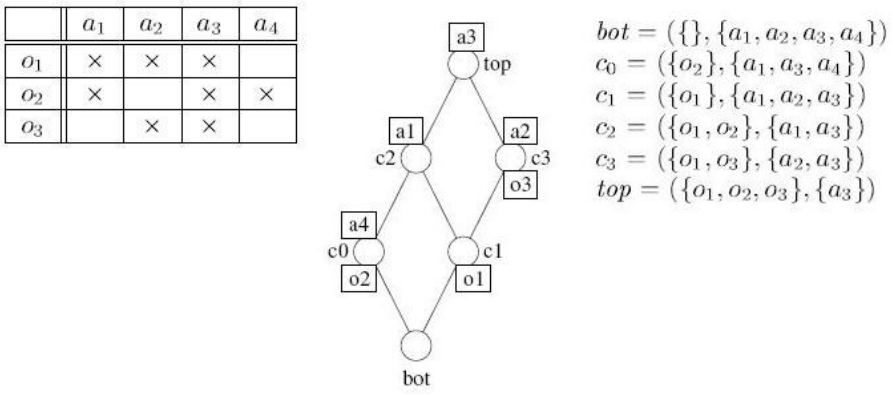

Fig. III - 10. Ejemplo de concept lattice.

9.2. Algoritmo

Los pasos que componen este enfoque se describen a continuación.

1. Obtener las trazas de ejecución generadas mediante distintas corridas del programa acorde a un conjunto de escenarios de ejecución.

2. Aplicar concept analysis a las relaciones entre las trazas de ejecución y las unidades computacionales (clases y métodos del sistema).

3. Identificar los casos de uso con los concepts lattices obtenidos en el paso anterior.

4. Derivar los aspectos candidatos a partir de los conceptos obtenidos que cumplan con las siguientes condiciones:

   4.1 Las unidades de compilación (métodos) de un concepto específico de un caso de uso pertenece a más de un módulo (clase).

   4.2 Diferentes unidades de compilación (métodos) de un mismo módulo (clase) etiquetan más de un caso de uso.

9.3. Ejemplo

Como ejemplo se plantea una búsqueda binaria en un árbol. La Fig. III - 11 muestra el diagrama de clases de la búsqueda. Se distinguen dos funcionalidades: inserción y búsqueda.

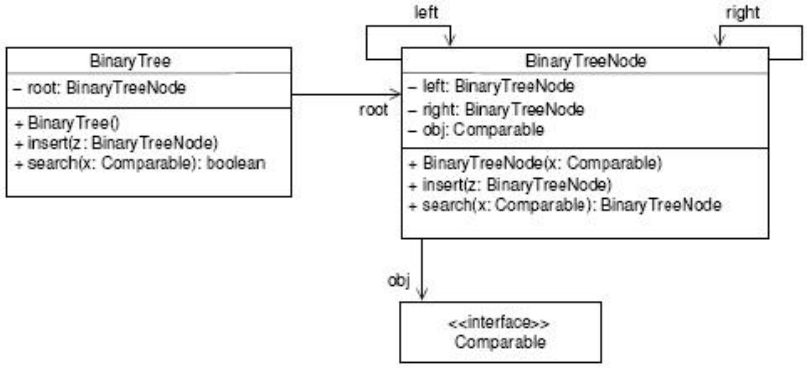

Fig. III - 11. Diagrama de clases de la aplicación de búsqueda binaria en un árbol.

La Tabla III - 4 muestra las trazas de ejecución para cada caso de uso.

	Inserción
m1	BinaryTree.BinaryTree()
m2	BinaryTree.insert(BinaryTreeNode)
m3	BinaryTreeNode.insert(BinaryTreeNode)
m4	BinaryTreeNode.BinaryTreeNode(Comparable)
	Search
m1	BinaryTree.BinaryTree()
m2	BinaryTree.search(Comparable)
m3	BinaryTreeNode.search(Comparable)

Tabla. III - 4. Relaciones entre casos de uso y métodos de ejecución.

La Fig. III - 12 permite visualizar los conceptos obtenidos luego de aplicar concept analysis a las relaciones de la tabla e indica que ambas funcionalidades son crosscutting concerns. El resultado puede ser interpretado por el hecho en que las dos clases no son cohesivas y realizan funciones múltiples y relativamente independientes.

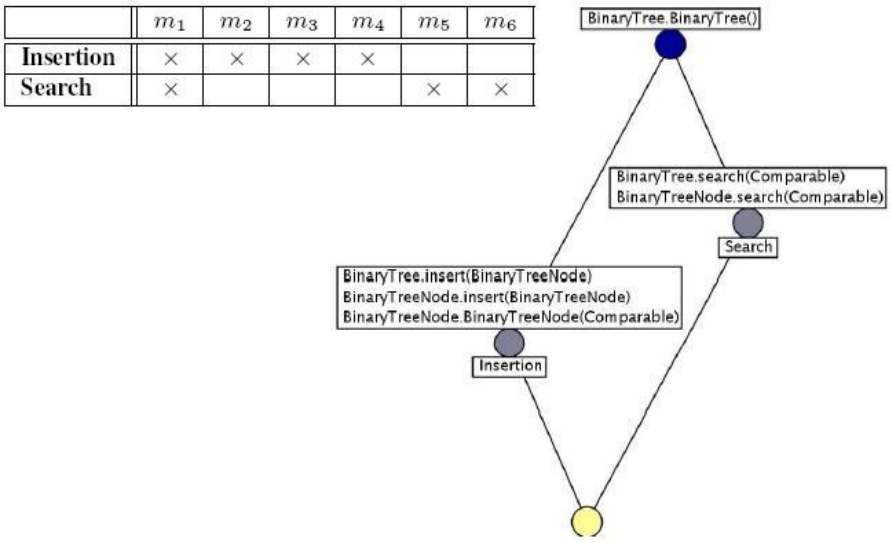

Fig. III - 12. Concept Lattice para la aplicación de búsqueda binaria en un árbol.

9.4. Herramienta

Los autores implementan la herramienta Dynamo [65] para aplicar la técnica descripta sobre aplicaciones Java.

10. Formal Concept Analysis Sobre los Identificadores del Código Fuente

Tourwé y Mens [66] proponen una técnica alternativa de aspect mining que depende de FCA. A diferencia de la técnica detallada en la sección 9, este enfoque realiza formal concept analysis sobre los identificadores de los métodos y las clases y se basa en la hipótesis de que los concerns en el código fuente se reflejan en el uso de convenciones de nombres tanto en las clases como en los métodos.

10.1. Algoritmo

El algoritmo FCA recibe como entrada tanto las clases y los métodos del sistema que son utilizadas como objetos dentro de FCA, como los identificadores de estos métodos y clases los cuáles representan las propiedades de esos objetos. A continuación, se describen los pasos para realizar el análisis:

1. Generar los objetos y los atributos.
2. Generar el lattice de conceptos.
3. Filtrar conceptos de menor importancia.
4. Analizar y clasificar los conceptos restantes.

En el paso uno se genera el espacio de búsqueda, donde los atributos están definidos por los substrings que poseen las entidades del programa utilizadas como objetos. Por ejemplo, una clase denominada *QuotedCodeConstant* se subdivide en las siguientes subcadenas: *Quoted, Code* y *Constant*.

Posteriormente, en el segundo paso, se aplica el algoritmo FCA para obtener el lattice de conceptos, el cuál agrupará aquellos elementos (métodos y clases) que compartan ciertas subcadenas según las restricciones impuestas por el algoritmo de FCA.

En el tercer paso se aplican filtros para reducir el espacio de soluciones, los cuáles utilizan heurísticas específicas a casos particulares. Sin embargo, también pueden utilizarse filtros generales, como por ejemplo la eliminación de conceptos que no contengan elementos o propiedades al inicio y o al final del mismo.

Finalmente, se analizan y clasifican los conceptos restantes de acuerdo a un conjunto de criterios predefinidos, por ejemplo, se podrían clasificar los conceptos acorde a las clases en que están definidos.

10.2. Herramienta

El enfoque propuesto es soportado por la herramienta DelfStof, la cuál permite analizar tanto código Java [67] como código Smalltalk [66].

11. Descubrimiento de Relaciones de Ejecución sobre el Grafo de Llamadas Estático

Krinke [68] describe una técnica en la cual se investiga el grafo de llamadas de un programa para descubrir patrones recurrentes de ejecución y así descubrir aspectos candidatos. La principal diferencia entre esta técnica y la técnica introducida en la Sección 8 [62] es el tipo de análisis de programa utilizado. Mientras que esta es una técnica estática, la anterior es una técnica basada en análisis dinámico. Si bien ambos tipos de análisis poseen sus pros y sus contras, algunos autores los consideran complementarios [68]. Por lo tanto, ambas técnicas deberían ser consideradas durante la migración hacia un sistema orientado a aspectos.

11.1. Representación de las Relaciones de Ejecución

Un grafo de control de flujo (CFG por sus siglas en inglés, Control Flow Graph), dirigido por atributos se define como G = (N, E, n^s, n^e) en donde N representa al conjunto de nodos y E al conjunto de arcos. Las sentencias son representadas por nodos $n \in N$, y el flujo de control entre nodos es representado por el par $(n, m) \in E$ y definido como $n \rightarrow m$. El conjunto E contiene al arco e, si y solo si la sentencia representada por el nodo terminal de e es ejecutado inmediatamente al nodo fuente, es decir que no se ejecuta ningún otro método entre el nodo fuente y el nodo terminal. Los símbolos distinguidos n^s y n^e denotan los nodos de comienzo y fin del programa.

Cada procedimiento o método $p \in P$ de un programa es representado con su propio grafo de control: Gp = (N_p, E_p, n^s_p, n^e_p), donde $\forall p, q: p \neq q \Rightarrow N_p \cap N_q = \phi \wedge E_p \cap E_q = \phi$ y $N^* = \cup_p N_p$, $E^* = \cup_p E_p$ representa el conjunto de nodos y arcos para todo el programa que está representado por el grafo G* = (N^*, E^*).

11.2. Algoritmo

Los pasos a seguir en el algoritmo son los siguientes:

1. Generar las relaciones de ejecución.
2. Aplicar filtros: restricción uniforme y crosscutting.
3. Ponderar métodos y relaciones.
4. Selección de aspectos candidatos.

Las relaciones que se pueden generar en las trazas de ejecución de un programa se clasifican en Outside-before-execution, Outside-after-execution, Inside-first-execution y Inside-last-execution. Estas relaciones son equivalentes a las definidas por el enfoque dinámico, expuesto anteriormente en la Sección 8 de este informe.

Una vez que las relaciones han sido generadas se aplican los filtros de uniformidad y crosscutting.

En el tercer paso se realiza una ponderación de las relaciones para obtener los aspectos candidatos. Por cada tipo de relación, éstas se ordenan de a pares (cantidad de relaciones, método) para obtener los métodos con mayor cantidad de relaciones del tipo en cuestión. Un método que presente gran número de relaciones indica que es parte de una funcionalidad crosscutting, ya que está presente en distintas zonas del código. La ponderación es independiente de cada tipo de relación.

Luego de ordenar cada tipo de relación, estas son inspeccionadas por el desarrollador para determinar si éstas pertenecen o no a un crosscutting concern.

11.3. Ejemplo

Los autores reportan la aplicación del enfoque sobre el sistema JHotDraw 5.4b1. La Tabla III - 5 muestra los resultados obtenidos para la relación Outside-before-execution luego de realizar los filtrados pertinentes. En ella se pueden ver la cantidad de métodos que tienen cierta cantidad de relaciones de este tipo. Por ejemplo, existen 53 métodos que tienen 2 relaciones Outside-before-relation.

Cantidad de relaciones	Cantidad de métodos
2	53
3	19
4	4
5	6
6	3
7	2
8	1
9	1

11	1
12	1
13	1
294	**92**

Tabla III - 5. Relaciones Outside-before-execution

Luego de obtener la tabla ordenada, donde los máximos candidatos se indican con la mayor cantidad de relaciones, se realiza un análisis de cada método. El candidato más probable consiste de 13 relaciones de ejecución donde el método involucrado es el "Iterator.next". Si se analiza el código con detenimiento, las 13 invocaciones revelan que el crosstutting es incidental, la operación es ejecutada luego de que se declaran ciertos contenedores. Por lo tanto, dicha relación se corresponde con un falso positivo. Sucede lo mismo con los próximos candidatos más probables (12, 11, 9 y 8). Recién el sexto máximo candidato corresponde a un crosscutting concern, el cual es identificado como el patrón observer/observable.

11.4. Herramienta

Los autores implementan el enfoque sobre el framework Soot [69], el cuál permite realizar este tipo de análisis sobre aplicaciones desarrolladas en Java.

12. Redirector Finder

Marin, Moonen y van Deursen [44] proponen una heurística para detectar clases en las que sus métodos redirijan sus llamadas consistentemente a métodos dedicados de otras clases. Como ejemplo se puede mencionar el patrón Decorator [50], el cual define una clase Decorator en la que sus métodos reciben llamadas, agregan funcionalidad opcionalmente, y luego redirigen estas llamadas a métodos específicos en la clase decorada.

12.1. Algoritmo

Para detectar este tipo de crosscutting concerns, se buscan las clases en las que sus métodos invoquen métodos específicos de otra clase. La regla de selección automática es la siguiente:

C.m[i] calls D.n[j] and only n[j] from D and *D.n[j] is called only by m[i] from C.*

Donde, *C* y *D* representan clases, y *m[i]* y *n[j]* los métodos de las clases respectivamente. La regla indica que el método *m[i]* de la clase *C* redirecciona al método *n[j]* de la clase *D* si *m[i]* llama únicamente al método *n[j]* de la clase *D* y a ningún otro método de *D*, y *n[j]* es llamado únicamente por el método *m[i]* de la clase *C*.

La clase *C* y sus métodos redireccionadores son reportados por la técnica si el número de los métodos que cumplen con esta condición está por encima de un cierto umbral elegido, o si el porcentaje de métodos redirectores en C con respecto al total de métodos de esta clase son mayores a un segundo valor de umbral.

12.2. Ejemplo

La técnica fue aplicada sobre el framework para edición de dibujo JHotDraw v5.4b1. Para el ejemplo, se utilizaron dos valores de umbral, el primero define la cantidad mínima de métodos redireccionadores que debe tener una clase y se le asignó el valor 3 en 3, y el segundo indica el porcentaje de métodos de una clase que deben ser redireccionadores, siendo este del 50%. En consecuencia, los candidatos reportados deben tener al menos 3 métodos redireccionadores, representando al menos un 50% de la cantidad total de los métodos de la clase.

Se obtuvieron resultados con los que se identificó el patrón Decorator en el código. Ejemplos de este patrón son las clases *Border* o *Animation-Decorator,* que provéen la funcionalidad básica para redirigir llamadas a la clase decorada *Figure.*

12.3. Herramienta

Los autores desarrollaron la herramienta FINT, la cual centra su funcionalidad en el análisis de Fan-in. No obstante, provee soporte para realizar la búsqueda de clases redireccionadoras, y da soporte para analizar los resultados obtenidos y seleccionar los aspectos candidatos a partir de los resultados.

13. Evaluación de las Técnicas Propuestas

Si bien los trabajos propuestos hasta el momento permiten, hasta cierto punto, identificar crosscutting concerns de manera semi-automática, sería interesante contar con una representación de la información que pueda ser analizada de diferentes maneras. En particular, la utilización de una representación lógica del código fuente de una aplicación permitiría el desarrollo de sistemas expertos que exploten las mejores características de las técnicas de aspect mining presentadas hasta el momento. Adicionalmente, este tipo de sistemas permitiría alcanzar la tan deseada automatización entre las técnicas de aspect mining y la sugerencia de los refactorings a aplicar con el objetivo de introducir aspectos en el sistema legado.

CAPÍTULO IV

Sistemas Expertos Basados en Reglas

1. Introducción

La Inteligencia Artificial (IA) es la parte de la Ciencia que se ocupa del diseño de sistemas de computación inteligentes, es decir, sistemas que exhiben las características que se asocian a la inteligencia en el comportamiento humano, esto es: la comprensión del lenguaje, el aprendizaje, el razonamiento, la resolución de problemas, etc [15]. Hoy en día, la inteligencia artificial abarca varias sub-áreas tales como los sistemas expertos, la demostración automática de teoremas, el juego automático, el reconocimiento de la voz y de patrones, el procesamiento del lenguaje natural, la visión artificial, la robótica, las redes neuronales, etc.

El presente capítulo tiene como objetivo introducir los conceptos referidos a sistemas expertos, y en particular sistemas expertos basados en reglas. Este último tipo de sistemas fue el seleccionado para implementar la propuesta de tesis que se detallará en el siguiente capítulo.

2. Sistemas Expertos

El diccionario de la Real Academia Española [16] define a este tipo de sistemas de la siguiente manera: “Sistema Experto: Programa de ordenador o computadora que tiene capacidad para dar respuestas semejantes a las que daría un experto en la materia.”

Los sistemas expertos surgen con la intención de emular el comportamiento de una persona experta en algún dominio. Los seres humanos llegan a convertirse en expertos mediante el estudio y la práctica continuada que realizan sobre determinada materia. El

conocimiento de un experto es una suerte de miscelánea entre los conceptos teóricos adquiridos y un conjunto de reglas heurísticas sobre un dominio, que la experiencia tanto propia como colectiva han establecido como válidas y efectivas [7].

Luego de un tiempo, una persona que en algún momento se la consideraba principiante en torno a un determinado dominio se transforma en un vasto conocedor del mismo, lo que suele llamarse, un experto en el tema. De la aplicación de este concepto a sistemas de software, surgen preguntas del estilo ¿Cómo es que un sistema llega a convertirse en un experto? ¿Estudia? ¿Practica?

A diferencia de los seres humanos, los sistemas expertos no se convierten en tales a partir del estudio y la practica constante. Los sistemas expertos no aprenden de su experiencia. El conocimiento humano de carácter experto sobre un dominio es traducido a un lenguaje formal con la finalidad de que las computadoras puedan hacer uso del mismo y sean capaces de generar respuestas a eventuales escenarios. Los sistemas expertos suelen estar focalizados en un reducido conjunto de problemas, es decir, su conocimiento es específico (al igual que los seres humanos, una persona no puede ser especialista en todas las materias). Generalmente, los sistemas expertos poseen las siguientes características [7]:

- Su estado mental, o base de conocimientos, es dinámico, es sencillo el agregado o substracción de conocimientos.
- Soportan y emulan el proceso de razonamiento humano mediante la manipulación de esta base de conocimientos.
- Razonan de forma heurística, usando su conocimiento para obtener la mejor solución al problema planteado.

2.1. Tipos de Sistema Experto

Los problemas con los que tratan los sistemas expertos pueden clasificarse en dos tipos: problemas esencialmente determinísticos y problemas esencialmente estocásticos

[24]. Los problemas determinísticos son aquellos que pueden definir sus salidas si las entradas están definidas, y los estocásticos son aquellos en los que la solución no viene dada únicamente por los datos de entrada, sino que presentan un grado de incertidumbre y varían de acuerdo a alguna variable. Consecuentemente, los sistemas expertos pueden clasificarse en dos tipos principales según la forma en que implementan el conocimiento. Cada uno apunta a resolver problemas de distinta naturaleza [12].

- **Sistemas expertos basados en reglas**: son formulados utilizando un conjunto de reglas que relacionan varios objetos bien definidos. Los sistemas expertos de este tipo sacan sus conclusiones basándose en un conjunto de reglas utilizando un mecanismo de razonamiento lógico e intentan resolver problemas determinísticos [7].

- **Sistemas expertos probabilísticos**: en situaciones inciertas, es necesario introducir medios para tratar la incertidumbre. Estos sistemas utilizan la probabilidad como medida de incertidumbre. La estrategia de razonamiento que usan se conoce como razonamiento probabilísticos, o inferencia probabilística, suelen implementarse mediante redes bayesianas e intentan resolver problemas estocásticos [12].

Otro tipo de sistemas expertos son los combinados con el razonamiento basado en casos (CBR, Case Based Reasoring), en donde se cuenta con el razonamiento del experto y experiencia en casos previos para resolver los problemas. Los casos se encuentran almacenados en la base de datos del sistema experto y son consultados para resolver los diferentes problemas. Estos sistemas pueden usarse para resolver problemas de ambos tipos (determinísticos y estocásticos) [18].

2.2. Aplicación de Sistemas Expertos

Los sistemas expertos pueden ser aplicados en diversas áreas donde se necesite emular el razonamiento humano. Una de las ventajas principales de estos sistemas es que

se pueden combinar conocimientos de varios expertos simulando el razonamiento conjunto de los mismos. Otra ventaja que presentan es que, estos sistemas, pueden utilizarse como fuente de conocimiento para realizar consultas del dominio sobre el cual se está resolviendo un problema.

El uso de sistemas expertos se recomienda especialmente en las siguientes situaciones [12]:

- Cuando el conocimiento es difícil de adquirir o se basa en reglas que sólo pueden ser aprendidas de la experiencia.
- Cuando la mejora continua del conocimiento es esencial y/o cuando el problema está sujeto a reglas o códigos cambiantes.
- Cuando los expertos humanos son caros o difíciles de encontrar.
- Cuando el conocimiento de los usuarios sobre el tema es limitado.

3. Sistema Basado en Reglas

Los programas basados en reglas surgen con la intención de ser usados para asistir en la resolución de problemas que con los métodos tradicionales de programación suelen ser difíciles de solucionar [7].

Es importante saber elegir el paradigma de programación adecuado para cada problema. Se puede dividir a los paradigmas en dos grandes grupos, aquellos en los que el programador codifica qué hacer y cómo hacerlo, y aquellos en los qué se codifica que hacer pero no se especifica de qué manera [7].

Dentro del primero grupo se encuentra la programación procedural y la programación orientada a objetos, entre otros. Aunque la estructura del programa difiera en los dos paradigmas, en ambos el programador codifica la lógica que controla a la

computadora: el programador le indica a la computadora qué hacer, de qué manera, y en qué orden. Este enfoque se adapta perfectamente cuando las entradas se encuentran bien especificadas y se conoce un conjunto de pasos para poder alcanzar la solución al problema. Los cálculos matemáticos, por ejemplo, son exitosamente resueltos por este estilo de programación.

En el segundo grupo, se encuentran los denominados programas basados en reglas, los cuales poseen una naturaleza declarativa. Un programa puramente declarativo especifica qué debe hacer la computadora, sin embargo no especifica **cómo** debe hacerlo. Esta característica los hace generalmente más sencillos de comprender que los programas procedurales. Un programa declarativo no está compuesto por una larga secuencia de instrucciones, por el contrario, este se materializa en un conjunto discreto de reglas las cuales describen subconjuntos del problema [7].

Los programas declarativos deben ser ejecutados por un sistema que sea capaz de manipular información declarativa para solucionar los problemas. Debido a que el flujo de ejecución es controlado por este sistema, este tipo de programas son más aptos que otros para alcanzar una solución válida cuando el conjunto de datos de entrada se presenta pobre o incompleto. La programación declarativa es usualmente la manera natural de atacar problemas que involucran control, diagnóstico, predicción, clasificación, reconocimiento de patrones y cualquier situación problemática que no tenga asociada una solución algorítmica clara.

En resumen, en un programa basado en reglas, se escriben las reglas que describen el problema. Luego, otro programa, denominado motor de reglas, determina qué regla se aplica en un determinado momento y la ejecuta de manera adecuada. Como resultado, una versión basada en reglas de un programa complejo, generalmente, será más corta y simple de entender que la versión procedural. Escribir el programa es sencillo, ya que el programador se puede concentrar en una situación a la vez y generar las reglas pertinentes.

3.1. Reglas y Motores de Reglas

Un sistema basado en reglas utiliza reglas para derivar conclusiones desde sus premisas.

Una regla es una especie de instrucción o comando que aplica en ciertas situaciones. En general, cualquier tipo de información que puede ser pensada en términos lógicos puede ser escrita en forma de regla. Las reglas se parecen mucho a las sentencias *if-then* de los lenguajes de programación tradicionales. La parte *if* de la regla, parte izquierda, es habitualmente conocida como predicados o premisas; y la parte *then*, parte derecha, como acciones o conclusiones. En ingles se las suele denominar como *LHS* (*Left-Hand Side*) a la parte izquierda y *RHS* (*Right-Hand Side*) a la parte derecha. Además, las reglas tienen asociado un dominio, el cuál es el conjunto de información con la que ésta puede trabajar.

Un sistema basado en reglas es aquel que utiliza un conjunto de reglas de inferencia para implementar el razonamiento de un experto. El conjunto de reglas que denotan el conocimiento son inyectadas dentro del motor de reglas, el cual tiene la capacidad de recordar, borrar o generar nuevas reglas.

Estos sistemas combinan la flexibilidad y eficiencia que provee un motor de reglas con la información experta obtenida sobre un dominio, pudiendo generar respuestas inmediatas y precisas a problemáticas particulares.

Algunos ejemplos de motores de inferencia utilizados para escribir sistemas expertos son: Dendral, XCon [22], Mycin [21], R1 [22], CLIPS [19] y Jess [4, 7], Prolog [20].

3.2. Arquitectura de un Sistema Basado en Reglas

En la Fig. IV - 1 Se presenta la arquitectura de un sistema basado en reglas. Se distinguen 4 componentes principales, los cuáles interactúan entre si para llevar a cabo la ejecución de un programa: el motor de inferencia (a su vez se subdivide en un analizador de

patrones y una agenda), la memoria de trabajo, las reglas base y el motor de ejecución [7]. A continuación, se realiza una breve descripción de cada componente:

- **Motor de inferencia (Inference Engine)**: Es el encargado de aplicar las reglas a la información. Controla el proceso por el cual se aplican las reglas a la información almacenada en la memoria de trabajo para obtener los resultados del problema (salida del sistema).

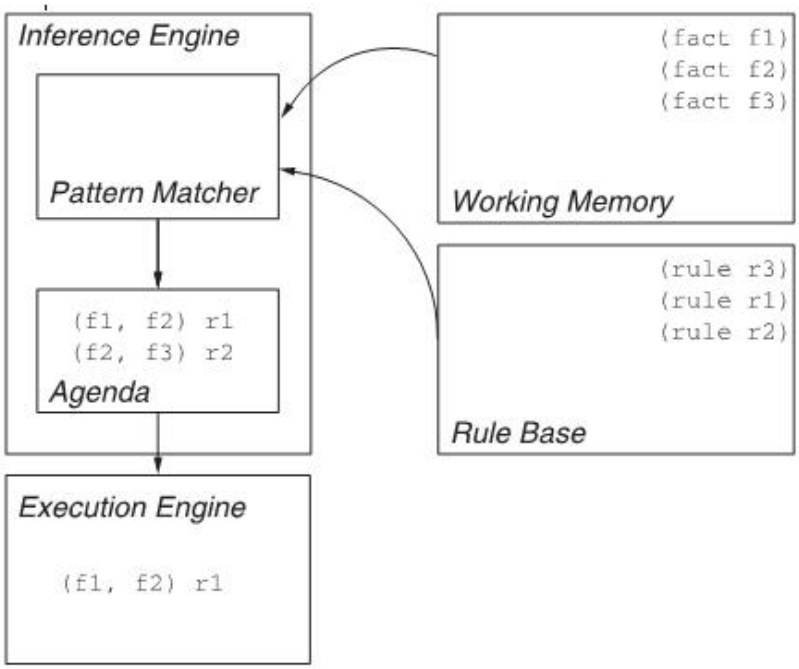

Fig. IV - 1. Arquitectura de un sistema basado en reglas.

- **Conjunto de reglas base (Rule Base)**: Contiene todas las reglas que el sistema conoce. Representa el conocimiento heurístico.

- **Memoria de trabajo (Working Memory)**: La memoria de trabajo almacena la información con la que el sistema basado en reglas trabaja. En esta es posible almacenar tanto las premisas como las conclusiones. Uno o más índices, similares a los utilizados en bases de datos, son mantenidos por el motor de reglas para hacer de la búsqueda en la memoria de trabajo una operación rápida.

- **Analizador de patrones (Pattern Matcher)**: El motor de inferencias debe decidir qué reglas disparar y en qué momento. Para eso hace uso del *pattern matcher*. El propósito de este último es decidir qué reglas ejecutar bajo un eventual estado de la memoria de trabajo.

- **Agenda**: Una vez que el motor de inferencia decide qué reglas debe disparar, tiene que decidir cuáles de estas ejecutar primero. La lista de las posibles reglas a ejecutar es almacenada en la agenda. Esta es responsable de usar una estrategia de resolución de conflictos para decidir qué regla tiene la prioridad más alta y debe ser ejecutada primero.

