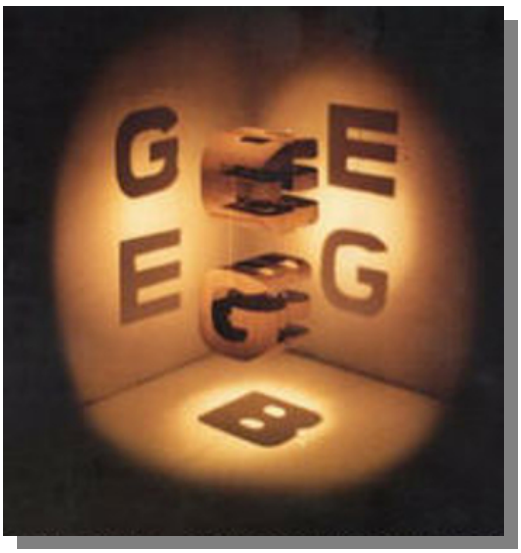
Aspect Mining Mediante Análisis Dinámico y Reglas de Asociación



TESIS DE GRADO DE LA CARRERA DE INGENIERÍA DE SISTEMAS
PRESENTADA A LA FACULTAD DE CIENCIAS EXACTAS DE LA UNCPBA

por

Esteban Sait Abait
Directora: Dra. Claudia Marcos

Diciembre, 2008

Universidad Nacional del Centro de la Provincia de Buenos Aires
Facultad de Ciencias Exactas – Tandil – Argentina

Índice de contenido

Índice de figuras

Índice de tablas

Agradecimientos

A mi familia y amigos por su apoyo paciente e incondicional, quienes me acompañaron aún en momentos difíciles.

A Claudia Marcos (*Pitty*) por sus permanentes sugerencias y correcciones que hicieron posible el desarrollo de este trabajo. Pero más importante por ser una gran amiga.

Capítulo I

Introducción

La evolución de los sistemas de software, es en la actualidad uno de los principales problemas que deben abordar aquellas personas relacionadas al desarrollo y mantenimiento de software. Esta evolución implica que los sistemas de software exitosos deben cambiar y adaptarse o volverse progresivamente menos útiles. Una de las formas para obtener sistemas más flexibles que se adapten mejor a los requerimientos cambiantes es mejorando la separación de concerns en el diseño de estos sistemas. Este principio establece que las diferentes partes que componen un sistema deben poseer un único propósito inherente a su propia naturaleza, y es la idea subyacente a la Programación Orientada a Aspectos (POA). Por lo tanto, la migración de los sistemas actuales orientados a objetos hacia POA, permitirá una mejora sustancial en la modularización de los mismos y un decremento en sus costos de mantenimiento.

1 Separación de *Concerns*

El principio de Separación de *Concerns* [Dijkstra 1982] se refiere al establecimiento de límites lógicos para definir y delinear propósito. El término *concern* ha sido definido como "cualquier cuestión de interés en un sistema de software" [Sutton y Rouvellou 2002]. Este principio básico de la ingeniería de software es de vital importancia, según Tarr et al. [Tarr et al. 1999]:

> "… nuestra habilidad para alcanzar los objetivos de la Ingeniería de Software (mejorar la calidad del software, reducir los costos de producción, y facilitar el mantenimiento y evolución) depende de nuestra habilidad para separar todos los *concerns* de importancia en los sistemas de software".

Sin embargo, en la práctica, es imposible separar todos los *concerns* en un sistema debido a diferentes razones, entre las cuales se incluyen un diseño inicial inadecuado, limitaciones que los lenguajes de programación imponen sobre la descomposición de un sistema de software, la aparición de *concerns* imprevistos a medida que el sistema evoluciona, y el deterioro que sufren las estructuras de código ha medida que el sistema sufre cambios [Robillard y Murphy 2007].

Las razones anteriores se hacen particularmente verdaderas para los sistemas legados. Este tipo de sistemas son a menudo sistemas de negocio críticos [Sommerville 2004], los cuales se mantienen porque es demasiado arriesgado y costoso reemplazarlos. Acorde a Demeyer et al. [Demeyer et al. 2002], cualquier sistema exitoso sufrirá los síntomas de los sistemas legados (documentación obsoleta, falta de conocimiento sobre el sistema, mayor tiempo para corregir defectos, código duplicado, entre otros). En particular, los sistemas legados orientados a objetos son exitosos sistemas cuya arquitectura y diseño ya no responden a los requerimientos cambiantes.

La ley de entropía del software dictamina que la mayoría de los sistemas pasado un tiempo tienden a decaer gradualmente en su calidad, a menos que éstos sean mantenidos y adaptados a los requerimientos cambiantes. En este sentido, Lehman y Belady [Lehman y Belady 1985] establecieron un conjunto de leyes (más bien hipótesis) concernientes a los cambios de los sistemas, las más importantes son:

- Ley del cambio continuado: Un programa que se usa en un entorno real necesariamente debe cambiar o se volverá progresivamente menos útil en ese entorno.
- Ley de la complejidad creciente: A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja. Se deben dedicar recursos extras para preservar y simplificar la estructura.

Como consecuencia, se desprende de estas leyes que el tiempo de vida de un sistema de software puede ser extendido manteniéndolo o reestructurándolo. Sin embargo, un sistema legado no puede ser ni reemplazado ni actualizado excepto a un alto costo. Por lo que el objetivo de la reestructuración es reducir la complejidad del sistema legado lo suficiente como para poder ser usado y adaptado a un costo razonable [Demeyer et al. 2002].

Uno de los principales problemas para el desarrollo del software es la "tiranía de la descomposición dominante" [Hannemann y Kiczales 2001], no importa cuan bien un sistema de software se descomponga en unidades modulares, siempre existirán *concerns* que cortan transversalmente la descomposición elegida. El código de estos *crosscutting concerns* estará presente sobre diferentes módulos, lo que tiene un impacto negativo sobre la calidad del software en

términos de comprensión, adaptación y evolución.

La Programación Orientada a Aspectos (POA) [Kiczales et al. 1997] ha sido propuesta como un nuevo paradigma que permite mejorar la *separación de concerns* en el software. El mismo se construye sobre tecnologías existentes, incluyendo la Programación Orientada a Objetos (POO) y la Programación Procedural (PP). De manera de capturar dichos *crosscutting concerns* de forma localizada, un nuevo mecanismo de abstracción (llamado aspecto) es agregado a los lenguajes de programación existentes (por ejemplo, AspectJ [Kiczales et al. 2001] para Java).

Para que la POA sea verdaderamente exitosa, es necesario migrar los sistemas de software existentes hacia su equivalente orientado a aspectos y reestructurarlos de manera continua. Sin embargo, aplicar manualmente técnicas orientadas a aspectos a un sistema legado es un proceso difícil y tendiente al error [Kellens et al. 2007]. Debido al gran tamaño de estos sistemas, la complejidad de su implementación, la falta de documentación y conocimiento sobre el mismo, es que existe la necesidad de herramientas y técnicas que ayuden a los desarrolladores a localizar y documentar los *crosscutting concerns* presentes en esos sistemas.

El estudio y desarrollo de tales técnicas es el objetivo de *aspect mining* y *aspect refactoring*. *Aspect mining* [Kellens et al. 2007] es la actividad de descubrir *crosscuting concerns* que potencialmente podrían convertirse en aspectos. A partir de allí, estos *concerns* pueden ser encapsulados en nuevos aspectos del sistema (mediante la utilización de técnicas conocidas como *aspect refactoring* [Kellens et al. 2007]) o podrían documentarse con el objetivo de mejorar la comprensión del programa.

La tarea de identificación de los *crosscutting concerns* adecuados para una implementación orientada a aspectos se denomina *aspect mining* [Kellens et al. 2007; Marin et al. 2007]. Las técnicas de *aspect mining* toman como entrada el código (o información obtenida ejecutando el código) de un sistema legado y de manera automática generan un conjunto de *seeds* o aspectos candidatos. Estos *seeds* o aspectos candidatos deberán ser analizados por el desarrollador para determinar si constituyen *crosscutting concerns* presentes en el código o no. Una vez que el desarrollador ha identificado los *crosscutting concerns*, los mismos podrán ser convertidos en aspectos reales del sistema mediante la aplicación de *refactorings* para aspectos [Kellens et al. 2007].

2 *Aspect Mining* Mediante Análisis Dinámico y Reglas de Asociación

En este trabajo, se presenta un proceso semiautomático de cinco pasos para *aspect mining* basado en análisis dinámico y reglas de asociación. El enfoque propuesto para *aspect mining* está basado en el análisis de las trazas de ejecución de un sistema mediante la utilización de algoritmos de reglas de asociación [Agrawal y Srikant 1994]. La salida del enfoque es un conjunto de reglas de asociación que indican posibles síntomas de código *scattering* en el sistema.

En la Fig. I-1, los principales pasos del enfoque de *aspect mining* propuesto son mostrados.

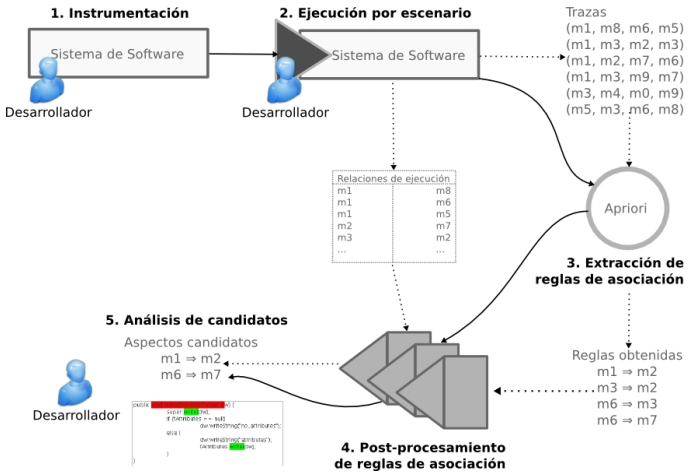


Fig. I-1. Workflow de la técnica propuesta.

