CAPITULO III

Aspect Mining: Trabajos Previos

1. Introducción

Aspect Mining es la actividad de descubrir potenciales crosscutting concerns a partir del análisis del código fuente de un sistema o desde un conjunto de trazas derivadas de su ejecución [10]. En este capítulo, se presentan los trabajos previos desarrollados en él área. Para cada una de las técnicas propuestas se describe su algoritmo, se exhibe un ejemplo y se muestra la herramienta que la implementa.

2. Conceptos de Aspect Mining

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

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

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

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

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

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

3. Procesamiento del Lenguaje Natural sobre el Código Fuente

Shepherd, Pollock y Tourwé [1] propusieron una técnica de aspect mining que intenta descubrir aspectos candidatos en el código fuente de un sistema por medio de técnicas de Procesamiento del Lenguaje Natural (PLN). Particularmente, se utiliza la técnica de lexical chaining [2] para identificar grupos de entidades de código fuente relacionadas semánticamente y evaluar cuándo estos representan crosscutting concerns.

El enfoque se basa en la hipótesis de que los crosscutting concerns generalmente están implementados mediante una rigurosa convención de nombres. Los desarrolladores recurren a estas convenciones con el fin de vincular entidades del código fuente relacionadas, y mejorar el entendimiento del software. Por ejemplo, implementaciones de patrones de diseño [3] abarcan múltiples clases y métodos que se relacionan mediante nombres bien específicos, se pueden citar nombres como “acceptVisitor” o “addObserver”. El punto clave es aplicar PLN para explotar relaciones semánticas entre palabras y entre nombres predefinidos.

La ventaja que presenta este enfoque sobre trabajos previos que utilizan convenciones de nombres [4] es que se reconocen relaciones semánticas más complejas entre palabras analizando identificadores y comentarios del código fuente para entender y comprender la semántica del mismo.

3.1. Algoritmo

El algoritmo propuesto consiste en dos pasos:

1. Obtener todas las palabras del código fuente de un programa.
2. Construir las cadenas léxicas procesando cada palabra: por cada palabra busca la cadena, por medio de lexical chaining, que esté relacionada

semánticamente y agrega la palabra a la cadena. Si no encuentra tal cadena comienza una nueva con dicha palabra. Luego de procesar cada palabra se obtiene una lista de cadenas de palabras de distintas longitudes.

3.2. Lexical Chaining

Lexical chaining es una técnica de PLN que agrupa en cadenas léxicas palabras semánticamente relacionadas de un documento [4]. El proceso de encadenamiento toma como entrada un texto y procede agrupando cada palabra en una cadena con otras palabras del mismo texto que están semánticamente relacionadas.

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

En función de automatizar el cálculo de distancia, se utiliza una base de datos de relaciones conocidas de palabras, como por ejemplo WordNet [6]. La distancia semántica se calcula midiendo la longitud de los caminos de relaciones entre dos palabras desde la base, donde a mayor distancia las palabras se encuentran menor es su relación semántica. En adición a la distancia obtenida, se debe tener en cuenta la naturaleza de la palabra, el tipo de relación puede variar en caso de que las palabras se utilicen como adjetivos, sustantivos, verbos, etc.

3.3. Utilización de Lexical Chaining para Identificar Concerns

Los autores suponen que algunas de las cadenas léxicas que se pueden derivar del código fuente se corresponden con concerns de alto nivel. Esto se debe a que los desarrolladores están forzados a utilizar pistas del lenguaje debido a la falta de estructuras adecuadas para modularizar correctamente estos crosscutting concerns. Estas pistas proveen información sobre la funcionalidad del código, por lo que si dos regiones de código

fuente contienen palabras similares, es probable que ambas regiones implementen funcionalidades relacionadas.