- **Motor de ejecución (Execution Engine):** El motor de ejecución es el componente del motor de inferencia que ejecuta la regla. En los sistemas de producción clásicos las reglas pueden agregar, eliminar y modificar hechos en la memoria de trabajo. En los motores de reglas modernos, la ejecución de una regla puede generar un amplio rango de efectos. Algunos de estos ofrecen un lenguaje de programación específico para determinar qué ocurre cuando una regla es ejecutada.

3.3. Desarrollo un Sistema Basado en Reglas

El proceso de desarrollo de un sistema basado en reglas, incluye las siguientes etapas [7].

1. **Ingeniería del conocimiento (Knowledge engineering)**: Es la primera etapa del proceso, en la que se recolecta la información mediante la cuál se derivan las reglas. Las personas encargadas de realizar esta tarea se las conoce como ingenieros de conocimiento (*knowledge engineer*).

2. **Estructuración de datos (Structuring data)**: Una vez que toda la información ha sido recolectada, las tareas concernientes a la programación comienzan.

Lo aconsejable es analizar la información y diseñar estructuras de datos convenientes, que ayuden a que la implementación de las reglas sea una labor clara y directa.

3. **Testing**: El testing en etapas tempranas de desarrollo ayuda a identificar errores que pueden resultar sumamente costosos si no son detectados a tiempo. Un sistema que es probado en cada etapa de su desarrollo será naturalmente más robusto y confiable que uno que no es probado hasta el final. Se recomienda, antes de escribir un conjunto de reglas, generar un script automático para probarlas. Es importante que los scripts de prueba sean automáticos, para que no requieran de mayor esfuerzo. Este ejercicio resulta en lo que se conoce como desarrollo dirigido por casos de prueba (*test-driven development*).

4. **Desarrollo de la interfaz (Interface building)**: Es importante tener en claro los límites del sistema. ¿Cómo interactuará el mismo con el mundo exterior, qué entradas necesitará para su procesamiento y cómo proveerá los resultados? Antes de comenzar con la codificación de las reglas, se aconseja realizar un diagrama de las conexiones entre el sistema y los componentes con los que se relaciona.

5. **Escribir las reglas (writing the rules)**: Una vez que las estructuras de datos están definidas, las interfaces especificadas, y los scripts de testeo existen, es momento de comenzar con la escritura de las reglas. Es aconsejable dividir las reglas en pequeños grupos. En general, un grupo de reglas es utilizado en un momento dado de la ejecución. Esta división permite generar un programa más sencillo de escribir y comprender.

6. **Desarrollo iterativo (Iterative development)**: Luego de desarrollar algunas reglas, es probable encontrarse en la situación que no se tiene la información

necesaria para continuar con el desarrollo. Cuando esto ocurre, es necesario volver a la fuente de información y reiniciar el proceso de desarrollo, atravesando nuevamente sus etapas. El desarrollo de un sistema basado en reglas tiende a amoldarse a este proceso de naturaleza iterativa.

4. Jess (Java Expert System Shell)

Jess es un motor de reglas para el lenguaje Java, el cual fue desarrollado en los laboratorios Sandia al final de la década del 90 por Ernest J. Friedman-Hill [4, 7].

En la utilización de Jess, el programador especifica la lógica del programa en forma de reglas usando uno de dos formatos: el lenguaje de reglas de Jess (recomendado y utilizado en este trabajo) o XML. A partir de allí, se deberán proveer datos de entrada para que las reglas puedan trabajar. Cuando el motor se pone en funcionamiento, las reglas son ejecutadas. Estas pueden crear nuevos datos, o pueden realizar cualquier operación que el lenguaje Java puede hacer, ya que es posible hacer un llamado a cualquier función Java desde Jess.

El motor de reglas de Jess usa una versión optimizada del algoritmo Rete [14] para asociar las reglas con la memoria de trabajo. Este algoritmo sacrifica espacio para ganar velocidad, es así que Jess puede llegar a usar una cantidad considerable memoria. No existen comandos que permitan reducir la performance a favor de disminuir el uso de memoria. No obstante, un programa considerable en magnitud se ajustará fácilmente en la pila de 64 megabytes que Java usa por defecto [7, 14].

Aunque Jess puede correr como una aplicación independiente, frecuentemente se importa la biblioteca desde el código Java y se manipula usando una API. Es versátil en cuanto a su uso, se pueden construir aplicaciones usando solamente el lenguaje de reglas de Jess, aplicaciones usando solamente las bibliotecas Jess desde Java, o una mezcla de ambas.

Es posible desarrollar código en lenguaje Jess en cualquier editor de texto, pero Jess provee un avanzado ambiente de desarrollo basado en la plataforma Eclipse [54]. La sintaxis de lenguaje Jess es simple y estándar. El lenguaje tiene pocos tipos de datos incorporados, entre ellos INTEGER, FLOAT, SYMBOL, STRING, y LONG. Además implementa estructuras de control simples, algunas de las cuales permiten transformar la información en código ejecutable. Todas las estructuras de control son, en realidad, funciones, Jess brinda casi 200 funciones predefinidas, y es posible definir nuevas usando el constructor "deffunction". También se permite modificar el comportamiento de las funciones predefinidas mediante el uso de "defadvice".

A continuación se listan los componentes básicos de este lenguaje y se provee una breve descripción de cada uno:

- ***deftemplate***: describe un tipo de hecho, a igual manera que una clase Java describe un tipo de objeto. Lista un conjunto de atributos, llamados *slots*, que el tipo de hecho puede contener. Se pueden realizar mapeos entre hechos Jess y clases Java mediante el uso conjunto de las sentencias *declare* y *from-class*. De esta manera, los *slots* del hecho estarán definidos por los atributos de la clase.

- ***defrule***: define una regla. Una regla está compuesta por una *LHS (Left-Hand Side)*, el símbolo "=>", y una *RHS (Right-Hand Side)*. En la parte izquierda - *LHS* – se encuentran uno o más elementos condicionales. La parte derecha - *RHS* - está compuesta por cero o más llamados a funciones. Los elementos condicionales son comprobados contra la memoria de trabajo de Jess. Si la comprobación es exitosa, el código en la parte derecha de la regla es ejecutado.

- ***deffunction***: define una función escrita en lenguaje Jess. Estas funciones pueden ser invocadas desde la línea de comandos, una regla, u otras funciones.

- ***defquery***: define una consulta, la cual consiste en una declaración opcional de una variable seguida por una lista de elementos condicionales. Las consultas (*queries*) son utilizadas para buscar hechos en la memoria de trabajo que satisfacen los elementos condicionales. Son de gran utilidad para obtener resultados una vez que las reglas que conforman la lógica del programa hayan sido ejecutadas.

5. Aspect Mining como un Problema de Decisión

Sería conveniente utilizar las ventajas que proveen los sistemas expertos para combinar la opinión de varios expertos simulando un razonamiento en conjunto. Para el caso particular de la identificación de crosscutting concerns en un sistema legado, las técnicas de aspect mining se modelan como expertos. De esta manera, es posible aprovechar las virtudes de los sistemas expertos junto con la automatización de dichas técnicas.

CAPÍTULO V

Aspect Mining mediante Sistemas Expertos

1. Introducción

Se ha comprobado que el mantenimiento de los programas orientados a aspectos resulta más sencillo y, por lo tanto, menos costoso que el de los programas orientados a objetos [58]. Por esta razón, es de suma importancia contar con la posibilidad de evolucionar de un paradigma a otro. Durante esta evolución, se pueden identificar, en principio, dos etapas bien definidas: la etapa de aspect mining y la de aspect refactoring. Aspect mining es la actividad de descubrir aquellos crosscutting concerns desde el código fuente o las trazas de ejecución de una aplicación que podrían ser encapsulados como aspectos del nuevo sistema. Aspect refactoring es la actividad de transformar los aspectos candidatos identificados en el código orientado a objetos en aspectos reales en el código fuente [6].

El presente trabajo de tesis tiene como objetivo crear una herramienta capaz de asistir en la primera etapa de este proceso. La propuesta radica en automatizar el proceso de identificación de crosscutting concerns sobre un código orientado a objetos mediante el uso de un sistema experto basado en reglas de inferencia. Este sistema experto soporta diversas técnicas de aspect mining, de forma tal de automatizar la ejecución y combinación de las mismas. La utilización de este tipo de sistema experto requiere la generación de una base de datos de hechos lógicos derivados desde el código fuente de la aplicación, a partir de los cuales se identifica los aspectos candidatos.

La obtención automatizada e inmediata de seeds candidatos mediante el uso de esta herramienta permite al programador ubicar los crosscutting concerns en el código legado

de forma más clara y precisa, agilizando la etapa de aspect mining, para luego determinar los posibles aspectos del sistema. Adicionalmente, se propone un enfoque en el cuál se combinan diferentes técnicas que permiten la obtención de seeds de una manera más certera.

La propuesta fue implementada como un plugin para la plataforma de desarrollo Eclipse [54]. A su vez, se utilizó el motor de inferencia Jess [4] para la implementación del sistema experto.

2. Sistema Experto para Aspect Mining

Para lograr una mayor automatización en el análisis e identificación de seeds candidatos, se definen algunos algoritmos de aspect mining como conocimiento dentro de un sistema experto. Dada la naturaleza determinística de la problemática a resolver, el sistema experto se basa en la utilización de reglas de inferencia. Los algoritmos se ejecutan sobre el motor de inferencia denominado Jess.

La Fig. V - 1 presenta el flujo del proceso de identificación de crosscutting concerns. Dada la aplicación que se desee analizar, su código fuente se debe transformar a una representación lógica de modo que pueda ser analizado desde el sistema experto. Para ello, se implementa un parser que deriva hechos lógicos a partir de una representación de árbol sintáctico del código (Parser AST). Una vez que se parsea el código, se crean los hechos que servirán como entrada al motor y se persisten en una base de datos. Posteriormente, se ejecutan las técnicas de aspect mining pertinentes y se obtienen los resultados para ser mostrados al usuario.

Se pueden distinguir los siguientes componentes y etapas del proceso:

- **Proyecto Java**: proyecto a analizar por la herramienta. Debe estar escrito en código Java, y se debe encontrar en el espacio de trabajo de Eclipse

(workspace). El mismo será analizado con el fin de obtener información de su estructura estática necesaria para el análisis.

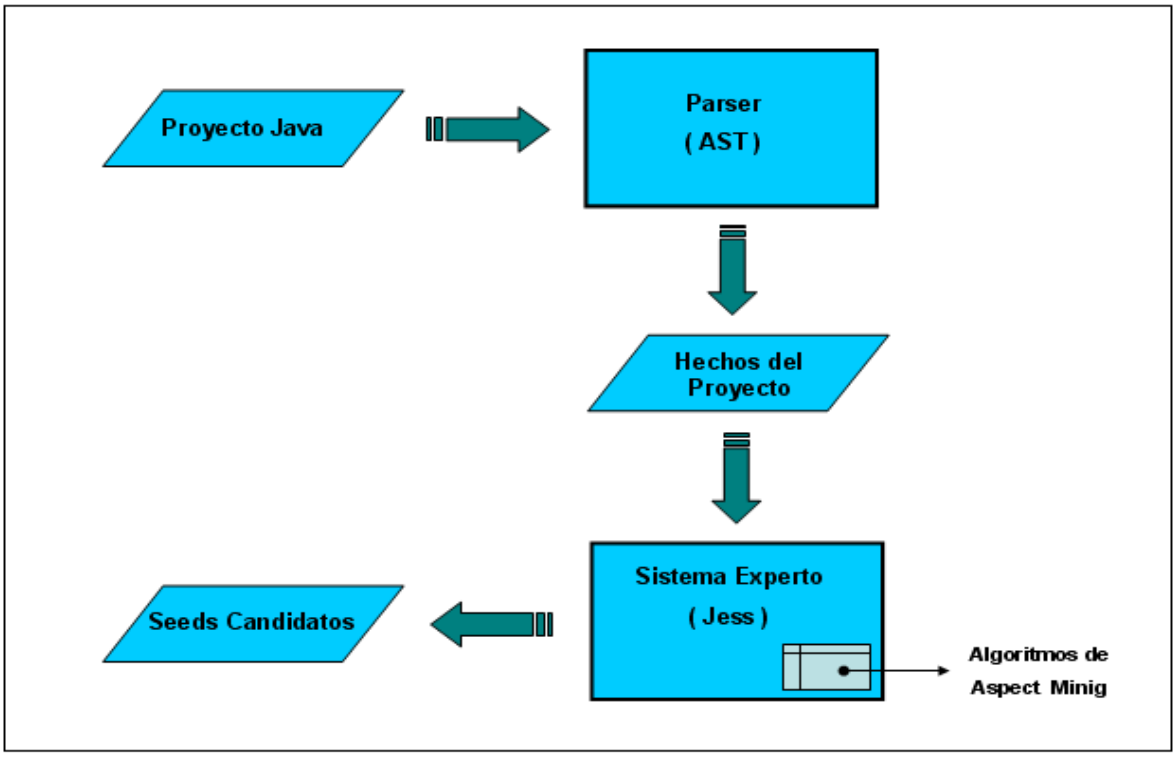

Fig. V - 1. Proceso de extracción de crosscutting concerns.

- **Parser (AST)**: componente encargado de obtener la información de la estructura de las clases del proyecto Java y traducirla a hechos lógicos.

- **Hechos del Proyecto**: salida obtenida del Parser. Estos hechos representan la estructura interna de cada clase del proyecto. Constituyen la entrada al motor de inferencia.

- **Sistema Experto (Jess) y Algoritmos de AM**: Jess es el motor de inferencia en donde se desarrollan los sistemas expertos. Los algoritmos de AM implementados corresponden a las siguientes técnicas: Análisis de Fan-in [58], Análisis de Métodos únicos [55], Análisis de Clases Redireccionadoras

[44], Análisis de Relaciones de Ejecución sobre el Grafo de Llamadas Estático [68] y Sinergia. Cada uno de ellos constituye un sistema experto en sí mismo.

- **Seeds Candidatos**: salida obtenida de la ejecución de los algoritmos de AM.

En las siguientes secciones de este informe se describirán con más detalle algunos componentes y etapas que participan en dicho proceso.

3. Parser (AST)

Los datos a manipular por un sistema experto deben presentarse en forma de hechos. Dado que el proyecto seleccionado para el análisis se encuentra especificado en código Java, se debe extraer del mismo toda la información referida al código y su representación. En consecuencia, se define un parser que permite obtener la información mencionada y convertirla en hechos.

El parser fue implementado utilizando la representación intermedia denominada AST (Abstract Sintax Tree o árbol sintáctico abstracto) [5]. Un AST es una representación en forma de árbol de la estructura sintáctica de un código fuente. Cada nodo de este árbol denota una construcción en el código fuente, el cuál contiene la información asociada a dicha construcción. Luego, se recorren estas construcciones con el fin de extraer su información.

En particular, en la herramienta implementada se utilizó el plugin JDT [9] de Eclipse con el fin de generar y consultar el árbol sintáctico abstracto del código. Este plugin es utilizado por diferentes herramientas provistas por Eclipse, como por ejemplo las funcionalidades de reemplazo en archivos y la actualización de dependencias. A continuación, se detalla el uso de JDT para generar la base de hechos lógicos a partir de un código dado.

En la estructura que provee AST, cada nodo del árbol es una subclase de ASTNode. Cada subclase de ASTNode representa un elemento del lenguaje de programación Java. Por ejemplo, se especifican nodos para las declaraciones de métodos (MethodDeclaration), declaraciones de variables (VariableDeclarationFragment), asignaciones, y demás.

La Fig. V – 3 presenta un ejemplo del árbol sintáctico obtenido del método de la Fig. V – 2. El nodo ASTNode se trata de una instancia del subtipo MethodDeclaration. En este objeto se puede encontrar toda la información referida al método, en forma de declaraciones de nombres simples o nuevos ASTNodes. Un ejemplo de nombres simples es el tipo de retorno del método ("void"), y un ejemplo de ASTNodes es el caso de los parámetros con el nodo SingleVariableDeclaration.

```
public void start(BundleContext context) throws Exception {
      super.start(context);
}
```

Fig. V - 2. Código de ejemplo.

La búsqueda de un elemento particular en el código no es una tarea sencilla debido a que el árbol resultante de un sistema puede llegar a ser muy complejo incluso para programas pequeños, como se puede apreciar en la Fig. V - 3. Por lo tanto, la búsqueda de un elemento no debería realizarse en todos los niveles del árbol, ya que dicha solución sería ineficiente. Se provee una alternativa para resolver este inconveniente: cada ASTNode puede ser accedido mediante el uso de un objeto visitor (patrón de diseño visitor [50]). Cada subclase de ASTNode contiene información específica sobre el elemento Java que representa. En el caso de la declaración de un método, MethodDeclaration, contendrá el nombre, el tipo de retorno, los parámetros, etc.

JDT define una clase llamada ASTVisitor que posee cuatro métodos: preVisit(), visit(), endvisit() y postVisit(). Estos métodos deben implementarse para acceder a los distintos tipos de nodos según la información que se desee obtener. La clase *ASTVisitor*

recorrerá recursivamente el árbol, invocando a los métodos previamente mencionados en el siguiente orden:

- preVisit(ASTNode node)
- visit(MethodInvocation node)
- ... los nodos hijos de la invocación al método son procesados recursivamente si visit() retorna true.
- endVisit(MethodInvocation node)
- postVisit(ASTNode node)

Fig. V - 3. AST para método ejemplo de Fig. V - 2.

La herramienta desarrollada define un visitor, FactsVisitor, que extiende de ASTVisitor y actúa de superclase para cualquier visitor que tenga como finalidad recorrer un conjunto de clases en busca de información para construir hechos lógicos. Como el lenguaje de reglas utilizado es Jess, los hechos deben ser generados para este lenguaje. Por lo tanto, se crea la clase JessFactsVisitor, que hereda de FactsVisitor e implementa la funcionalidad necesaria para extraer información de un nodo del árbol generado y plasmarlo en un hecho que respete la sintáxis de Jess. FactsVisitor permite extender a futuro la clase con el fin de generar hechos a cualquier otro lenguaje lógico.

Finalmente, el proceso de generación de la base de hechos lógicos para una aplicación dada es como sigue:

1. Identificar la unidad de compilación.
2. Generar el árbol sintáctico utilizando las clases del plugin de AST.
3. Visitar el árbol de salida del paso anterior en busca de la información para generar los hechos.

4. Hechos del Proyecto

La herramienta fue desarrollada utilizando un motor de inferencia sobre el cual se ejecutan los sistemas expertos de distintos enfoques de aspect mining. Las reglas definidas por cada sistema experto toman como entrada un conjunto de hechos lógicos y a partir de ellos se realizan los razonamientos.

Los 5 algoritmos implementados (Análisis de Fan-in, Análisis de Métodos Únicos, Análisis de Clases Redireccionadoras, Análisis de Relaciones de Ejecución y Sinergia) realizan un análisis sobre la información estática de un sistema. Debido a esto, es necesario contar con la representación en hechos lógicos de la información estructural de las clases que componen el proyecto.

La Tabla V -1 muestra los hechos definidos para representar la información estática de cada clase de un sistema. Estos hechos se obtienen luego de ejecutar el parser sobre el código de un sistema para ser utilizados como entrada para los algoritmos mencionados.

Hecho (Sintaxis Jess)	Atributos	Semántica del hecho
Call(caller_id, callee_id, precedence, id)	*caller_id*: identificador del método llamador. callee_id: identificador del método llamado. *precedence*: valor entero que indica en qué orden se ejecuta la misma en el método con respecto a las demás llamadas. *id*:identificador de la llamada de un método a otro.	El método identificado por *caller_id invoca al* método identificado por *callee_id* en el orden que indica el atributo *precedence.*
Class(id, name)	*id*: identificador de la clase. *name*: nombre de la clase.	Clase de nombre *name* y identificador *id.*
Abstract(id, name)	*id*: identificador de la clase abstract. *name*: nombre de la clase abstacta.	Clase abstracta de nombre *name* y identificador *id.*
Interface(id, name)	*id*: identificador de la interface. *name*: nombre de la interface.	Interface de nombre name e identificados id.
Implements(child_id, father_id)	*child_id*: identificador del elemento implementador. *father_id*: identificador del elemento implementado.	La clase identificada por el valor child_id implementa a la clase interface identificada por father_id.
Inherits(child_id, father_id)	*child_id*: identificador del tipo. *father_id*: identificador del subtipo.	El elemento (clase o interface) identificada por el child_id implementa a la clase identificada por father_id.
Method(id, name, returnType, class_id, param)	*id*: identificador del método. *returnType*: tipo de retorno. *class_id*: identificador de la clase. *param*: parámetros del método	El método de nombre *name* es identificado por *id*, pertenece a la clase identificada por *class_id* y tiene los parámetros *param.*

Tabla V - 1. Hechos derivados del parser.

Este conjunto de hechos fue definido en base a la mínima información requerida por las técnicas de aspect mining utilizadas. Se puede concluir que todos los algoritmos utilizados requieren, en mayor o menor medida, de información estructural similar del

proyecto, ya que basan su análisis en información estática de un sistema. Adicionalmente, cada algoritmo expande este conjunto con hechos que son propios a sus necesidades.

5. Sistema Experto y Algoritmos de Aspect Mining

Se implementaron cuatro algoritmos de aspect mining dentro del sistema experto: análisis mediante Fan-in, detección de Métodos Únicos, descubrimiento de Relaciones de Ejecución sobre el grafo de llamadas estático y análisis de Clases Redireccionadoras. De hecho, cada uno de ellos puede interpretarse como un experto en sí mismo. Adicionalmente, se presenta un quinto enfoque, el cual converge los 3 primeros.

Cada algoritmo necesita definir hechos con el objetivo de persistir los razonamientos intermedios y resultados obtenidos de la ejecución de las reglas. Estos hechos son propios de cada algoritmo.

Los hechos particulares para cada uno de estos enfoques, las reglas utilizadas para su implementación y un ejemplo de su uso se describen a continuación.

5.1. Análisis mediante Fan-in

Esta técnica calcula el valor de Fan-in de cada método del sistema con la suposición de que si un método tiene un alto valor de Fan-in, es probable que dicho método implemente un comportamiento crosscutting. En esta sección se explicará cómo se lleva a cabo el cálculo de la métrica propuesta en [Marin 2007] utilizando un sistema experto.

5.1.1. Hechos Particulares del Enfoque

En la Tabla V – 2, se puede observar el conjunto de hechos de mayor relevancia definidos por la técnica de Fan-in. Los mismos son utilizados por el algoritmo para persistir en la base de datos los razonamientos parciales y los resultados finales obtenidos a partir de las reglas. Por ejemplo, el hecho finalFan-inMetric se utiliza para presentar el valor de Fan-in total de cada método.

Hecho	Atributos	Semántica del hecho
familiar (elemento1) (elemento2)	*elemento1*: identificador de una clase o una interfaz. *elemento2*: identificador de una clase o de una interfaz.	La clase o interfaz identificada por elemento1 es familiar de la clase o interfaz identificada por el elemento2. Esta característica se cumple si el elemento1 hereda o implementa el elemento2.
metodoFamiliar(metodo1)(metodo2)	*metodo1*: identificador de un método. *metodo2*: identificador de un método.	El método identificado por metodo1 es familiar del método identificado por metodo2 si las clases que contienen a dichos métodos son familiares, y metodo1 y metodo2 identifican al mismo método reimplementado en la jerarquía.
llamadoNoDirecto(caller_id) (callee_id)	*caller_id*: identificador de un método. *callee_id*: identificador de un método	El método identificado por caller_id llama de manera no directa al método identificado por callee_id.
fan-in_metric(method_id, metric)	*method_id*: identificador de un método. *metric*: valor entero que representa la cantidad de invocaciones directas al método.	El método identificado por method_id tiene un valor de Fan-in correspondiente a los llamados directos indicado por la variable metric.
fan-in_metric_acum (method_id, metric)	*method_id*: identificador de un método. *metric*: valor entero que representa la cantidad de invocaciones no directas al método.	El método identificado por method_id tiene un valor de Fan-in correspondiente a los no llamados directos indicado por la variable metric.
finalFan-inMetric(method_id, metric)	*method_id*: identificador de un método. *metric*: valor entero que representa la cantidad de invocaciones al método.	El método identificado por method_id tiene una métrica final de Fan-in indicado por la variable metric.

Tabla V - 2. Hechos propios del algoritmo Fan-in.

5.1.2. Implementación del Algoritmo

Una de las consideraciones que presenta este algoritmo es la forma en que se calcula el Fan-in para los métodos polimórficos. Es por esto, que debe existir una manera de encontrar estos métodos polimórficos a partir de las relaciones de las clases del sistema. En el algoritmo implementado, se diferencian dos pasos para alcanzar esta información. El primero de ellos consiste en calcular las clases familiares, y el segundo consiste en calcular los métodos familiares.

```
(defrule assert_familiar_1
       (declare (salience 10000))
       (Inherits (child_id ?X) (father_id ?Y))
        =>
       (assert (familiar(clase1 ?X)(clase2 ?Y))))
(defrule assert_familiar_3
       (declare (salience 5000))
       (Inherits (child_id ?X) (father_id ?Y))
       (familiar (clase1 ?Y) (clase2 ?Z))
       =>
       (assert (familiar(clase1 ?X)(clase2 ?Z)))
)
```

Fig. V - 4. Reglas para el cálculo de elementos familiares.

- **Calcular clases familiares**: calcula las clases que se relacionan entre sí tanto en forma de herencia como en implementaciones de interfaces. La Fig. V - 4 muestra dos de las cuatro reglas utilizadas para este cálculo. La primera de ellas calcula los familiares directos, y la segunda busca los familiares con más de un nivel de relación. Las dos reglas omitidas son similares, variando el hecho *Inherits* por *Implements*.

- **Calcular métodos familiares**: calcula los métodos que son reimplementados por elementos familiares. Identifica al mismo método presente en una jerarquía de clases/interfaces. La Fig. V - 5 muestra las reglas que se utilizan para derivar está información, en donde un método es familiar de otro si, la clase a la que pertenece tiene una clase familiar que define el mismo método.

Una vez obtenidas las relaciones entre métodos se pueden calcular las llamadas a sus métodos polimórficos. Esto se debe a que el algoritmo especifica que si el *metodo1* llama al *metodo2,* el *metodo1* se agregará a la lista de potenciales llamadores del *metodo2,* así como a sus métodos familiares, estos últimos se denominan dentro del algoritmo como llamados no directos. Por ejemplo, en la Fig. V – 5 se muestra la regla utilizada para persistir en la base de datos los llamados no directos de cada método, en donde si un método m1 es llamado por otro método m2, se buscan los métodos familiares m_i de m1, y se agrega cómo llamado no directo m2, a m_i.

En este punto, la base de datos contiene los llamados directos a los métodos (representado por el hecho Call) y los llamados no directos (representado por el hecho llamadoNoDirecto). En consecuencia, es posible realizar el cálculo de Fan-in para cada método, mediante las siguientes reglas: contar las llamadas directas, contar las llamadas indirectas y calcular el Fan-in total.

```
(defrule metodos_familiares_padres
       (Method (id ?Method)(name ?MethodName)(class_id ?Class)(parametros ?p))
       (familiar (clase1 ?Class) (clase2 ?Familiar))
       (not (father_counted (clase1 ?Class) (clase2 ?Familiar)(metodo ?Method)))
       (Method (id ?FamiliarMethod) (name ?MethodName)
              (class_id ?Familiar)(parametros ?p))
       =>
       (assert (father_counted (clase1 ?Class) (clase2 ?Familiar)
       (metodo ?Method)))
       (assert (metodoFamiliar (metodo1 ?Method) (metodo2 ?FamiliarMethod))))
(defrule metodos_familiares_hijos
       (Method (id ?Method)(name ?MethodName)(class_id ?Class)(parametros ?p))
       (familiar (clase1 ?Familiar) (clase2 ?Class))
       (not (son_counted (clase1 ?Class) (clase2 ?Familiar)(metodo ?Method)))
       (Method (id ?FamiliarMethod) (name ?MethodName)
       (class_id ?Familiar)(parametros ?p))
       =>
       (assert (son_counted (clase1 ?Class) (clase2 ?Familiar)
                             (metodo ?Method)))
       (assert (metodoFamiliar (metodo1 ?Method) (metodo2 ?FamiliarMethod)))
)
```

Fig. V - 5. Reglas para el cálculo de métodos familiares.

```
(defrule propagarLlamadas
    (Call (callee_id ?metodoLlamado)(caller_id ?metodoLlamador))
    (metodoFamiliar (metodo1 ?metodoLlamado)(metodo2 ?metodoFamiliar))
    (not (metodoFamiliar_counted(metodo1 ?metodoLlamado)
         (metodo2 ?metodoFamiliar)))
      =>
    (assert (metodoFamiliar_counted(metodo1 ?metodoLlamado)
            (metodo2 ?metodoFamiliar)))
    (assert (llamado_no_directo(callee_id ?metodoFamiliar)
            (caller_id ?metodoLlamador)))
)
```

Fig. V - 6. Cálculo de llamados no directos de los métodos.