La instrumentación del sistema (paso 1) implica modificar el sistema analizado para establecer la infraestructura necesaria que permita el análisis dinámico del mismo. En particular, el mecanismo de *tracing* es añadido a la aplicación como un nuevo paquete que contiene el aspecto de *tracing* y las clases de soporte. Si la aplicación posee una interfase gráfica, la misma es modificada para reflejar la funcionalidad correspondiente a la habilitación e inhabilitación del mecanismo de *tracing*.

Durante el segundo paso del proceso, el sistema es ejecutado a partir de un conjunto de escenarios de ejecución.

Cada escenario representa una funcionalidad de alto nivel desde el punto de vista del usuario. La elección del conjunto de escenarios a probar en el sistema depende del tipo de análisis a realizar:

- *Estrategia orientada al objetivo*. De adoptarse este tipo de estrategia, los escenarios a elegir dependerán del objetivo que se desee alcanzar. Por ejemplo, si se deseara analizar el *concern* de persistencia en la aplicación JHotdraw, los escenarios que se deben probar son los correspondientes al almacenamiento y recuperación de un gráfico.

- *Análisis general*. Si lo que se desea es realizar un análisis general del sistema con el objetivo de ver qué *crosscutting concerns* pueden estar presentes en el mismo, los escenarios se deben elegir de forma tal que cada uno represente una funcionalidad de la aplicación visible externamente. Se deben evitar escenarios que sean equivalentes funcionalmente, ya que estos podrían ejercitar las mismas secciones de código del sistema induciendo un sesgo en el análisis posterior mediante reglas de asociación.

La salida de este segundo paso es un conjunto de trazas (una por escenario ejecutado) y las relaciones de ejecución que se dieron durante los escenarios ejecutados.

El tercer paso se corresponde con la ejecución del algoritmo Apriori [Agrawal y Srikant 1994] de reglas de asociación. Este algoritmo toma como entrada un conjunto de transacciones (cada transacción se corresponde con una traza) y el mínimo valor de soporte y confianza. La salida de este paso es un conjunto de reglas de asociación.

En el cuarto paso, algunas reglas de asociación son descartadas y otras clasificadas mediante diferentes filtros de post-procesamiento. Estos filtros deben eliminar las reglas redundantes y eliminar reglas que no sean de interés para el desarrollador. A su vez, otros filtros implementan heurísticas para reconocer que reglas de asociación representan aspectos candidatos.

El quinto paso corresponde al análisis de los aspectos candidatos obtenidos por parte del desarrollador. Para esto, el desarrollador puede valerse de técnicas que visualicen las reglas de asociación sobre el código fuente del sistema, o de otras técnicas de navegación y exploración que tomen como entrada alguno de los métodos dentro de las reglas.

La participación del desarrollador dentro del *workflow* propuesto está dada principalmente en las primeras actividades y en la última de ellas. Podría considerarse la participación del usuario para establecer los valores mínimos de soporte y confianza durante la generación de las reglas de asociación (paso 3), pero la generación propiamente dicha está automatizada por el algoritmo correspondiente y su implementación.

3 YAAM (*Yet Another Aspect Mining Tool*)

YAAM (*Yet Another Aspect Miner*) es una herramienta prototípica implementada en Java que permite la extracción de reglas de asociación a partir de trazas y relaciones de ejecución, su posterior post-procesamiento y la visualización de las mismas sobre el código fuente de la aplicación bajo análisis.

De esta manera, la herramienta permite automatizar, en gran parte, los pasos tres (extracción de reglas de asociación), cuatro (post-procesamiento de las reglas de asociación) y cinco (análisis de los candidatos) del proceso de *aspect mining* propuesto.

La principal característica de YAAM es que permite visualizar una regla de asociación sobre el código fuente de la aplicación. Cada vez que una regla de asociación es visualizada, se puede observar:

- El *concern* que esa regla de asociación representa (en caso de que el usuario haya especificado alguno), y el filtro mediante el cual esa regla fue clasificada.

- Los métodos correspondientes tanto al antecedente como al consecuente.

- Información correspondiente a cada uno de esos métodos, tal como su nombre, la clase y paquete al que pertenece y su tipo de retorno.

- A su vez, el código fuente de las clases del método del antecedente y del método del consecuente es cargado y puede ser navegado por el desarrollador.

4 Organización del Trabajo

Este trabajo esta organizado de la siguiente manera:

En el capítulo 2, se presenta una introducción al Desarrollo de Software Orientado a Aspectos. Se definen los

conceptos principales del paradigma y se muestra un ejemplo de utilización del paradigma y los beneficios que este conlleva.

En el capítulo 3, se introduce la problemática relacionada a la evolución de los sistemas legados orientados a objetos, y cómo mediante la identificación de *crosscuting concerns* sobre el código legado (*aspect mining*) y su refactorización en forma de aspectos es posible extender el tiempo de vida útil de los mismos.

En el capítulo 4, se describen los trabajos existentes en el área de *aspect mining*, y se realiza un análisis y una clasificación de los mismos. Por cada trabajo analizado, se incluye el enfoque propuesto, se muestra un ejemplo de su aplicación y se describe la herramienta que da soporte al mismo.

En el capítulo 5, se presenta el enfoque propuesto para la identificación de *crosscutting concerns*. Se describen tanto las técnicas de análisis dinámico como de reglas de asociación, haciendo énfasis en sus ventajas y desventajas. A su vez, se muestra la herramienta que sistematiza al enfoque propuesto y se presenta un pequeño ejemplo que ilustra su utilización.

En el capítulo 6, se presenta un completo caso de estudio y una evaluación de la técnica tomando como base los *concerns* previamente descubiertos por otros autores sobre el sistema analizado.

Adicionalmente, en el Anexo I, se presenta el paper *Dynamic Analysis and Association Rules for Aspects Identification*, publicado en II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2008). Campinas, Brasil. 2008.

Por otra parte, en el Anexo II, se presenta el paper *Aspect Mining Mediante Análisis Dinámico y Reglas de Asociación*, publicado en el Concurso de Trabajos Estudiantiles - EST 2008 realizado como parte de las 37 Jornadas Argentinas de Informática (JAIIO). Santa Fé, Argentina, Septiembre de 2008.

Capítulo II

Introducción al Desarrollo de Software Orientado a Aspectos

El desarrollo basado en componentes es un enfoque ampliamente utilizado al momento de construir complejos sistemas de software, la idea de este tipo de desarrollo es que cada requerimiento del sistema sea realizado por un único componente. El problema, es que muchos de los requerimientos no pueden ser directamente localizados a una única unidad. La imposibilidad de mantener los concerns[1] separados durante diseño e implementación hace que el sistema sea difícil de entender y mantener, inhibiendo la posibilidad de un desarrollo en paralelo de los componentes y dificultando futuras extensiones al sistema. En este contexto, una solución efectiva a este problema debería proveer dos cosas: una técnica de ingeniería que permita separar los concerns desde la toma de requerimientos hasta la fase de implementación y un mecanismo de composición que combine el diseño y la implementación de cada concern. Si bien muchas soluciones han sido propuestas, en la actualidad el Desarrollo de Software Orientado a Aspectos ha surgido como el enfoque más apto para resolver los problemas planteados por la separación de concerns[2] en el software. En este capítulo se introducen los conceptos principales del paradigma y se muestra un ejemplo de utilización del paradigma y los beneficios que este conlleva.

1 Introducción

La Programación Orientada a Aspectos (POA) [Kiczales et al. 1997] ha sido propuesta como un nuevo paradigma que permite mejorar la *separación de concerns* en el software. El mismo se construye sobre tecnologías existentes, incluyendo la Programación Orientada a Objetos (POO) y la Programación Procedural (PP).

La idea central de la POA, es que, mientras los mecanismos jerárquicos de modularización de la POO son muy útiles, están inherentemente imposibilitados de modularizar todos los *concerns* de interés presentes en sistemas complejos de software. Considerando además que, en la implementación de cualquier sistema complejo de software, existirán *concerns* que inherentemente atravesarán (*crosscut*) la modularidad natural del resto de la implementación, tales *crosscuting concerns* no están dedicados a una única unidad de modularización (como un único paquete, jerarquía de clases, clase o método) sino que están dispersos (*scattered*) a través de todas esas unidades. La consecuencia de esto es *tangling*: las unidades modulares no pueden tratar únicamente con su *concern*, sino que tienen que administrar la implementación de otros *concerns* que atraviesan su modularización.

1.1 Síntomas de Mala Modularización

Es posible clasificar, de manera muy amplia, los síntomas de mala modularización en dos categorías: código *tangling* (entremezclado) y código *scattering* (diseminado) [Hannemann y Kiczales 2001]. De existir estos síntomas en un sistema es muy probable que se deban a la implementación de *crosscutting concerns*.

El código *tangling* surge cuando se implementa un módulo que maneja múltiples *concerns* simultáneamente. En tal caso, el desarrollador deberá considerar *concerns* tales como lógica del negocio, *performance*, sincronización o *logging* al momento de implementar dicho módulo. El resultado es la presencia simultánea de elementos pertenecientes a diferentes *concerns* lo que resulta en el entremezclado del código. En la Fig. II-1, se presenta un ejemplo de código *tangling* extraído de la aplicación JHotDraw 5.4b1 [JHotDraw]. En la misma, se ven resaltados elementos del código fuente correspondientes a diferentes *concerns* presentes en la misma clase, tales como la persistencia (métodos que incluyen la cadena 'read') o la notificación de eventos (métodos que incluyen la cadena 'changed' o 'listener'). Ninguno de estos *concerns* están relacionados con la funcionalidad principal de la clase TextFigure, la cual representa una figura gráfica que permite agregar texto a un dibujo.

El código *scattering* es causado cuando “algo” es implementado usando múltiples módulos. Por ejemplo, en un sistema que usa una base de datos, los *concerns* de *performance* podrían afectar a todos los módulos que acceden a la base de datos. A su vez, existen dos tipos de código *scattering* [Laddad 2003]: código duplicado y bloques de código

1 *Concern*. Cualquier cuestión de interés en un sistema de software.

