CAPITULO V

Propuesta

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 [5.12]. 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 [15].

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 [12]. 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 los análisis 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.

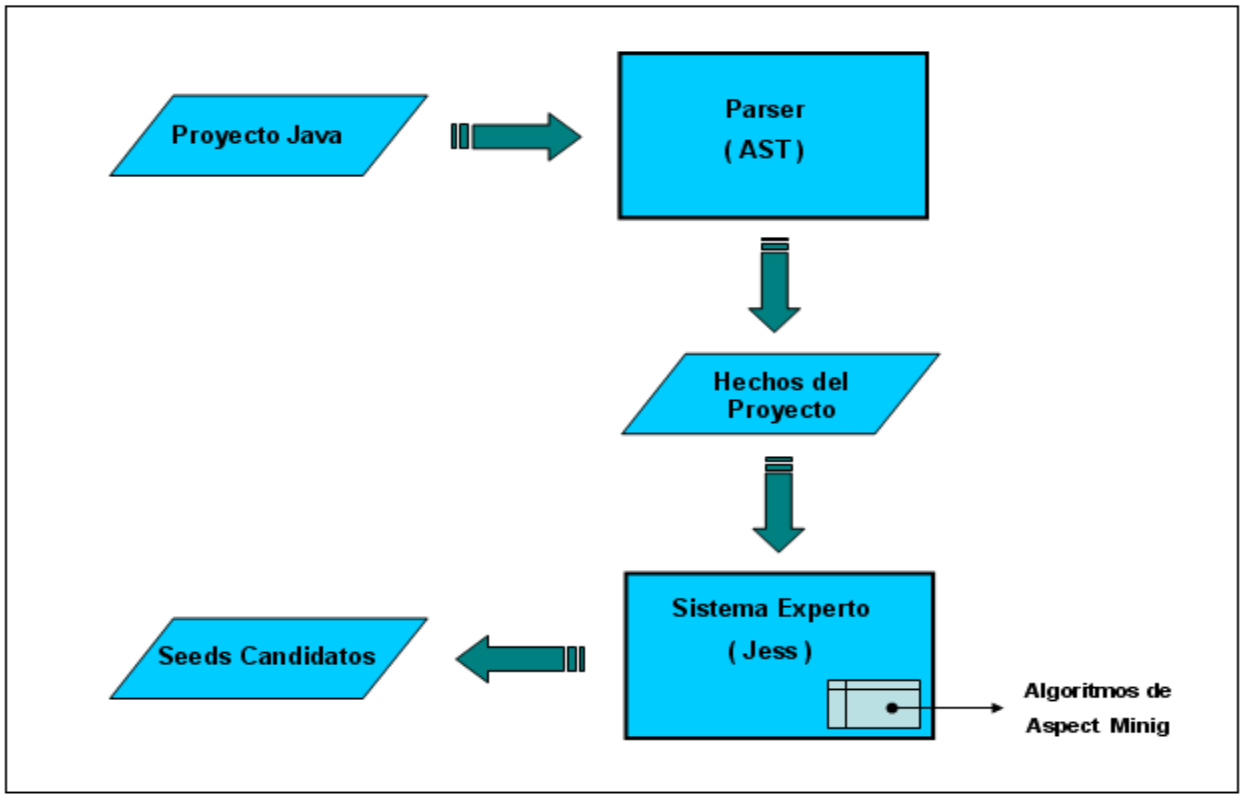

Fig. V - 1. Proceso de extracción de crosscutting concerns.

- **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 implementados corresponden a las siguientes técnicas: Análisis de Fan-in [10.10], Análisis de Métodos únicos [7.7], Análisis de Clases Redireccionadoras [22.22], Análisis

de Relaciones de Ejecución sobre el Grafo de Llamadas Estático [20.20] 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 permita 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) [3]. 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 [13] 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 que se puede apreciar 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

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

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 [6]). 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)