- **Contar llamadas directas:** esta regla (Fig. V - 7) recorre todas las llamadas de un método e incrementa el valor de la variable del hecho fan-in_metric asociado al método en cuestión si la llamada no ha sido previamente contada. Los valores de Fan-in son inicializadas en 0 al comienzo del algoritmo para todos los métodos. Para evitar contar más de una vez un llamado a un método se utiliza el hecho call_counted, de esta manera, el algoritmo solo aumenta el valor de Fan-in de un método si los métodos que lo invocan poseen cuerpos distintos, por lo que si se tiene dos llamados al metodo1 desde el metodo2 solo se cuenta una vez.

```
(defrule count_callers
    (Call (caller_id ?Caller) (callee_id ?Method))
    (not (call_counted (caller_id ?Caller) (callee_id ?Method)))
    ?OldFanInMetric <- (fan-in_metric (method_id ?Method)
                                      (metric ?Metric))
    =>
    (assert (call_counted (caller_id ?Caller)(callee_id ?Method)))
    (bind ?NewMetric (+ ?Metric 1))
    (modify ?OldFanInMetric (metric ?NewMetric)))
)
```

Fig. V - 7. Regla que calcula el valor de fan-in proveniente de las llamadas directas.

- **Contar llamadas no directas**: la Fig. V - 8 muestra la regla que calcula el valor de Fan-in para los llamados no directos. Esta regla es similar a la anterior, pero en vez de contar los llamados directos a un método, cuenta los no directos. Para esto utiliza el hecho *llamado_no_directo* previamente generado. Si este llamado no ha sido contado con anterioridad se incrementa

en 1 la métrica acumulada. Nótese que en esta regla se utiliza el hecho *fan-in_metric_acum* y no *fan-in_metric.*

```
(defrule count_Callers_noDirectos
    (llamado_no_directo(callee_id ?metodoLlamado)(caller_id ?metodoLlamador))
    (not (call_counted(callee_id ?metodoLlamado)(caller_id ?metodoLlamador)))
    ?OldFanInMetricAcum <- (fan-in_metric_acum (method_id ?metodoLlamado)
                                               (metric ?MetricAcum))
     =>
    (assert (call_counted (callee_id ?metodoLlamado)
                          (caller_id ?metodoLlamador)))
    (bind ?NewMetricAcum (+ ?MetricAcum 1))
    (modify ?OldFanInMetricAcum (metric ?NewMetricAcum))
)
```

Fig. V - 8. Regla que calcula el valor de fan-in proveniente de las llamadas no directas.

- **Cálculo final de Fan-in:** La regla de la Fig. V - 9 calcula el Fan-in final para cada método. Para esto hace uso de los hechos *fan-in_metric* y *fan-in_metric_acum* previamente almacenados. La suma entre ambos dará el valor de la métrica del hecho *final_fan-in_metric.*

```
(defrule final_fan-in
       (fan-in_metric(method_id ?Method) (metric ?OwnValue))
       (fan-in_metric_acum (method_id ?Method) (metric ?AcumValue))
    =>
       (bind ?NewValue (+ ?OwnValue ?AcumValue))
       (assert (final_fan-in_metric (method_id ?Method)
                                    (metric ?NewValue)))
)
```

Fig. V - 9. Regla que calcula el Fan-In total de un método.

5.1.3. Consulta de Resultados

La Fig. V - 10 muestra las dos consultas implementadas para obtener los resultados del algoritmo y mapearlos a objetos java a fin de manipularlos dentro de la aplicación. La primera de ellas es *fanInTotal*, la cual retorna todos los métodos con su respectivo valor de Fan-in, junto con la clase a la que pertenecen. La segunda consulta es utilizada para consultar cuáles métodos llaman a un método pasado por parámetro (Method).

```
(defquery fanInTotal
       (declare (variables ?ln))
       (final_fan-in_metric(method_id ?mi)(metric ?m))
       (Method (id ?mi) (class_id ?class))
)
(defquery llamados
       (declare (variables ?Method))
       (call_counted (caller_id ?Caller) (callee_id ?Method))
)
```

Fig. V - 10. Consultas utilizadas para obtener las salidas del algoritmo de Fan-in.

5.1.4. Ejemplo

A continuación, se presenta un ejemplo del cálculo de Fan-in; se muestran los hechos generados por el parser, y los hechos derivados del algoritmo.

La Fig. V - 11 presenta el diagrama de clases del ejemplo y la Tabla V - 3 muestra los llamados entre los métodos. Las filas representan el método llamador, y las columnas el método llamado.

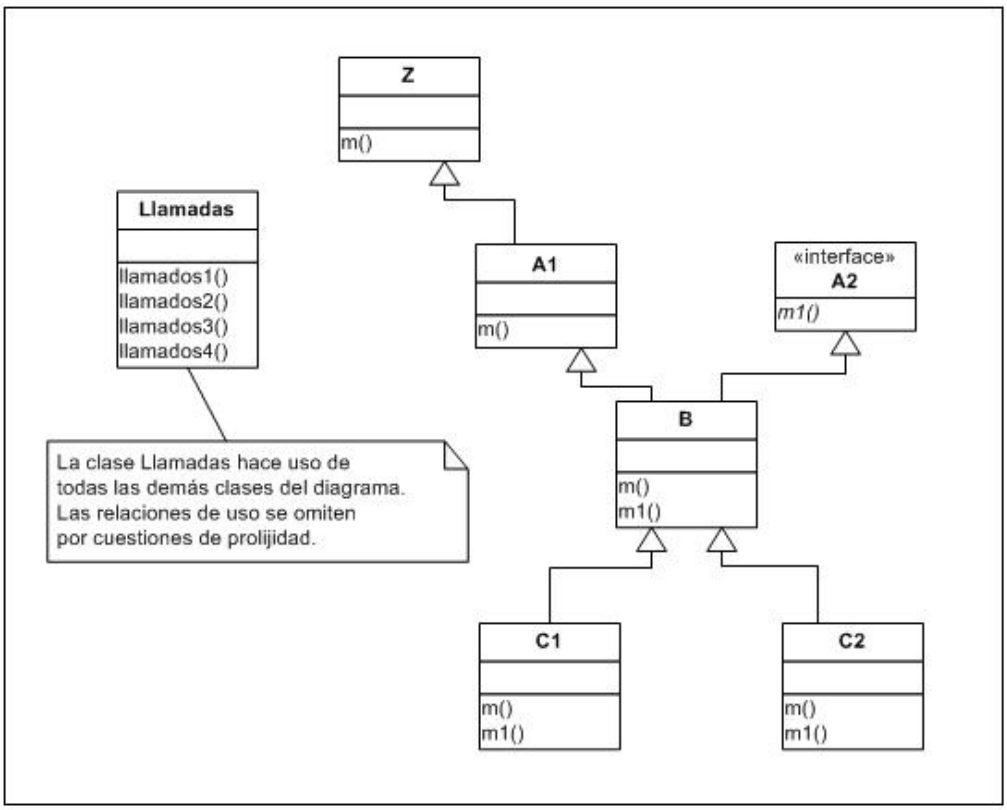

Fig. V - 11. Diagrama de clases para ejemplo de Fan-in.

	A1.m	B.m	B.m1	C1.m	C1.m1	C2.m	C2.m1
llamados1()	X						
llamados2()		X	X				
llamados3()				X	X		
llamados4()						X	X

Tabla V - 3. Llamados entre los métodos de las clases.

A continuación, se listan los hechos de entrada al sistema experto, los cálculos parciales provenientes del razonamiento experto y los resultados obtenidos.

5.1.4.1. Hechos de Entrada

El proyecto de Java que contiene las clases presentadas en el ejemplo (Fig. V - 11) será el sistema de entrada a la herramienta. Al ejecutar el parser sobre dicho código se obtienen los siguientes hechos que representan la estructura interna de cada clase:

Hechos derivados de la estructura de la clase Z:

```
Class  (id "classes/Z") (name "Z")
Method (class_id "classes/Z") (id "classes/Z//m///") (name "m") (parametros "")
       (returnType "java.lang.String")
```

Estos hechos indican la existencia de la clase Z, con id "classes/Z" y el método m perteneciente a la misma. Dicho método no posee parámetros y tiene un tipo de retorno String.

Hechos derivados de la estructura de la clase A1:

```
Class  (id "classes/A1") (name "A1")
Inherits (child_id "classes/A1") (father_id "classes/Z")
Method (class_id "classes/A1") (id "classes/A1//m///") (name "m") (parametros "")
       (returnType "java.lang.String")
```

Hechos derivados de la estructura de la interface A2:

```
Interface (id "classes/A2") (name "A2")
Method (class_id "classes/A2") (id "classes/A2//m1///") (name "m1") (parametros "")
```

```
      (returnType "void")
```

Hechos derivados de la estructura de la clase B:

```
Class (id "classes/B") (name "B")
Inherits (child_id "classes/B") (father_id "classes/A1")
Implements (child_id "classes/B") (father_id "classes/A2")
Method (class_id "classes/B") (id "classes/B//m1///") (name "m1") (parametros "")
      (returnType "void")
Method (class_id "classes/B") (id "classes/B//m///") (name "m") (parametros "")
      (returnType "java.lang.String")
```

Hechos derivados de la estructura de la clase C1:

```
Class (id "classes/C1") (name "C1")
Inherits (child_id "classes/C1") (father_id "classes/B")
Method (class_id "classes/C1") (id "classes/C1//m///") (name "m") (parametros "")
      (returnType "java.lang.String")
Method (class_id "classes/C1") (id "classes/C1//m1///") (name "m1") (parametros "")
      (returnType "void")
```

Hechos derivados de la estructura de la clase C2:

```
Class (id "classes/C2") (name "C2")
Inherits (child_id "classes/C2") (father_id "classes/B")
Method (class_id "classes/C2") (id "classes/C2//m///") (name "m") (parametros "")
      (returnType "java.lang.String")
Method (class_id "classes/C2") (id "classes/C2//m1///") (name "m1") (parametros "")
     (returnType "void")
```

Hechos derivados de la estructura de la clase Llamadas:

```
Class (id "llamadas/Llamadas") (name "llamadas")

Method (id "llamadas/Llamadas//llamados1///") (class_id "llamadas/Llamadas")
       (name "llamados1")(parametros "") (returnType "void")
Call (caller_id "llamadas/Llamadas//llamados1///") (callee_id "classes/A1//m///")
     (id "llamadas/Llamadas//llamados1///classes/A1//m///1") (precedence "1")

Method (id "llamadas/Llamadas//llamados2///") (class_id "llamadas/Llamadas")
       (name "llamados2")(parametros "") (returnType "void")
Call (caller_id "llamadas/Llamadas//llamados2///") (callee_id "classes/B//m///")
     (id "llamadas/Llamadas//llamados2///classes/B//m///1") (precedence "1")
Call (caller_id "llamadas/Llamadas//llamados2///") (callee_id "classes/B//m1///")
     (id "llamadas/Llamadas//llamados2///classes/B//m1///2") (precedence "2")

Method (id "llamadas/Llamadas//llamados3///") (class_id "llamadas/ Llamadas ")
       (name "llamados3")(parametros "")(returnType "void")
Call (caller_id "llamadas/Llamadas//llamados3///")(callee_id "classes/C1//m///")
     (id "llamadas/Llamadas//llamados3///classes/C1//m///1") (precedence "1")
Call (caller_id "llamadas/Llamadas//llamados3///")(callee_id "classes/C1//m1///")
     (id "llamadas/Llamadas//llamados3///classes/C1//m1///2") (precedence "2")
```

```
Method (id "llamadas/Llamadas//llamados4///") (class_id "llamadas/ Llamadas ")
      (name "llamados4")(parametros "") (returnType "void")
Call (caller_id "llamadas/Llamadas//llamados4///") (callee_id "classes/C2//m///")
      (id "llamadas/Llamadas//llamados4///classes/C2//m///1") (precedence "1")
Call (caller_id "llamadas/Llamadas//llamados4///") (callee_id "classes/C2//m1///")
      (id "llamadas/Llamadas//llamados4///classes/C2//m1///2") (precedence "2")
```

Los métodos de la clase Llamadas invocan a métodos del resto de las clases. Por esta razón, se generan los hechos Call que representan dichas llamadas. Por ejemplo, el hecho *"Call (caller_id "llamadas/Llamadas//llamados1///") (callee_id "classes/A1//m///") (id "llamadas/Llamadas//llamados1///classes/A1//m///1") (precedence "1")"* indica que el método llamados1 invoca en primer lugar al método m perteneciente a la clase A1.

5.1.4.2. Razonamiento del Sistema

Luego de aplicarse las reglas sobre los hechos de entrada, el algoritmo genera hechos intermedios. Los más importantes son *fan-in_metric*, *llamado_no_directo* y *fan-in_metric_acum*. A continuación, se muestran dichos hechos para cada método agrupados por la clase a la que pertenecen.

- **Clase Z**

El siguiente código muestra el Fan-in del método m perteneciente a la clase Z. El valor total de Fan-in es la suma del Fan-in propio del método, y del Fan-in acumulado proveniente de los llamados no directos del método. Como se puede ver en los hechos de entrada, no existe un llamado directo al método, por lo tanto su fan-in-metric es igual a cero. El caso es distinto para los llamados no directos. Dicho valor es igual a 4, a razón de que acumula los llamados al método polimórfico que implementan sus hijos (llamados a A1.m, B.m, C1.m y C2.m):

```
fan-in_metric (method_id "classes/Z//m///") (metric 0)

fan-in_metric_acum (method_id "classes/Z//m///") (metric 4)
llamado_no_directo (caller_id "llamadas/llamadas//llamados1///")
                   (callee_id "classes/Z//m///")
llamado_no_directo (caller_id "llamadas/llamadas//llamados2///")
                   (callee_id "classes/Z//m///")
llamado_no_directo (caller_id "llamadas/llamadas//llamados4///")
                   (callee_id "classes/Z//m///")
```

```
llamado_no_directo (caller_id "llamadas/llamadas//llamados3///")
                   (callee_id "classes/Z//m///”)

final_fan-in_metric (method_id "classes/Z//m///") (metric 4)
```

- **Clase A1**

Como se puede ver en el siguiente código, el valor total de Fan-in para el método m perteneciente a la clase A1 es igual a 4. Este valor es el resultado de la suma del Fan-in propio y acumulado del método. El método llamados1 de la clase Llamados invoca a m (hecho que se omite ya que pertenece a los datos de entrada previamente descriptos para el ejemplo), en consecuencia el valor de Fan-in propio para A1.m es igual a 1. El Fan-in acumulado del método es igual a 3, el cual representa los llamados al método m de sus clases familiares (B, C1 y C2).

```
fan-in_metric (method_id "classes/A1//m///") (metric 1)

fan-in_metric_acum (method_id "classes/A1//m///") (metric 3)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados2///")
                   (callee_id "classes/A1//m///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
                   (callee_id "classes/A1//m///”)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                   (callee_id "classes/A1//m///”)

final_fan-in_metric (method_id "classes/A1//m///") (metric 4)
```

- **Interface A2**

El valor total de Fan-in del método m1 de la interface A2 es igual a 3. Debido a que A2 es una interface, el valor de Fan-in de sus métodos proviene únicamente de los llamados no directos a cada uno de ellos, ya que sus métodos nunca se llamarán en forma directa. En este caso en particular, el método m1 de A2 acumula Fan-in por llamados que se realizan a los método B.m1(), C1.m1() y C2.m1(). Los métodos que llaman a estos últimos se contabilizan como llamadores de A2.m1() (hecho llamado_no_directo). A continuación se muestran estos hechos:

```
fan-in_metric (method_id "classes/A2//m1///") (metric 0)

fan-in_metric_acum (method_id "classes/A2//m1///") (metric 3)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
```

```
                    (callee_id "classes/A2//m1///”)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                    (callee_id "classes/A2//m1///”)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados2///")
                    (callee_id "classes/A2//m1///")

final_fan-in_metric (method_id " classes/A2//m1///") (metric 3)
```

- **Clase B**

El valor de Fan-in de los métodos de la clase B es el resultado de la sumatoria de la cantidad de llamados directos a cada uno (m y m1) y los llamados no directos provenientes de los llamados a sus familiares. Para el caso de m, el Fan-in acumulado proviene de los llamados a A1.m, C1.m y C2.m. Para el caso del método m1, los métodos que suman al valor acumulado son C1.m1 y C2.m1. A continuación se muestran los hechos correspondientes a los valores de Fan-in y los llamados no directos a los métodos B.m() y B.m1().

```
fan-in_metric (method_id "classes/B//m///") (metric 1)

fan-in_metric_acum (method_id "classes/B//m///") (metric 3)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
                    (callee_id "classes/B//m///”)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                    (callee_id "classes/B//m///”)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados1///")
                    (callee_id "classes/B//m///")

final_fan-in_metric (method_id "classes/B//m///") (metric 4)


fan-in_metric (method_id "classes/B//m1///") (metric 1)

fan-in_metric_acum (method_id "classes/B//m1///") (metric 2)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
                    (callee_id "classes/B//m1///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                    (callee_id "classes/B//m1///”)

final_fan-in_metric (method_id "classes/B//m1///") (metric 3)
```

- **Clase C1**

El siguiente código presenta los hechos correspondientes a los métodos de la clase C1. Estos métodos solo acumulan llamadores provenientes de la jerarquía de padres e hijos, no acumulan de sus hermanos (C2). Por lo tanto, el método m suma las invocaciones a A1.m

y B.m, y el método m1 de A2.m1, B.m1. Para este último caso, al ser A2 una interface, solo suma del método B.m1.

```
fan-in_metric (method_id "classes/C1//m///") (metric 1)

fan-in_metric_acum (method_id "classes/C1//m///") (metric 2)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados1///")
                   (callee_id "classes/C1//m///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados2///")
                   (callee_id "classes/C1//m///")

final_fan-in_metric (method_id "classes/C1//m///") (metric 3)


fan-in_metric (method_id "classes/C1//m1///") (metric 1)

fan-in_metric_acum (method_id "classes/C1//m1///") (metric 1)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados2///")
                    (callee_id "classes/C1//m1///")

final_fan-in_metric (method_id "classes/C1//m1///") (metric 2)
```

- **Clase C2**

Los siguientes hechos corresponden a los valores de Fan-in y llamados no directos de los métodos de la clase C2. El caso es semejante a C1.

```
fan-in_metric (method_id "classes/B//m///") (metric 1)

fan-in_metric_acum (method_id "classes/B//m///") (metric 3)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
                   (callee_id "classes/B//m///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                   (callee_id "classes/B//m///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados1///")
                   (callee_id "classes/B//m///")

final_fan-in_metric (method_id "classes/B//m///") (metric 4)


fan-in_metric (method_id "classes/B//m1///") (metric 1)

fan-in_metric_acum (method_id "classes/B//m1///") (metric 2)
llamado_no_directo (caller_id "llamadas/Llamadas//llamados4///")
                   (callee_id "classes/B//m1///")
llamado_no_directo (caller_id "llamadas/Llamadas//llamados3///")
                   (callee_id "classes/B//m1///")

final_fan-in_metric (method_id "classes/B//m1///") (metric 3)
```

- **Clase Llamadas**

Los valores de Fan-in (acumulado, propio y final) son igual a 0 para todos los métodos de la clase Llamados, ya que corresponde a la clase cliente, que tiene como función realizar llamados al resto de las clases:

```
fan-in_metric (method_id "llamadas/Llamadas//llamados1///") (metric 0)
fan-in_metric_acum (method_id "llamadas/Llamadas//llamados1///") (metric 0)
final_fan-in_metric (method_id "llamadas/Llamadas//llamados1///") (metric 0)

fan-in_metric (method_id "llamadas/Llamadas//llamados4///") (metric 0)
fan-in_metric_acum (method_id "llamadas/Llamadas//llamados4///") (metric 0)
final_fan-in_metric (method_id "llamadas/Llamadas//llamados4///") (metric 0)

fan-in_metric (method_id "llamadas/llamadas//llamados2///") (metric 0)
fan-in_metric_acum (method_id "llamadas/Llamadas//llamados2///") (metric 0)
final_fan-in_metric (method_id "llamadas/Llamadas//llamados2///") (metric 0)

fan-in_metric (method_id "llamadas/Llamadas//llamados3///") (metric 0)
fan-in_metric_acum (method_id "llamadas/Llamadas//llamados3///") (metric 0)
final_fan-in_metric (method_id "llamadas/Llamadas//llamados3///") (metric 0)
```

5.1.4.3. Salidas del Ejemplo

Una vez que se ejecutan todas las reglas, los resultados del algoritmo quedan persistidos en la base de datos. Para obtener la información se hace uso de las consultas *fanInTotal* y *llamados* explicadas previamente. La Tabla V – 4 resume el valor final de Fan-in por cada método, obtenido al ejecutar la consulta *fanInTotal* y los métodos que aportan a dicho valor de Fan-in, obtenidos a partir de la consulta *llamados.*

	Valor de Fan-in	Métodos Cliente
Z.m()	4	Llamados.llamados1() - Llamados.llamados2() - Llamados.llamados3() - Llamados.llamados4()
A1.m()	4	Llamados.llamados1() - Llamados.llamados2() - Llamados.llamados3() - Llamados.llamados4()
A2.m1()	3	Llamados.llamados2() - Llamados.llamados3() – Llamados.llamados4()
B.m()	4	Llamados.llamados1() - Llamados.llamados2() - Llamados.llamados3() - Llamados.llamados4()
B.m1()	3	Llamados.llamados2() - Llamados.llamados3() – Llamados.llamados4()

C1.m()	3	Llamados.llamados1() - Llamados.llamados2() - Llamados.llamados3()
C1.m1()	2	Llamados.llamados2() - Llamados.llamados3()
C2.m()	3	Llamados.llamados1() - Llamados.llamados2() - Llamados.llamados4()
C2.m1()	2	Llamados.llamados2() - Llamados.llamados4()

Tabla V - 4. Salidas para ejemplo de Fan-in.

5.2. Detección de Métodos Únicos

Como se describió en el Capítulo III del presente informe, el enfoque de métodos únicos se puede implementar a partir del algoritmo de Fan-in. Por esta razón, los hechos y las reglas son iguales a los de dicha técnica. La única diferencia reside en los datos que se consultan a la base de datos luego de la ejecución de las reglas.

5.2.1. Hechos Particulares del Enfoque

Los hechos de este enfoque son iguales a los explicados en la técnica de Fan-in.

5.2.2. Implementación del Algoritmo

Las reglas definidas por el algoritmo son iguales a las explicados en la técnica de Fan-in.

5.2.3. Consulta de Resultados

La Fig. V – 12 muestra la consulta utilizada para obtener los métodos únicos resultantes de la ejecución del algoritmo. Esta consulta devuelve los métodos y su valor de Fan-in asociado en los que el tipo de retorno es igual a void.

```
(defglobal ?*x* = "void")
(defquery UniqueMethods
       (declare (variables ?ln))
       (Method (id ?mi)(class_id ?class)(returnType ?*x*)(name ?name
                (parametros ?parametros))
       (final_fan-in_metric(method_id ?mi)(metric ?m)))
```

Fig. V - 12. Consulta de Unique Methods.

5.2.4. Ejemplo

Se realiza la ejecución del mismo ejemplo que se propuso para el algoritmo de Fan-in. A continuación se presenta el análisis para dicho ejemplo.

5.2.4.1. Hechos de Entrada

Los hechos de entrada son iguales a los hechos de entrada para el algoritmo de Fan-in.

5.2.4.2. Razonamiento del Sistema

Las reglas ejecutadas son las mismas que para el algoritmo de Fan-in, debido a esto los resultados y razonamientos parciales son iguales para ambos casos.

5.2.4.3. Salidas del Ejemplo

La Tabla V – 5 muestra los métodos resultaltantes con su respectivo valor de Fan-in obtenidos al ejecutar la consulta *UniqueMethods*. El resto de los métodos del ejemplo no serán devueltos como solución ya que no cumplen con la restricción de métodos únicos. Las llamadas a cada uno de ellos se obtienen al ejecutar la consulta *llamados* descripta para el enfoque de Fan-in.

	Valor de Fan-in	**Métodos Cliente**
A2.m1()	3	Llamados.llamados2() - Llamados.llamados3() – Llamados.llamados4()
B.m1()	3	Llamados.llamados2() - Llamados.llamados3() –

		Llamados.llamados4()
C1.m1()	2	Llamados.llamados2() - Llamados.llamados3()
C2.m1()	2	Llamados.llamados2() - Llamados.llamados4()

Tabla V - 5. Salidas para ejemplo de Métodos Únicos.

5.3. Descubrimiento de Relaciones de Ejecución

Este algoritmo identifica seeds candidatos a partir de un análisis del grafo de llamadas estático del sistema. El mismo define cuatro tipos de relaciones de ejecución: *Outside Before*, *Outside After*, *Inside First* e *Inside Last.* Estas relaciones pueden darse entre dos métodos si se cumple alguna de las siguientes condiciones:

- B->A es una relación de ejecución Outside Before si el método B es llamado antes que el método A.
- B<-A es una relación de ejecución Outside After si el método A es llamado antes que el método B.
- $G \in_{\tau} C$ es una relación de ejecución Inside First si el método G es el primero en ser invocado durante la ejecución del método C.
- $H \varepsilon_{\perp} C$ es una relación de ejecución Inside Last si el método H es el último en ser invocado durante la ejecución del método C.

Una vez que estas relaciones son identificadas, se puede obtener el tamaño de las relaciones en término de cantidad de métodos que participan en la misma. Luego, aquellas relaciones que posean gran cantidad de métodos constituirán el conjunto de seeds candidatos.

5.3.1. Hechos Particulares del Enfoque

En la Tabla V – 6, se puede observar el conjunto de hechos de mayor relevancia definidos por la técnica de relaciones de ejecución. Estos hechos son utilizados por el algoritmo para persistir datos de interés (información parcial y final de las relaciones de ejecución) por las reglas del sistema experto. Por ejemplo, los hechos InsideFirstExecution y OutsideAfterExecutionMetric indican una relación de ejecución del tipo inside first de un método y la cantidad de veces que un método participa en una relación del tipo outside after respectivamente.

Hecho	Atributos	Semántica del hecho
InsideFirstExecution(call_id,method_id)	*call_id*: identificador de llamada de un método a otro. *Method_id*:identificador de un método.	La llamada identificada por call_id es ejecutada antes que cualquier otra dentro del método indentificado por method_id.
InsideFirstExecutionMetric(method_id, metric)	*method_id*: identificador de un método. *metric*: número entero. Describe la cantidad de apariciones del método en este tipo de relación de ejecución.	El método identificado por method_id participa de tantas relaciones del tipo InsideFirstExecution como indica metric.
InsideLastExecution(call_id, method_id)	*call_id*: identificador de llamada de un método a otro. *method_id*: identificador de un método.	La llamada identificada por call_id es ejecutada después que cualquier otra dentro del método indentificado por method_id.
InsideLastExecutionMetric(method_id, metric)	*method_id*: identificador de un método. *metric*: número entero. Describe la cantidad de apariciones del método en este tipo de relación de ejecución.	El método identificado por method_id participa de tantas relaciones del tipo InsideLastExecutionMetric como indica metric.
OutsideAfterExecution(call_id, call_id2)	*call_id*: identificador de llamada de un método a otro. *call_id2*: identificador de llamada de un método a otro.	La llamada identificada por call_id es ejecutada inmediatamente después que la llamada identificada por call_id2.
OutsideAfterExecutionMetric(	*method_id*: identificador de un	El método identificado por

method_id, metric)	método. *metric*: número entero. Describe la cantidad de apariciones del método en este tipo de relación de ejecución.	method_id participa de tantas relaciones del tipo OutsideAfterExecutionMetric como indica metric.
OutsideBeforeExecution(call_id, call_id2)	*call_id*: identificador de llamada de un método a otro. *call_id2*: identificador de llamada de un método a otro.	La llamada identificada por call_id es ejecutada justo antes que la llamada identificada por call_id2. No hay llamadas entre estas.
OutsideBeforeExecutionMetric(method_id, metric)	*method_id*: identificador de un método. *metric*: número entero. Describe la cantidad de apariciones del método en este tipo de relación de ejecución.	El método identificado por method_id participa de tantas relaciones del tipo OutsideBeforeExecutionMetric como indica metric.

Tabla V - 6. Hechos propios del algoritmo Relaciones de Ejecución.

5.3.2. Implementación del Algoritmo

El primer paso en la implementación del algoritmo consiste en construir las relaciones de ejecución a partir de los hechos de entrada.