2 Separación de *concerns,* implica el diseño o la implementación del sistema usando las unidades naturales de *concerns* en vez de las unidades impuestas por las herramientas (o paradigmas) utilizadas.

complementarios. El primer tipo se caracteriza por el código repetido de naturaleza casi idéntica. Por ejemplo, en JHotDraw la persistencia de los datos es una funcionalidad que se encuentra dispersa a través de toda la jerarquía de figuras (Fig. II-2).

TextFigure.java

```
private void readObject(ObjectInputStream s) throws ClassNotFoundException, IOException {

    s.defaultReadObject();

    if (fObservedFigure != null) {
        fObservedFigure.addFigureChangeListener(this);
    }
    markDirty();
}
public void figureChanged(FigureChangeEvent e) {
    updateLocation();
}

public void figureRemoved(FigureChangeEvent e) {
    if (listener() != null) {
        listener().figureRequestRemove(new FigureChangeEvent(this));
    }
}
```

Fig. II-1. Ejemplo de código *tangling* en JHotDraw.

El segundo tipo de código *scattering* ocurre cuando diferentes módulos implementan partes complementarias del mismo *concern*. Por ejemplo, para un sistema de control de acceso, en un módulo se realiza la autenticación del usuario, el usuario autenticado es pasado a los módulos que solicitan la autorización, y entonces esos módulos llevan a cabo la autorización. Todas esas piezas implementan la funcionalidad requerida.

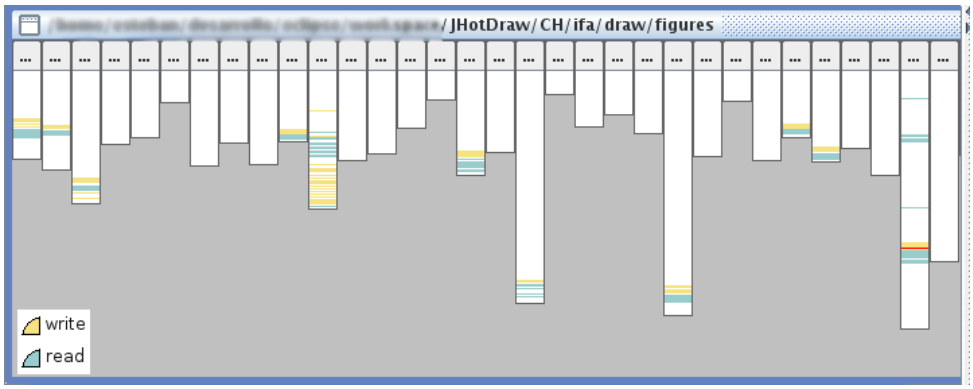


Fig. II-2. *Scattering* de las operaciones read y write, encargadas de la persistencia en JHotDraw.

1.2 Elementos de una Implementación Orientada a Aspectos

La POA introduce nuevos mecanismos con el objetivo de capturar la estructura transversal de los *crosscuting concerns* tratando de solucionar los síntomas de *scattering* y *tangling*. Esto se traduce en los típicos beneficios de una mayor modularidad: código más simple y fácil de mantener y con un mayor nivel de reuso. Se denominará, a partir de ahora, *aspecto* a aquel *crosscuting concern* que ha sido correctamente modularizado.

Los siguientes elementos son necesarios para implementar una solución orientada a aspectos:

- Un lenguaje base (como Java), con el cual programar componentes.
- Un lenguaje de aspectos (como AspectJ), con el cual encapsular los *crosscuting concerns* del sistema.
- Un *weaver* (tejedor), el cual permite combinar ambos lenguajes.
- Un programa de componentes, el cual implementa los componentes usando el lenguaje base.
- Un programa de aspectos, el cual implementa los aspectos usando el lenguaje de aspectos.

A continuación, se describen los dos elementos más importantes: aspecto y *aspect weaver* (tejedor de aspectos).

1.2.1 Aspecto

Un aspecto es una nueva unidad de modularización cuyo objetivo es encapsular un *crosscuting concern*. Los aspectos no son, por lo general, unidades que surgen de la descomposición funcional del sistema sino que son propiedades que afectan la performance o la semántica de los componentes del mismo. Por lo tanto, a la hora de desarrollar un nuevo sistema mediante la orientación a aspectos, el sistema podría descomponerse utilizando tanto objetos (los cuales, por lo general, representan conceptos del dominio) como aspectos (los cuales encapsulan cualquier *concern* de tipo *crosscutting*).

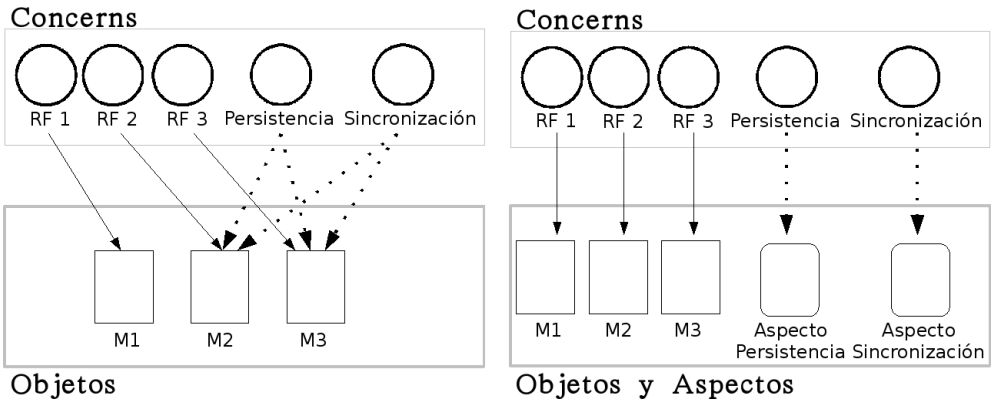


Fig. II-3. Mapeo de *concerns* a objetos (*izq.*) y de *concerns* a objetos y aspectos (*der.*). Las flechas continuas indican que objeto implementa que *concern*. Las flechas punteadas indican que objeto/aspecto implementa un *crosscuting concern*.

En la Fig. II-3, se ilustran las ventajas de utilizar aspectos a la hora de descomponer el sistema. Si el único mecanismo de descomposición son los objetos, muchos *concerns* (que podrían corresponderse tanto a requerimientos funcionales como a decisiones de diseño o implementación) se verán implementados en más de una unidad. Por ejemplo, los *concerns* RF 1, RF 2 y RF 3 son implementados por los objetos M1, M2 y M3, pero a su vez, estos objetos necesitan implementar la funcionalidad requerida por los *concerns* de persistencia y sincronización. Por otra parte, dividir el sistema usando objetos y aspectos, permite lograr la tan deseada separación de *concerns* haciendo que cada unidad de implementación se corresponda con un único *concern*. Como consecuencia, los objetos M1, M2 y M3 no sufren de código *tangling* haciéndolos más reutilizables y fáciles de modificar. El mismo ejemplo podría presentarse usando procedimientos o cualquier otro tipo de unidad distinta a objetos.

1.2.2 Aspect Weaver (o Tejedor de Aspectos)

Un *aspect weaver* tiene la responsabilidad de procesar el lenguaje base (o de componentes) y el lenguaje de aspectos, y componerlos con el objetivo de obtener la implementación del sistema. La composición del código aspectual con el código base se realiza en un punto del sistema en particular, el cual se denomina *join point*. Los *aspect weavers* trabajan generando una representación de los *join-points* del programa base, y luego ejecutando (o compilando) los aspectos respecto a esta.

Este proceso de composición se denomina *weaving* y está guiado por las reglas de *weaving* [Laadad 2003], las cuales determinan cómo integrar los *concerns* implementados mediante aspectos y componentes de manera de formar el sistema final. Por ejemplo, para un aspecto que implementa funcionalidad concerniente al *concern logging* las reglas de *weaving* especifican los puntos donde se introduce el log (*join point*), la información a ser registrada, etc. El sistema utilizará estas reglas para invocar correctamente al aspecto encargado del *logging* desde los componentes base.

Por otra parte, el *weaver* puede ser implementado de muchas maneras [Laadad 2003]. La manera más simple es mediante la traducción fuente-a-fuente, en este caso el código fuente de los componentes base y de los aspectos son pre-procesados por el compilador del lenguaje de aspectos para producir el código fuente final. Luego, el compilador del lenguaje de aspectos entrega este código convertido al compilador del lenguaje base para producir la versión ejecutable de la aplicación. Este proceso se denomina *weaving* estático.

Otro enfoque de *weaving* consiste en primero compilar el código fuente de los componentes utilizando el compilador del lenguaje base, luego compilar el código de los aspectos mediante el compilador del lenguaje de aspectos y utilizar el *aspect weaver* para componer el código binario de los aspectos con el código binario de los componentes a medida que estos son cargados. En este caso, el código de los aspectos no es compuesto con el código de los componentes hasta tiempo de ejecución. Este proceso de *weaving* se denomina *weaving* dinámico.

2 El Desarrollo de Software Orientado a Aspectos

Las técnicas de Desarrollo de Software Orientado a Aspectos (DSOA) proveen mecanismos sistemáticos para identificar, modularizar, representar y componer *crosscutting concerns*. Estas técnicas están construidas sobre trabajos previos existentes en metodologías de desarrollo de software, de manera de resolver el problema planteado por los *crosscutting concerns* sistemáticamente. Si bien la mayoría del trabajo inicial en DSOA se enfocó en el desarrollo de lenguajes, *frameworks* y plataformas orientados a aspectos (POA), un número de métodos y técnicas se han enfocado en solucionar la problemática de los *crosscutting concern* en el nivel de análisis y diseño [Chitchyan et al. 2005]. No es suficiente dedicarse a los enfoques orientados a aspectos únicamente durante la etapa de diseño y posteriores pues los aspectos no sólo están confinados en artefactos producidos durante esas etapas. Más bien, es beneficioso soportar la orientación a aspectos desde el comienzo del ciclo de vida del software.