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, Anásls 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,	*caller_id*: identificador del método llamador.	El método identificado por *caller_id*

precedence, id)	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.	*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.

```
(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.

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.

- **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.

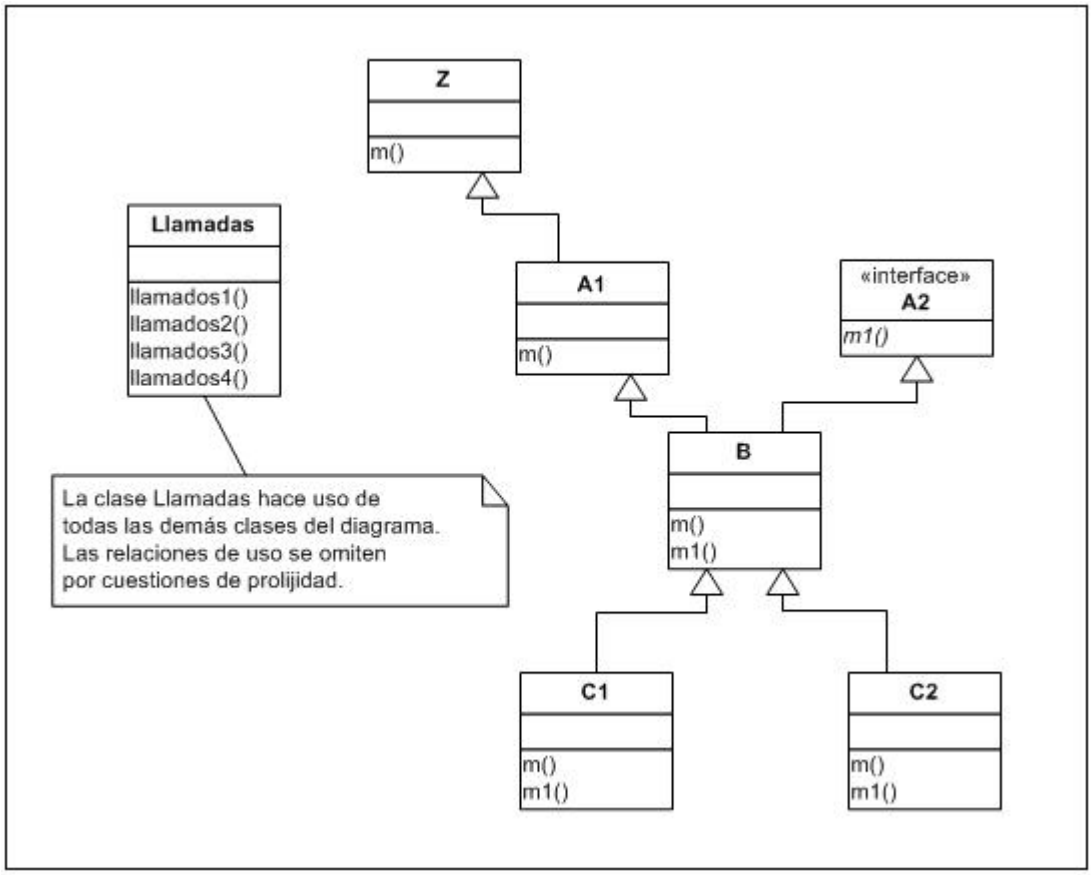

Fig. V - 11. Diagrama de clases para ejemplo de Fan-in.

	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 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 presentas 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 simétrico 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 resultados 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 Unique Methods.

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.

- $G \epsilon_{\top} C$ 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, el hecho 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	El método identificado por method_id participa de tantas relaciones del tipo InsideFirstExecution como indica metric.

	de ejecución.	
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, 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 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 Unique Methods.

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 que 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 queries 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 Execution Relations.

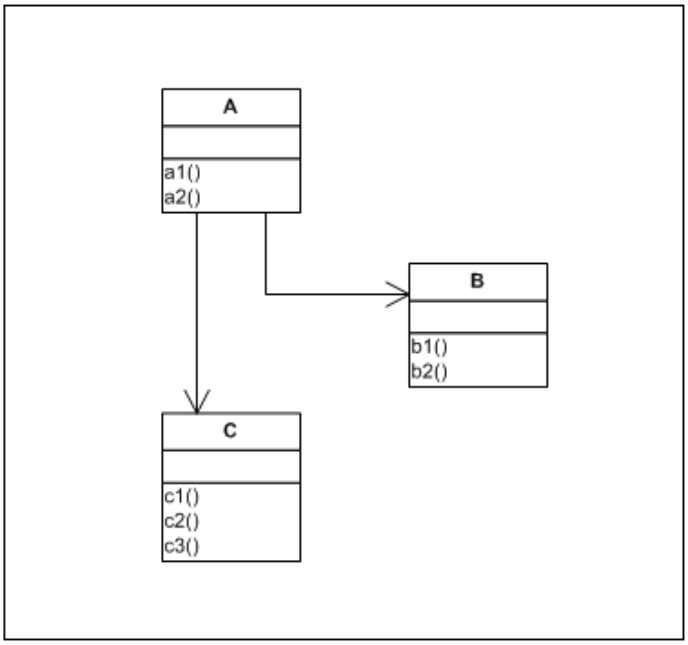

Fig. V - 22. Diagrama de clases para el ejemplo de Execution Relations.

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, podemos ver 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. Podemos ver 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**

Podemos 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 podemos ver 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 Execution Relations.

5.4. Redirector Finder

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 Redirector Finder 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 redirectores 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 Redirector Finder.

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 redirector methods. 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.

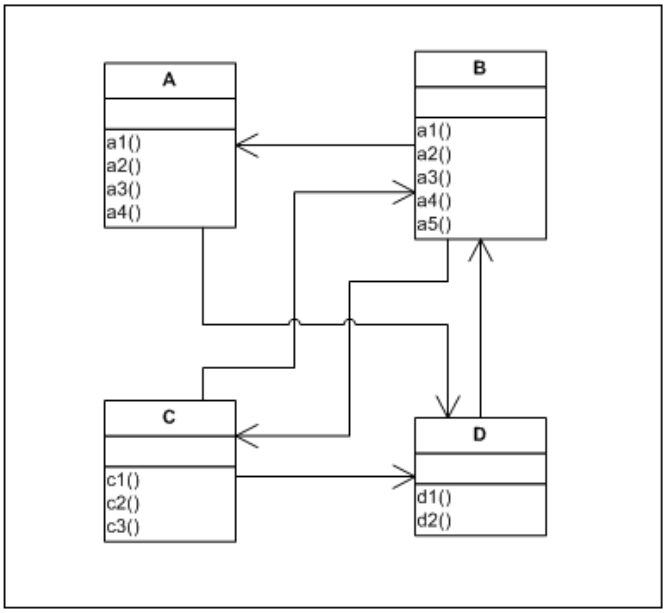

Fig. V - 27. Diagrama de clases del ejemplo.

	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 ejecuta las reglas del algoritmo Redirector Finder.

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 redirectores 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 Redirector Finder. 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 -1. Salidas para ejemplo de Redirector Methods.

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. Este análisis conjunto aporta mayor grado de fiabilidad a los resultados finales. Se deben especificar 7 parámetros de entrada, entre ellos tres umbrales y cuatro niveles de confianza. Los umbrales son valores que se definen 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 cuan confiables son los resultados que arrojan, y en consecuencia, cuanto aportará cada algoritmo a la solución final. 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 Execution Relations.

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 Unique Methods.