- **Cálculo de relaciones Outside Execution**: la Fig. V – 13 muestra la regla que genera dichas relaciones. Esta regla toma dos llamados (*Call*) realizados desde un mismo método (*method_X*) y compara sus precedencias. La distancia entre dos métodos se calcula mediante el atributo precedencia, el cual indica en qué orden se realizan los llamados a los métodos. Si la distancia entre dos llamadas es igual a 1, indica que no se ejecuta ningún otro método entre ellas y por lo tanto indica dos llamados contiguos. Si las llamadas seleccionadas por la regla cumplen con esta restricción, las relaciones de ejecución correspondientes entre estas llamadas, *OutsideBeforeExecution* y *OutsideAfterExecution,* son generadas.

```
(defrule generate_OutsideExecution_relations
    (Call (id ?call_1) (caller_id ?method_X) (callee_id ?method_Y)
          (precedence ?precedence_1))
    (Call (id ?call_2) (caller_id ?method_X) (callee_id ?method_Z)
          (precedence ?precedence_2))
    (not (OutsideBeforeExecution (call_id ?call_1) (call_id2 ?call_2)))
    =>
    (bind ?distance (- ?precedence_2 ?precedence_1))
    (if (= ?distance 1) then
        (bind ?relation_1 (new OutsideBeforeExecution ?call_1 ?call_2))
        (bind ?relation_2 (new OutsideAfterExecution ?call_2 ?call_1))
        (add ?relation_1)
        (add ?relation_2))
)
```

Fig. V - 13. Regla para obtener las relaciones OutsideExecution.

- **Cálculo de relaciones InsideFirstExecution**: la regla definida en la Fig. V - 14 toma una llamada (Call) y verifica si su precedencia es igual a 1. Esto último indica la presencia de una llamada que es ejecutada antes que cualquier otra llamada en ese método (method_X). Por lo tanto, la relación de ejecución correspondiente, InsideFirstExecution, es generada.

- **Cálculo de relaciones InsideLastExecution:** la regla mostrada en la Fig. V - 15 toma una llamada (*Call*) realizada desde un método (*method_X*) y verifica si su precedencia es mayor que la precedencia de la llamada establecida hasta el momento como la última del método *method_X*. Si esto se cumple se actualiza la relación *InsideLastExecution* asociada al método *method_X,* ya que fue encontrada una llamada desde ese método con mayor precedencia.

```
(defrule generate_InsideFirstExecution_relations
    (Call (id ?call) (caller_id ?method_X) (callee_id ?method_Y)
          (precedence ?precedence))
    (not (InsideFirstExecution (call_id ?call_1) (method_id ?method_X)))
    =>
    (if (= ?precedence 1) then
        (bind ?relation (new InsideFirstExecution ?call ?method_X))
        (add ?relation)
    )
)
```

Fig. V - 14. Regla para obtener las relaciones InsideFirstExecution.

```
(defrule generate_InsideLastExecution_relations
    (Call (id ?call_1) (caller_id ?method_X) (callee_id ?method_Y)
          (precedence ?precedence_1))
    ?actualRelation <- (InsideLastExecution (call_id ?call_2)
                                            (method_id ?method_X))
    (Call (id ?call_2) (caller_id ?method_X) (callee_id ?method_Z)
          (precedence ?precedence_2))
      =>
    (if (> ?precedence_1 ?precedence_2) then
      (retract ?actualRelation)
       (bind ?newRelation (new InsideLastExecution ?call_1 ?method_X))
      (add ?newRelation)
    )
)
```

Fig. V - 15. Regla para obtener las relaciones InsideLastExecution.

Una vez que se persistieron en la base de datos las relaciones de ejecución, se puede calcular para cada tipo de relación, la cantidad de métodos que estas poseen. Mediante esta métrica se puede hacer un ranking de los seeds para esta técnica.

- **Cálculo de métricas de Outside Before Execution:** la regla de la Fig. V - 16 calcula la métrica para las relaciones de tipo *OutsideBeforeExecution*. El hecho *OutsideBeforeExecutionMetric* es generado para todos los métodos que participan en relaciones de este tipo. Cuando la regla encuentra una relación que no fue contada la métrica es sumada en una unidad. Para controlar que dos relaciones no sean contabilizadas en más de una ocasión se utiliza el hecho *OutsideBeforeRelationCounted*.

```
(defrule generate_OutsideBeforeExecution_Metric
    (OutsideBeforeExecution (call_id ?call_1) (call_id2 ?call_2))
    (not (OutsideBeforeRelationCounted (call_1 ?call_1) (call_2 ?call_2)))
    (Call (id ?call_1)(callee_id ?method))
    ?oldMetric <- (OutsideBeforeExecutionMetric (method ?method)(metric ?metric))
      =>
    (assert (OutsideBeforeRelationCounted (call_1 ?call_1) (call_2 ?call_2)) )
    (bind ?newMetric (+ ?metric 1))
    (modify ?oldMetric (metric ?newMetric))
)
```

Fig. V - 16. Regla que calcula la métrica para las relaciones OutsideBeforeExecution.

- **Cálculo de métrica para relaciones Outside After Execution**: la regla de la Fig. V - 17 calcula la cantidad de métodos presentes en este tipo de relación.

Para esto, cada vez que la regla encuentra una relación que no fue contada, el valor de la métrica *OutsideAfterExecutionMetric* aumenta. El hecho *OutsideAfterRelationCounted* es utilizado para no contar una relación más de una vez.

```
(defrule generate_OutsideAfterExecution_Metric
    (OutsideAfterExecution (call_id ?call_1) (call_id2 ?call_2))
    (not (OutsideAfterRelationCounted (call_1 ?call_1) (call_2 ?call_2)))
    (Call (id ?call_1)(callee_id ?method))
    ?oldMetric <- (OutsideAfterExecutionMetric (method ?method)(metric ?metric))
       =>
    (assert (OutsideAfterRelationCounted (call_1 ?call_1) (call_2 ?call_2)) )
    (bind ?newMetric (+ ?metric 1))
    (modify ?oldMetric (metric ?newMetric))
)
```

Fig. V - 17. Regla que calcula la métrica para las relaciones Outside After Execution.

- **Cálculo de métrica para las relaciones Outside Before Execution:** la regla que se presenta en la Fig. V – 18 es utilizada para obtener la cantidad de métodos en las relaciones de tipo *OutsideBeforeExecution*. Para esto, cada vez que la regla encuentra una relación que no fue contada, el valor de la métrica *OutsideBeforeExecutionMetric* aumenta. El hecho *OutsideBeforeRelationCounted* es utilizado para no contar una relación más de una vez.

```
(defrule generate_OutsideBeforeExecution_Metric
    (OutsideBeforeExecution (call_id ?call_1) (call_id2 ?call_2))
    (not (OutsideBeforeRelationCounted (call_1 ?call_1) (call_2 ?call_2)))
    (Call (id ?call_1)(callee_id ?method))
    ?oldMetric <- (OutsideBeforeExecutionMetric (method ?method)(metric ?metric))
       =>
    (assert (OutsideBeforeRelationCounted (call_1 ?call_1) (call_2 ?call_2)) )
    (bind ?newMetric (+ ?metric 1))
    (modify ?oldMetric (metric ?newMetric))
)
```

Fig. V - 18. Regla que calcula la métrica para las relaciones Outside Before Execution.

- **Cálculo de métrica para relaciones Inside First Execution:** la regla de la Fig. V - 19 calcula la cantidad de métodos presentes en este tipo de relación. Para esto, cada vez que la regla encuentra una relación que no fue contada, el

valor de la métrica *InsideFirstExecutionMetric* aumenta. El hecho *InsideFirstRelationCounted* es utilizado para no contar una relación más de una vez.

```
(defrule generate_InsideFirstExecution_Metric
    (InsideFirstExecution (call_id ?call) (method_id ?method))
    (not (InsideFirstRelationCounted (call_id ?call) (method_id ?method)))
    (Call (id ?call)(callee_id ?method2))
    ?oldMetric <- (InsideFirstExecutionMetric (method ?method2)(metric ?metric))
       =>
    (assert (InsideFirstRelationCounted (call_id ?call) (method_id ?method)) )
    (bind ?newMetric (+ ?metric 1))
    (modify ?oldMetric (metric ?newMetric))
)
```

Fig. V - 19. Regla que calcula la métrica para las relaciones Inside First Execution.

- **Cálculo de métrica para relaciones Inside Last Execution**: la regla de la Fig. V - 20 calcula la cantidad de métodos presentes en este tipo de relación. Para esto, cada vez que la regla encuentra una relación que no fue contada, el valor de la métrica *InsideLastExecutionMetric* aumenta. El hecho *InsideLastRelationCounted* es utilizado para no contar una relación más de una vez.

```
(defrule generate_InsideLastExecution_Metric
    (InsideLastExecution (call_id ?call) (method_id ?method))
    (not (InsideLastRelationCounted (call_id ?call) (method_id ?method)))
    (Call (id ?call)(callee_id ?method2))
    ?oldMetric <- (InsideLastExecutionMetric (method ?method2) (metric ?metric))
       =>
    (assert (InsideLastRelationCounted (call_id ?call) (method_id ?method)) )
    (bind ?newMetric (+ ?metric 1))
    (modify ?oldMetric (metric ?newMetric)))
```

Fig. V - 20. Regla que calcula la métrica para las relaciones Inside Last Execution.

5.3.3. Consulta de Resultados

La Fig. V – 21 muestra las consultas implementadas para obtener los resultados del algoritmo y mapearlos a objetos java para manipularlos dentro de la aplicación. Todas estas consultas retornan un par método-valor, como por ejemplo OutsideBeforeExecutionMetric (method "m1") (metric "2") para la relación de ejecución Outiside Before del método m1.

- **get_OutsideBeforeExecution_Value**: devuelve el par método – valor para las relaciones de tipo Outside Before Execution.

- **get_OutsideAfterExecution_Value**: devuelve el par método – valor para las relaciones de tipo Outside After Execution.

- **get_InsideFirstExecution_Value**: devuelve el par método – valor para las relaciones de tipo Inside First Execution.

- **get_InsideLastExecution_Value***:* devuelve el par método – valor para las relaciones de tipo Inside Last Execution.

```
(defquery get_OutsideBeforeExecution_Value
    (Method (id ?method))
    (OutsideBeforeExecutionMetric (method ?method)(metric ?metric)))
(defquery get_OutsideAfterExecution_Value
    (Method (id ?method))
    (OutsideAfterExecutionMetric (method ?method)(metric ?metric)))
(defquery get_InsideFirstExecution_Value
    (Method (id ?method))
    (InsideFirstExecutionMetric (method ?method)(metric ?metric)))
(defquery get_InsideLastExecution_Value
    (Method (id ?method))
    (InsideLastExecutionMetric (method ?method)(metric ?metric)))
```

Fig. V - 21. Consultas utilizadas para obtener los resultados del análisis Execution Relations.

5.3.4. Ejemplo

A continuación se presenta un ejemplo del enfoque Execution Relations. Se muestran los hechos generados por el parser, y los hechos derivados del algoritmo.

La Fig. V - 22 muestra el diagrama de clases del ejemplo y la Tabla V - 7 muestra los llamados realizados hacia los métodos. Las filas representan el método llamador, y las columnas el método llamado. El número en la celda indica que el método de la fila llama al método en la columna con ese nivel de precedencia. Es decir, si en la celda se halla un 1, la llamada al método es realizada antes que cualquier otra llamada, si se halla un 2 esta es realizada luego de otra llamada, y así sucesivamente.

	A.a1()	A.a2()	B.b1()	B.b2()	C.c1()	C.c2()	C.c3()
A.a1()			1	3		2	
A.a2()			1		4	2	3

Tabla V - 7. Llamadas entre métodos del ejemplo para Relaciones de Ejecución.

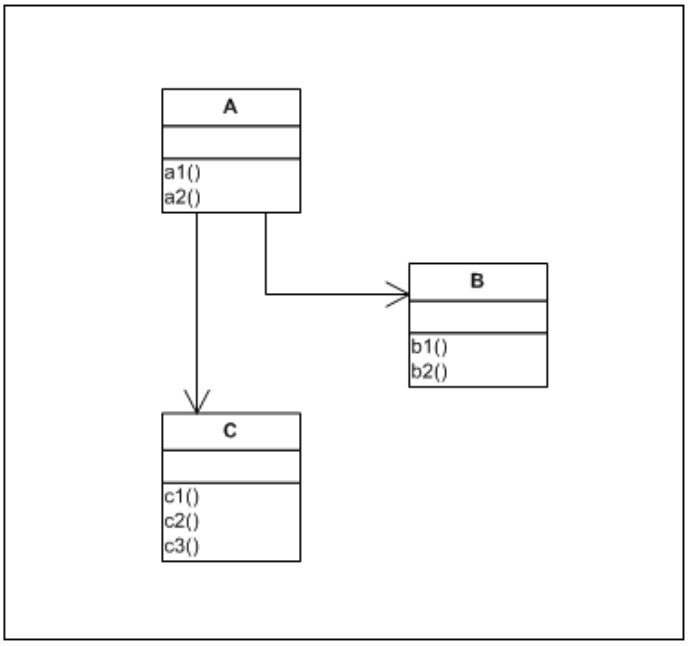

Fig. V - 22. Diagrama de clases para el ejemplo de Execution Relations.

5.3.4.1. Hechos de Entrada

A continuación, se listan los hechos de entrada por cada clase del ejemplo. Por cada una de estas se muestra el hecho que representa a la misma, los métodos que contiene, y por cada método los llamados que realiza:

Hechos derivados de la estructura de la clase A:

```
Class (id "classes/A")(name "A")

Method (id "classes/A//a1///")(class_id "classes/A")(name "a1")(parametros "")
       (returnType "void")
Call (caller_id "classes/A//a1///")(callee_id "classes/B//b1///")
     (id "classes/A//a1///classes/B//b1///1") (precedence "1")
Call (caller_id "classes/A//a1///")(callee_id "classes/C//c2///")
```

```
     (id "classes/A//a1///classes/C//c2///2") (precedence "2")
Call (caller_id "classes/A//a1///")(callee_id "classes/B//b2///")
     (id "classes/A//a1///classes/B//b2///3") (precedence "3")

Method (id "classes/A//a2///")(class_id "classes/A")(name "a2")(parametros "")
       (returnType "void")
Call (caller_id "classes/A//a2///")(callee_id "classes/B//b1///")
     (id "classes/A//a2///classes/B//b1///1") (precedence "1")
Call (caller_id "classes/A//a2///")(callee_id "classes/C//c2///")
     (id "classes/A//a2///classes/C//c2///2") (precedence "2")
Call (caller_id "classes/A//a2///")(callee_id "classes/C//c3///")
     (id "classes/A//a2///classes/C//c3///3") (precedence "3")
Call (caller_id "classes/A//a2///")(callee_id "classes/C//c1///")
     (id "classes/A//a2///classes/C//c1///4") (precedence "4")
```

El primero de estos hechos indica que existe una clase identificada por el id “classes/A”. El siguiente hecho que esta clase contiene un método identificado por el id “classes/A//a1///”. Los tres hechos siguientes representan llamadas desde este método a otros métodos de otras clases. A su vez, en los hechos restantes, es posible observar la existencia de otro método perteneciente a esta clase, identificado por el id “classes/A//a2///”, y las llamadas que realiza este método.

Hechos derivados de la estructura de la clase B:

```
Class (id "classes/B") (name "B")
Method (id "classes/B//b1///") (class_id "classes/B") (name "b1") (parametros "")
       (returnType "void")
Method (id "classes/B//b2///") (class_id "classes/B") (name "b1") (parametros "")
       (returnType "void")
```

Hechos derivados de la estructura de la clase C:

```
Class (id "classes/C") (name "C")
Method (id "classes/C//c1///")(class_id "classes/C")(name "c1")(parametros "")
       (returnType "void")
Method (id "classes/C//c2///")(class_id "classes/C")(name "c2")(parametros "")
       (returnType "void")
Method (id "classes/C//c3///")(class_id "classes/C")(name "c3")(parametros "")
       (returnType "void")
```

5.3.4.2. Razonamiento del Sistema

Una vez que los hechos fueron introducidos a la base de datos, se ejecuta el algoritmo Relaciones de Ejecución. A continuación, se muestran los hechos de salida

obtenidos. Es posible observar los tipos de relaciones de ejecución encontrados, y luego los tipos de métricas para cada clase.

- **Outside Before Execution Relations**

A continuación se presentan las relaciones de ejecución Outside Before, junto con el valor de la métrica asociada a cada método. Con el hecho OutsideBeforeExecution se indica que la llamada al método asociado al id call_id se ejecuta inmediatamente antes que la llamada al método call_id2. El hecho OutsideBeforeExecutionMetric acumula las relaciones de ejecución por método, por ejemplo el método B.b1() presenta dos relaciones de este tipo.

```
OutsideBeforeExecutionMetric (method "classes/B//b1///") (metric "2")
OutsideBeforeExecution (call_id "classes/A//a1///classes/B//b1///1")
                       (call_id2 "classes/A//a1///classes/C//c2///2")
OutsideBeforeExecution (call_id "classes/A//a2///classes/B//b1///1")
                       (call_id2 "classes/A//a2///classes/C//c2///2")

OutsideBeforeExecutionMetric (method "classes/C//c3///") (metric "1")
OutsideBeforeExecution (call_id "classes/A//a2///classes/C//c3///3")
                       (call_id2 "classes/A//a2///classes/C//c1///4")

OutsideBeforeExecutionMetric (method "classes/C//c2///") (metric "2")
OutsideBeforeExecution (call_id "classes/A//a2///classes/C//c2///2")
                         (call_id2 "classes/A//a2///classes/C//c3///3")
OutsideBeforeExecution (call_id "classes/A//a1///classes/C//c2///2")
                       (call_id2 "classes/A//a1///classes/B//b2///3")
```

- **Outside After Execution Relations**

Se puede ver en el conjunto de hechos listado a continuación las relaciones de ejecución Outside After, junto con el valor de la métrica asociada a cada método. Para cada caso se muestra la métrica correspondiente a cada método y las relaciones que aportan al valor. Con el hecho OutsideAfterExecution se indica que la llamada al método asociado al id call_id se ejecuta inmediatamente después que la llamada al método call_id2. Por ejemplo, el método B.b2() presenta una relación de ejecución OutsideAfter con el método A.a1().

```
OutsideAfterExecutionMetric (method "classes/B//b2///") (metric "1")
OutsideAfterExecution (call_id "classes/A//a1///classes/B//b2///3")
                      (call_id2 "classes/A//a1///classes/C//c2///2")
```

```
OutsideAfterExecutionMetric (method "classes/C//c3///") (metric "1")
OutsideAfterExecution (call_id "classes/A//a2///classes/C//c3///3")
                      (call_id2 "classes/A//a2///classes/C//c2///2")


OutsideAfterExecutionMetric (method "classes/C//c2///") (metric "2")
OutsideAfterExecution (call_id "classes/A//a2///classes/C//c2///2")
                      (call_id2 "classes/A//a2///classes/B//b1///1")
OutsideAfterExecution (call_id "classes/A//a1///classes/C//c2///2")
                      (call_id2 "classes/A//a1///classes/B//b1///1")


OutsideAfterExecutionMetric (method "classes/C//c1///") (metric "1")
OutsideAfterExecution (call_id "classes/A//a2///classes/C//c1///4")
                      (call_id2 "classes/A//a2///classes/C//c3///3")
```

- **Inside First Execution Relations**

Se observa a continuación la métrica de la relación de ejecución Inside After para el método B.b1(). Seguido a este hecho se listan las relaciones de las cuales se obtiene este valor. No se presenta el valor de las métricas para el resto de los métodos, ya que el método en cuestión es el único que presenta la característica de ejecutarse primero en dos métodos distintos.

```
InsideFirstExecutionMetric (method "classes/B//b1///") (metric "2")
InsideFirstExecution (call_id "classes/A//a2///classes/B//b1///1")
                     (method_id "classes/A//a2///")
InsideFirstExecution (call_id "classes/A//a1///classes/B//b1///1")
                     (method_id "classes/A//a1///")
```

- **Inside Last Execution Relations**

En el conjunto de hechos siguiente se puede observar la métrica de la relación de ejecución Inside Last para los métodos B.b2() y C.c1(). Ambos métodos son los únicos que cumplen con la condición de ser llamados en último lugar desde otros métodos.

```
InsideLastExecutionMetric (method "classes/B//b2///") (metric "1")
InsideLastExecution (call_id "classes/A//a1///classes/B//b2///3")
                    (method_id "classes/A//a1///")

InsideLastExecutionMetric (method "classes/C//c1///") (metric "1")
InsideLastExecution (call_id "classes/A//a2///classes/C//c1///4")
                    (method_id "classes/A//a2///")
```

5.3.4.3. Salidas del Ejemplo

Cuando el algoritmo termina de ejecutarse, los hechos que representan los resultados finales quedan alojados en la base de datos del sistema experto. Para recuperar estos se hace uso de las consultas implementadas previamente explicadas. La Tabla V – 8 expone los resultados finales obtenidos por el análisis de Execution Relations para el ejemplo en cuestión.

Método	Valor de la Métrica	Tipo de Relación de Ejecución
b1	2	OutsideBefore
c3	1	OutsideBefore
c2	2	OutsideBefore
b2	1	OutsideAfter
c3	1	OutsideAfter
c2	2	OutsideAfter
c1	1	OutsideAfter
b1	2	InsideFirst
b2	1	InsideLast
c1	1	InsideLast

Tabla V - 8. Salidas del ejemplo de Relaciones de Ejecución.

5.4. Análisis de Métodos Redireccionadores

Esta técnica identifica aquellas clases que funcionan como una capa de indirección de otra clase (i.e. wrapper). Para esto se calcula la cantidad de métodos que como parte de su lógica redireccionan la llamada a un método de otra clase. Luego, este valor es utilizado para decidir si estás clases forman parte de una capa de redirección y en consecuencia aspectos candidatos.

5.4.1. Hechos Particulares del Enfoque

La Tabla V – 9 presenta los hechos que define la técnica de métodos redireccionadores con el objetivo de utilizarlos para persistir información de los razonamientos en la base de datos. Por ejemplo, el hecho *cantRedirecPorClase* indica la cantidad de métodos redireccionadores que posee cada clase.

Hecho	Atributos	Semántica del hecho
cantMetodosPorClase (slot idClase) (slot cantMet)	*idClase*: identificador de una clase. *cantMet*: valor numérico que representa cantidad de métodos.	La clase identificada por idClase contiene tantos métodos como indica cantMet
redirectMethod (slot metodoBase) (slot claseBase) (slot metodoRedireccionado) (slot claseRedireccionada)	*metodoBase*: identificador de un método. *claseBase*: identificador de la clase a la que pertenece el método identificado por metodoBase. *metodoRedireccionado*: identificador de un método. claseRedireccionada: identificador de la clase a la que pertenece el método identificado por metodoRedireccionado.	El método identificado por metodoBase perteneciente a la clase identificada por claseBase, redirecciona su llamada al método identificado por metodoRedireccionado que pertenece a la clase identificada por claseRedireccionada.
cantRedirecPorClase (slot claseBase) (slot claseRedireccionada) (slot cant)	*claseBase*: identificador de una clase. *claseRedireccionada*: identificador de una clase. *cant*: valor numérico que representa la cantidad de métodos redireccionadores.	La clase identificada con claseBase contiene la cantidad de métodos definida por cant que que redireccionan a métodos de la clase identificada con claseRedireccionada.

Tabla V - 9. Hechos propios del enfoque Métodos Redireccionadores.

5.4.2. Implementación del Algoritmo

A continuación, se listan y explican las reglas de mayor relevancia para la implementación de la técnica.

Mediante la regla de la Fig. V – 23 es posible consultar si un método redirecciona su llamada a un método de otra clase. Para ello, selecciona los métodos que participan en

llamadas (Call: con ids *metodoLlamador* y *metodoLlamado*) y comprueba las siguientes restricciones:

- que no exista otro método de la clase a la que pertenece *metodoLlamador* que llame a *metodoLlamado*.
- que metodoLlamador no llame a otro método de la clase a la cual pertenece MetodoLlamado.

De cumplirse ambas restricciones, se persiste el hecho *redirectMethod* para dejar especificado el método redireccionador encontrado.

```
(defrule redirectorMethod
    (Call (caller_id ?MetodoLlamador) (callee_id ?MetodoLlamado))
    (Method (id ?MetodoLlamador)(class_id ?classIdLlamador))
    (Method (id ?MetodoLlamado)(class_id ?classIdLlamada))
    (not (and
            (Call (caller_id ?otroMetodoClaseLlamadora)(callee_id ?MetodoLlamado))
            (Method (id ?otroMetodoClaseLlamadora&~?MetodoLlamador)
                    (class_id ?classIdLlamador))
        )
    )
    (not (and
          (Call (caller_id ?MetodoLlamador)(callee_id ?otroMetodoClaseLlamada))
          (Method (id ?otroMetodoClaseLlamada&~?MetodoLlamado)
                  (class_id ?classIdLlamada))
         )
     )
    =>
    (assert (redirectMethod(metodoBase ?MetodoLlamador)
                           (claseBase ?classIdLlamador)
                           (metodoRedireccionado ?MetodoLlamado)
                           (claseRedireccionada ?classIdLlamada)))
)
```

Fig. V - 23. Regla que detecta los métodos redireccionadores.

Una vez que se persisten en la base de datos todos los hechos que representan a los métodos redireccionadores, es posible obtener la cantidad de métodos redireccionadores por clase. Para esto, se utiliza la regla de la Fig. V - 24, la cual cuenta la cantidad de métodos de una clase (representada por el atributo *claseBase* del hecho *redirectMethod*) que redireccionan a otra clase (representada por el atributo *claseRedireccionada* del hecho

redirectMethod). Para evitar contar más de una vez el mismo método se utiliza el hecho *redirectMethodCounted* en forma auxiliar.

```
(defrule calculaCantidadMetodosClase
    (Class (id ?idClass))
    (Method (id ?metodo) (class_id ?idClass))
    (not (methodCounted(idMethod ?metodo) (idClass ?idClass)))
    ?OldCantMetodos <- (cantMetodosPorClase (idClase ?idClass)(cantMet ?Cant))
    =>
    (assert (methodCounted (idMethod ?metodo)(idClass ?idClass)))
    (bind ?NewMetric (+ ?Cant 1))
    (modify ?OldCantMetodos (cantMet ?NewMetric))
)
```

Fig. V - 24. Regla que cuenta la cantidad de métodos por clase.

```
(defrule finalRedirectMetodosPorClase
    ?OldCantRedicMetodos <- (cantRedirecPorClase
                            (claseBase ?classIdeLlamador)
                            (claseRedireccionada ?classIdLlamada)
                            (cant ?Cant))
    (redirectMethod (claseBase ?classIdeLlamador)
                    (claseRedireccionada ?classIdLlamada)
                    (metodoBase ?MetodoLlamador)
                    (metodoRedireccionado ?MetodoLlamado))
    (not (redirectMethodCounted(metodoBase ?MetodoLlamador)
                               (claseBase ?classIdeLlamador)
                               (metodoRedireccionado ?MetodoLlamado)
                               (claseRedireccionada ?classIdLlamada)))
    =>
    (assert (redirectMethodCounted (metodoBase ?MetodoLlamador)
                                   (claseBase ?classIdeLlamador)
                                   (metodoRedireccionado ?MetodoLlamado)
                                   (claseRedireccionada ?classIdLlamada)))
    (bind ?NewMetric (+ ?Cant 1))
    (modify ?OldCantRedicMetodos (cant ?NewMetric))
)
```

Fig. V - 25. Regla que cuenta la cantidad de métodos redireccionadores por clase.

Como define la heurística, se debe determinar la cantidad y el porcentaje del total de métodos que redireccionan sus llamados desde una clase a otra. En consecuencia, se debe calcular la cantidad total de métodos que posee cada clase. La Fig. V - 25, muestra la regla que realiza dicho cálculo. La misma busca todos los métodos de una clase y se aumenta en 1 el valor que representa la cantidad de métodos. Este valor se guarda en la base de datos en el hecho *cantMetodosPorClase*.

5.4.3. Consulta de Resultados

La Fig. V - 26 muestra las consultas implementadas para obtener los resultados del algoritmo y mapearlos a objetos java para manipularlos dentro de la aplicación. Estas consultas son:

- **cantMetodos**: dado el identificador de una clase, devuelve la cantidad de métodos que posee la misma.

- **cantRedirectorMethods**: devuelve los hechos *cantRedirectorMethods* de la base de datos. Del mismo se obtiene: la clase base, la clase redireccionada y la cantidad de métodos que se redireccionan desde la clase base hacia la redireccionada.