Early aspects son *crosscutting concerns* que han sido identificados durante fases tempranas del ciclo de vida del desarrollo de un sistema [Araújo et al. 2005]. Por lo general, estas fases incluyen el análisis del dominio, la captura de requerimientos y el diseño de la arquitectura. Esto implica, que los *early aspects* son una generalización del concepto de "aspecto" anteriormente introducido, ya que durante las primeras etapas de desarrollo de un sistema no existen unidades de implementación que puedan ser atravesadas por *crosscutting concerns*. En cambio, los *early aspects* tienden a estar dispersos sobre múltiples artefactos generados al comienzo del desarrollo; artefactos tales como documentos de requerimientos, modelos del dominio o elementos del diseño de una arquitectura. La localización y gestión de estos *crosscutting concerns* permite aumentar la modularidad de los artefactos facilitando el futuro mantenimiento e implementación del sistema.

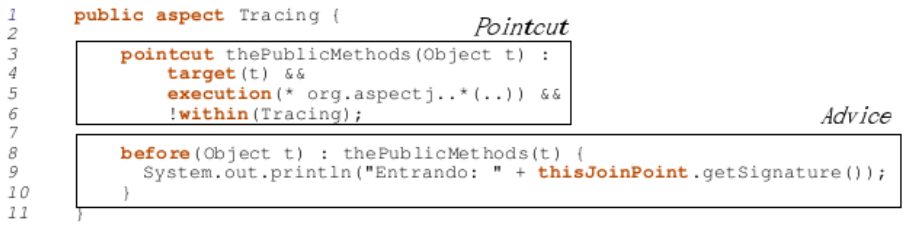
3 Plataformas POA

A continuación, se presentan algunas de las plataformas que dan soporte a la POA más conocidas, en particular, se describe al lenguaje orientado a aspectos AspectJ [Kiczales et al. 2001], el cual cuenta con una gran adopción dentro de la comunidad de desarrolladores.

3.1 AspectJ

AspectJ [Kiczales et al. 2001] es una simple y práctica extensión al lenguaje Java con la cual es posible modularizar un amplio rango de *crosscuting concerns*. Dado que AspectJ es una extensión del lenguaje Java, cada programa válido en Java también lo es en AspectJ, debido a que el compilador de AspectJ genera archivos de clases que conforman con la especificación byte-code de Java permitiendo que cualquier JVM ejecute tales archivos de clases [Laddad 2003]. AspectJ consiste en dos partes: la especificación del lenguaje y la implementación del lenguaje [Laadad 2003]. La especificación del lenguaje define el lenguaje correspondiente para escribir código; con AspectJ, se implementan los *concerns* utilizando el lenguaje Java y las extensiones provistas por AspectJ para implementar la composición de los *crosscutting concerns*. La parte de la implementación del lenguaje provee herramientas para compilar, hacer *debugging*, e integrar el lenguaje con los IDE (*Integrated Development Environment*) más populares.

AspectJ extiende a Java otorgándole soporte para dos tipos de implementación de *crosscuting concerns* [Kiczales et al. 2001]. El primer tipo hace posible definir implementaciones adicionales para que se ejecuten en puntos bien definidos de la ejecución de un programa, es denominado *dynamic crosscuting*. El segundo hace posible definir nuevas operaciones sobre los tipos (o clases) existentes, es denominado *static crosscuting* ya que afecta la signatura estática de los tipos de un programa.




```
1   public aspect Tracing {
2
3       pointcut thePublicMethods(Object t) :
4           target(t) &&
5           execution(* org.aspectj..*(..)) &&
6           !within(Tracing);
7
8       before(Object t) : thePublicMethods(t) {
9         System.out.println("Entrando: " + thisJoinPoint.getSignature());
10      }
11  }
```

Fig. II-4. Ejemplo de un aspecto de *tracing* escrito en AspectJ.

Dynamic crosscuting se basa en un pequeño (aunque poderoso) conjunto de constructores:

- *Join Points*: puntos bien definidos en la ejecución de un programa (por ejemplo, la llamada de un método o

constructor, la ejecución de un método o un constructor, el acceso a un atributo de una clase, entre otros).

- *Pointcuts*: construcción declarativa que permite hacer referencia a un conjunto de *join points* y sus valores.
- *Advices*: mecanismo usado para especificar el código que se debería ejecutar en cada *join poin*t referenciado por un *pointcut*.

El mecanismo de *static crosscutting*, también denominado *inter-type declarations*, permite declarar miembros que atraviesan múltiples clases, o cambiar las relaciones de herencia entre las mismas. Este tipo de mecanismo opera de manera estática en tiempo de compilación.

Finalmente, las declaraciones correspondientes a ambos mecanismos se empaquetan en una unidad denominada *aspect*. En la Fig. II-4, se muestra un aspecto de *tracing* implementado en AspectJ, el cual intercepta mediante un *pointcut* (líneas 3 a 6) todas las ejecuciones de métodos públicos dentro del paquete 'org.aspectj' e imprime por salida estándar (*advice*, líneas 8 a 10) la signatura del método en ejecución.

3.2 Spring AOP

Spring [SpringAOP] es un *framework open source* implementado en Java que provee básicamente un *container* IoC [Fowler 2004] para el desarrollo de aplicaciones Java y un componente (denominado AOP *framework*) que da soporte a la orientación a aspectos. El modelo de *join points* soportado por este *framework* no es tan expresivo como el de AspectJ debido a que sólo soporta *join points* de llamada a métodos. Otra desventaja, es que sólo permite asociar aspectos a objetos administrados por un *container*. Los aspectos en Spring son clases Java estándar, y los *pointcuts* se pueden definir ya sea mediante archivos XML o a través de anotaciones.

3.3 JBoss AOP

JBoss AOP [JBossAOP] es un *framework* implementado puramente en Java, que puede ser utilizado en cualquier ambiente de desarrollo o integrado al servidor de aplicaciones JBoss [JBoss]. Este *framework* soporta inyección de dependencias, *mixins* e introducciones, también permite interceptar invocaciones a métodos, a constructores y acceso a atributos. Además, es posible realizar el *deployment* de los aspectos en tiempo de ejecución. Una de las principales ventajas de este *framework* es que permite realizar *weaving* dinámico de aspectos.

3.4 AspeCt-oriented C

AspeCt-oriented C es un proyecto de investigación que aplica los conceptos de la POA al lenguaje de programación C de manera de permitir la orientación de aspectos en sistemas basados en C [Gong et al. 2006]. Este proyecto ofrece una extensión orientada a aspectos al lenguaje C y un compilador que implementa la extensión del lenguaje. Un programa desarrollado en AspeCt-oriented C consiste de dos parte, comúnmente llamadas código base y código aspectual, donde el código base se encuentra escrito en C y el código aspectual está escrito en AspeCt-oriented C y C. El compilador de AspeCt-oriented C es un traductor fuente-a-fuente, el cual toma como entrada código AspeCt-oriented C y C y genera como salida código C.

3.5 LOOM .NET

LOOM .NET [Schult et al. 2003] es un *weaver* que permite realizar el *weaving* de código de aspectos sobre código .NET ya compilado. LOOM .NET utiliza mecanismos de metadatos y reflexión para examinar el código compilado y generar un proxy. Debido a que no importa si el código .NET compilado fue escrito en lenguaje C# o Java, LOOM .NET es independiente del lenguaje.

4 Ejemplo: Patrón *Observer*

En esta sección se presentan los problemas de modularización que posee una implementación del patrón de diseño *Observer* [Gamma et al. 1995] realizada en un lenguaje orientado a objetos y los beneficios que se obtendrían utilizando un lenguaje orientado a aspectos para implementar el patrón.

El patrón *Observer* [Gamma et al. 1995] permite "definir una dependencia de uno a muchos entre objetos tal que cuando el estado de un objeto cambia, todas sus dependencias son notificadas y actualizadas". La implementación orientada a objetos de este patrón usualmente añade un campo (observers) a todos los sujetos (Subject) para almacenar la lista de interesados (Observer) (Fig. II-5). Cuando un sujeto desea reportar un cambio de estado a sus interesados invoca a su método notifyObservers, el cual a su vez invocará el método update de cada uno de los interesados en su lista. En la Fig. II-5, se presenta una instancia particular de este patrón (adaptada de [Hannemann y Kiczales 2002]), donde la clase Point además de implementar funcionalidad propia debe implementar los mecanismos propios impuestos

por el rol Subject (agregar y eliminar Observers y notificar el cambio cada vez que su estado varíe). La clase Screen, además de representar una pantalla es tanto Subject como Observer de Point y de sí misma.

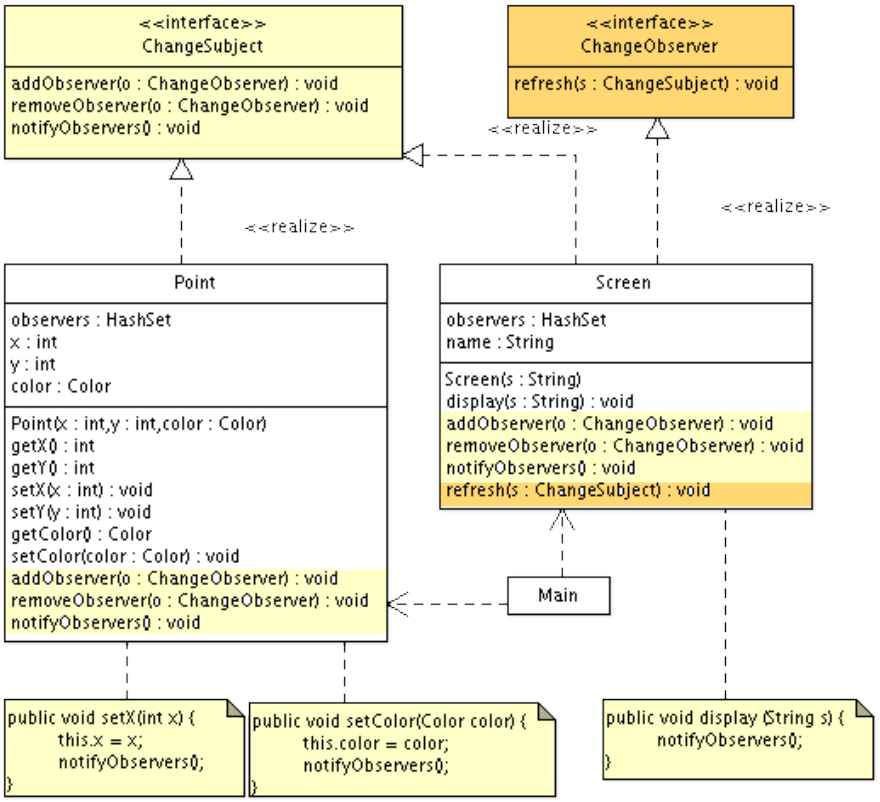


Fig. II-5. Diagrama de clases del ejemplo.