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

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

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

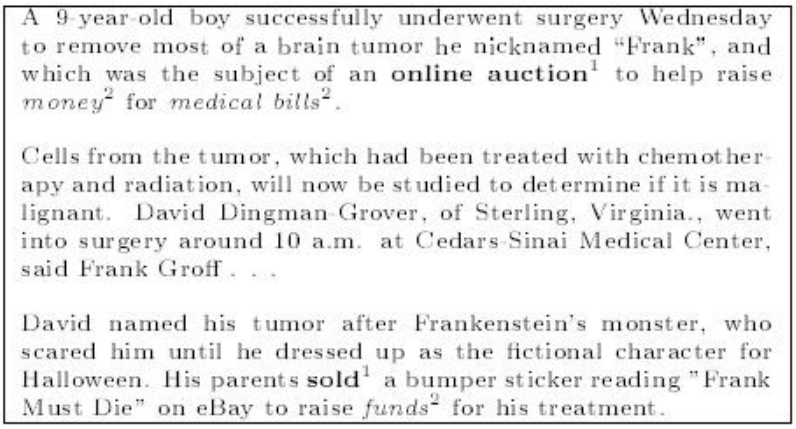
- **Comentarios:** usualmente se escriben en forma de sentencias o frases. Por esta razón, se utiliza un speech tagger, el cual etiqueta las palabras de un comentario y elimina la ambigüedad entre ellas. Por ejemplo, las palabras “dirección” y “destino” podrían relacionarse si ambas se utilizan como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

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

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

3.4. Ejemplo

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



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

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

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

Fig. III- 1. Párrafo con cadenas léxicas

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

3.5. Herramienta

Los autores implementaron la herramienta de lexical chaining como un Plug-in de Eclipse [6]. Tiene como propósito automatizar el algoritmo propuesto, incluyendo la selección de palabras y el lexical chaining. Luego de correr la aplicación, los usuarios navegan las cadenas obtenidas.

4. Detección de Métodos Únicos

Gybels y Kellens [7] utilizaron heurísticas para descubrir crosscutting concerns mediante la identificación de métodos únicos. Este enfoque intenta identificar los concerns que fueron implementados centralizados en un único método, al cual se lo invoca desde varios lugares del código fuente.

En los lenguajes no orientados a aspectos los desarrolladores implementan los crosscutting concerns de tal forma que el código resultante se encuentra disperso y entremezclado [23]. Dichos concerns son implementados de manera que el weaving que se realiza en los lenguajes orientados a aspectos es manual. Una forma de hacerlo es tener métodos centralizados para realizar tareas específicas e invocarlos de diversos lugares del código, dando lugar así a código disperso.

Un ejemplo típico es el caso del logging, en donde la funcionalidad se encuentra bien implementada mediante los mecanismos provistos por la programación orientada a objetos: una clase Singleton [3] encapsula el manejo del archivo del logging y representa con el método log el servicio mencionado. A pesar de que no se evidencia código entrelazado, el método log será llamado de diversos lugares donde se requiera registrar información (Fig. III-2).

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

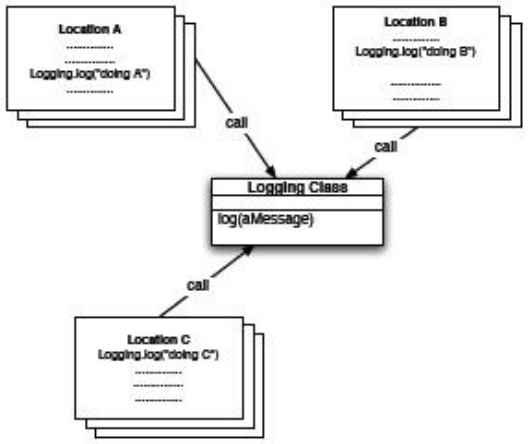


Fig. III- 2. Logging como clase central que provee la funcionalidad de loggeo.