```
(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 Execution Relations.

- **Acumulación de la confianza en Fan-in:** la regla definida en la Fig. V – 31 toma cada una de las seeds que el algoritmo de Fan-in considera como tal (*fan-in_seed*) y suma el nivel de confianza de este algoritmo 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 el algoritmo de Fan-in.

```
(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.

- **Acumulación de la confianza en Métodos Únicos:** la regla definida en la Fig. V – 32 toma cada una de las seeds que el algoritmo de Métodos Únicos considera como tal (*unique_method_seed*) y suma el nivel de confianza de este algoritmo al nivel de certeza general que se tiene sobre el seed. Así, el atributo *trust* del hecho *seed* se ve incrementado por el valor de confianza que se tiene sobre el algoritmo de Métodos Únicos.

```
(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 Unique Methods en los seeds.

- **Acumulación de la confianza en Relaciones de Ejecución:** la regla definida en la Fig. V – 33 toma cada una de las seeds que el algoritmo de Relaciones de Ejecución considera como tal (*execution_relation_seed*) y suma el nivel de confianza de este algoritmo al nivel de certeza general que se tiene sobre el mismo. Así, el atributo *trust* del hecho *seed* se ve incrementado por el valor de confianza que se tiene sobre el algoritmo de Relaciones de Ejecución.