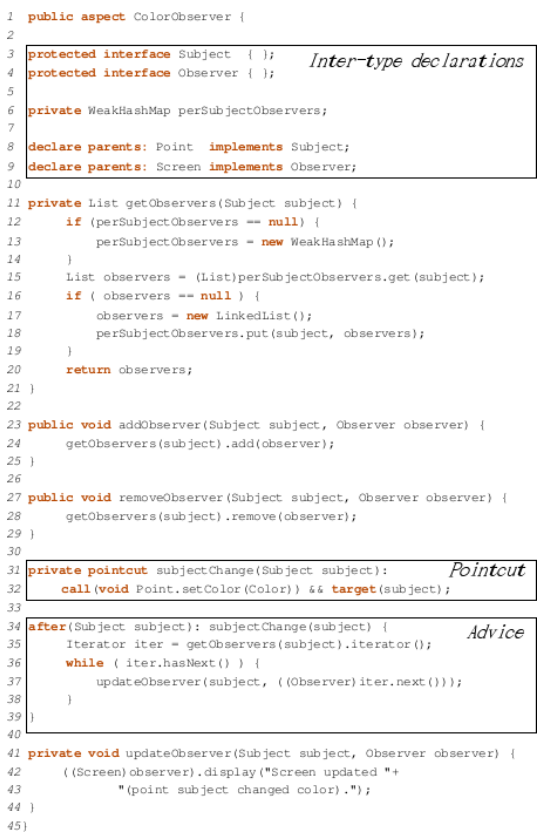
La aplicación de este patrón sobre las clases Point y Screen supone la imposición de funcionalidad adicional, la cual se considera como *crosscutting* y provoca, para este caso, código repetido en ambas clases (ver notas en Fig. II-5). Por lo tanto, las clases participantes deben poseer código correspondiente tanto a su funcionalidad principal (saber dibujarse o gestionar sus coordenadas) como a la de los roles impuestos por el patrón. La consecuencia de esto es código *tangling* en cada clase (por ejemplo cada invocación al método setY en Point genera una invocación a su método notifyObservers) y código *scattering* debido a que la funcionalidad correspondiente del patrón no está debidamente localizada en una unidad sino que está dispersa en todas las clases participantes en el patrón (ver métodos addObserver, removeObserver y notifyObservers).

Una primera solución sería encapsular todo lo referente al patrón en un aspecto, el cual deberá diferenciar entre Subjects y Observers, mantener el mapeo entre ambos roles, implementar el mecanismo de actualización e indicar el conjunto de cambios en un Subject que desencadena la posible actualización en un Observer.

La implementación de esta solución en AspectJ [Hannemann y Kiczales 2002] se muestra en la Fig. II-6. Mediante el mecanismo de *inter-type declarations* se definen dos interfaces Subject y Observer y se hace que las clases Point y Screen las implementen (líneas 3 a 9). A su vez, se define la representación interna utilizada para almacenar la lista de Observers de un Subject en particular. En las líneas 11 a 29 se definen tres métodos (al estilo Java), los cuales se corresponden con la administración de los Observers. El *pointcut* subjectChange será verdadero cada vez que se invoque el método setColor de la clase Point (líneas 31 a 32), y el *after advice* especifica el código a ejecutar cuando el *pointcut* se cumpla. Por lo tanto, posterior a cada ejecución del método setColor de la clase Point se recorrerá la lista de Observers acorde al Subject que cambió y se invocará al método update definido en las líneas 41 a 44.

Esta implementación en particular, mantiene una lista de Observers interesados en cambios en el color del punto. Esto podría variarse cambiando la definición del *pointcut* y del método update.

Otro aspecto podría implementarse pero que sea específico a la clase Screen, el mismo sería muy similar al desarrollado para la clase Point. Por lo que una mejor solución sería crear un aspecto abstracto, que encapsule el protocolo común para el patrón Observer y después implementar aspectos concretos que hereden y redefinan los mecanismos particulares a cada clase.




```
1  public aspect ColorObserver {
2
3  protected interface Subject  { };
4  protected interface Observer { };
5
6  private WeakHashMap perSubjectObservers;
7
8  declare parents: Point  implements Subject;
9  declare parents: Screen implements Observer;
10
11 private List getObservers(Subject subject) {
12      if (perSubjectObservers == null) {
13          perSubjectObservers = new WeakHashMap();
14      }
15      List observers = (List)perSubjectObservers.get(subject);
16      if ( observers == null ) {
17          observers = new LinkedList();
18          perSubjectObservers.put(subject, observers);
19      }
20      return observers;
21 }
22
23 public void addObserver(Subject subject, Observer observer) {
24      getObservers(subject).add(observer);
25 }
26
27 public void removeObserver(Subject subject, Observer observer) {
28      getObservers(subject).remove(observer);
29 }
30
31 private pointcut subjectChange(Subject subject):
32     call(void Point.setColor(Color)) && target(subject);
33
34 after(Subject subject): subjectChange(subject) {
35      Iterator iter = getObservers(subject).iterator();
36      while ( iter.hasNext() ) {
37          updateObserver(subject, ((Observer)iter.next()));
38      }
39 }
40
41 private void updateObserver(Subject subject, Observer observer) {
42      ((Screen)observer).display("Screen updated "+
43              "(point subject changed color).");
44 }
45}
```

Fig. II-6. Implementación en AspectJ del aspecto de la participación de Point en el patrón.

El resultado es que las clase Point y Screen ya no poseen código correspondiente al patrón logrando así que cada una implemente un único *concern*. Las ventajas de esta implementación son las siguientes [Hannemann y Kiczales 2002]:

- *Localidad*: Todo el código correspondiente al patrón está localizado en los aspectos, además posibles cambios en los mecanismos del patrón están confinados a un único lugar.

- *Reusabilidad*: En caso de utilizar un aspecto abstracto, es posible reutilizar el mismo código cada vez que se usen instancias de este patrón.

- *Transparencia de composición*: Dado que los participantes de la implementación no están acoplados al patrón, si un Subject o un Observer participa en múltiples relaciones Observer/Subject su código no se vuelve más complicado y las instancias del patrón no se confunden.

- *(Un) pluggability*: Dado que los Subject y los Observers no son conscientes de su rol en la instancia del patrón, es posible cambiar entre usar el patrón y no usarlo.

5 Conclusiones

La Programación Orientada a Aspectos (POA) es un paradigma que permite alcanzar una mejor *separación de concerns* en el diseño de sistemas de software. Para esto, provee de un nuevo mecanismo (el aspecto) que permite encapsular *crosscutting concers*, los cuales son *concerns* (algo de interés en un sistema de software) cuya implementación es imposible de localizar en una única unidad mediante los mecanismos provistos por anteriores paradigmas de programación. La presencia de *crosscutting concerns* conlleva a la aparición de síntomas de código *scattering*, código correspondiente a un *concern* esparcido a través de diferentes unidades de implementación y código *tangling*, el cual se da cuando existe código que pertenece a más de un *concern* en una unidad de implementación. Posterior al surgimiento de la POA, surge el Desarrollo de Software Orientado a Aspectos (DSOA), el cual subsume a la

POA y provee de técnicas para manejar los *crosscutting concerns* durante todas las etapas del ciclo de vida de un desarrollo de software y no solo durante la implementación o diseño del mismo. Dentro de los lenguajes o plataformas que dan soporte a la orientación a aspectos, AspectJ no sólo es el más utilizado por los desarrolladores sino que también los conceptos que este introdujo (*pointcuts*, *advices*, *inter-type declarations*, etc) fueron de gran influencia al definirse nuevos lenguajes para el paradigma (por ejemplo, JBoss). Finalmente, las ventajas de utilizar este nuevo paradigma de programación se traducen en la obtención de implementaciones con mayor grado de localidad y reusabilidad.

Capítulo III

Evolución del Software y *Aspect Mining*

El término 'evolución' describe un fenómeno encontrado en diferentes dominios. Clases de entidades tales como especies, sociedades, ciudades, artefactos, conceptos, teorías, ideas, entre otras, evolucionan en el tiempo, cada una en su propio contexto. Los programas de software también evolucionan o como dijo Parnas "envejecen" provocando que la calidad interna de los mismos comience a decaer. El presente capítulo introduce la problemática relacionada a la evolución de los sistemas legados orientados a objetos, y cómo mediante la identificación de crosscuting concerns sobre el código legado (aspect mining) y su refactorización en forma de aspectos es posible extender el tiempo de vida útil de los mismos.

1 Sistemas Legados

Un legado es algo valioso que fue heredado, análogamente, el software legado es software valioso que ha sido heredado. El hecho de que haya sido heredado puede significar que sea un poco antigüo [Demeyer et al. 2002], por lo que el mismo puede haber sido desarrollado utilizando un lenguaje de programación obsoleto, o un método de desarrollo que ya no se utiliza. Por otra parte, según [Demeyer et al. 2002], no es la edad lo que convierte una pieza de software en un sistema legado, sino la tasa en que ha sido desarrollada y adaptada sin haber sido reestructurada.