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

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

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

4.1. Algoritmo

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

4.2. Ejemplo

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

Clase	Método Único
CodeComponent	#startLoad
Locale	#currentPolicy:
Menu	#addItem:value:
ScheduledWindow	#updateEvent:

UIFinderVW2	#showClasses:
ComposedText	#centered
UIBuilder	#wrapWith:
Text	#emphasizeAllWith:
Cursor	#show
Image	#pixelsDo:

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

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

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

5. Clustering Jerárquico de Métodos Relacionados

Shepherd y Pollock [8] reportaron un experimento en el cual utilizaron Clustering Jerárquico Aglomerativo (CJO) [REF] para agrupar métodos relacionados.

Algunos trabajos previos en aspect mining [9] han utilizado análisis conceptual para agrupar código relacionado a un concern particular a partir de los nombres de métodos y clases. Los resultados de estos enfoques presentan al usuario una lista de nodos conceptuales, donde cada uno contiene una lista de hijos y cada hijo es una clase o un método. En estos casos no se reporta el lattice de conceptos y tampoco se permite la visualización de cuerpos de métodos simultáneamente. En consecuencia muchas relaciones entre vecinos, por mínima distancia que haya entre ellos, se pierden de vista.

A diferencia de trabajos basados en análisis conceptual [14, 15], este enfoque le facilita al usuario la visualización de los crosscutting concerns del sistema ya que relaciona el código de los mismos. Para ello, los autores utilizan una función de distancia semántica basada en (PLN), la cual permite agrupar en clusters métodos semánticamente relacionados del sistema. En consecuencia, cada cluster contendrá métodos que colaboren para satisfacer funcionalidad relacionada o correspondiente a un único concern. Luego, mediante una inspección de estos clusters es posible identificar cuáles de ellos se corresponden a crosscutting concerns. Cada cluster contiene métodos relacionados semánticamente, representando, generalmente, un crosscutting concerns.

5.1. Algoritmo

Se aplica CJO con el fin de agrupar métodos relacionados [10] y se ubica cada método en un cluster. A continuación, se detallan los pasos de este enfoque:

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

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

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

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

5.2. Ejemplo

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

1. Crosscutting concerns representados como una interface y sus métodos implementados de manera consistente.
2. Crosscutting concerns representados como una interface con sus métodos implementados que contengan código duplicado y lógica dispersa y que se encuentran implementados de forma inconsistente
3. Crosscutting concerns representados como métodos con nombres similares entre las clases, sin interfaces explícitas, con gran cantidad de código duplicado pero implementados de forma consistente.

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

5.3. Herramienta

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

6. Análisis de Fan-in

La técnica de análisis de Fan-in fue propuesta por Marin, van Deursen y Moonen [10]. Este enfoque se basa en la aplicación de la métrica de fan-in con el objetivo de

descubrir aquellos métodos del sistema que presenten una mayor dispersión (scattering). Los autores argumentan que es muy común que el código duplicado en un sistema legado sea refactorizado en un único método cuyo cuerpo está formado por dicho código duplicado. En consecuencia, estos métodos serán llamados desde diversos lugares, obteniendo así un alto valor alto de fan-in. En una reestructuración orientada a aspectos de un código legado, los concerns identificados con un valor alto de fan-in constituirán parte de un advice y el sitio de llamado corresponderá al contexto que necesita ser capturado usando un pointcut.

6.1. Algoritmo

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

6.1.1. Cálculo de Métrica de Fan-in

Se define a la métrica de fan-in como la medida del número de métodos que llaman a otro método [11]. Para el cálculo de la métrica se reúnen el conjunto de potenciales llamadores de un método y se toma la cardinalidad de este conjunto. Sin embargo, el valor exacto de fan-in depende de la manera en que se interprete el polimorfismo de los métodos (tanto métodos llamadores como métodos llamados). Por esta razón, se plantean refinamientos para el cálculo de este valor.