- **metodosRedirectorsPorClase:** devuelve los métodos de la clase identificada por *claseLlamadora* que redireccionan a la clase representada por el identificador *claseLlamada*.

```
(defquery cantMetodos
   (declare (variables ?idClase))
   (cantMetodosPorClase (idClase ?idClase)(cantMet ?cant))
)
(defquery cantRedirectorMethods
   (cantRedirecPorClase (claseBase ?classIdeLlamador)
                        (claseRedireccionada ?classIdLlamada)
                        (cant ?Cant))
)
(defquery metodosRedirectorsPorClase
   (declare (variables ?claseLlamadora ?claseLlamada))
   (redirectMethod  (claseBase ?claseLlamadora)
                    (claseRedireccionada ?claseLlamada)
                    (metodoBase ?MetodoLlamador)
                    (metodoRedireccionado ?MetodoLlamado))
)
```

Fig. V - 26. Consultas definidas para el algoritmo de.Redirector Methods.

5.4.4. Ejemplo

A continuación, se presenta un ejemplo del enfoque Métodos Redireccionadores. La Fig. V - 27 muestra el diagrama de clases del mismo y la Tabla V - 10 muestra los llamados

realizados hacia los métodos. Las filas representan el método llamador, y las columnas el método llamado. La intersección que contiene X indica que el método de la fila llama al método indicado en la columna.

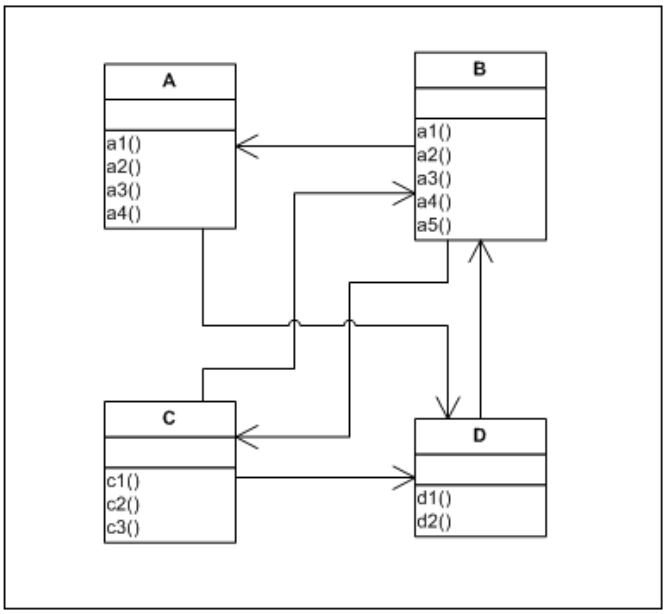

Fig. V - 27. Diagrama de clases del ejemplo.

	A.a1()	A.a2()	A.a3()	B.b3()	B.b4()	C.c1()	C.c2()	D.d1()	D.d2()
A.a1()								X	
A.a2()								X	
B.b1()	X								
B.b2()		X							
B.b3()			X						
B.b5()							X		
C.c1()					X				
C.c3()									X
D.d1()					X				

Tabla V - 10. Tabla de llamados del ejemplo.

A continuación, se presentan los hechos de entrada generados por el parser, los razonamientos parciales del razonamiento y las salidas del mismo.

5.4.4.1. Hechos de Entrada

A continuación, se listan los hechos de entrada por cada clase del ejemplo. Por cada una de ellas se muestra el hecho que representa a la clase, los métodos que contiene, y por cada método los llamados que realiza:

Hechos derivados de la estructura de la clase A:

```
Class (id "classes/A")(name "A")

Method (id "classes/A//a1///") (class_id "classes/A")(name "a1") (parametros "")
       (returnType "void")
Call (caller_id "classes/A//a1///")(callee_id "classes/D//d1///")
     (id "classes/A//a1///classes/D//d1///1") (precedence "1")

Method (id "classes/A//a2///")(class_id "classes/A")(name "a2")(parametros "")
       (returnType "void")
Call (caller_id "classes/A//a2///") (callee_id "classes/D//d1///")
     (id "classes/A//a2///classes/D//d1///1") (precedence "1")

Method (id "classes/A//a3///")(class_id "classes/A")(name "a3")(parametros "")
       (returnType "void")

Method (id "classes/A//a4///")(class_id "classes/A")(name "a4")(parametros "")
       (returnType "void")
```

La semántica de los hechos indica lo siguiente: El hecho Class indica la existencia de la clase A, con id "classes/A". Los métodos a1(), a2(), a3(), y a4() pertenecen a dicha clase y se representan con el hecho Method. Luego, los hechos Call indican llamadas entre métodos, por ejemplo el método A.a1() invoca al método d1() de la clase D.

Este análisis es igual para todos las clases presentadas a continuación.

Hechos derivados de la estructura de la clase B:

```
Class (id "classes/B") (name "B")

Method (id "classes/B//b1///")(class_id "classes/B")(name "b1")(parametros "")
       (returnType "void")
Call (caller_id "classes/B//b1///") (callee_id "classes/A//a1///")
     (id "classes/B//b1///classes/A//a1///1") (precedence "1")
```

```
Method (id "classes/B//b2///")(class_id "classes/B")(name "b2")(parametros "")
       (returnType "void")
Call (caller_id "classes/B//b2///") (callee_id "classes/A//a2///")
     (id "classes/B//b2///classes/A//a2///1") (precedence "1")

Method (id "classes/B//b3///")(class_id "classes/B") (name "b3")(parametros "")
       (returnType "void")
Call (caller_id "classes/B//b3///") (callee_id "classes/A//a3///")
     (id "classes/B//b3///classes/A//a3///1") (precedence "1")

Method (id "classes/B//b4///")(class_id "classes/B")(name "b4")(parametros "")
       (returnType "void")

Method (id "classes/B//b5///")(class_id "classes/B")(name "b5")(parametros "")
       (returnType "void")
Call (caller_id "classes/B//b5///")(callee_id "classes/C//c2///")
     (id "classes/B//b5///classes/C//c2///1") (precedence "1")
```

Hechos derivados de la estructura de la clase C:

```
Class (id "classes/C") (name "C")
Method (class_id "classes/C") (id "classes/C//c1///") (name "c1")
       (parametros "") (returnType "void")
Call (caller_id "classes/C//c1///") (callee_id "classes/B//b4///")
     (id "classes/C//c1///classes/B//b4///1") (precedence "1")

Method (class_id "classes/C") (id "classes/C//c2///") (name "c2")
       (parametros "") (returnType "void")

Method (id "classes/C//c3///")(class_id "classes/C")(name "c3")
       (parametros "") (returnType "void")
Call    (callee_id    "classes/D//d2///")    (caller_id    "classes/C//c3///")(id
"classes/C//c3///classes/D//d2///1") (precedence "1")
```

Hechos derivados de la estructura de la clase D:

```
Class (id "classes/D") (name "D")

Method (class_id "classes/D") (id "classes/D//d1///") (name "d1")(parametros "")
       (returnType "void")

Call (caller_id "classes/D//d1///") (callee_id "classes/B//b4///")
     (id "classes/D//d1///classes/B//b4///1") (precedence "1")

Method (class_id "classes/D") (id "classes/D//d2///") (name "d2")(parametros "")
       (returnType "void")
```

5.4.4.2. Razonamiento del Sistema

Una vez que los hechos fueron introducidos a la base de datos, se ejecutan las reglas del algoritmo Métodos Redireccionadores.

A continuación, se muestran los hechos de salida obtenidos agrupados por clase. Para cada uno de ellos se muestran sus métodos redireccionadores, la cantidad de métodos redireccionadores, y la cantidad total de métodos presentes en la clase:

- **Clase A**

La clase A no presenta métodos redireccionadores, ya que dos de sus métodos llaman al mismo método de la clase B y esta característica hace fallar la regla *redirectorMethod*. En consecuencia, no se generan hecho para esta clase.

- **Clase B**

El siguiente código muestra los hechos que indican la cantidad de métodos redireccionadores de la clase B, y los métodos que aportan a dicho valor. Como se puede observar, la clase B redirecciona sus llamados a los métodos de la clase A y de la clase C. Para este ejemplo, 3 métodos de la clase B - b1(), b2() y b3() - llaman a métodos de la clase A - a1(), a2() y a3() -. A su vez, ningún otro método de esta clase B llama a1(), a2() y a3(), y b1(), b2() y b3() tampoco llaman a otro método de A. Finalmente, la clase B redirecciona a 3 métodos de la clase A. Lo mismo sucede con la clase C, excepto que solo 1 método es redireccionado.

```
cantMetodosPorClase (idClase "classes/B") (cantMet 5)

cantRedirecPorClase (claseBase "classes/B")(claseRedireccionada "classes/A")(cant 3)

cantRedirecPorClase (claseBase "classes/B") (claseRedireccionada "classes/C")(cant 1)

redirectMethod (metodoBase "classes/B//b5///") (claseBase "classes/B")
               (metodoRedireccionado "classes/C//c2///")
               (claseRedireccionada "classes/C")

redirectMethod (metodoBase "classes/B//b1///") (claseBase "classes/B")
               (metodoRedireccionado "classes/A//a1///")
               (claseRedireccionada "classes/A")
redirectMethod (metodoBase "classes/B//b2///") (claseBase "classes/B")
               (metodoRedireccionado "classes/A//a2///")
               (claseRedireccionada "classes/A")
redirectMethod (metodoBase "classes/B//b3///") (claseBase "classes/B")
               (metodoRedireccionado "classes/A//a3///")
               (claseRedireccionada "classes/A")
```

- **Clase C**

El siguiente código muestra los hechos para la clase C luego de ejecutar el algoritmo de Métodos Redireccionadores. La clase C redirecciona a la clase B - el método c1() llama a b4() - y también lo hace a la clase D - el método c3() llama a d2() -. Los dos casos cumplen con las restricciones que plantea la regla *redirectorMethod*.

```
cantMetodosPorClase (idClase "classes/C") (cantMet 3)

cantRedirecPorClase (claseBase "classes/C") (claseRedireccionada "classes/B")(cant 1)
redirectMethod (metodoBase "classes/C//c1///") (claseBase "classes/C")
               (metodoRedireccionado "classes/B//b4///")
               (claseRedireccionada "classes/B")

cantRedirecPorClase (claseBase "classes/C") (claseRedireccionada "classes/D")(cant 1)
redirectMethod (metodoBase "classes/C//c3///") (claseBase "classes/C")
               (metodoRedireccionado "classes/D//d2///")
               (claseRedireccionada "classes/D")
```

- **Clase D**

Los siguientes hechos muestran que la clase D redirecciona a la clase B (el método d1 llama a b4). Este método no llama a ningún otro método de B, y el método d2 de D tampoco llama a b4, en consecuencia, el método d1 cumple con las restricciones de método redireccionador.

```
cantMetodosPorClase (idClase "classes/D") (cantMet 2)

cantRedirecPorClase (claseBase "classes/D") (claseRedireccionada "classes/B")
                    (cant 1)
redirectMethod (metodoBase "classes/D//d1///") (claseBase "classes/D")
               (metodoRedireccionado "classes/B//b4///")
               (claseRedireccionada "classes/B")
redirectMethod (metodoBase "classes/D//d1///") (claseBase "classes/D")
               (metodoRedireccionado "classes/B//b4///")
               (claseRedireccionada "classes/B")
```

5.4.4.3. Salidas del Ejemplo

Una vez que se ejecutan todas las reglas, los resultados del algoritmo quedan persistidos en la base de datos. Para obtener dicha información se hace uso de las consultas *cantMetodos, metodosRedirectoresPorClase* y *cantRedirectrMethods* explicadas

previamente. La Tabla V – 11 resume las clases redireccionadoras, junto con la cantidad de métodos que participan, el porcentaje del total de métodos de la clase que redirecciona, y las llamadas que aportan a dichos valores.

Clase Redireccionadora	Clase Redireccionada	Cantidad de Métodos	% de Métodos	Llamados
B	A	3	60%	b1() -> a1() b2() -> a2() b3() -> a3()
B	C	1	20%	b5() -> c2()
C	B	1	33,33%	C1() -> b4()
C	D	1	33,33%	C3() -> d2()
D	B	1	50%	D1() -> b4()

Tabla V - 11. Salidas para ejemplo de Métodos Redireccionadores.

5.5. Sinergia

El algoritmo realiza un análisis conjunto de los resultados de los algoritmos de Fan-in, Unique Methods y Execution Relations con el propósito de detectar seeds presentes en el código de una aplicación y en consecuencia, aporta mayor grado de fiabilidad a los resultados finales. Es necesario definir ciertos parámetros de entrada con el objetivo de automatizar la selección de seeds de dichos algoritmos. Estos parámetros son valores de umbral y niveles de confianza:

- **Valor de umbral:** el valor de umbral indica a partir de qué número asociado a cada método un algoritmo lo considera como seed. Por ejemplo, si el valor de Fan-in asociado al método m1 es igual a 10, y el valor de umbral definido para dicho enfoque es igual a 5, el método m1 será considerado como seed candidata.

- **Nivel de confianza:** el nivel de confianza establece el porcentaje de certeza que posee cada algoritmo individual.

Se deben especificar tres umbrales y cuatro niveles de confianza. Los umbrales son definidos para cada algoritmo con el fin de filtrar las seeds reportadas por ellos. Los niveles de confianza se establecen tanto para cada algoritmo independiente como para Sinergia. Los valores de confianza para cada algoritmo indican cuanto aportará cada algoritmo a la solución final, y el valor de confianza de Sinergia indica el porcentaje de certeza que un método debe cumplir para ser reportado como seed.

5.5.1. Hechos Particulares del Enfoque

Los hechos de entrada corresponden a las salidas de los algoritmos de Fan-in, Métodos Únicos y Relaciones de Ejecución. Luego, se definen los hechos que se utilizarán para persistir los resultados parciales y finales del algoritmo (Tabla V – 12). Por ejemplo, los métodos fan-in_seed, unique_method_seed y execution_relation_seed se utilizan para indicar que un método corresponde a una seed reportada por Fan-in, Método Únicos o Relaciones de Ejecución, respectivamente.

Hecho	Atributos	Semántica del hecho
fan-in_seed (method_id)	*method_id*: identificador de un método.	El método identificado por method_id es considerado como seed por el algoritmo de Fan-in.
unique_method_seed(method_id)	*method_id*: identificador de un método.	El método identificado por method_id es considerado como seed por el algoritmo de Unique Methods.
execution_relation_seed(method_id)	*method_id*: identificador de un método.	El método indentificado por method_id es considerado como seed por el algoritmo de Execution Relations.
fan-in_trust(trust)	*trust*: número entero. Representa un nivel de confianza.	Se tiene un nivel de confianza igual al valor indicado por el atributo trust sobre el algoritmo de Fan-in.
unique_method_trust(trust)	*trust*: número entero. Representa un nivel de confianza.	Se tiene un nivel de confianza igual al valor indicado por el atributo trust sobre el algoritmo de Métodos Únicos.

execution_relation_trust(trust)	*trust*: número entero. Representa un nivel de confianza.	Se tiene un nivel de confianza igual al valor indicado por el atributo trust sobre el algoritmo de Relaciones de Ejecución.
seed(method_id, trust)	*method_id*: identificador de un método. *trust*: número entero. Representa un nivel de confianza.	El método identificado por method_id es considerado como seed con una seguridad igual al valor indicado por el atributo trust.

Tabla V - 12. Hechos propios del enfoque Sinergia.

5.5.2. Implementación del Algoritmo

A continuación se presentan y explican las reglas más importantes del algoritmo del presente enfoque. El primer conjunto de reglas es utilizado para marcar como seeds a aquellos métodos que son considerados como tales por los algoritmos particulares. Luego, se hace una ponderación general, acumulando la confianza que se tiene sobre cada algoritmo. Por último, se eliminan los seeds que no alcanzan el umbral de confianza general.

- **Seeds considerados por Fan-in:** la regla definida en la Fig. V – 28 toma cada una de los hechos resultados del algoritmo Fan-in (*fan-in_metric*) y verifica si estas satisfacen el mínimo umbral establecido. Para los casos que el valor de Fan-in sea mayor o igual al umbral, el método con dicha métrica es marcado como seed mediante el agregado del hecho *fan-in_seed* a la memoria de trabajo. Los hechos que han sido analizados son eliminados con la sentencia *retract* para evitar computarlas más de una vez.

```
(defrule mark_as_fan-in_seed
    ?Fan-in_metric <- (fan-in_metric (method ?Method) (metric ?Metric))
    (fan-in_umbral (umbral ?Umbral))
    =>
    (if (>= ?Metric ?Umbral) then
       (assert (fan-in_seed (method ?Method)))
    )
    (retract ?Fan-in_metric)
)
```

Fig. V - 28. Regla que marca seeds según el criterio de Fan-in.

- **Seeds considerados por Métodos Únicos:** la regla definida en la Fig. V – 29 toma los resultados del algoritmo de Métodos Únicos (*unique_method_metric*) y verifica si estas satisfacen el umbral establecido. Para los casos que el valor de Fan-in sea mayor o igual al umbral, los métodos son marcados como seed mediante el agregado del hecho *unique_method_seed* a la memoria de trabajo. Los hechos que han sido analizados son eliminadas con la sentencia *retract* para evitar computarlos más de una vez.

- **Seeds considerados por Relaciones de Ejecución:** la regla definida en la Fig. V – 30 toma cada uno de los hechos de tipo Outside Before Execution calculados por el algoritmo de Relaciones de Ejecución (*OutsideBeforeExecutionMetric*) y verifica si estas satisfacen el umbral establecido. Para los casos en que el valor asociado al método sea mayor o igual al umbral, se lo marca como seed mediante el agregado del hecho *execution_relation_seed* a la memoria de trabajo. Los hechos que ya han sido analizados son eliminados con la sentencia *retract* para evitar computarlas más de una vez. Las reglas para los tipos de relaciones de ejecución restantes (*Outside After, Inside First* e *Inside Last*) son similares, la única diferencia se encuentra en el tipo de hecho analizado, y por lo tanto se omiten estas reglas.

```
(defrule mark_as_unique_method_seed
    ?Unique_method_metric <- (unique_method_metric (method ?Method)
                                                   (metric ?Metric))
    (unique_method_umbral (umbral ?Umbral))
    =>
    (if (>= ?Metric ?Umbral) then
       (assert (unique_method_seed (method ?Method)))
    )
    (retract ?Unique_method_metric)
)
```

Fig. V - 29. Regla que marca seeds según el criterio de Métodos Únicos.

```
(defrule mark_as_execution_relation_seed
    ?OutsideBeforeExecutionMetric <- (OutsideBeforeExecutionMetric
                                                        (method?Method)
                                                        (metric ?Metric))
    (OutsideBeforeExecution_umbral (umbral ?Umbral))
    =>
    (if (>= ?Metric ?Umbral) then
       (assert (execution_relation_seed (method ?Method)))
    )
    (retract ?OutsideBeforeExecutionMetric)
)
```

Fig. V - 30. Regla que marca seeds según el criterio de Relaciones de Ejecución.

- **Acumulación de la confianza:** las regla definidas en la Fig. V – 31, la Fig. V -32 y la Fig. V - 33 toman las seeds de los algoritmos Fan-in, Métodos Únicos y Relaciones de Ejecución respectivamente y suman el nivel de confianza de que aporta cada uno al nivel de certeza general que se tiene sobre este seed. Así, el atributo *trust* del hecho *seed* se ve incrementado por el valor de confianza que se tiene sobre cada algoritmo.

```
(defrule acum_seed_fan-in
    ?FanInSeed <- (fan-in_seed (method ?Method))
    ?Seed <- (seed (method ?Method) (trust ?Trust))
    (fan-in_trust (trust ?FanInTrust))
      =>
    (bind ?NewTrust (+ ?FanInTrust ?Trust))
    (modify ?Seed (trust ?NewTrust))
    (retract ?FanInSeed)
)
```

Fig. V - 31. Regla que acumula la confianza de Fan-in en los seeds.

```
(defrule acum_seed_unique_method
    ?UniqueMethodSeed <- (unique_method_seed (method ?Method))
    ?Seed <- (seed (method ?Method) (trust ?Trust))
    (unique_method_trust (trust ?UniqueMethodTrust))
      =>
    (bind ?NewTrust (+ ?UniqueMethodTrust ?Trust))
    (modify ?Seed (trust ?NewTrust))
    (retract ?UniqueMethodSeed)
)
```

Fig. V - 32. Regla que acumula la confianza de Métodos Únicos en los seeds.

- **Eliminación de seeds candidatos que no superan el umbral general:** la regla definida en la Fig. V – 34 elimina aquellos métodos que no alcanzan el nivel

de confianza general establecido para un seed. El hecho *seed* es removido de la memoria de trabajo si su atributo *trust* es menor al nivel de confianza general. Los seeds restantes (hechos *seed* restantes) componen el resultado del algoritmo.

```
(defrule acum_seed_execution_relation
    ?ExecutionRelationSeed <- (execution_relation_seed (method ?Method))
    ?Seed <- (seed (method ?Method) (trust ?Trust))
    (execution_relation_trust (trust ?ExecutionRelationTrust))
       =>
    (bind ?NewTrust (+ ?ExecutionRelationTrust ?Trust))
    (modify ?Seed (trust ?NewTrust))
    (retract ?ExecutionRelationSeed)
)
```

Fig. V - 33. Regla que acumula la confianza de Relaciones de Ejecución en los seeds.

```
(defrule remove_false_seeds
    ?Seed <- (seed (method ?Method) (trust ?Trust))
    (umbral_trust (trust ?GeneralTrust))
       =>
    (if (< ?Trust ?GeneralTrust) then
       (retract ?Seed)
    )
)
```

Fig. V - 34. Regla que elimina los seeds candidatos que no alcanzan el umbral de confianza general.

5.5.3. Consulta de Resultados

La Fig. V – 35 muestra las consultas implementadas para algoritmo Sinergia. De esta forma, se obtienen los resultados que fueron persistidos en la base de datos y se mapean dichos resultados a a objetos java para manipularlos dentro de la aplicación. Estas consultas son:

- ***getSeeds***: devuelve las seeds identificadas por el algoritmo.
- ***getFanInSeeds***: devuelve las seeds que son consideradas como tal por el algoritmo de Fan-in. Esta información es utilizada para indicar en los resultados finales (presentados al usuario) si el algoritmo considera a la seed como tal.

- ***getUniqueMethodsSeeds***: devuelve las seeds que son consideradas como tal por el algoritmo de Métodos Únicos. Esta información es utilizada para indicar en los resultados finales (presentados al usuario) si este algoritmo considera a la seed como tal.

- ***getExecutionRelationsSeeds****:* devuelve las seeds que son consideradas como tal por el algoritmo de Relaciones de Ejecución. Esta información será utilizada para indicar en los resultados finales (presentados al usuario) si este algoritmo considera a la seed como tal.

```
(defquery getSeeds
    (declare (variables ?ln))
    (seed (method ?method)(trust ?trust))
)
(defquery getFanInSeeds
    (declare (variables ?method))
    (fan-in_seed_Counted (method ?method))
)
(defquery getUniqueMethodsSeeds
    (declare (variables ?method))
    (unique_method_seed_Counted (method ?method))
)
(defquery getExecutionRelationsSeeds
    (declare (variables ?method))
    (execution_relation_seed_Counted (method ?method))
)
```

Fig. V - 35. Consultas definidas en el algoritmo de Sinergia.

5.5.4. Ejemplo

La Fig. V - 36 muestra el diagrama de clases del ejemplo que será utilizado para el algoritmo Sinergia y la Tabla V – 13 presenta los llamados realizados hacia los métodos. Las filas corresponden a los métodos llamadores y las columnas a los métodos que son invocados. El valor numérico en las celdas no vacías indica la precedencia del llamado (orden en que se realiza con respecto a los demás llamados), las celdas vacías representan que el llamado no existe.

	A1.m	B.m	B.m1	C1.m	C1.m1	C2.m	C2.m1

Llamadas.llamados1()	**1**			**2**			
Llamadas.llamados2()		**1**	**2**		**3**		
Llamadas.llamados3()	**1**			**2**	**3**		**4**
Llamadas.llamados4()	**1**					**2**	**3**

Tabla V - 13. Llamados entre los métodos de las clases.

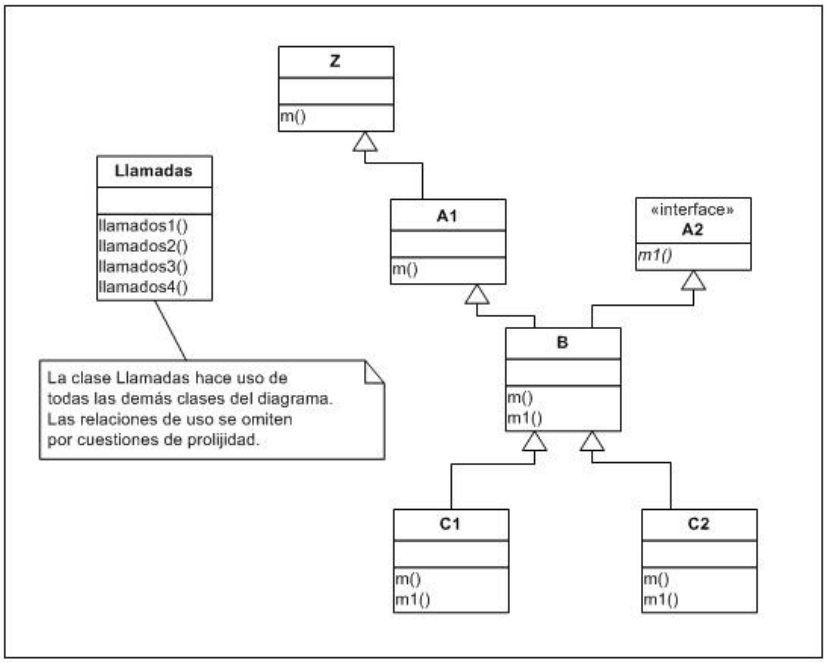

Fig. V - 36. Diagrama de clases para ejemplo de Sinergia.

Para realizar el análisis se han establecido los siguientes parámetros de umbral y confianza para cada algoritmo. La Tabla V – 14 muestra dichos valores.

	Umbral	**Confianza**
Fan-in	**3**	**33%**
Unique Methods	**2**	**33%**
Execution Relations	**2**	**33%**

Tabla V - 14. Valores de confianza y umbral.

El valor de confianza general que un seed candidato tiene que alcanzar para ser considerado como válido se estableció a 50%. Este valor indica que la sumatoria de confianzas de los tres algoritmos individuales debe ser mayor a este número. En el ejemplo

planteado, tanto Fan-in como Métodos Únicos y Relaciones de Ejecución poseen el mismo valor de confianza, por lo tanto, el 50% indica que la mitad de los algoritmos deben haberlo seleccionado como seed.

5.5.4.1. Hechos de Entrada

A continuación, se presentan los hechos que conforman la entrada del algoritmo de Sinergia para este ejemplo, agrupados por el algoritmo que aporta dicha información:

- **Resultados de Fan-in**

El hecho fan-in_metric indica el valor de Fan-in que posee cada método. Por ejemplo, el método C2.m1() tiene un valor de Fan-in igual a 3.

```
fan-in_metric (method_id "llamadas/Llamadas//llamados1///") (metric 0)
fan-in_metric (method_id "llamadas/Llamadas//llamados2///") (metric 0)
fan-in_metric (method_id "llamadas/Llamadas//llamados3///") (metric 0)
fan-in_metric (method_id "llamadas/Llamadas//llamados4///") (metric 0)
fan-in_metric (method_id "classes/Z//m///") (metric 3)
fan-in_metric (method_id "classes/A//m///") (metric 4)
fan-in_metric (method_id "classes/A2//m1///") (metric 3)
fan-in_metric (method_id "classes/B//m1///") (metric 3)
fan-in_metric (method_id "classes/B//m///") (metric 3)
fan-in_metric (method_id "classes/C1//m///") (metric 4)
fan-in_metric (method_id "classes/C1//m1///") (metric 2)
fan-in_metric (method_id "classes/C2//m///") (metric 2)
fan-in_metric (method_id "classes/C2//m1///") (metric 3)
```

- **Resultados de Métodos Únicos**

El hecho UniqueMethodsMetric representan los métodos únicos del ejemplo, junto con su valor de Fan-in.

```
UniqueMethodsMetric (method_id "llamadas/Llamadas//llamados3///") (metric 0)
UniqueMethodsMetric (method_id "classes/C1//m1///") (metric 2)
UniqueMethodsMetric (method_id "classes/C2//m1///") (metric 3)
UniqueMethodsMetric (method_id "classes/A2//m1///") (metric 3)
UniqueMethodsMetric (method_id "llamadas/Llamadas//llamados4///") (metric 0)
UniqueMethodsMetric (method_id "llamadas/Llamadas//llamados1///") (metric 0)
UniqueMethodsMetric (method_id "llamadas/Llamadas//llamados2///") (metric 0)
UniqueMethodsMetric (method_id "classes/B//m1///") (metric 3)
```

- **Resultados para el algoritmo de Relaciones de Ejecución**

Los hechos OutsideBeforeExecutionMetric, OutsideAfterExecutionMetric, InsideFirstExecutionMetric y InsideLastExecutionMetric corresponden a los resultados obtenidos para las relaciones Outise Before, Outside After, Inside First e Inside Last respectivamente. Para un determinado método, por ejemplo B.m() identifica las relaciones de ejecución del tipo en las que participa junto con la cantidad de veces que participa en dicha ejecución.

```
OutsideBeforeExecutionMetric (method "classes/B//m///") (metric "1")
OutsideBeforeExecutionMetric (method "classes/C1//m///") (metric "1")
OutsideBeforeExecutionMetric (method "classes/C1//m1///") (metric "1")
OutsideBeforeExecutionMetric (method "classes/C2//m///") (metric "1")
OutsideBeforeExecutionMetric (method "classes/A1//m///") (metric "3")
OutsideBeforeExecutionMetric (method "classes/B//m1///") (metric "1")

OutsideAfterExecutionMetric (method "classes/C1//m///") (metric "2")
OutsideAfterExecutionMetric (method "classes/C1//m1///") (metric "2")
OutsideAfterExecutionMetric (method "classes/C2//m///") (metric "1")
OutsideAfterExecutionMetric (method "classes/C2//m1///") (metric "2")
OutsideAfterExecutionMetric (method "classes/B//m1///") (metric "1")

InsideFirstExecutionMetric (method "classes/B//m///") (metric "1")
InsideFirstExecutionMetric (method "classes/A1//m///") (metric "3")
InsideLastExecutionMetric (method "classes/C1//m///") (metric "1")
InsideLastExecutionMetric (method "classes/C1//m1///") (metric "1")
InsideLastExecutionMetric (method "classes/C2//m1///") (metric "2")
```

5.5.4.2. Razonamiento del Sistema

Luego de la ejecución del algoritmo con los datos de entrada listados previamente, se obtienen los siguientes seeds.