Los sistemas legados son a menudo sistemas de negocio críticos [Sommerville 2004], los cuales se mantienen porque es demasiado arriesgado y costoso reemplazarlos. Por ejemplo, para la mayoría de los bancos el sistema contable de clientes fue uno de los primeros sistemas. Las políticas y procedimientos organizacionales pueden depender de este sistema. Si el banco fuera a descartar y reemplazar el sistema contable de clientes, entonces habría un serio riesgo de negocio si el sistema de recambio no funcionara adecuadamente. Además, los procedimientos existentes tendrían que cambiar, provocando que las personas que trabajan con dichos procedimientos se deban adaptar a los nuevos mecanismos. Por lo tanto, algunas veces es demasiado caro y peligroso descartar tales sistemas de negocio críticos después de unos pocos años de uso. Su desarrollo continúa toda su vida con cambios para satisfacer nuevos requerimientos, nuevas plataformas operativas, y así sucesivamente [Sommerville 2004].

Mantener un sistema legado en funcionamiento evita los riesgos asociados a su reemplazo, aunque los cambios que se deben realizar al mismo son más costosos a medida que pasa el tiempo. Existen varias razones por las que un sistema de software legado es difícil de modificar [Demeyer et al. 2002]:

- **Documentación obsoleta o falta de la misma**. Documentación obsoleta es un claro síntoma de un sistema que ha sufrido muchos cambios. La ausencia de documentación es un problema a largo plazo, el cual se hará real en el mismo instante en que los desarrolladores actuales dejen el proyecto.

- **Ausencia de *tests***. Más importante que la presencia de documentación actualizada es la ausencia de *tests* de unidad para todos los componentes del sistema, y de *tests* de sistema para todos los escenarios y casos de uso más importantes.

- **Los desarrolladores y usuarios originales se han ido**. A menos que se posea un sistema con documentación actualizada y buena cobertura de *tests*, el mismo comenzará a transformarse en un sistema de menor calidad cuya documentación es cada vez peor.

- **Pérdida del conocimiento sobre el sistema**. Como la documentación está desactualizada respecto al código fuente, nadie realmente entiende el funcionamiento interno del sistema.

- **Entendimiento limitado del sistema en su conjunto**. No sólo nadie entiende el funcionamiento interno del sistema en detalle, sino que el entendimiento global del mismo también es limitado.

- **Demasiado tiempo para llegar a producción**. En alguna parte el proceso no esta funcionando, puede que tome demasiado tiempo aprobar los cambios, tal vez los *tests* de regresión no estén funcionando, o los cambios tardan demasiado en distribuirse (*deploy*). A menos que se comprendan y atiendan estas cuestiones las mismas sólo empeorarán.

- **Demasiado tiempo para hacer cambios pequeños**. El sistema es tan complejo que hasta los cambios más simples al mismo toman mucho tiempo en realizarse. La consecuencia es que los grandes cambios que deban hacerse al sistema deberán retrasarse indefinidamente.

- **Necesidad constante de corregir defectos (*bugs*)**. Los defectos parecen nunca desaparecer. Cada vez que se arregla un defecto, otro surge a continuación. Esto es un síntoma de que la aplicación es tan compleja que es muy difícil analizar el impacto que puede tener un cambio. Además, la arquitectura de la aplicación ya no satisface las necesidades, por lo que inclusive los pequeños cambios tendrán consecuencias inesperadas.

- **Gestión de las dependencias**. Cuando se soluciona un defecto en un lugar, otro defecto surge en otro lugar. Por lo general, este es un síntoma de que la arquitectura se ha deteriorado tanto que los componentes del sistema inicialmente independientes ya no lo son tanto.

- **Grandes tiempos de construcción (*build time*)**. Largos tiempos de recompilación reducen la capacidad de efectuar cambios al sistema. A su vez, indican que el sistema es complejo inclusive para que el compilador realice su trabajo eficientemente.

- **Dificultad para separar productos**. Si existen diferentes clientes para el mismo producto, y se tienen problemas para ajustar las entregas (*releases*) para cada cliente, entonces la arquitectura del producto no está cumpliendo su trabajo.

- **Código duplicado**. El código duplicado surge naturalmente ha medida que el sistema evoluciona, como atajo para implementar código casi idéntico, o para juntar diferentes versiones de un sistema de software. Si el código duplicado no es eliminado refactorizando las partes comunes en abstracciones adecuadas, el mantenimiento se vuelve muy dificultoso ya que el mismo código debe ser arreglado en diferentes partes.

- ***Code smells***. Los *code smells* [Fowler 1999] son ciertas estructuras observables en el código que sugieren la posibilidad de la aplicación de un *refactoring* [Fowler 1999]. Ejemplos de *code smells* son código duplicado, clases de tamaños excesivos, o la presencia de sentencias condicionales (cases o switchs) entre otros. Los mismos son la evidencia de que un sistema ha sido expandido y adaptado muchas veces sin haber sido aplicadas actividades de reingeniería.

Aquellas organizaciones que cuentan con un gran número de sistemas legados se enfrentan a un dilema fundamental. Si continúan usando sus sistemas legados y haciendo cambios sobre los mismos, sus costos de mantenimiento aumentarán inevitablemente. Si deciden reemplazar sus sistemas legados con sistemas nuevos, puede que estos no provean un soporte tan efectivo a sus negocios como los sistemas legados. Consecuentemente, muchas de estas organizaciones están buscando técnicas que permitan extender el tiempo de vida de sus sistemas legados y que permitan reducir los costos de mantener a estos en funcionamiento.

2 Dinámica de Evolución de los Programas

La ley de entropía del software dictamina que la mayoría de los sistemas pasado un tiempo tienden a decaer gradualmente en su calidad, a menos que estos sean mantenidos y adaptados a los requerimientos cambiantes [Wilde y Huitt 1992]. En tal sentido, Lehman y Belady [Lehman y Belady 1985] establecieron un conjunto de leyes (más bien hipótesis) concernientes a los cambios de los sistemas, las cuales se describen a continuación:

- *Ley del cambio continuado*: Un programa que se usa en un entorno real necesariamente debe cambiar o se volverá progresivamente menos útil en ese entorno.

- *Ley de la complejidad creciente*: A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja. Se deben dedicar recursos extras para preservar y simplificar la estructura.

- *Ley de la evolución prolongada del programa*: La evolución de los programas es un proceso

autorregulativo. Los atributos de los sistemas, tales como el tamaño, tiempo entre entregas y el número de errores documentados, son aproximadamente invariantes para cada entrega del sistema.

- *Ley de la estabilidad organizacional*: Durante el tiempo de vida de un programa, su velocidad de desarrollo es aproximadamente constante e independiente de los recursos dedicados al desarrollo del sistema.

- *Ley de la conservación de la familiaridad*: Durante el tiempo de vida de un programa, el cambio incremental en cada entrega es aproximadamente constante.

- *Ley del crecimiento continuado*: La funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción de los usuarios.

- *Ley del decremento de la calidad*: La calidad de los sistemas comenzará a disminuir a menos que dichos sistemas se adapten a los cambios en su entorno de funcionamiento.

- *Ley de la realimentación del sistema*: Los procesos de evolución incorporan sistemas de realimentación multiagente y multibucle y éstos deben ser tratados como sistemas de realimentación para lograr una mejora significativa.

Del conjunto de leyes o hipótesis detalladas anteriormente, los dos primeras son las más importantes. La primera ley establece que el mantenimiento de los sistemas es un proceso inevitable. A medida que el entorno cambia, surgen nuevos requerimientos y el sistema debe ser modificado. Cuando el sistema modificado vuelve a introducirse en el entorno, éste promueve más cambios en el entorno, de forma que el proceso de evolución se recicla. La segunda ley establece que, a medida que se cambia un sistema, su estructura se degrada. La única forma de evitar este hecho, es invertir en mantenimiento preventivo en el que se consume tiempo mejorando la estructura del software sin añadir funcionalidades [Sommerville 2004].

Por su parte, Parnas [Parnas 1994] establece que existen dos causas por las cuales un sistema puede envejecer, la primera se debe a la incapacidad de los dueños del producto de adaptar este a las necesidades cambiantes; la segunda, es el resultado de los cambios realizados sobre el producto. Respecto de la primera causa, Parnas afirma que, a menos que el software sea actualizado frecuentemente, sus usuarios se volverán progresivamente insatisfechos con el mismo y cambiarán a otro producto en cuanto puedan. La segunda de las causas, hace referencia a los cambios que se deben realizar a un sistema de software. Según Parnas, si estos cambios son realizados por desarrolladores que no entienden el concepto del diseño original, los mismos causarán la degradación de la estructura interna del programa. De esta manera, el software que ha sido repetidamente modificado (o mantenido) de esta manera se vuelve muy costoso de actualizar. El corolario de este hecho es que los desarrolladores sienten que no poseen tiempo para actualizar la documentación.

Por lo tanto, se desprende que tanto Lehman y Belady como Parnas coinciden en las causas que provocan la evolución de un sistema de software y el problema que surge cuando se introducen nuevos cambios en un sistema de software.

En consecuencia, el tiempo de vida de un sistema de software puede ser extendido manteniéndolo o reestructurándolo. Pero, un sistema legado no puede ser ni reemplazado ni actualizado excepto a un alto costo. Por lo que el objetivo de la reestructuración es reducir la complejidad del sistema legado lo suficiente como para poder ser usado y adaptado a un costo razonable [Demeyer et al. 2002].

3 Mantenimiento de Sistemas de Software

El mantenimiento del software es el proceso general de cambiar un sistema después de que éste ha sido entregado [Sommerville 2004]. El término aplica normalmente a software a medida en donde grupos de desarrollo distintos están implicados antes y después de la entrega. Los cambios realizados al software pueden ser sencillos para corregir errores de código, cambios más extensos para corregir errores de diseño o mejoras significativas para corregir errores de especificación o acomodar nuevos requerimientos. Los cambios se implementan modificando los componentes del sistema existente y añadiendo nuevos componentes al sistema donde sea necesario.

Existen tres tipos diferentes de mantenimiento de software [IEEE 1998]:

- *Mantenimiento para reparar defectos del software (o correctivo)*. Por lo general, los errores de código son relativamente baratos de corregir; los errores de diseño son mucho más caros ya que implican reescribir varios componentes de los programas. Los errores de requerimientos son los más caros de reparar debido a que puede ser necesario un rediseño extenso del sistema.