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

Teniendo en cuenta que se intenta encontrar métodos que son llamados desde diferentes lugares, siendo estos potenciales crosscutting concerns, se plantea un segundo refinamiento que comprende los llamados a métodos polimórficos. En el caso de que se encuentre un método *m* que pertenece a cierto concern, es muy probable que tanto el

método redefinido de las superclases como de las subclases pertenezcan al mismo concern. Por esta razón, si un método *m'* llama a un método *m* de la clase *C,* se agrega también a *m'* como método llamador de cada método *m* declarado en las superclases y subclases de *C.* Con esta definición, los métodos abstractos actúan como acumuladores: cuando una implementación específica de una de sus subclases es invocada, no solo se aumenta el fan-in del método específico, sino que también aumenta el valor de fan-in del método padre.

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

6.1.2. Filtrado de Resultados

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

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

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

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

6.1.3. Análisis de Resultados

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

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

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

6.2. Ejemplo

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

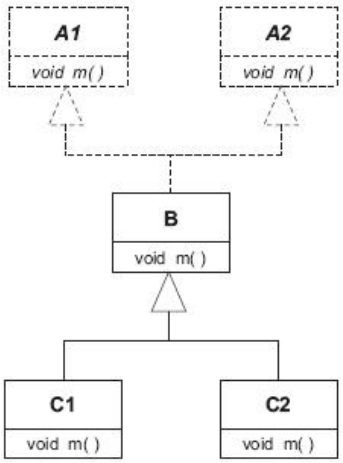


Fig. III- 4. Ejemplo de Jerarquía de Clases

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

Tabla. III-3 – Valores de Fan-in.

La Tabla. III-3 muestra las llamadas directas a un método, y su respectivo impacto en el fan-in de cada uno de los métodos de la jerarquía de clases (Fig. III- 5). Por ejemplo, el llamado *f1(A1 a1){a1.m();}* suma 1 al fan-in de *A1.m* por ser un llamado directo, y acumula 1 al fan-in de *B.m*, *C1.m* y *C2.m* debido a la redefinición del método llamado por sus clases hijas. El valor total muestra el valor de fan-in de cada método luego de haber realiado todas las llamadas definidas en la tabla. Por ejemplo, el método *m* de la clase *B* presenta un valor de fan-in de 5. Esto se debe a que el método es llamado una vez directamente (fan-in = 1) y a que actúa como acumulador de sus clases hijas y padres (fan-in = 1 + 4).

6.3. Herramienta

Los autores desarrollaron FINT (Fan-in Tool), un plug-in para eclipse [6] que provee soporte automático para calcular la métrica y filtrar métodos, y provee ayuda para realizar el análisis de los aspectos candidatos.

7. Detección de Clones como Indicadores de Crosscutting Concerns

El código duplicado puede considerarse como un síntoma de la presencia de crosscutting concerns en el código fuente de un sistema [12]. Esto se debe a que los

crosscutting concerns no se encuentran bien modularizados y ciertas partes de su implementación deben ser replicadas en diferentes partes del sistema.

Shepherd, E. Gibson, y L. Pollock [12] presentan un método basado en grafos de dependecia (PDG) con el fin de identificar aspectos candidatos en el código fuente de un sistema. Cada método es representado mediante un grafo, los cuáles pueden ser comparados de manera de encontrar código similar.

7.1. Algoritmo

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

1. Construir el grafo PDG a nivel código de cada método existente.
2. Identificar el conjunto de refactorings candidatos (control-based).
3. Filtrar refactorings candidatos no deseados (data-based).
4. Combinar conjuntos de candidatos relacionados en clases.

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

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

Los dos pasos previos dan como resultado pares de clones candidatos. Dado que la comparación es realizada método a método, es posible que candidatos similares o hasta

idénticos hayan sido reportados en diferentes pares. La cuarta fase intenta eliminar estos duplicados combinando los conjuntos que presentan esta característica.