- **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 Execution Relations 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.

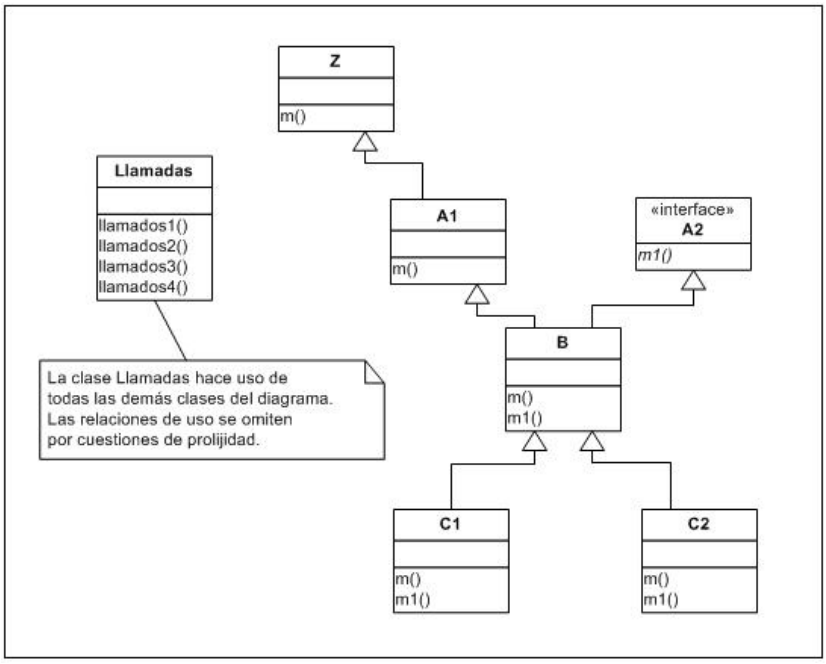

Fig. V - 36. Diagrama de clases para ejemplo de Fan-in.

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 Únicos	Relaciones de 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. Valores de confianza y umbral.

6. Aspect Mining Tool

Aspect Mining Tool es una herramienta implementada como un plugin para eclipse [12] 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).