El siguiente código muestra los hechos que fueron considerados como seeds según el algoritmo de Fan-in. Esto indica que el valor de Fan-in de cada método era mayor o igual al umbral previamente establecido:

```
fan-in_seed_Counted (method "classes/Z//m///")
fan-in_seed_Counted (method "classes/A1//m///")
fan-in_seed_Counted (method "classes/A2//m1///")
fan-in_seed_Counted (method "classes/B//m///")
fan-in_seed_Counted (method "classes/B//m1///")
fan-in_seed_Counted (method "classes/C1//m///")
fan-in_seed_Counted (method "classes/C2//m1///")
```

Los hechos que se listan a continuación presentan los métodos que fueron considerados como seeds según el algoritmo de Unique Methods y su umbral previamente establecido:

```
unique_method_seed_Counted (method "classes/A2//m1///")
unique_method_seed_Counted (method "classes/B//m1///")
unique_method_seed_Counted (method "classes/C1//m1///")
unique_method_seed_Counted (method "classes/C2//m1///")
```

A continuación se muestran los hechos que representan los métodos considerados como seeds por el algortimo de Relaciones de Ejecución.

```
execution_relation_seed_Counted (method "classes/A1//m///")
execution_relation_seed_Counted (method "classes/C1//m///")
execution_relation_seed_Counted (method "classes/C1//m1///")
execution_relation_seed_Counted (method "classes/C2//m1///")
```

Luego de calcular los seeds que aporta cada algoritmo, se debe calcular el valor de confianza de cada método, esto es, la sumatoria de las confianzas que aporta cada algoritmo al resultado. En el caso de ejemplo, cada algoritmo aporta un 33% de confianza. Los hechos luego de realizar este cálculo se listan a continuación:

```
seed (method "classes/Z//m///")(trust "33.0")
seed (method "classes/B//m///")(trust "33.0")
seed (method "classes/A1//m///") (trust "66.0")
seed (method "classes/A2//m1///") (trust "66.0")
seed (method "classes/B//m1///") (trust "66.0")
seed (method "classes/C1//m///") (trust "66.0")
seed (method "classes/C1//m1///") (trust "66.0")
seed (method "classes/C2//m1///") (trust "99.0")
```

5.5.4.3. Salidas del Ejemplo

Como salida del algoritmo se obtienen los métodos que fueron considerados como seeds según Sinergia y su umbral de confianza previamente establecido. El umbral establecido fue del 50%. La Tabla V- 15 presenta los resultados arrojados por el algoritmo Sinergia. Las columnas de Fan-in, Métodos Únicos y Relaciones de Ejecución indican si fueron o no reportados por tales algoritmos.

Método	Confianza	Fan-in	Métodos	Relaciones de

			Únicos	Ejecución
A1.m()	66%	Si	No	Si
A2.m1()	66%	Si	Si	No
B.m1()	99%	Si	Si	Si
C1.m()	99%	Si	Si	Si
C1.m1()	99%	Si	Si	Si
C2.m1()	99%	Si	Si	Si

Tabla V - 15. Salida del Ejemplo para el enfoque Sinergia.

6. Synergy Analysis Tool

Synergy Analysis Tool (SAT) es una herramienta implementada como un plugin para eclipse [54] que permite extraer los seeds candidatos de un sistema legado. Esta herramienta ofrece la posibilidad de ejecutar cinco algoritmos de aspect mining. Tres de ellos realizan un análisis a nivel de métodos con el fin de determinar cuál de ellos resulta en un seed candidato. Un cuarto algoritmo arroja como resultado seeds candidatos a nivel de clase y el quinto y último presenta un análisis híbrido entre los tres algoritmos que se ejecutan a nivel de métodos. Este último, realiza la selección de seeds ponderando los resultados obtenidos por los tres algoritmos mencionados. La Fig. V - 37 muestra los 5 análisis que provee la herramienta.

Una vez ejecutado un algoritmo, los resultados pueden visualizarse en forma de vistas de eclipse.

La Fig. V – 38 presenta como ejemplo la vista asociada al algoritmo de Fan-in. Las vistas correspondientes a los resultados de los algoritmos Unique Methods Analysis y Execution Relations Analysis poseen el mismo formato que la de Fan-in Analysis, por lo que se omitirán dichas imágenes. Las vistas ofrecen un menú desde el cual se puede abrir el/los recursos seleccionados y seleccionarlos como seeds definitivos.

La Fig. V -39 muestra la vista de seeds, la cual se muestra al seleccionar un seed candidato como definitivo. En esta vista, los seeds pueden manipularse para poder seleccionarlos en forma más detallada (por ejemplo marcar aquellos métodos que están relacionados con el seed y aquellos que no).

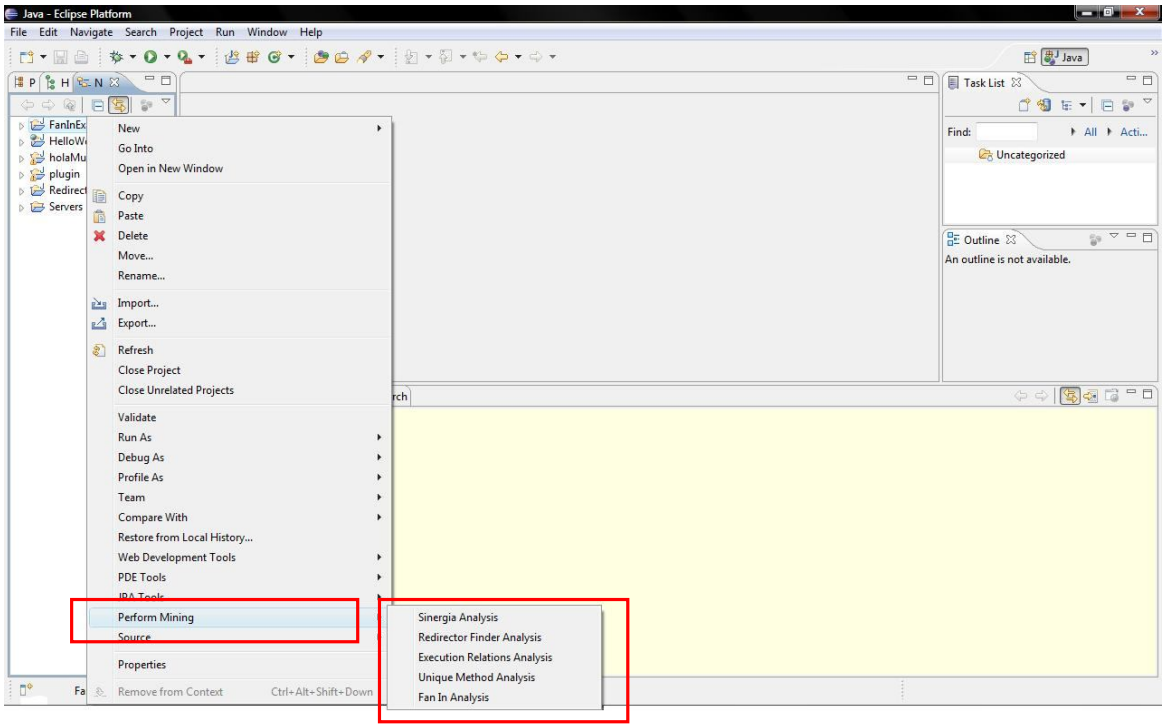

Fig. V - 37. Análisis ofrecidos por Synergy Analysis Tool.

La Fig V – 40 muestra la pantalla de configuración para el algoritmo Sinergia, en la cual se ingresan los valores de umbral para los algoritmos de entrada (Fan-in, Unique Methods y Execution Relations), el valor de confianza de cada algoritmo y el valor porcentaje utilizado como umbral para los resultados finales del análisis conjunto. Luego, la Fig. V – 41 presenta la vista en la cual se plasman los seeds candidatos arrojados por el análisis, junto con los algoritmos que indican dicho seed como positivo.

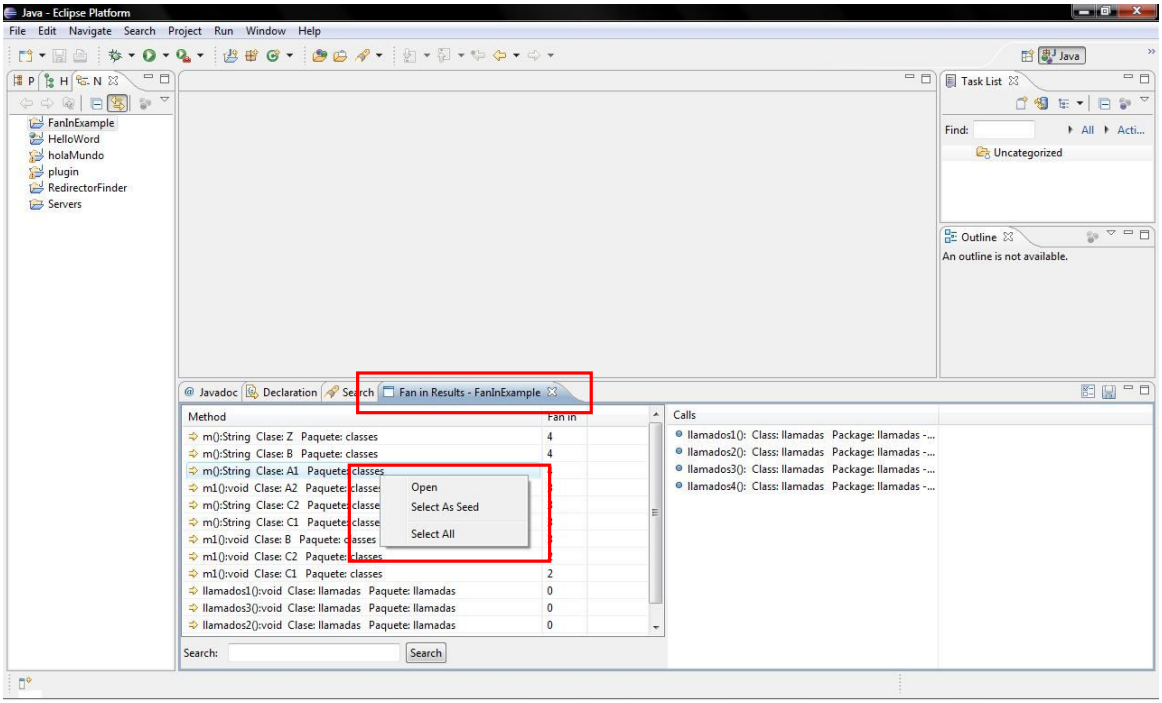

Fig. V - 38. Vista de Fan-in.

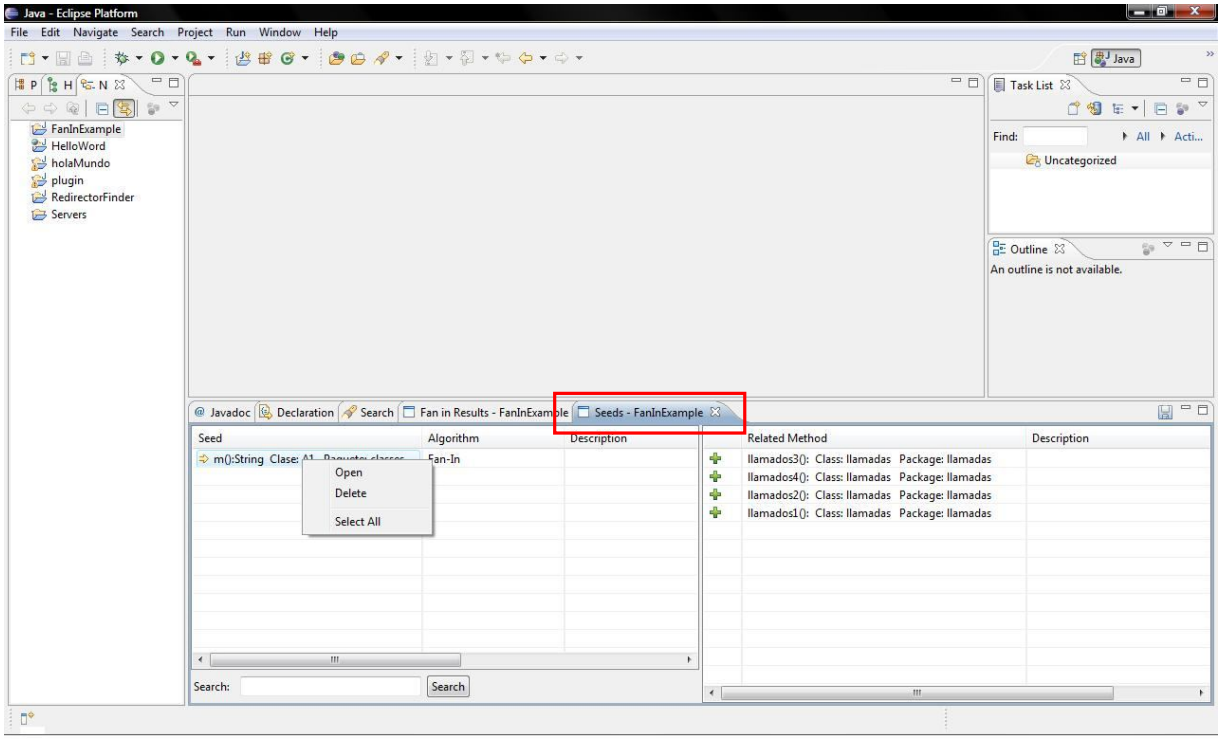

Fig. V - 39. Vista de Seeds a nivel de métodos.

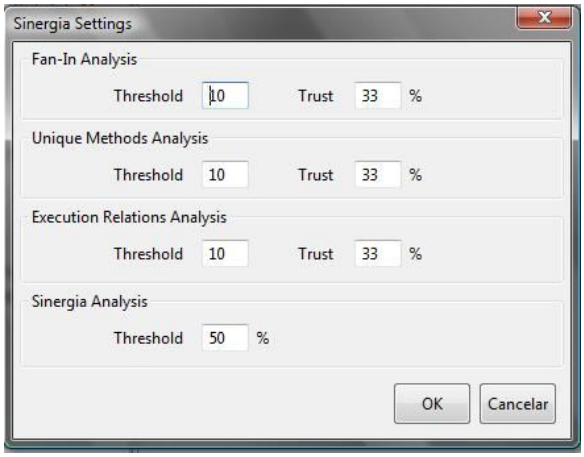

Fig. V - 40. Pantalla de Configuración de análisis de Sinergia

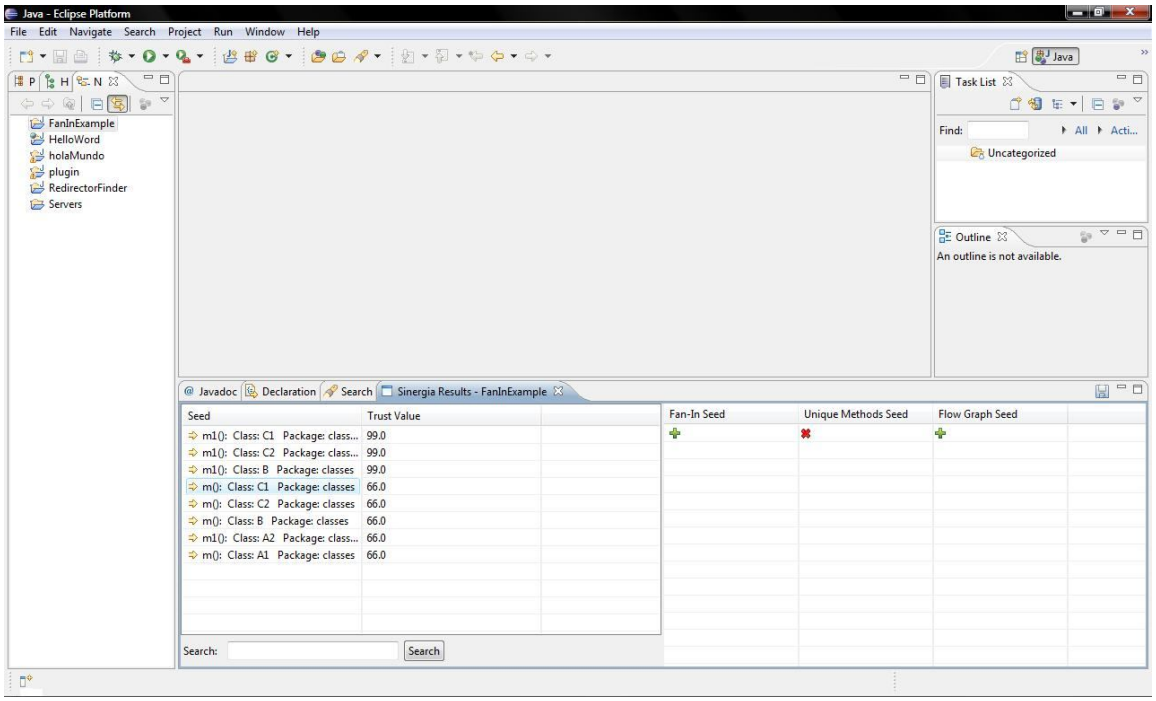

Fig. V - 41. Vista de resultados provenientes del análisis Sinergia.

CAPÍTULO VI

Caso de Estudio

1. Introducción

El presente capítulo tiene como objetivo mostrar los resultados obtenidos por la herramienta luego de analizar un sistema real, y evaluar la capacidad de la misma para identificar crosscutting concerns en este tipo de sistemas. Se aplica el análisis sobre la versión 1 del sistema Health Watcher [8, 10, 75].

2. Health Watcher

Health Watcher [8, 10, 75] es una aplicación desarrollada acorde a una arquitectura por capas utilizando tecnología J2EE [74]. El propósito principal de este sistema es permitir a ciudadanos registrar sus quejas referidas a temas de salud. El mismo fue seleccionado como caso de estudio por las siguientes razones:

- Es un sistema real y suficientemente complejo, con implementaciones tanto en el paradigma de programación orientado a objetos como en el paradigma orientado a aspectos. Ambas versiones fueron diseñadas aplicando principios de modificabilidad [8, 11, 79].

- Se han reportado análisis previos del sistema [10], lo cual provee un análisis cualitativo de ambas implementaciones. La disponibilidad de la versión orientada a aspectos permite comprobar la capacidad de la herramienta desarrollada para identificar los crosscutting concerns en la versión orientada a objetos.

- Es un sistema real que involucra un gran número de concerns clásicos como por ejemplo concerns de concurrencia, persistencia y distribución. Adicionalmente, la aplicación hace uso de tecnologías comúnmente utilizadas en contextos industriales como RMI (Java Remote Method Invocations) [17], Servlets [63] y JDBC (Java Database Connectivity) [73].

- La disponibilidad de ambas versiones permite analizar de manera cuantitativa y objetiva los resultados del enfoque propuesto.

2.1. Versión Orientada a Objetos

La versión orientada a objetos del sistema HW fue implementada utilizando el lenguaje de programación Java [70]. A su vez, con el objetivo de lograr modificabilidad y extensibilidad, la arquitectura de este sistema fue organizada en capas. La utilización de este patrón arquitectónico (Layers), junto con la aplicación de patrones de diseño relacionados al mismo, ayudan no sólo a satisfacer dichos atributos de calidad, sino también a disminuir el código entrelazado [50, 23, 29].

Las 4 capas principales del sistema son: interfaz gráfica de usuario (View Layer), distribución (Distribution Layer), código de negocio (Business Layer) y datos (Data Layer). (Fig. VI – 1)

La capa de interfaz gráfica fue implementada como una interfaz web, utilizando para ello la API de Servlets [63] de Java [70]. Por su parte, la capa de distribución es la responsable de proveer los servicios del negocio en forma distribuida. El acceso a los servicios de HW se realiza mediante la interface *IFacade,* la cual es implementada por *HealthWatcherFacade.* Esta capa es implementada utilizando la tecnología RMI (Remote Method Invocation) [17].

La capa de negocio contiene aquellas clases que representan conceptos del dominio y que contienen las reglas del negocio. Algunas de estas clases, como *ComplaintRecord* y

EmployeeRecord, permiten el acceso a la capa de datos mediante la interface *IComplaintRep*. A través de esta interface se desacopla la lógica de negocio de la lógica específica de gestión de datos.

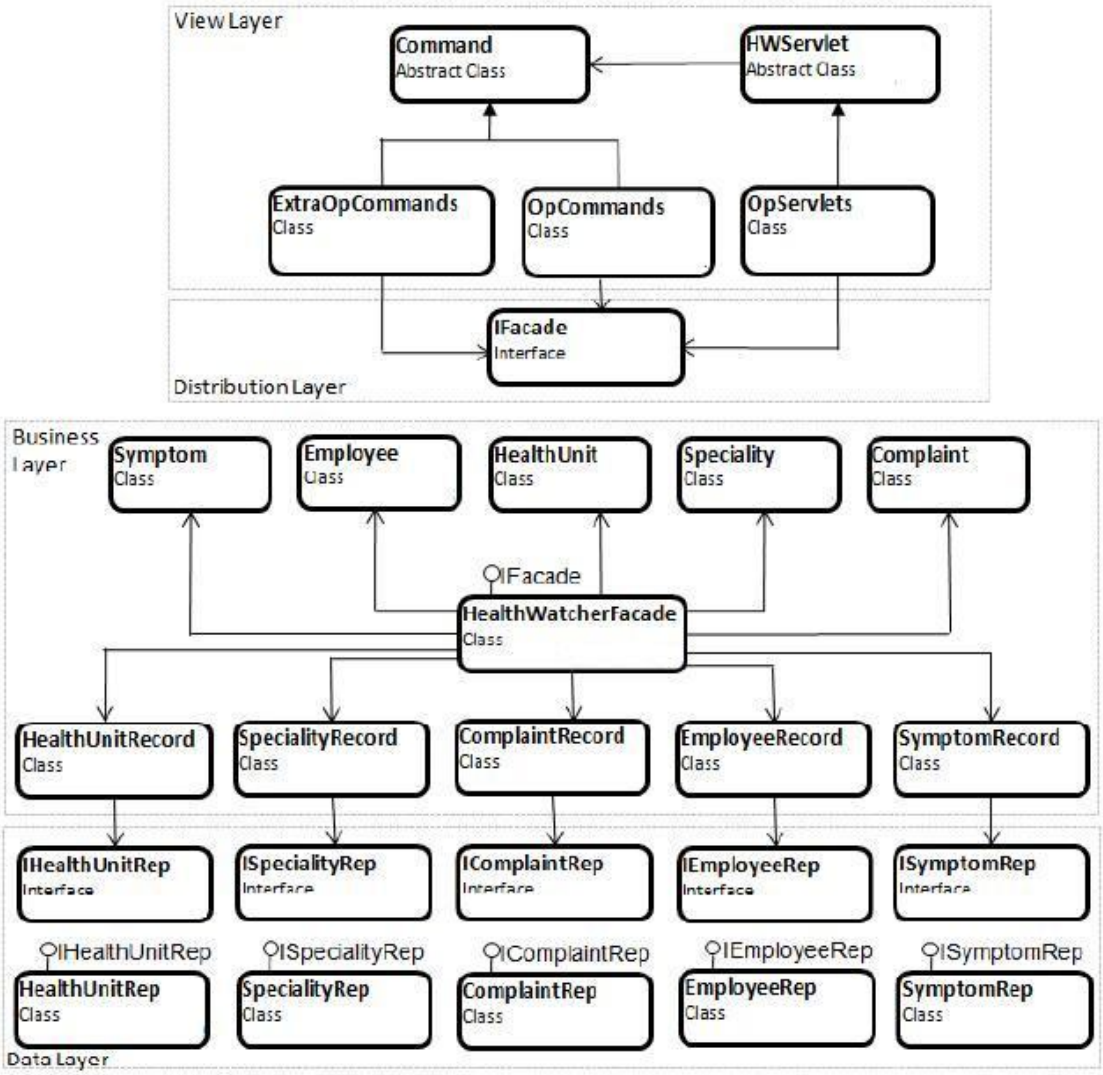

Fig. VI - 1. Arquitectura de HW de la versión OO.

Finalmente, la capa de persistencia fue implementada mediante la API de JDBC, y permite acceder y persistir los datos de las quejas de los usuarios del sistema.

Adicionalmente, mediante el uso de patrones de diseño, como Command, Adapter y Decorator [23, 50, 72], los requerimientos de reusabilidad y mantenibilidad de la implementación pudieron ser alcanzados.

2.2. Versión Orientada a Aspectos

Con el objetivo de lograr una mejor separación de concerns, y para evitar código entrelazado y diseminado, este sistema fue reestructurado [8, 10, 75] con el objetivo de introducir aspectos que encapsulen aquellos crosscutting concerns presentes en el mismo.

El diseño de la versión AO se centra en los mismos principios de reusabilidad y mantenibilidad de la versión OO. La única diferencia es que el diseño de esta versión fue llevado a cabo con el fin de separar los crosscutting concerns de persistencia, distribución y concurrencia, del resto de la implementación. Si bien la arquitectura elegida persigue este objetivo, el aislamiento no es absoluto y se pueden encontrar concerns diseminados por las diferentes capas del sistema. La nueva versión implementa de igual manera el patrón arquitectónico por capas, aunque se omite la capa de distribución y se implementa con aspectos (Fig. VI - 2).

A continuación, se describen aquellos crosscutting concerns que fueron encapsulados en aspectos en esta versión del sistema: distribución, persistencia y concurrencia.

2.2.1. Concern de Distribución

El primer paso para encapsular el código correspondiente al concern de distribución en un aspecto consiste en remover el código específico de RMI de la versión escrita puramente en Java. Por lo general, en un sistema con este tipo de arquitectura, el código RMI se encuentra diseminado en la clase *facade* (del lado del servidor) y en las clases de interfaz de usuario (lado del cliente). No obstante, algunas clases de la lógica del negocio

también poseen código relacionado a RMI, ya que sus objetos son parámetros y valores de retornos de los métodos del *facade,* los cuales son ejecutados remotamente.