7.2. Ejemplo

Los autores reportan la aplicación del enfoque sobre el código fuente del contenedor Tomcat [13], el cual contiene 430 archivos, 7.704 métodos y 38.495 líneas de código. Se determinó que más del 90% de los aspectos candidatos reportados fueron extremadamente útiles. La Fig. III-5 muestra un ejemplo de dos métodos específicos reportados para la aplicación Tomcat, los cuales son léxicamente diferentes, aunque de acuerdo a la representación PDG son equivalentes. Las líneas 10 y 11 son semánticamente idénticas a las líneas 19 y 20 aunque planteadas en contextos diferentes.

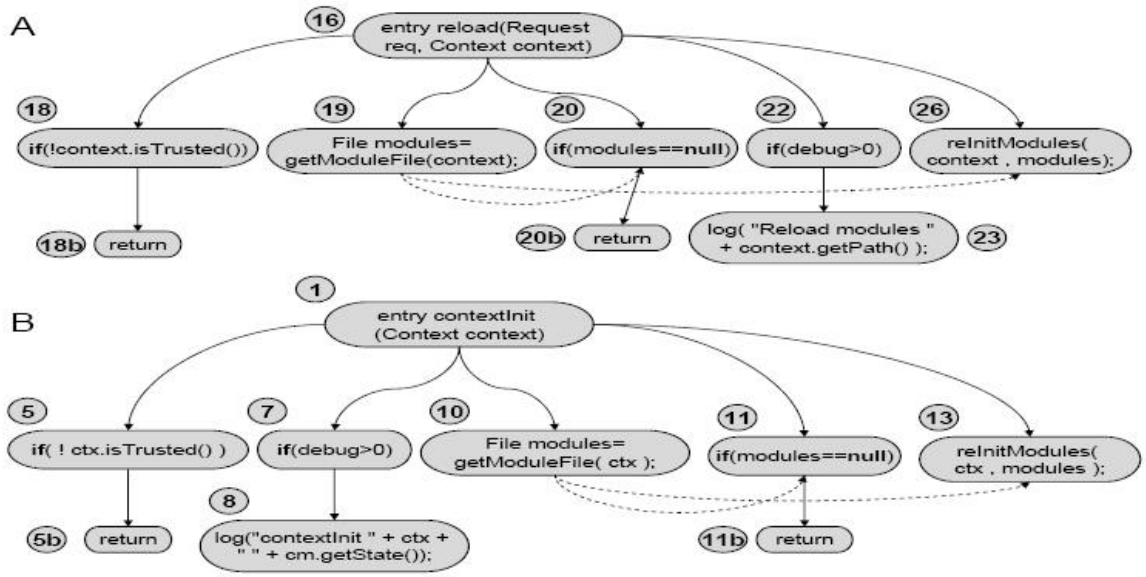


Fig. III- 6. Ejemplo de PDGs de dos métodos de la aplicación Tomcat.

7.3. Herramienta

Los autores presentan el enfoque como una extensión del framework Ophir. El mismo está implementado como un plug-in de eclipse [6] y permite el análisis automático de los clones y el refactoring manual y automático de los aspectos.

8. Análisis de Patrones Recurrentes en Trazas de Ejecución

Breu y Krinke [14] propusieron un enfoque dinámico para la extracción de aspectos de un programa. El mismo, comienza con la extracción de un conjunto de trazas de ejecución, las cuales son generadas durante diferentes corridas del programa y usadas como pool de datos. Luego, estas trazas son investigadas con el fin de obtener patrones de ejecución recurrentes basados en diferentes restricciones que indican cuando un patrón se repite. Los autores consideran que este conjunto de patrones de ejecución recurrentes corresponde a potenciales crosscutting concerns que describen funcionalidad recurrente del sistema y por lo tanto posibles crosscutting concerns.

8.1. Algoritmo

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

8.1.1. Clasificación de Relaciones de Ejecución

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

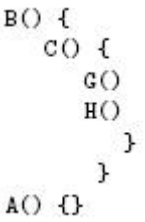


Fig. III- 7. Ejemplo de traza de ejecución.