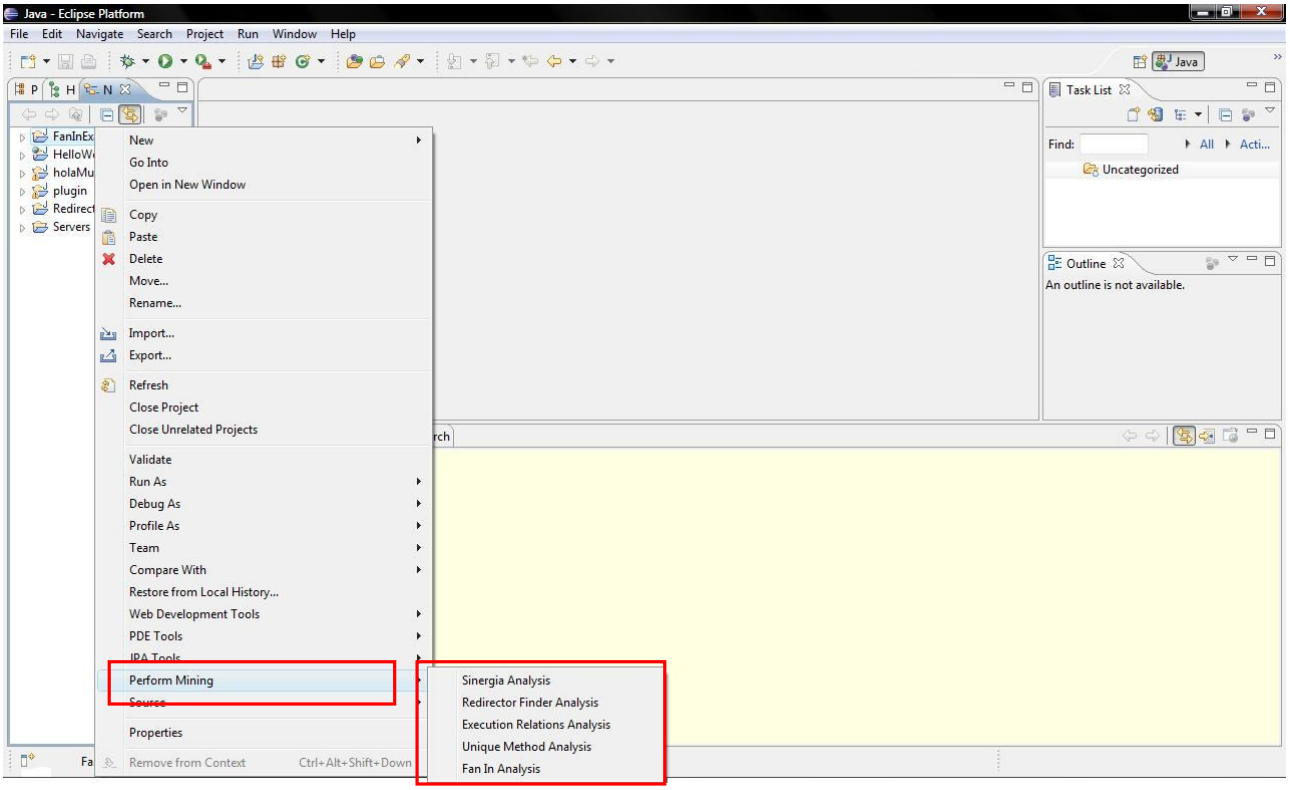

Fig. V - 37. Análisis ofrecidos por la Aspect Mining Tool.

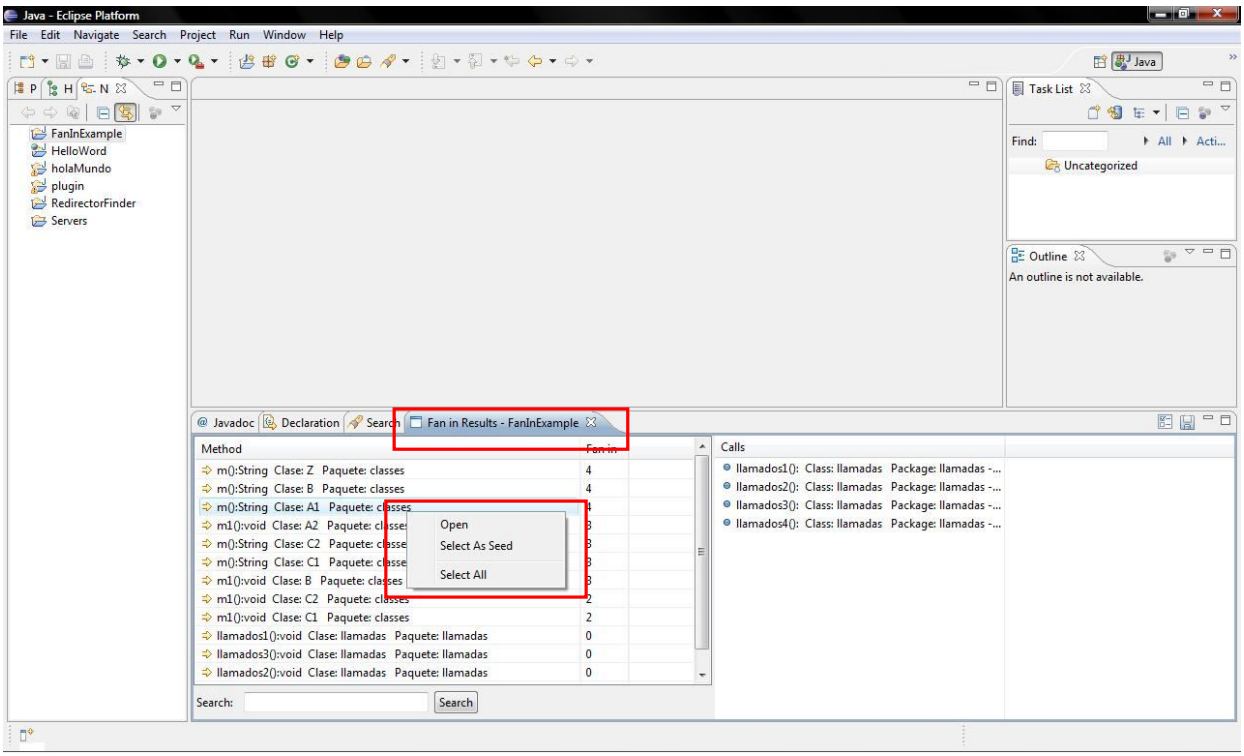

Fig. V - 38. Vista de Fan-in.