- *Mantenimiento para adaptar el software a diferentes entornos operativos (o adaptativo)*. Este tipo de mantenimiento se requiere cuando cambia algún aspecto del entorno del sistema, como por ejemplo el hardware, la plataforma del sistema operativo u otro software de soporte. El sistema de aplicaciones debe modificarse para adaptarse a estos cambios en el entorno.

- *Mantenimiento para añadir o modificar las funcionalidades del sistema (o perfectivo)*. Este tipo de mantenimiento es necesario cuando los requerimientos del sistema cambian como respuesta a cambios organizacionales o del negocio.

Los trabajos de Lientz y Swanson [Lientz y Swanson 1980], y Nosek y Palvia [Nosek y Palvia 1990] sugieren que alrededor del 65% del mantenimiento está relacionado con la implementación de nuevos requerimientos, el 18% con cambios en el sistema para adaptarlo a un nuevo entorno operativo y el 17% para corregir defectos del sistema (Fig. III-1).

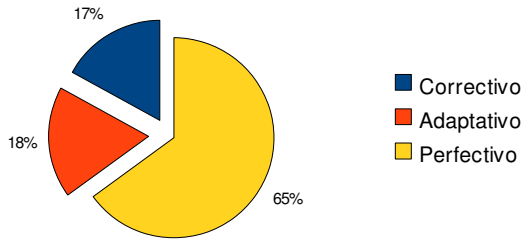


Fig. III-1. Distribución de los tipos de mantenimiento.

Por lo tanto, la evolución del sistema para adaptarse a nuevos entornos y requerimientos nuevos o cambios en los mismos consume la mayor parte del esfuerzo de mantenimiento.

4 Evolución de Sistemas Orientados a Objetos

Según [Demeyer et al. 2002] cualquier sistema exitoso sufrirá los síntomas de los sistemas legados. En particular, los sistemas legados orientados a objetos son exitosos sistemas orientados a objetos cuya arquitectura y diseño ya no responden a los requerimientos cambiantes. Una cultura de continua reingeniería es un pre-requisito para alcanzar sistemas orientados a objetos flexibles y mantenibles. Si bien, estos procesos de reingeniería procuran mejorar la estructura interna de un sistema orientado a objetos, la presencia de *crosscutting concerns* es el principal obstáculo a la evolución de los mismos. Por lo tanto, estos procesos de reingeniería deberán llevarse a cabo con el objetivo de obtener un nuevo sistema que encapsule los *crosscutting concerns* como aspectos.

4.1 Reingeniería de Software

La esencia de la reingeniería de software es mejorar o transformar software existente de forma que pueda ser entendido, controlado, y usado como si fuese nuevo. El concepto de reingeniería se define como la evaluación y alteración de un sistema objetivo para reconstituirlo en una nueva forma y la subsecuente implementación de esa nueva forma [Chikofsky y Cross 1992]. Básicamente, este proceso toma sistemas legados costosos en su mantenimiento, o cuya arquitectura o implementación son obsoletos, y los rehace mediante la utilización de tecnología de software o hardware actual. La reingeniería de sistemas es importante para recuperar y reutilizar activos existentes de software, controlar los altos costos de mantenimiento del mismo, y establecer una base para su futura evolución.

Un proceso de reingeniería, toma como entrada el código fuente de un sistema legado y genera como salida el código fuente de un nuevo sistema. Por su parte, la ingeniería inversa, se define como el proceso de analizar un sistema objetivo para identificar sus componentes y relaciones y crear representaciones del mismo en otra forma o en un nivel mayor de abstracción [Chikofsky y Cross 1992]. Mientras que la ingeniería inversa se encarga de entender como funciona un sistema, la reingeniería se encarga de reestructurar un sistema en preparación para futuros desarrollos y extensiones. La ingeniería directa, en tanto, es el proceso tradicional de moverse de diseños y abstracciones independientes de la implementación a la implementación física de un sistema [Chikofsky y Cross 1992].

Si la ingeniería directa implica moverse desde vistas de alto nivel como requerimientos y modelos hacia realizaciones concretas, entonces la ingeniería inversa implica moverse desde alguna realización concreta a modelos más abstractos, y la reingeniería implica transformar una implementación concreta en otra implementación concreta (Fig. III-2) [Demeyer et al. 2002].

El punto clave a observar, es que la reingeniería no es sólo una cuestión de transformar el código fuente, sino transformar un sistema en todos sus niveles. Es por esto, que tiene sentido hablar de ingeniería inversa y directa en el mismo contexto. En un típico sistema legado, es posible que no solo el código sino toda la documentación y especificación sean inconsistentes. La ingeniería inversa es por lo tanto un prerequisito a la reingeniería debido a que no se puede transformar lo que no se entiende [Demeyer et al. 2002].

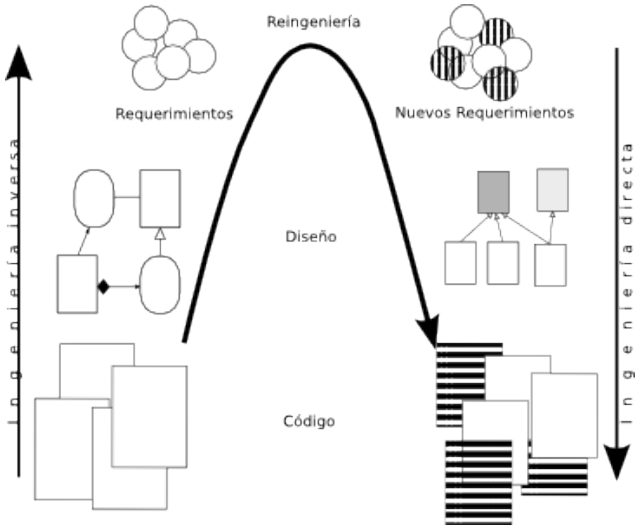


Fig. III-2. Ingeniería reversa, directa y reingeniería de un sistema legado.

La transformación efectiva del diseño o del código dentro de un proceso de reingeniería, es conocida como reestructuración. Según [Chikofsky y Cross 1992], la reestructuración es la transformación desde una representación hacia otra que se encuentra en su mismo nivel de abstracción, preservando a su vez el comportamiento externo del sistema.

Dentro de la investigación de trasformación de programas, se distinguen dos enfoques de reestructuración diferentes [Visser 2001]. El término *rephrasing* es utilizado para referirse a técnicas que mejoran la estructura del software sin cambiar el lenguaje de implementación. El ejemplo típico es *software refactoring* [Opdyke 1992], una técnica que intenta mejorar la estructura interna de un programa sin cambiar su comportamiento externo. Por otra parte, el término *translation* se refiere a técnicas que reestructuran el software a través de lenguajes de programación. Un ejemplo típico es la migración de código legado a un nuevo paradigma que permita mejoras en términos de calidad interna del mismo.

4.2 El Problema de los *Crosscutting Concerns*

Uno de los principales problemas para el desarrollo del software es la "tiranía de la descomposición dominante" [Hannemann y Kiczales 2001], no importa cuan bien un sistema de software se descomponga en unidades modulares, siempre existirán *concerns* que cortan transversalmente la descomposición elegida. El código de estos *crosscutting concerns* estará presente sobre diferentes módulos, lo que tiene un impacto negativo sobre la calidad del software en términos de comprensión, adaptación y evolución. Por ejemplo, en la Fig. III-3, se observa el código extraído de un método de la aplicación Apache Tomcat [Tomcat] en el cual sólo el 10% del código corresponde a la funcionalidad original del método (eliminar un hijo de un contenedor). El resto del código corresponde a *crosscutting concerns* tales como la sincronización, el manejo del ciclo de vida de componentes (presente en al menos cuarenta clases), el *logging* de información y la notificación de eventos. Algunos de los *crosscutting concerns* más importantes presentes en el código de Apache Tomcat son discutidos en [Marin et al. 2007].

La presencia de código *tangling* como se muestra en la Fig. III-3, dificulta las tareas de comprensión del código, las cuales juegan un rol fundamental al momento de hacer el mantenimiento de un sistema de software. Por lo que, desde el punto de vista de comprensión de un programa, los *crosscuting concerns* son un problema por dos razones. Primero, descubrir o entender la implementación de este tipo de *concern* es difícil ya que el mismo no está localizado en un único módulo sino que está disperso (*scattered*) a través de muchos de ellos. Segundo, entender la implementación de los módulos en sí mismo se vuelve más complicado, ya que el código de los diferentes *concerns* está mezclado (*tangled*) con la funcionalidad principal de estos.

A su vez, si la evolución del sistema genera cambios sobre alguno de los *crosscutting concerns* el costo del cambio dependerá del nivel de dispersión que el *concern* posea sobre el código fuente. Para el caso del *concern* "ciclo de vida"

del ejemplo anterior el costo del cambio deberá multiplicarse por cuarenta, ya que son cuarenta las clases donde ese *concern* está presente.

Por lo tanto, la migración de un sistema legado orientado a objetos hacia un sistema orientado a aspectos favorecerá la separación de estos *crosscuting concerns* del código base. La consecuencia es un sistema más fácil de entender, mantener y evolucionar.

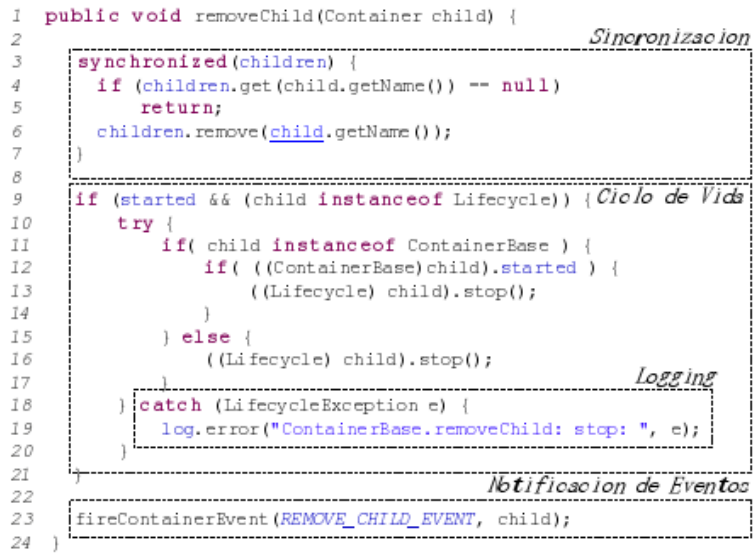


Fig. III-3. Método extraído del servidor de aplicaciones *open-source* Apache Tomcat, en el cual se observa una gran cantidad de código correspondientes a *crosscutting concerns*.