Los autores proponen 4 diferentes tipos de relaciones de ejecución:

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

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

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

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

8.1.2. Restricciones de Relaciones de Ejecución

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

- **Restricción uniforme:** se refiere a relaciones de ejecución que ocurren siempre de la misma manera. Siendo *u* -> *v* una relación outside-before-execution se define como una relación recurrente si cada ejecución de *v* es precedida por *u*. La argumentación para las relaciones outside-after-execution es análoga. Para las relaciones inside-execution, $u \in_{\top} v$ (o $u \varepsilon_{\perp} v$) la restricción de uniformidad exige que *v* nunca sea ejecutado en primer lugar (o último) por otro método que no sea *u*.

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

8.1.3. Análisis de Relaciones de Ejecución

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

8.2. Ejemplo

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

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

Fig. III- 8. Ejemplo de traza de ejecución.

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

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

Fig. III- 9. Conjunto $S^{\rightarrow}$.

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

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

Fig. III- 10. Conjunto $S^{\in T}$ *y* $S^{\in \square} S^{\rightarrow}$.

8.3. Herramienta

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

9. Análisis Formal de Trazas de Ejecución (FCA)

Tonella y Ceccato [15] proponen una técnica de aspect mining dinámica, la cual aplica algoritmos de formal concept analysis o análisis conceptual (FCA sus siglas en inglés, Formal Concept Analysis) a las trazas de ejecución con el fin de identificar posibles aspectos. FCA es una rama de lattice theory, el cual, dado un conjunto de objetos y atributos que los describen, genera conceptos (grupos máximos de objetos con atributos en común).

Esta técnica utiliza trazas de ejecución generadas mediante casos de uso correspondientes a la funcionalidad principal de la aplicación. A continuación, la relación entre las trazas, la funcionalidad asociada y las unidades de compilación (método, clases) invocadas en cada ejecución son exploradas en función del lattice de conceptos producido por FCA. Los conceptos son clasificados como aspectos candidatos si presentan código entrelazado y disperso.

9.1. Concept Analysis para Feature Locations

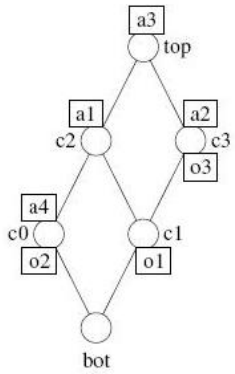
Concept análisis [REF] es una rama de lattice theory que provee una forma de agrupar objetos maximizando la cantidad de atributos que tienen en común. Dado un contexto (O, A, R), siendo O el conjunto de objetos, A el conjunto de atributos y R el conjunto de relaciones binarias entre O y A, un concepto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

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

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

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

	a_1	a_2	a_3	a_4
o_1	×	×	×	
o_2	×		×	×
o_3		×	×	



$bot = (\{\}, \{a_1, a_2, a_3, a_4\})$
$c_0 = (\{o_2\}, \{a_1, a_3, a_4\})$
$c_1 = (\{o_1\}, \{a_1, a_2, a_3\})$
$c_2 = (\{o_1, o_2\}, \{a_1, a_3\})$
$c_3 = (\{o_1, o_3\}, \{a_2, a_3\})$
$top = (\{o_1, o_2, o_3\}, \{a_3\})$

Fig. III- 11. Ejemplo de concept lattice.

9.2. Algoritmo

Los pasos que componen este enfoque se describen a continuación.

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

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

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

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

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

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

9.3. Ejemplo

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

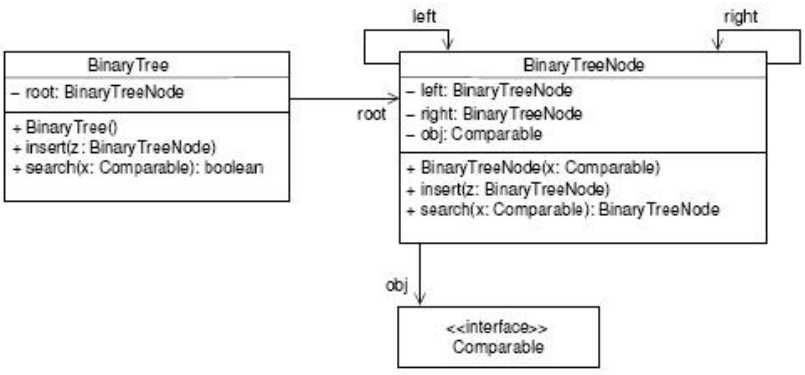


Fig. III- 12. Diagrama de clases de la aplicación de búsqueda binaria en un árbol.