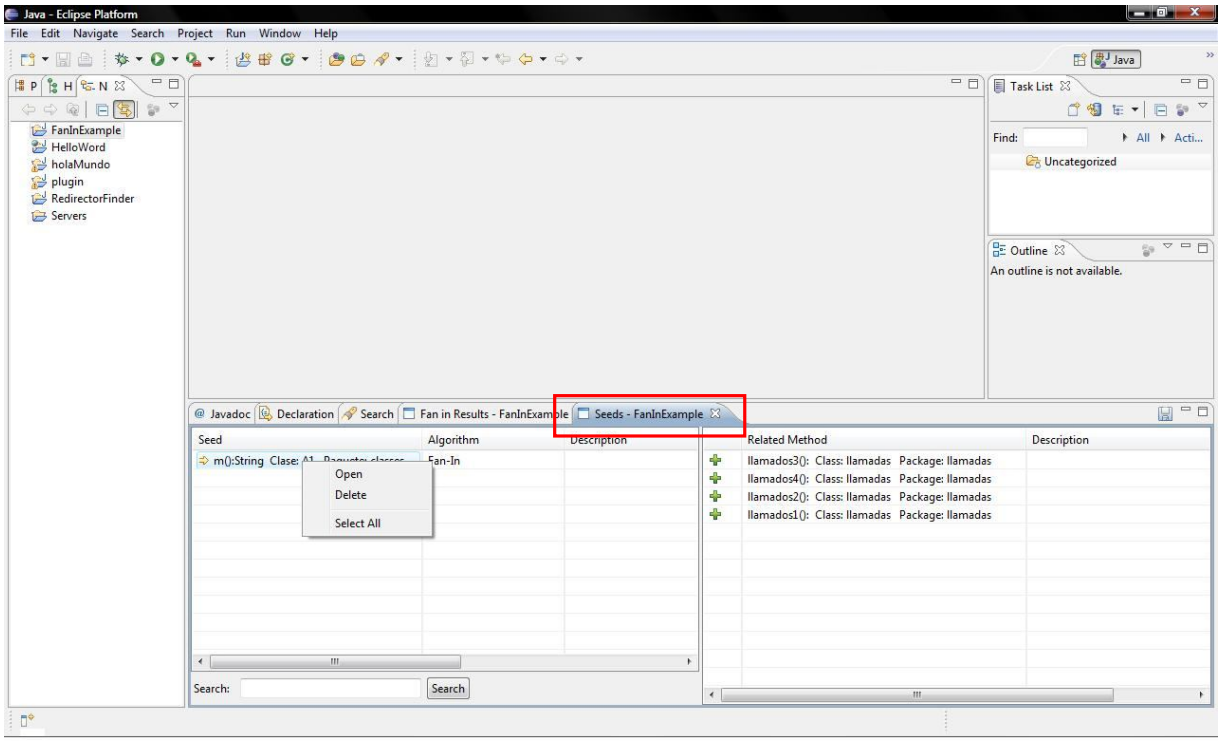

Fig. V - 39. Vista de Seeds a nivel de métodos.

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.