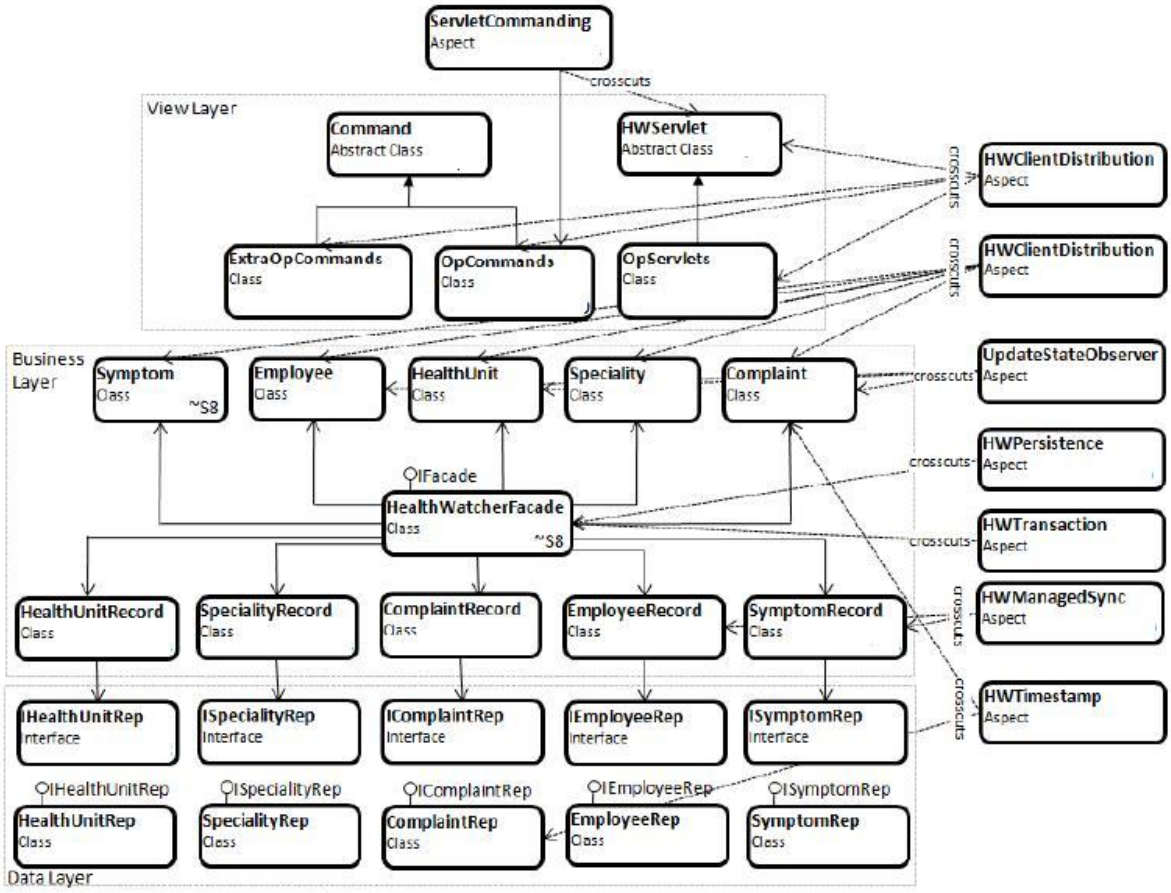

Fig. VI - 2. Arquitectura de HW de la versión AO.

En la versión orientada a aspectos se eliminó el código correspondiente a RMI de todas las clases del sistema, y se implementó la misma funcionalidad de forma separada en un conjunto de aspectos. De esta manera el concern de distribución consiste en un conjunto de aspectos y clases e interfaces auxiliares. Cuando este código contenido en los aspectos es ligado con el código de las clases del sistema, efectivamente aspectualiza el *facade* y las clases correspondientes a la interfaz de usuario; la comunicación entre estas se vuelve remota mediante la distribución de la instancia del *facade* por el cliente y el servidor.

El aspecto *ServerSideHWDistribution* es responsable de la disponibilidad remota de la instancia del *facade* y de asegurar que los métodos de éste último tengan parámetros y valores de retorno serializables, ya que es requerido por RMI. Para que el *facade* esté disponible en forma remota, los aspectos del lado del servidor deben modificar la clase facade (*HWFacade*) para implementar la interface remota correspondiente (*IHWFacade*).

2.2.2. Concern de Persistencia

Este concern está relacionado a la persistencia y resurrección del estado de un objeto. El mismo está concentrado principalmente en las colecciones de datos y en las clases de lógica del negocio.

El código de persistencia incluye, no sólo la interface *IPersistenceMechanism* junto con sus implementaciones, sino también las interfaces de *IBusinessData*. Los aspectos de persistencia afectan el *facade* y las colecciones de datos de la lógica de negocio.

En la nueva versión orientada a aspectos de Health Watcher, el código de persistencia incluye aspectos y clases e interfaces auxiliares para satisfacer los siguientes concerns: mecanismo de control de conexión y transacciones, carga parcial y caché de objetos para mejorar la performance y sincronización del estado de los objetos con las entidades de la base de datos para asegurar consistencia.

2.2.2.1. Mecanismo de Control de Persistencia

Los aspectos que implementan el mecanismo de control de persistencia están encargados de administrar los objetos que dan acceso a los datos. Para ello, los aspectos crean una instancia de la clase que representa al mecanismo de persistencia (una implementación de *IPersistenceMechanism*) y mediante esta clase, manejan la inicialización de la base de datos, la conexión y liberación de los recursos.

El aspecto que controla la persistencia se encarga de inicializar y retornar el mecanismo de persistencia al comenzar la ejecución del sistema, así como también se ocupa de la liberación del recurso.

2.2.2.2. Mecanismo de Control de Transacciones

Es esencial en un sistema distribuido poder garantizar las propiedades ACID [72]: atomicidad de las operaciones, consistencia de los datos, aislamiento al realizar operaciones y durabilidad de los datos ante fallas. En la versión de HW orientada a objetos, la mayor parte de las operaciones que deben cumplir con las restricciones ACID son invocadas desde la clase *facade*. En consecuencia, esta clase posee código entrelazado relacionado a la transaccionalidad de sus operaciones.

La versión orientada a aspectos intenta modularizar este concern y separar el código entrelazado hacia un aspecto. Debido a esto, se define un aspecto, *TransactionControlHW,* que identifica los métodos transaccionales - métodos que deben ser ejecutados como una transacción lógica – y se encarga de comenzar, terminar satisfactoriamente o abortar transacciones. Los métodos transaccionales están incluidos en la interface del facade. De esta manera, el uso de aspectos permitió eliminar del facade todo el código relacionado a este concern.

2.2.2.3. Acceso a Datos Bajo Demanda

Dado que los objetos pueden tener estructuras complejas y estar compuestos de otros objetos, es necesario contar con políticas de acceso a los mismos con el fin de evitar la degradación de la performance del sistema. Por ejemplo, un servicio que permite consultar una lista de quejas posiblemente solo necesite la descripción y el código de cada queja, mientras que un servicio que genere un reporte completo necesitará todos los datos asociados al objeto. El sistema Health Watcher implementa dicha característica agregando un parámetro a dichos métodos de acceso. Este enfoque tiene dos problemas principales, el primero es que el parámetro mencionado no tiene relación con el método en cuestión, y el

segundo es que cualquier método que necesite acceder al estado del objeto deberá especificar dicho parámetro.

Con el objetivo de evitar estos problemas, se define un aspecto llamado *ParameterizedLoading* para efectuar el acceso por demanda. Este aspecto agrega a los métodos de acceso el parámetro extra, aunque evita que sea visible para el resto de los servicios del sistema.

2.2.3. Concurrencia

El concern de concurrencia se encuentra relacionado directamente con el concern de persistencia, ya que identifica los puntos de sincronización sobre los datos que se modifican y posteriormente se persisten en la base de datos.

Las capas de negocio y presentación gestionan objetos que deben ser persistidos en la base de datos. Con el fin de garantizar la consistencia de los datos almacenados, el acceso a estos métodos debe estar sincronizado.

Con el fin de separar este concern, las capas mencionadas no deben saber cuándo un objeto es persistente o cuando no lo es. Por lo tanto, se deben separar las llamadas a los métodos de sincronización, e implementar la misma funcionalidad en un aspecto que se encargue de controlar la sincronización de los datos. El aspecto encargado de dicha funcionalidad se denomina *UpdateStateControl,* el cual captura los mensajes a los objetos que se deben persistir y al final del servicio provisto actualiza los datos en el repositorio.

2.2.4. Gestión de Excepciones

La gestión de excepciones representa un crosscutting concern cuya cobertura es a nivel sistema, es decir, está presente en la mayoría de las clases del mismo. Por lo tanto, el aspecto que implemente dicha funcionalidad tendrá que definir aquellos puntos donde deberán lanzarse las excepciones, y saber cuáles de estas retornar. Por ejemplo, para el

caso particular de la gestión de excepciones en los servlets de la aplicación, el aspecto deberá conocer aquellas excepciones específicas a los servlets.

3. Identificación de crosscutting concerns en HW

En esta sección se presentan los resultados obtenidos de aplicar las técnicas de aspect mining implementadas en la herramienta desarrollada al sistema Health Watcher (versión orientada a objetos). Para el análisis desarrollado, se optó por aplicar el enfoque Sinergia y la técnica de identificación de métodos redireccionadores, ambos descriptos en el capitulo anterior. Se realizaron dos análisis con Sinergia variando los parámetros de entrada y comparando los resultados entre ambos. Por cada experimento se calcularon tanto la precisión como el recall de la herramienta [80]. La precisión se define como el número de seeds confirmados dividido el número total de seeds reportados. El recall se define como el número de crosscutting concerns encontrados mediante el enfoque respecto del total de crosscutting concerns existentes en el sistema.

3.1. Sinergia: Análisis I

La Fig. V – 3 muestra los parámetros de entrada seleccionados para la ejecución del enfoque. Se eligió un valor de umbral de 10 e igual valor de confianza (33%) para los tres enfoques. El valor de confianza de Sinergia es del 50%, lo que indica, junto con los valores de confianza seleccionados para los algoritmos individuales, que se reportarán los seeds que hayan sido seleccionados por al menos 2 de los 3 algoritmos.

La Tabla VI -1 exhibe los resultados obtenidos luego de realizado el análisis.

Seed Candidatos	Confianza	Fan-in Seed	Unique Methods Seed	Execution Relations Seed
Método: commitTransaction() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	99.0%	X	X	X

Método: rollbackTransaction() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	99.0%	X	X	X
Método: releaseCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	99.0%	X	X	X
Método: Object getCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	99.0%	X	X	X
Método: beginTransaction() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	99.0%	X	X	X
Método: getPm() **Clase: HealthWatcherFacadeInit** **Paquete: healthwatcher.business**	66.0%	X	-	X
Método: getCodigo() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	66.0%	X		X
Método: errorPage(String): **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X

Tabla VI - 1. Resultados Sinergia I.

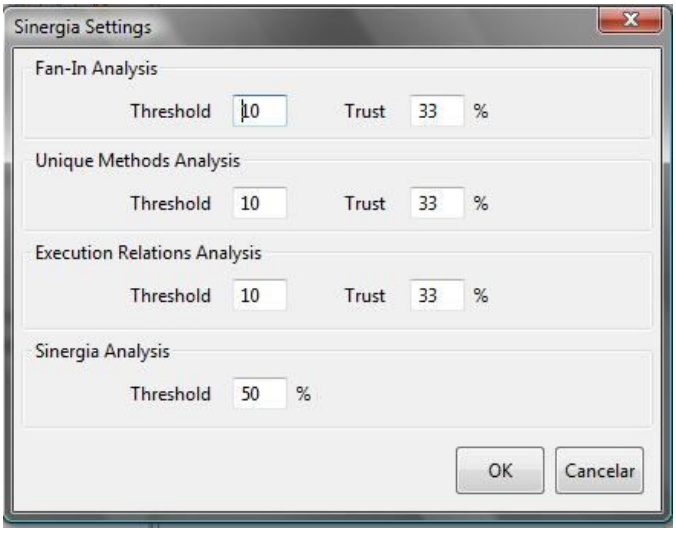

Fig. VI - 3. Valores de entrada para Sinergia.

La Tabla VI – 2 presenta la correspondencia entre los seeds candidatos y los concerns involucrados con cada seed. A continuación, se describe cada seed en particular.

Seed Candidatos	Concern Asociado
Método: commitTransaction() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	Persistencia (Transacciones)
Método: rollbackTransaction() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	Persistencia (Transacciones)
Método: Object getCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	Persistencia
Método: beginTransaction() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	Persistencia (Transacciones)
Método: releaseCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	Falso Positivo
Método: getPm() **Clase: HealthWatcherFacadeInit** **Paquete: healthwatcher.business**	Persistencia (Control de Persistencia)
Método: getCodigo() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	Falso Positivo
Método: errorPage(String): **Clase: HTMLCode** **Paquete: lib.util**	Gestión de Excepciones

Tabla VI - 2. Seeds candidatos y concerns asociados.

3.1.1. Seeds Candidatos: *commitTransaction* y *rollbackTransaction*

Ambos métodos pertenecen a la clase *PersistenceMechanism, commitTransaction* indica que se ha finalizado con éxito la ejecución de una operación de tipo transaccional, en

cambio el método *rollbackTransaction* permite volver a un estado consistente de la aplicación.

En HW, la mayoría de los métodos transaccionales están definidos en la clase *HealthWatcherFacadeInit.* Dichos métodos no deberían realizar los llamados a *commitTransaction* y *rollbackTransaction,* debido a que no es parte de la funcionalidad básica de la clase. Por lo tanto, se evidencia la presencia de llamados a métodos que implementan el concern de persistencia diseminados en la aplicación. Específicamente, corresponden al control de transacciones y deberían ser encapsulados en un aspecto.

3.1.2. Seeds Candidatos: *getCommunicationChannel* y *releaseCommunicationChannel*

Si bien los dos métodos fueron reportados como seeds por las tres técnicas de Sinergia (confianza del 100%), los mismos se corresponden a falsos positivos.

El método *getCommunicationChannel* tiene como objetivo devolver un objeto *Statement* que permite ejecutar consultas SQL a la base de datos, y el método *releaseCommunicationChannel* tiene como objetivo liberar el canal de comunicación. Ambos métodos pertenecen a la clase *PersistenceMechanism*. El contexto en que ambos métodos son utilizados, indica que los mismos implementan funcionalidad relacionada a las clases y métodos que hacen uso de los mismos. Por ejemplo, el método *insert(Address)* de la clase *AddressRepositoryRDB* invoca a *getCommunicationChannel* con el fin obtener el objeto Statement, y utilizarlo para ejecutar la inserción.

El hecho de que estos métodos hayan sido reportados como seeds se debe ha que los mismos son llamados desde las diferentes implementaciones de los repositorios, haciendo que estos posean un gran Fan-in y estén presentes en muchas relaciones de ejecución.

3.1.3. Seed Candidato: *beginTransaction*

El método *beginTransaction()* fue reportado como seed con un 100% de confianza. El mismo pertenece a la interface *IPersistenceMechanism,* y es implementado en forma concreta por *PersistenceMechanism*. La finalidad del método es dar comienzo a una transacción luego de obtener el canal de comunicación.

Los llamados al mismo provienen de la clase *HealthWatcherFacadeInit,* donde los métodos de la misma lo invocan a fin de dar comienzo a una transacción. Dicha funcionalidad no es propia de los métodos de la clase mencionada, por lo tanto se considera a este seed como parte de un crosscutting concern correspondiente a la gestión de las transacciones. Esta funcionalidad se corresponde con aquella implementada en el aspecto *TransactionControlHW* de la versión AO.

3.1.4. Seed Candidato: *getPm*

El método *getPm()* es reportado como seed sólo por dos de los tres enfoques. El enfoque Unique Methods no lo reporta ya que no cumple con la restricción de poseer el tipo de retorno void.

En el contexto de la aplicación, el método getPm pertenece a la clase *HealthWatcherFacadeInit*. Este método utiliza el método con la misma signatura de la clase *HealthWatcherFacade* de forma de obtener la instancia del mecanismo de persistencia. Este último es un singleton, el cual retorna la instancia de *IPersistenceMechanism*, y en caso de que la misma no haya sido creada, realiza previamente la inicialización de ésta llamando al método *pmInit* de *IPersistenceMechanism*.

Este método corresponde al concern de persistencia, específicamente al control de la persistencia (obtener el mecanismo de persistencia e inicializarlo). Claramente, este concern no pertenece a la lógica de inicialización del facade, simplemente se realiza la inicialización en este punto por necesidad, ya que al inicializar los servicios de la aplicación,

el mecanismo de persistencia debe ser inicializado. Es por esto, que se decide extraer esta funcionalidad en un aspecto, y refactorizar ambos facades.

3.1.5. Seed Candidato: *getCodigo*

El método *getCodigo* es reportado como seed por dos de los tres enfoques. A pesar de esto, este seed es un falso positivo, ya que es un método getter de la clase *Complaint*.

3.1.6. Seed Candidato: *errorPage*

Este método tiene un valor de confianza del 66%, dado que el análisis de Unique Method no lo reporta como seed debido al tipo de retorno (String) que posee.

El método es utilizado para devolverle al usuario un mensaje de error. Este es invocado desde los distintos servlets al momento en que se presenta una excepción de tipo *RemoteException*, correspondiente a RMI. El método *errorPage* corresponde al concern de manejo de excepciones, por lo que el llamado al mismo es refactorizado en un aspecto. En la nueva versión, este método no será invocado desde los servlets sino será llamado desde dicho aspecto.

La versión orientada a aspectos define un aspecto, *HWDistributionExceptionHandler*, encargado del manejo de excepciones referentes a este tipo de concerns. Tiene como funcionalidad interceptar las excepciones, obtener un objeto sobre el cual presentar los errores, y finalmente presentar el error a los usuarios del servicio.

3.1.7. Evaluación del experimento

A continuación se presentan algunas métricas obtenidas a partir de los resultado de la ejecución del análisis de Sinergia bajo los parámetros previamente mencionados.

- **Número de seeds reportados**: 8.
- **Número de seeds confirmados**: 6.

- **Número de falsos positivos**: 2.
- **Número de falsos negativos**: 3.
- **Precisión**: 6 / 8 = 0,75 (Calculado como el Número de Seeds Confirmados sobre el Número de Seeds Reportados).
- **Recall**: 3 / 6 = 0,50 (Calculado como el Número de Crosscutting Concerns Detectados sobre el Número de Crosscutting Concerns Posibles de Detectar).

3.2. Sinergia: Análisis II

En esta sección se presentan los resultados de ejecutar nuevamente el algoritmo de Sinergia seleccionando valores de umbrales inferiores a los establecidos en el primer análisis. Como resultado se obtuvo un conjunto de resultados diferente y lógicamente más amplio al conseguido anteriormente. Esto se realiza con el objetivo de comparar ambos resultados, analizando cuánto afecta el establecimiento de umbrales adecuados en la identificación de crosscutting concerns.

La Fig. VI – 4 muestra los parámetros de entrada elegidos para la ejecución del enfoque. El valor de confianza de Sinergia es del 50%, lo que indica, que se reportarán los seeds que hayan sido seleccionados al menos por 2 de los 3 algoritmos. Por otra parte, los umbrales mínimos para cada técnica fueron drásticamente disminuidos pasando de un valor de 10 a un valor de 4. La idea de este segundo experimento es observar cómo varían la precisión y el recall respecto del primer experimento. Al disminuir los valores de los umbrales se espera un aumento en el recall (cantidad de crosscutting concerns identificados) y una disminución de la precisión (porcentaje de seeds confirmados sobre seeds reportados).

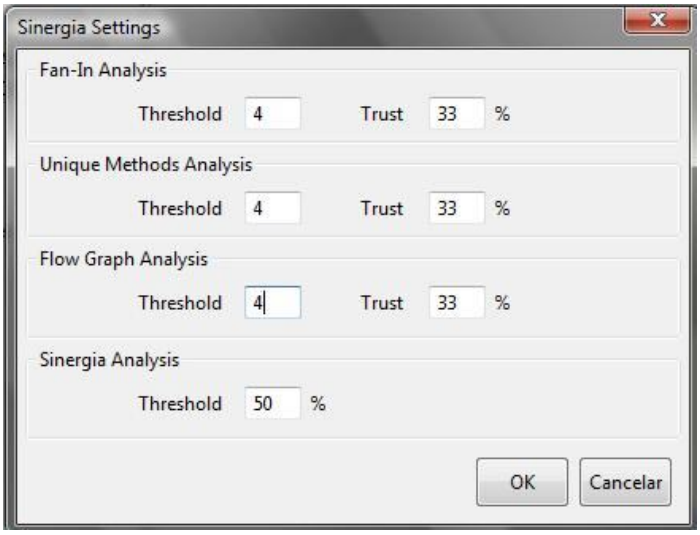

Fig. VI - 4. Valores de entrada para Sinergia II.

La Tabla VI -3 exhibe los resultados obtenidos luego de realizado el análisis.

Seed Candidatos	Confianza	Fan-in Seed	Unique Methods Seed	Execution Relations Seed
Método: commitTransaction() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	99.0%	X	X	X
Método: rollbackTransaction() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	99.0%	X	X	X
Método: releaseCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	99.0%	X	X	X
Método: Object getCommunicationChannel() **Clase: IPersistenceMechanism** **Paquete: lib.persistence**	99.0%	X	X	X
Método: beginTransaction() **Clase: IPersistenceMechanism** **Paquete: : lib.persistence**	99.0%	X	X	X
Método: close () **Clase: IteratorDsk** **Paquete: lib.util**	99.0%	X	X	X

Método: validaData(int,int,int) **Clase: Date** **Paquete: lib.util**	99.0%	X	X	X
Método: getPm() **Clase: HealthWatcherFacadeInit** **Paquete: healthwatcher.business**	66.0%	X	-	X
Método: getCodigo() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	66.0%	X		X
Método: errorPage(String): **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: errorPageAdministrator(String) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: errorPageQueries(String) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: htmlPage(String,String) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: htmlPageAdministrator(String,String) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: htmlPage(String,String,int) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: hasNext() **Clase: IteratorDsk** **Paquete: lib.util**	66.0%	X	-	X
Método: next() **Clase: IteratorDsk** **Paquete: lib.util**	66.0%	X	-	X
Método: insert(Address) **Clase: AddressRepositoryRDB** **Paquete: healthwatcher.data.rdb**	66.0%	X	-	X

Método: search(int) **Clase: AddressRepositoryRDB** **Paquete: healthwatcher.data.rdb**	66.0%	X	-	X
Método: search(int) **Clase: HealthUnitRepositoryRDB** **Paquete: healthwatcher.data.rdb**	66.0%	X	-	X
Método: getPm() **Clase: HealthWatcherFacade** **Paquete: healthwatcher.business**	66.0%	X	-	X
Método: getCommunicationChannel(boolean) **Clase: PersistenceMechanism** **Paquete: lib.persistence**	66.0%	X	-	X
Método: open(String) **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: getLogin() **Clase: HTMLCode** **Paquete: healthwatcher.model.employee**	66.0%	X	-	X
Método: closeAdministrator () **Clase: HTMLCode** **Paquete: lib.util**	66.0%	X	-	X
Método: getCode() **Clase: Address** **Paquete: healthwatcher.model.address**	66.0%	X	-	X
Método: getDescricao() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	66.0%	X	-	X
Método: getSituacao() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	66.0%	X	-	X
Método: getOccurenceLocalAddress() **Clase: AnimalComplaint** **Paquete: healthwatcher.model.complaint**	66.0%	X	-	X
Método: getCode() **Clase: HealthUnit** **Paquete: healthwatcher.model.healthguide**	66.0%	X	-	X

Método: getDescription() **Clase: HealthUnit** **Paquete: healthwatcher.model.healthguide**	66.0%	X	-	X
Método: getCodigo() **Clase: MedicalSpeciality** **Paquete: healthwatcher.model.healthguide**	66.0%	X	-	X
Método: getDescricao() **Clase: MedicalSpeciality** **Paquete: healthwatcher.model.healthguide**	66.0%	X	-	X

Tabla VI - 3. Resultados Sinergia II.

La Tabla VI – 4 presenta la correspondencia entre los seeds candidatos y los concerns involucrados con cada seed. A continuación se hará una explicación para cada uno de ellos, omitiendo los descriptos en Sinergia I. Los resultados de Sinergia II amplían los seeds candidatos obtenidos en Sinergia I, ya que disminuir los umbrales de cada algoritmo tiene como consecuencia ampliar el espacio de solución de cada uno de ellos en particular.

Seed Candidatos	Concern Asociado
Método: close () **Clase: IteratorDsk** **Paquete: lib.util**	Falso Positivo
Método: validaData(int,int,int) **Clase: Date** **Paquete: lib.util**	Falso Positivo
Método: errorPageAdministrator(String) **Clase: HTMLCode** **Paquete: lib.util**	Falso Positivo
Método: errorPageQueries(String) **Clase: HTMLCode** **Paquete: lib.util**	Gestión de Excepciones
Método: htmlPage(String,String) **Clase: HTMLCode** **Paquete: lib.util**	Falso Positivo
Método: htmlPageAdministrator(String,String)	Falso Positivo

Clase: HTMLCode **Paquete: lib.util**	
Método: htmlPage(String,String,int) **Clase: HTMLCode** **Paquete: lib.util**	Falso Positivo
Método: hasNext() **Clase: IteratorDsk** **Paquete: lib.util**	Falso Positivo
Método: next() **Clase: IteratorDsk** **Paquete: lib.util**	Falso Positivo
Método: insert(Address) **Clase: AddressRepositoryRDB** **Paquete: healthwatcher.data.rdb**	Falso Positivo
Método: search(int) **Clase: AddressRepositoryRDB** **Paquete: healthwatcher.data.rdb**	Persistencia (Acceso Bajo Demanda)
Método: search(int) **Clase: HealthUnitRepositoryRDB** **Paquete: healthwatcher.data.rdb**	Persistencia (Acceso Bajo Demanda)
Método: getPm() **Clase: HealthWatcherFacade** **Paquete: healthwatcher.business**	Persistencia (Control de Acceso)
Método: getCommunicationChannel(boolean) **Clase: PersistenceMechanism** **Paquete: lib.persistence**	Falso Positivo
Método: getLogin() **Clase: Employee** **Paquete: healthwatcher.model.employee**	Falso Positivo
Método: open(String) **Clase: HTMLCode** **Paquete: lib.util**	Falso Positivo
Método: closeAdministrator () **Clase: HTMLCode** **Paquete: lib.util**	Falso Positivo
Método: getCode()	Falso Positivo

Clase: Address **Paquete: healthwatcher.model.address**	
Método: getDescricao() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	Falso Positivo
Método: getSituacao() **Clase: Complaint** **Paquete: healthwatcher.model.complaint**	Falso Positivo
Método: getOccurenceLocalAddress() **Clase: AnimalComplaint** **Paquete: healthwatcher.model.complaint**	Falso Positivo
Método: getCode() **Clase: HealthUnit** **Paquete: healthwatcher.model.healthguide**	Falso Positivo
Método: getDescription() **Clase: HealthUnit** **Paquete: healthwatcher.model.healthguide**	Falso Positivo
Método: getCodigo() **Clase: MedicalSpeciality** **Paquete: healthwatcher.model.healthguide**	Falso Positivo
Método: getDescricao() **Clase: MedicalSpeciality** **Paquete: healthwatcher.model.healthguide**	Falso Positivo

Tabla VI - 4. Seeds candidatos y concerns asociados.

3.2.1. Seeds Candidatos: Métodos Utilitarios

Bajo este título se agrupan los métodos utilitarios que la herramienta reporta como seed, aunque corresponden a falsos positivos (Tabla VI - 5). Estos seeds son en realidad métodos que presentan funcionalidades utilitarias de la aplicación, como por ejemplo iterar sobre una colección de objetos o funciones referidas a la presentación de los datos.

Método	Clase	Paquete
close ()	IteratorDsk	lib.util
hasNext()	IteratorDsk	lib.util

next()	IteratorDsk	lib.util
validaData(int,int,int)	Date	lib.util
errorPageAdministrator(String)	HTMLCode	lib.util
htmlPage(String,String)	HTMLCode	lib.util
htmlPageAdministrator(String,String)	HTMLCode	lib.util
htmlPage(String,String,int)	HTMLCode	lib.util
open(String)	HTMLCode	lib.util
closeAdministrator ()	HTMLCode	lib.util

Tabla VI - 5. Métodos utilitarios.

3.2.2. Seed Candidato: *errorPageQueries(String)*

Este método es ejecutado desde los Servlets al reportarse la excepción *ObjectNotFoundException.* En consecuencia corresponde a la lógica del manejo de excepciones y debe ser refactorizado en un aspecto.

3.2.3. Seeds Candidatos: *search(int), AddressRepositoryRDB y search(int), HealthUnitRepositoryRDB*

Los métodos *search(int)* de la clase *AddressRepositoryRDB, y search(int)* de la clase *HealthUnitRepositoryRDB* tienen un valor de confianza del 66%,. El enfoque de Unique Methods no los reporta ya que no cumplen con la restricción de poseer un tipo de retorno void.

El objetivo de estos métodos es obtener información según el valor entero pasado como parámetro. Los llamados a los mismos provienen de los servlets *ServletUpdateHealthUnitSearch* y *ServletSearchComplaintData,* pasando por clases intermedias, incluidos los facades. Este comportamiento se repite en los distintos tipos de repositorios, y servlets que presentan información al usuario.

Ambos métodos corresponden al concern de acceso bajo demanda, específicamente a la funcionalidad de acceso bajo demanda explicado previamente. El parámetro utilizado en la búsqueda no tiene relación con la lógica del método *search*, sino que se adiciona para implementar este tipo de acceso. Por esta razón, se separa el concern en el aspecto *ParameterizedLoading,* el cual intercepta los llamados a este tipo de métodos.