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

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

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

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

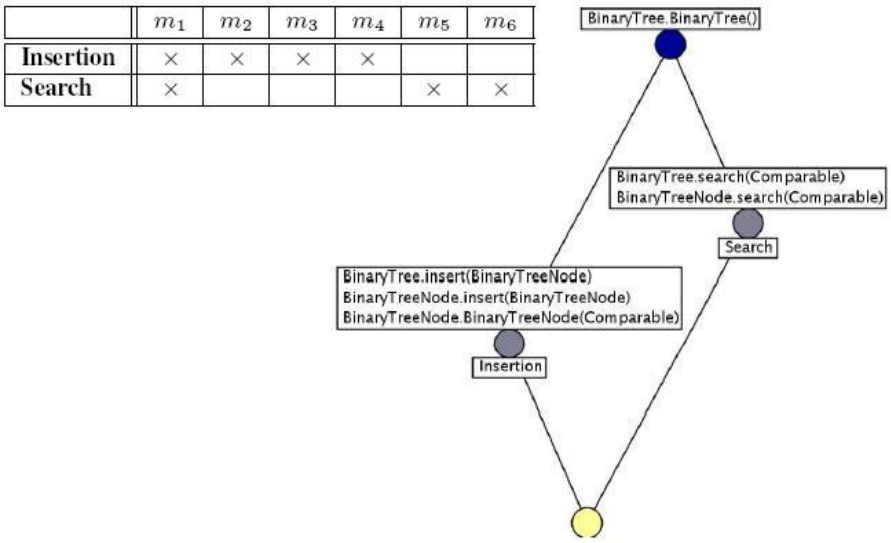


Fig. III- 13. Concept Lattice para la aplicación de búsqueda binaria en un árbol.

9.4. Herramienta

Los autores implementan la herramienta Dynamo [17] para aplicar la técnica descripta sobre aplicaciones Java.

10. Formal Concept Analysis Sobre los Identificadores del Código Fuente

Tourwé y Mens [18] proponen una técnica alternativa de aspect mining que depende de FCA. A diferencia de la técnica detallada en la sección 9, este enfoque realiza formal concept analysis sobre los identificadores de los métodos y las clases y se basa en la hipótesis de que los concerns en el código fuente se reflejan en el uso de convenciones de nombres tanto en las clases como en los métodos.

10.1. Algoritmo

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

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

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

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

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

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

10.2. Herramienta

El enfoque propuesto es soportado por la herramienta DelfStof, la cuál permite analizar tanto código Java [19] como código Smalltalk [18].

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

Krinke [20] describe una técnica en la cual se investiga el grafo de llamadas de un programa para descubrir patrones recurrentes de ejecución y así descubrir aspectos candidatos. La principal diferencia entre esta técnica y la técnica introducida en la Sección 8 [14] es el tipo de análisis de programa utilizado. Mientras que esta es una técnica estática, la anterior es una técnica basada en análisis dinámico. Si bien ambos tipos de análisis poseen sus pros y sus contras, algunos autores los consideran complementarios [REF]. Por lo tanto, ambas técnicas deberían ser consideradas durante la migración hacia un sistema orientado a aspectos.