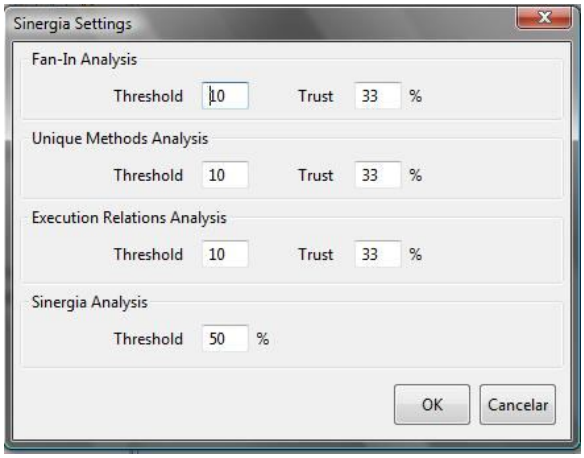

Fig. V - 40. Pantalla de Configuración de análisis de Sinergia

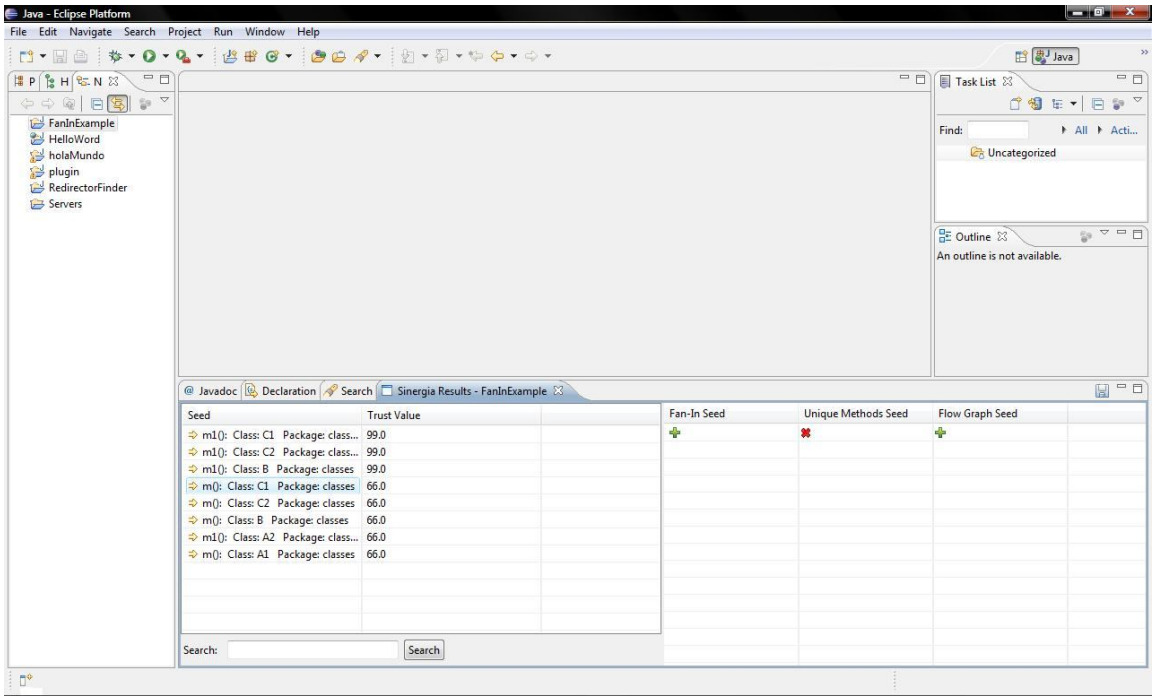

Fig. V - 41. Vista de resultados provenientes del análisis Sinergia.

Referencias

[1] Joel Jones. Abstract Syntax Tree Implementation Idioms. The 10th Conference on Pattern Languages of Programs, Sep. 8th-12th, 2003.

[2] Sistemas Expertos y Modelos de Redes Probabil´ısticas - Enrique Castillo, Jos´e Manuel Guti´errez, y Ali S. Hadi

[3] Iulian Neamtiu , Jeffrey S. Foster , Michael Hicks, Understanding source code evolution using abstract syntax tree matching, Proceedings of the 2005 international workshop on Mining software repositories, p.1-5, May 17-17, 2005, St. Louis, Missouri

[4] http://www.jessrules.com/

[5] Charles L. Forgy, “Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem,” Artificial Intelligence 19 (1982): 17–37.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, desing Patterns, Addison-Wesley, 1995.

[12] http://www.eclipse.org/

[13] http://www.eclipse.org/jdt/

[14] 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.

[5.12] [12] 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.

[15] Kellens, A., Mens, K. A survey of aspect mining tools and techniques. Technical report, INGI 2005-07, Universite catholique de Louvain, Belgium (2005)

[7.7] 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.

[10.10] 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.

[20.20] 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.

[22.22]Marius Marin, Leon Moonen, Arie van Deursen, "A common framework for aspect mining based on crosscutting concern sorts," wcre, pp.29-38, 13th Working Conference on Reverse Engineering (WCRE 2006), 2006