3.2.4. Seed Candidato: *getCommunicationChannel(boolean)*

El método tiene como objetivo obtener un canal de comunicación con la base de datos. El seed reportado es un falso positivo debido a que este método es utilizado por otros métodos con funcionalidad relacionada.

3.2.5. Seed Candidato: *getPm()*

El método getPm() de HealthWatcherFacade lleva a cabo la creación e inicialización de la instancia del mecanismo de persistencia. El mismo se corresponde con el concern de persistencia. Esto ha sido expuesto en Sinergia I, para el método de igual nombre perteneciente a la clase HealthWatcherFacadeInit.

3.2.6. Seed Candidato: *insert(Address)*

El método *insert(Address)* pertenece a la clase *AddressRepositoryRDB* y tiene como objetivo realizar la inserción de una dirección (Address) en el repositorio. Esta funcionalidad es propia de la clase a la que pertenece. En consecuencia, no corresponde a la implementación de un crosscutting concern. El seed reportado es un falso positivo.

3.2.7. Seeds Candidatos: métodos getters

La Tabla VI – 6 resume los métodos getters reportados por este segundo análisis como seeds candidatos. Todos ellos son falsos positivos debido a que son métodos propios de cada clase utilizados para consultar su estado.

Método	Clase	Paquete
getCode()	**Address**	**healthwatcher.model.address**
getDescricao()	**Complaint**	**healthwatcher.model.complaint**
getSituacao()	**Complaint**	**healthwatcher.model.complaint**
getOccurenceLocalAddress()	**AnimalComplaint**	**healthwatcher.model.complaint**
getCode()	**HealthUnit**	**healthwatcher.model.healthguide**
getDescription()	**HealthUnit**	**healthwatcher.model.healthguide**
getCodigo()	**MedicalSpeciality**	**healthwatcher.model.healthguide**
getDescricao()	**MedicalSpeciality**	**healthwatcher.model.healthguide**
getLogin()	**Employee**	**healthwatcher.model.employee**

Tabla VI - 6. Métodos getters reportados por Sinergia II.

3.2.8. Evaluación del Experimento

A continuación, se presentan algunas métricas a partir de los resultados de la segunda ejecución del análisis de Sinergia bajo los parámetros previamente mencionados.

- **Número de seeds reportados**: 33.
- **Número de seeds confirmados**: 10.
- **Número de falsos positivos**: 23.
- **Número de falsos negativos**: 2.
- **Precisión**: 10 / 33 = 0.3030.
- **Recall**: 4/ 6 = 0,6666.

3.3. Análisis de Métodos Redireccionadores

La Tabla VI – 7 presenta los resultados provenientes de la ejecución del análisis Métodos Redireccionadores. Se seleccionan las clases candidatas que devuelven un porcentaje de métodos redireccionadores igual o mayor a 50%. A continuación se realiza un análisis de los candidatos encontrados, se muestran en la tabla sólo aquellos que al menos tienen 3 métodos redireccionadores. Este filtro se decidió aplicar dado que sin él se obtiene un gran número de seeds candidatos con tan sólo uno o dos métodos redireccionadores, que no corresponden en general a estructuras de tipo adapter o decorador, resultando en falsos positivos.

Seed Candidatos	Confianza	Cant. Métodos	% de Met. Redirecc
Clase: HealthWatcherFacade **Paquete:** healthwatcher.business	**Clase:** HealthWatcherFacadeInit **Paquete:** healthwatcher.business	16	69,56%
Clase: ComplaintRecord **Paquete:** healthwatcher.business.complaint	**Clase:** IComplaintRepository **Paquete:** healthwatcher.data	3	50%
Clase:HealthUnitRecord **Paquete:** healthwatcher.business.healthguide	**Clase**:IHealthUnitRepository **Paquete:** healthwatcher.data	4	57,1%

Tabla VI - 7. Seeds reportados por el análisis Métodos Redireccionadores.

La Tabla VI – 8 muestra las relaciones entre las clases identificadas como redireccionadoras y aquellas a donde redireccionan. Luego se presenta un análisis para cada una de estas seeds.

Clase Base	Clase Redireccionada	Propósito
Clase: HealthWatcherFacade **Paquete:** healthwatcher.business	**Clase:** HealthWatcherFacadeInit **Paquete:** healthwatcher.business	Implementación del patron decorator
Clase: ComplaintRecord **Paquete:** healthwatcher.business.complaint	**Clase:** IComplaintRepository **Paquete:** healthwatcher.data	Implementación del patrón wrapper
Clase:HealthUnitRecord **Paquete:** healthwatcher.business.healthguide	**Clase**:IHealthUnitRepository **Paquete:** healthwatcher.data	Implementación del patrón wrapper

Tabla VI - 8. Propósito de redirecciones de las clases reportadas como Seeds.

3.3.1. Seed Candidato: clase HealthWatcherFacade

La clase *HealthWatcherFacade* está destinada a ser el punto de entrada a los servicios de la aplicación. Sin embargo, al realizar un análisis en profundidad, se puede ver que la misma implementa el patrón Decorator. Esta clase cuenta con una instancia de la clase *HealthWatcherFacadeInit* a la cual redirecciona un 69,56% de sus métodos (16 de 23). La clase *HealthWatcherFacade* tiene como objetivo implementar el concern de distribución, convirtiendo los servicios provistos por *HealthWatcherFacadeInit* en servicios remotos.

La versión orientada a aspectos refactoriza este concern utilizando el aspecto ServerSideHWDistribution, el cual permite la conexión remota de los métodos, y elimina la clase *HealthWatcherFacadeInit,* dejando solo la clase *HealthWatcherFacade.*

3.3.2. Seeds Candidatos: clases Records

Las clases de la Tabla VI – 9 implementan el patrón adapter para los repositorios de datos. Las clases adaptadoras forman parte de la capa de negocio y brindan una interfaz de acceso que encapsula los repositorios y presenta los servicios de acuerdo a una interfaz específica.

Clase	Paquete
DiseaseRecord	healthwatcher.business.complaint
MedicalSpecialityRecord	healthwatcher.business.healthguide
EmployeeRecord	healthwatcher.business.employee
ComplaintRecord	healthwatcher.business.complaint
HealthUnitRecord	healthwatcher.business.healthguide

Tabla VI - 9. Clases adpaters.

Se puede ver, que el patrón adapter se encuentra diseminado en dos clases, y que si se necesita realizar un cambio se deberían modificar ambas clases. Si se utiliza una

implementación orientada aspectos para encapsular este comportamiento, la implementación aumentaría la reusabilidad y centralizaría el código en un aspecto.

3.3.3. Evaluación del experimento

A continuación, se presentan algunas métricas a partir de los resultados de la ejecución del análisis de Redirector Methods de la herramienta bajo los parámetros previamente mencionados.

- **Número de seeds reportados**: 3.
- **Número de seeds confirmados**: 3.
- **Número de falsos positivos**: 0.
- **Número de falsos negativos**: 5.
- **Precisión**: 3 / 3 = 1.
- **Recall**: 1 / 6 = 0,1666.

4. Análisis de los Resultados

En esta sección, se presenta una comparación de los resultados obtenidos de la ejecución de las técnicas utilizadas. Luego se muestra una comparativa entre los concerns existentes en el sistema y los encontrados por cada una.

4.1. Comparación entre Técnicas

Los análisis de Sinergia difieren en el valor de umbral seleccionado para cada algoritmo. El establecimiento de los valores de umbral impacta directamente sobre la cantidad de crosscutting concerns identificados. Si se disminuyen los valores de umbral, trae como consecuencia que la magnitud de la solución aumente, dando lugar a un número

mayor de falsos positivos, lo que reduce la precisión del algoritmo. A su vez, el número de crosscutting concerns identificados aumenta, elevando el valor de recall de la herramienta.

El primero de los análisis exhibe mayor precisión ya que se definen valores de umbrales relativamente altos para el sistema, un valor de 10 para cada uno de los algoritmos. Por otra parte, Sinergia II reduce los valores de umbral a 4 para cada algoritmo. En consecuencia, se obtiene mayor cantidad de aspectos candidatos, a costa de encontrar más falsos positivos. En el Gráfico VI – 1 y la Tabla VI – 10 se presentan los valores de seeds confirmadas y falsos positivos de ambos enfoques. Se puede notar que el valor de falsos positivos aumenta considerablemente para el segundo análisis, y no así el valor de seeds confirmadas. El costo/beneficio que acarrea disminuir el umbral y analizar manualmente los seeds no es favorable.

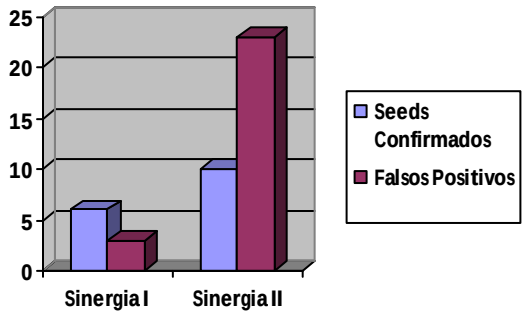

Gráfico VI - 1. Resultados Sinergia I y II.

	Sinergia I	Sinergia II
Seeds Confirmadas	6	10
Falsos Positivos	3	23

Tabla VI - 10. Resultados Sinergia I y II.

El Gráfico VI - 2 y la Tabla VI - 11 reflejan los resultados obtenidos luego de realizar el análisis de métodos redireccionadores sobre el sistema Health Watcher. Estos resultados no pueden ser comparados directamente con los obtenidos en los enfoques anteriores debido a que esta técnica devuelve concerns a nivel de clases, y no a nivel de métodos.

Los resultados anteriores reflejan la precisión de cada algoritmo (Tabla VI – 12). La precisión de los mismos se calcula como seedsConfirmados/seedsReportados.

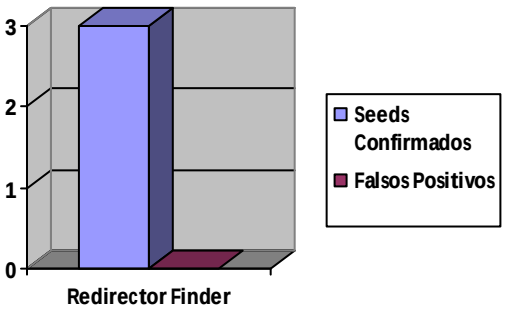

Gráfico VI - 2. Resultados Métodos Redireccionadores.

	Métodos Redireccionadores
Seeds Confirmadas	3
Falsos Positivos	0

Tabla VI - 11. Resultados de Métodos Redireccionadores.

	Sinergia I	Sinergia II	Métodos Redireccionadores
Seeds Confirmadas	6	10	3
Seeds Reportados	8	33	3
Precisión	0,75	0,3030	1

Tabla VI - 12. Precisión por algoritmo.

4.2. Concerns Existentes y Concerns Reportados por las Técnicas

El sistema Health Watcher presenta 4 tipos de concerns implementados: distribución, persistencia, concurrencia y gestión de excepciones. El concern de persistencia se puede subdividir en 3 concerns más específicos: mecanismo de control de persistencia, mecanismo de control de transacciones y acceso a datos bajo demanda [8, 10]. A continuación, se presenta un análisis de los concern identificados por cada uno de los enfoques. En la Tabla VI – 13 se reflejan los seeds confirmados de cada algoritmo, los falsos negativos y el valor de recall. Este último se calcula como concernEncontrados/concernsExistentes.

- **Sinergia I**: detecta los concerns de control de persistencia, control de transacciones y gestión de excepciones. El resto de los concerns caen en la

definición de falsos positivos. Esto se debe a dos razones. En primer lugar, el concern de acceso a datos no está contemplado debido a los valores de umbral seleccionados. El resto no son contemplados debido a la naturaleza de seeds que reporta este enfoque, los cuales son aquellos métodos usados desde otros métodos. Los concerns de distribución y concurrencia no están implementados de tal forma. Por ejemplo, para el último de estos, el control en la concurrencia se logra haciendo que los métodos involucrados estén sincronizados, agregando en su declaración el modificador synchronized. El Gráfico VI – 3 y la Tabla VI – 13 reflejan estos valores.

- **Sinergia II**: detecta los concerns de control de persistencia, control de transacciones, acceso a datos y gestión de excepciones. El análisis es similar al realizado en el punto anterior por Sinergia I, solo difiere en el valor de umbral seleccionado. A razón de esto se detecta el concern de acceso de datos. El Gráfico VI – 4 y la Tabla VI – 13 reflejan estos valores.

- **Métodos Redireccionadores**: detecta el concern de distribución, y reconoce patrones adapters en los cuales su refactorización a aspectos es óptima. Estos últimos no fueron encapsulados en aspectos en análisis previos del sistema [8, 10, 75]. Este análisis está destinado a encontrar patrones como adapters y decorators, es por eso que el resto de los concerns presentes en el sistema no son detectados por la técnica y se consideran como falsos negativos. El Gráfico VI – 5 y la Tabla VI – 13 reflejan los valores.

	Sinergia I	Sinergia II	Métodos Redireccionadores
Seeds Confirmados	6	10	3
Falsos Positivos	2	23	0
Recall	0,5	0,66	0,1666

Tabla VI - 13. Precisión por algoritmo.

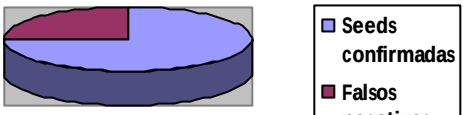

Gráfico VI - 3. Concerns Sinergia I.

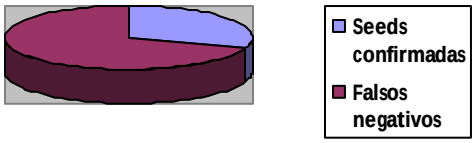

Gráfico VI - 4. Concerns Sinergia II.

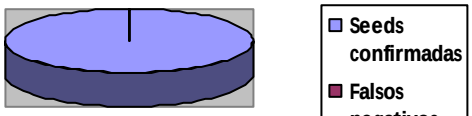

Gráfico VI - 5. Concerns Métodos Redireccionadores

Los concerns detectados por los análisis de la herramienta son refactorizados, en la versión AO de Health Watcher [8], en aspectos. A continuación, se indica para cada técnica, la correspondencia entre concerns y aspectos:

- **Sinergia I**: el control de persistencia se encapsula en el aspecto *PersistenceControlHW*, el control de transacciones en *TransactionControlHW* y la gestión de excepciones en el aspecto *ExceptionHandlingAspect*.

- **Sinergia II**: adicionalmente a los concerns reconocidos por Sinergia I, este enfoque detecta el concern de acceso a datos, el cual es refactorizado en el aspecto *ParameterizedDataLoading*.

- **Métodos Redireccionadores**: el concern de distribución se refactoriza en dos nuevos aspectos denominados *ServerSideHWDistribution* y *ClienteSideHWDistribution*.

CAPÍTULO VII

Conclusión

1. Análisis del Enfoque Propuesto

Se define a un concern como cualquier cuestión de interés en un sistema de software. Involucra todo lo que un stakeholder quiera considerar como una unidad conceptual, incluyendo características, requerimientos no funcionales y decisiones de diseño e inclusive *programming idioms* [76]. Los concerns que atraviesan los módulos correspondientes a la descomposición principal del sistema son llamados crosscutting concerns.

El enfoque propuesto tiene como objetivo asistir en la búsqueda de crosscutting concerns que pueden convertirse en potenciales aspectos en un código fuente. De esta manera, se puede realizar una futura refactorización con el fin de mejorar el entendimiento y mantenimiento del sistema.

SAT (Synergy Analysis Tool) es una herramienta desarrollada como un plugin para la plataforma Eclipse. La misma analiza el código fuente de un sistema desarrollado en Java utilizando técnicas de aspect mining estáticas con el fin de identificar los crosscutting concerns presentes en el sistema. Se implementaron 5 técnicas, cada una constituyendo un sistema experto. Las técnicas implementadas son: análisis de Fan-in [25], análisis de Métodos Únicos [55], análisis de Relaciones de Ejecución [68], análisis de Métodos Redirectores [44] y un 5to enfoque que converge a los 3 primeros.

La herramienta fue probada con el sistema Health Watcher (HW) [8]. HW es una aplicación desarrollada acorde a una arquitectura por capas utilizando tecnología J2EE, y el

propósito principal del mismo es permitir a ciudadanos registrar sus quejas referidas a temas de salud. El mismo es seleccionado debido a que es un sistema real y suficientemente complejo. El sistema involucra un gran número de concerns clásicos como por ejemplo sistemas usables, concurrentes, persistentes y distribuidos. Adicionalmente, la aplicación utiliza tecnologías utilizadas en contextos industriales como RMI (Java Remote Method Invocations) [17], Servlets [63] y JDBC (Java Database Connectivity).

Se realizaron 3 análisis sobre dicho sistema, dos correspondiente al enfoque denominado sinergia (variando sus parámetros de entrada) y un tercer análisis utilizando métodos redireccionadores. Los concerns identificados se compararon con los concerns identificados en trabajos previos [8]. El enfoque Sinergia identificó el 50% y 66.66% de los concerns. El análisis de métodos redirectores identificó el 16.66% de los crosscutting y agregó al conjunto la identificación de patrones adapters diseminados en la capa de negocio.

Las técnicas implementadas por SAT obtienen sus resultados a partir de la información estática del código fuente de un sistema. Una ventaja importante que presenta esta herramienta es la flexibilidad con que se implementan los análisis, debido a que se utiliza un sistema experto para cada uno, el cual mapea el conocimiento en reglas. Este tipo de implementación permite combinar más de una técnica con facilidad (por ejemplo Sinergia). Otra ventaja importante es la rapidez con la que se ejecutan los análisis sobre todo el código fuente de una aplicación una vez que se cuenta con la información representada como hechos en la base de datos. Esto se debe a la utilización de un motor de inferencia para la ejecución de las reglas de cada algoritmo.

Una desventaja de esta herramienta reside en la naturaleza de las técnicas implementadas. Dichas técnicas identifican dos tipos de concerns: aquellos que fueron implementados como llamadas entre métodos y aquellos que fueron implementados utilizando patrones del estilo adapter o decorator. Debido a esto, los concerns que pueden ser identificados por la herramienta se reducen al grupo listado previamente.

2. Trabajos Futuros

A continuación se listan posibles proyectos que puedan extender o continuar el trabajo realizado:

- Creación de nuevos sistemas expertos con el fin de implementar nuevos algoritmos de aspect mining.
- Creación de nuevos sistemas expertos con el objetivo de combinar más de una técnica de aspect mining realizando un estudio detallado de los parámetros y las variaciones de los mismos.
- Implementación de recomendación de refactorings utilizando las salidas obtenidas y nuevas reglas de inferencia.
- Implementación de refactorings automatizados utilizando las salidas obtenidas y nuevas reglas de inferencia.

Bibliografía

1. B Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and Aspectj. In: 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 161--173. ACM Press, New York, USA (2002).
2. Ramnivas Laadad. "AspectJ in Action". ©2003 by Manning Publications Co. All rights reserved.
3. Khalid Al-Jasser, Peter Schachte, Ed Kazmierczak, "Suitability of Object and Aspect Oriented Languages for Software Maintenance," aswec, pp.117-128, 2007 Australian Software Engineering Conference (ASWEC'07), 2007 .
4. http://www.jessrules.com/
5. Joel Jones. Abstract Syntax Tree Implementation Idioms. The 10th Conference on Pattern Languages of Programs, Sep. 8th-12th, 2003.
6. Kellens, A., Mens, K. A survey of aspect mining tools and techniques. Technical report, INGI 2005-07, Universite catholique de Louvain, Belgium (2005)
7. Friedman-Hill E. (2003). "Jess in Action". Manning Publications, ISBN 1-930110-89-8.
8. Soares, S.; Borba, P.; Laureano, E.: Distribution and Persistence as Aspects. In: Software Practice and Experience, Wiley, vol. 36 (7), (2006) 711-759.
9. http://www.eclipse.org/jdt/
10. Greenwood, P.; et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP'07. LNCS, vol. 4609, Springer (2007) 176–200.
11. S. Soares, et al. "Implementing Distribution and Persistence Aspects with AspectJ". Proc. OOPSLA'02.
12. Hadi A. S., Castillo E., Guti´errez J. M.(1997), Expert Systems and Probabilistic Network Models, Springer Verlag, New York.
13. http://www.jessrules.com/
14. Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern MatchProblem," Artificial Intelligence 19 (1982): 17–37.
15. Barr, A. and Feigenbaum, E. A. (1981), The Handbook of Artificial Intelligence, Volume I. William Kaufman, Los Altos, CA.
16. [16] http://www.rae.es/rae.html
17. [17] S. Microsystems. Java Remote Method Invocation (RMI). At http://java.sun.com/products/jdk/1.2/docs/guide/rmi, 2001.
18. Agnar Aamodt , Enric Plaza, Case-based reasoning: foundational issues, methodological variations, and system approaches, AI Communications, v.7 n.1, p.39-59, March 1994
19. CLIPS (2008) http://www.ghg.net/clips/CLIPS.html
20. Leon Sterling and Ehud Shapiro, The Art of Prolog: Advanced Programming Techniques, 1994, ISBN
21. http://lazax.com/software/Mycin/mycin.html
22. McDermott, J. R1: an expert in the computer systems domain. In Proceedings of the 1st Annual National Conference on Artificial Intelligence, pages 269-271. Stanford University, 1980.
23. V. Alves and P. Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In First Latin American Conference on Pattern Languages of Programming | SugarLoafPLoP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

24. *Chiang C., An Introduction to Stochastic Processes and their Applications, R.Krieger Publishing Company, 1968*
25. *P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04). Washington, DC, USA: IEEE Computer Society, 2004, pp. 112121.*
26. *A. Marcus, R. Koschke, A. van Deursen, V. Rajlich, P. Tonella, and H. Sneed. Identification of concepts, features, and concerns in source code. Panel Discussion at the International Conference on Software Maintenance, 2005.*
27. *P. Tarr, H Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In ICSE, 1999.*
28. *Kim Mens, Andy Kellens, Jens Krinke, "Pitfalls in Aspect Mining," wcre, pp.113-122, 2008 15th Working Conference on Reverse Engineering, 2008.*
29. *T. Massoni, V. Alves, S. Soares, and P. Borba. PDC: Persistent Data Collections pattern. In First Latin American Conference on Pattern Languages of Programming | SugarLoafPLoP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.*
30. *T. Mens, T Tourwé. A Survey of Software Refactoring. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. XX, NO. Y, MONTH 2004.*
31. *M. Fowler, Refactoring: Improving the Design of Existing Programs, Addison-Wesley, 1999. [28]*
32. *M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.*
33. *G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of Aspectj," in ECOOP, ser. Lecture Notes in Computer Science, J. L. Knudsen and J. L. Knudsen, Eds., vol. 2072. Springer, 2001, pp. 327353.*
34. *Jboss Home page: http://jboss.org/jbossaop/*
35. *Spring Home page: http://static.springsource.org/spring/docs/2.5.x/reference/aop.html*
36. *M. P. Robillard and G. C. Murphy, "Representing concerns in source code," ACM Trans. Softw. Eng. Methodol., vol. 16, no. 1, pp. 3+, February 2007.*
37. *F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons, August 1996.*
38. *Kellens, A., Mens, K., Tonella, P.: A survey of automated code-level aspect mining techniques. Trans. Aspect-Oriented Software Development (2007)*
39. *Baniassad, E., Clements, P.C., Araujo, J., Moreira, A., Rashid, A., Tekinerdogan, B.: Discovering early aspects. IEEE Software 23(1) (January-February 2006) 61–70*
40. *Bass, L., Klein, M., Northrop, L.: Identifying aspects using architectural reasoning. Position paper presented at Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design, Workshop of the 3rd Int'l Conf. Aspect-Oriented Software Development (AOSD) (2004)*
41. *Robillard, M.P., Murphy, G.C.: Concern graphs: Finding and describing concerns using structural program dependencies. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press (2002) 406–416*
42. *Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving code and design with intensional views – a case study. Computer Languages, Systems and Structures 32(2–3) (July-October 2006) 140–156 Special Issue: Smalltalk.*
43. *Griswold, W., Kato, Y., Yuan, J.: Aspect browser: Tool support for managing dispersed aspects. In:Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems. (1999) [213]*

44. Marin, M.,Moonen, L., van Deursen, A.: A common framework for aspect mining based on crosscutting concern sorts. In Sim, S.E., Di Penta, M., eds.: Proc. Working Conf. Reverse Engineering (WCRE), IEEE Computer Society Press (2006) 29–38
45. Zhang, C., Jacobsen, H.A.: PRISM is research in aspect mining. In: Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM Press (2004) 20–21
46. K. MENS & T. TOURWE. Evolution Issues in Aspect-Oriented Programming. Chapter in book on "Software Evolution edited by T. Mens & S. Demeyer, pp. 197–224. Springer, 2008. ISBN 978-3-540-76439-7. DOI 10.1007/978-3-540-76440-3.
47. Lehman, M.M., Belady, L.A.: Program Evolution: Processes of Software Change. Apic Studies In Data Processing. Academic Press (1985)
48. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
49. Shepherd, D., Pollock, L. L., Tourwé, T. "Using Language Clues to Discover Crosscutting Concerns," ACM SIGSOFT Software Engineering Notes 30 (4), 1--6 2005.
50. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, desing Patterns, Addison-Wesley, 1995.
51. J. Hannemann and G. Kiczales. Overcoming the prevalent descomposition of legacy code. In Wkshp on Advances Separation of concerns, 2001.
52. J. Morris and G. Hirst. Lexical cohesion computed by theasural relations as an indicator of the structure of text. Comput. Linguist., 17(1):21-48,1991.
53. A. Budanitsky. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures, 2001.
54. Eclipse Homepage. http://www.eclipse.org. 2005.
55. Gybels, K. and Kellens, A. "Experiences with Identifying Aspects in SmallItalk Using Unique Methods," in: International Conference on Aspect Oriented Software Development. Amsterdam, The Netherlands 2005.
56. D. Shepherd and L. Pollock. Interfaces, aspects and views. In Linking Aspect Technology and Evolution (LATE) Workshop, 2005.
57. T. Tourwé and K. Mens. Mining Aspectural Views using Formal Concept Analysis. In Proceedings of the 4th International Workshop on Source code Analysis and Manipulation (SCAM), pages 97 – 106. IEEE Computer Science, 2004.
58. M. Marin, A. Van Deursen, and L. Moonen. "Identifying crosscutting concerns using fan-in analysis," ACM Trans. Softw. Eng. Methodol., vol. 17, no. 1, pp. 1-37, December 2007.
59. I. Sommerville, Software Engineering (7th Edition) (International Computer Science Series). Addison Wesley, May 2004.
60. D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In International Conference on Software Engineering Research and Practice, 2004.
61. Tomcat homepage, http://jakarta.apache.org/tomcat/.
62. Breu, S., Krinke, J. Aspect Mining Using Event Traces. In: 19th IEEE International Conference on Automated Software Engineering, pp. 310--315. IEEE Computer Society, Washington DC, USA (2004).
63. Hunter, J., Crawford, W.: Java Servlet Programming. O'Reilly and Associates Inc. 1998
64. B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Spring-Verlag, 1999.
65. Dynamo homepage, http://star.itc.it/dynamo/
66. Tourwe, T., Kim Mens, K. Mining Aspectual Views using Formal Concept Analysis. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 97—106. (2004)
67. M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonello, and T. Tourw´e. A qualitative comparison of three aspect mining techniques. In International Workshop on Program Comprehension (IWPC), 2005.

68. *J. Krinke. Mining Control Flow Graphs for Crosscutting Concerns. In Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), pages 334–342, Washington, DC, USA, 2006. IEEE Computer Society.*
69. *R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In Proc. CASCON, 1999.*
70. *J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Addison-Wesley, second edition, 2000.*
71. *Hannemann, J., Kiczales, G.: Overcoming the Prevalent Decomposition of Legacy Code. In:Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering. Toronto, Ontario, Canada, (2001)*
72. *R. Elmasri and S. Navathe. Fundamentals of Database Systems. Addison{Wesley, second edition, 1994.*
73. *http://java.sun.com/javase/technologies/database/*
74. *http://java.sun.com/j2ee/1.4/docs/tutorial/doc/*
75. *Uirá Kulesza, Cláudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, Carlos Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," icsm, pp.223-233, 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006*
76. *Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of Aspectj. In: Knudsen, J. L., Knudsen, J. L. (eds.) ECOOP 2001. LNCS, vol.2072, pp. 327--353, Springer (2001)*
77. *A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," ACM Comput. Surv., vol. 31, no. 3, pp. 264323, September 1999.*
78. *B. Ganter and R. Wille, "Applied lattice theory: Formal concept analysis," in In General Lattice Theory, G. Grätzer editor, Birkhäuser, 1997.*
79. *S. Soares. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, 2004.*
80. *David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to find and understand action-oriented concerns. In Int. Conf. on Aspect-oriented Software Development, 2007.*