11.1. Representación de las Relaciones de Ejecución

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

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

11.2. Algoritmo

Los pasos a seguir en el algoritmo son los siguientes:

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

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

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

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

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

11.3. Ejemplo

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

Cantidad de relaciones	Cantidad de métodos
2	53
3	19
4	4
5	6
6	3
7	2
8	1
9	1
11	1
12	1
13	1
294	92

Tabla. III-5. Relaciones Outside-before-execution

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

11.4. Herramienta

Los autores implementan el enfoque sobre el framework Soot [21], el cuál permite realizar este tipo de análisis sobre aplicaciones desarrolladas en Java.

12. Redirector Finder

Marin, Moonen y van Deursen [22] proponen una heurística para detectar clases en las que sus métodos redirijan sus llamadas consistentemente a métodos dedicados de otras clases. Como ejemplo se puede mencionar el patrón Decorator [3], el cual define una clase Decorator en la que sus métodos reciben llamadas, agregan funcionalidad opcionalmente, y luego redirigen estas llamadas a métodos específicos en la clase decorada.

12.1. Algoritmo

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

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

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

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

12.2. Ejemplo

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

métodos redireccionadores, representando al menos un 50% de la cantidad total de los métodos de la clase.

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

12.3. Herramienta

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

Referencias

[1] Shepherd, D., Pollock, L. L., Tourwé, T. “Using Language Clues to Discover Crosscutting Concerns,” ACM SIGSOFT Software Engineering Notes 30 (4), 1--6 2005.

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

[2] J. Hannemann and G. Kiczales. Overcoming the prevalent descomposition of legacy code. In Wkshp on Advances Separation of concerns, 2001.

[4] J. Morris and G. Hirst. Lexical cohesion computed by theasural relations as an indicator of the structure of text. Comput. Linguist., 17(1):21-48,1991.

[5] A. Budanitsky. Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures, 2001.

[6] Eclipse Homepage. http://www.eclipse.org. 2005.

[7] Gybels, K. and Kellens, A. “Experiences with Identifying Aspects in Smalltalk Using Unique Methods,” in: International Conference on Aspect Oriented Software Development. Amsterdam, The Netherlands 2005.

[8] D. Shepherd and L. Pollock. Interfaces, aspects and views. In Linking Aspect Technology and Evolution (LATE) Workshop, 2005.

[9] T. Tourwé and K. Mens. Mining Aspectural Views using Formal Concept Analysis. In Proceedings of the 4th International Workshop on Source code Analysis and Manipulation (SCAM), pages 97 – 106. IEEE Computer Science, 2004.

[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.

[11] I. Sommerville, Software Engineering (7th Edition) (International Computer Science Series). Addison Wesley, May 2004.

[12] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In International Conference on Software Engineering Research and Practice, 2004.

[13] Tomcat homepage, http://jakarta.apache.org/tomcat/.

[14] Breu, S., Krinke, J. Aspect Mining Using Event Traces. In: 19th IEEE International Conference on Automated Software Engineering, pp. 310--315. IEEE Computer Society, Washington DC, USA (2004).

[10] Tonella, P., Ceccato, M. Aspect Mining through the Formal Concept Analysis of Execution Traces. In: 11th Working Conference on Reverse Engineering, pp. 112--121. IEEE Computer Society, Washington DC, USA (2004)

[15] B. Ganter and R. Wille. Formal Concept Analysis: Mathematical Foundations. Spring-Verlag, 1999.

[16] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. IEEE Transactions on Software Engineering, 29(3):195.209, March 2003.

[17] Dynamo homepage, http://star.itc.it/dynamo/

[18] Tourwe, T., Kim Mens, K. Mining Aspectual Views using Formal Concept Analysis. In: 4th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 97—106. (2004)

[19] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonello, and T. Tourw´e. A qualitative comparison of three aspect mining techniques. In International Workshop on Program Comprehension (IWPC), 2005.

[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.

[21] R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In Proc. CASCON, 1999.

[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

[23] Hannemann, J., Kiczales, G.: Overcoming the Prevalent Decomposition of Legacy Code. In: Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering. Toronto, Ontario, Canada, (2001)