4.3 Aspect Mining

Para que la POA [Kiczales et al. 1997] sea verdaderamente exitosa, es necesario migrar los sistemas de software existentes hacia su equivalente orientado a aspectos y reestructurarlos de manera continua. Sin embargo, aplicar manualmente técnicas orientadas a aspectos a un sistema legado es un proceso difícil y tendiente al error [Kellens et al. 2007]. Debido al gran tamaño de estos sistemas, la complejidad de su implementación, la falta de documentación y conocimiento sobre el mismo, es que existe la necesidad de herramientas y técnicas que ayuden a los desarrolladores a localizar y documentar los *crosscutting concerns* presentes en esos sistemas.

El estudio y desarrollo de tales técnicas es el objetivo de *aspect mining* y *aspect refactoring*. *Aspect mining* [Kellens et al. 2007] es la actividad de descubrir *crosscuting concerns* que potencialmente podrían convertirse en aspectos. A partir de allí, estos *concerns* pueden, ser encapsulados en nuevos aspectos del sistema (mediante la utilización de técnicas conocidas como *aspect refactoring* [Kellens et al. 2007]) o podrían documentarse con el objetivo de mejorar la comprensión del programa.

4.3.1 Clasificación de Técnicas de Descubrimiento de Aspectos

El proceso de migrar un sistema legado a un sistema basado en aspectos consiste de dos pasos, el descubrimiento de aspectos candidatos y la refactorización de algunos de ellos en aspectos. Según Kellens, Mens y Tonella [Kellens et al. 2007] existen tres tipos de enfoques para el descubrimiento de aspectos:

-Técnicas de descubrimiento de *early-aspects* implica descubrir aspectos durante etapas tempranas del ciclo de vida del software, tales como requerimientos, análisis de dominio o diseño de la arquitectura. Aunque, en el contexto de sistemas legados, donde los documentos de requerimientos y de la arquitectura no están actualizados, estas técnicas no pueden ser aplicadas.

***-Browsers* específicos** que ayudan a los desarrolladores a navegar manualmente el código fuente de un sistema para explorar los *crosscuting concerns*.

-Técnicas de *aspect mining* automatizan el proceso de descubrimiento de aspectos y proponen a su usuario uno o más aspectos candidatos. Este tipo de técnicas razonan acerca del código fuente del sistema o sobre los datos obtenidos mediante la ejecución o manipulación del código.

Luego, en base a la clasificación anterior, Kellens, Mens y Tonella [Kellens et al. 2007] proponen la siguiente definición de *aspect mining*:

> "*Aspect mining* es la actividad de descubrir aquellos *crosscutting concerns* que potencialmente podrían convertirse en aspectos, desde el código fuente y/o desde el comportamiento del sistema de software. Nos referimos a tales *concerns* como 'aspectos candidatos'".

Según [Marin et al. 2007], el origen de *aspect mining* está emparentado con técnicas correspondientes al problema de asignación de concepto (el problema de descubrir conceptos de dominio y asignarlos a sus realizaciones dentro de un programa en particular [Biggerstaff et al. 1994]). Trabajos dentro de este problema han resultado en áreas de investigación tales como *feature location* [Koschke y Quante 2005; Wilde y Scully 1995; Xie et al. 2006], *design pattern mining* [Ferenc et al. 2005], y *program plan recognition* [Rich y Wills 1990; Wills 1990; van Deursen et al. 2000].

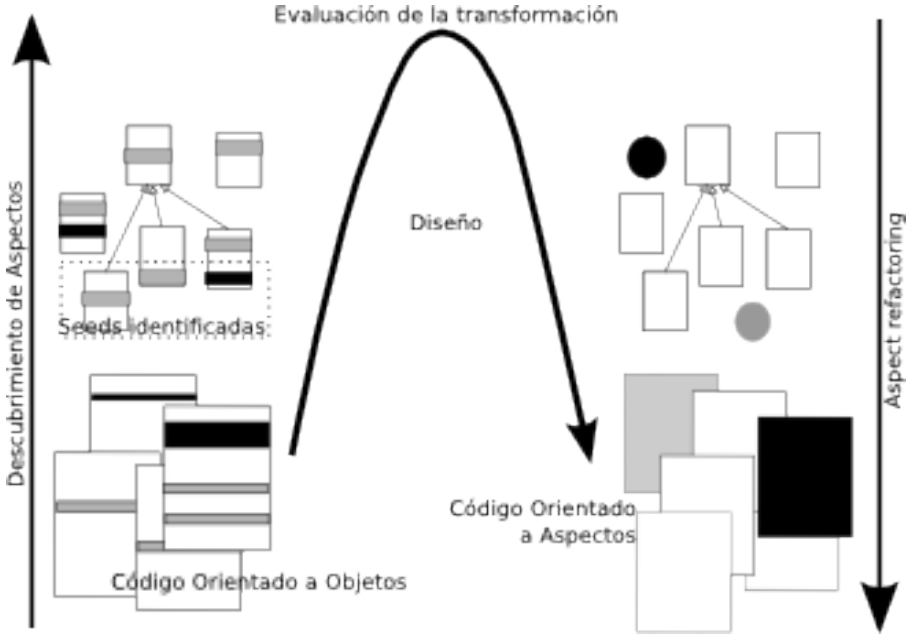


Fig. III-4. Ciclo de reingeniería revisado.

4.3.2 El Ciclo de Reingeniería Revisado

El objetivo de las técnicas de *aspect mining* es el de identificar *crosscutting concerns* en código no orientado a aspectos. Esta tarea implica la búsqueda de elementos del código fuente que pertenezcan a la implementación de algún *crosscutting concern*, estos elementos del código se denominan aspectos candidatos o *seeds* [Marin et al. 2007; Kellens et al. 2007]. La mayoría de las técnicas de *aspect mining* son semi-automáticas [Ceccato et al. 2005], por lo que requieren interacción con el desarrollador. Como consecuencia, la salida de las mismas es un conjunto de *seeds* candidatas [Marin et al. 2007], las cuales pueden convertirse en un conjunto de *seeds* confirmadas (o *seeds*) si son aprobadas por el desarrollador, o no-*seeds* si las mismas son desechadas (también llamadas *falsos positivos*). A su vez, un *falso negativo* es una parte de un *crosscutting concern* conocido, potencialmente detectable por la técnica, pero no encontrado debido a las limitaciones inherentes a la misma. El desafío principal de las técnicas de *aspect mining* es mantener el porcentaje de *seeds* confirmadas lo más alto posible sobre el total de *seeds* candidatas, sin aumentar el número de falsos negativos [Marin et al. 2007].

Por su parte, en [Marin et al. 2007] se distingue entre la intención (*intent*) y la extensión (*extent*) de un *concern*. Donde la intención de un *concern* se define como el objetivo del mismo, por ejemplo, la intención del *concern tracing* es la de registrar todas las llamadas a los métodos públicos de un sistema. La extensión de un *concern* es la representación concreta del mismo en el código fuente del sistema, para el ejemplo del *concern* de *tracing* esta será el conjunto de todas las sentencias que efectivamente generan las trazas.

En base a esta diferenciación entre intención y extensión de un *concern*, es posible realizar las siguientes afirmaciones:

- La salida de las técnicas de *aspect mining*, denominadas *seeds*, son un subconjunto de la extensión de un

posible *crosscutting concern*.

- Las técnicas de *aspect refactoring* necesitan, para llevar a cabo su trabajo, la extensión del *crosscutting concern* objetivo.

- Es necesario, por lo tanto, la utilización de *browsers* específicos que permitan la localización y extracción de un *crosscutting concern*.

Finalmente, el ciclo de reingeniería aplicado a un sistema legado orientado a objetos comenzará mediante la aplicación de técnicas de *aspect mining* y navegación del código (*browsers* específicos) de manera de identificar y localizar todos los *crosscutting concerns* de un sistema. Luego, la ingeniería directa se realizará mediante la aplicación de *refactorings* orientados a aspectos, que transformen incrementalmente la implementación orientada a objetos en una implementación orientada a aspectos. A su vez, es posible incluir una tercera actividad posterior al descubrimiento de aspectos que evalúe mediante la aplicación de métricas de software [Sant' Anna et al. 2003] el impacto que tendrá en términos de reuso y modificabilidad la reestructuración hacia aspectos del código legado. Este nuevo ciclo de reingeniería puede observarse en la Fig. III-4.

5 Conclusiones

La evolución de un sistema de software es un proceso inevitable en cualquier sistema de software considerado exitoso. La misma se dará inclusive en sistemas desarrollados bajo el paradigma orientado a objetos, en los cuales el principal problema para su correcta evolución son los denominados *crosscutting concerns*. La POA permite solucionar el problema de los *crosscutting concerns* mediante la introducción de un nuevo constructor llamado aspecto, con el que es posible encapsular un *crosscutting concern*. La migración de un sistema legado hacia un sistema orientado a aspectos se realizará en dos etapas bien definidas. La primera de ellas, denominada *aspect mining*, tiene como objetivo la identificación de los *crosscutting concerns* en el código legado. La segunda, denominada *aspect refactoring*, consiste en la transformación de los *crosscutting concerns* en aspectos. Ambas etapas permiten proponer un nuevo ciclo de reingeniería que describe los pasos que se deben tomar para transformar un sistema orientado a objetos en uno orientado a aspectos.

Capítulo IV

Aspect Mining: Resumen de Técnicas y Enfoques

Aspect mining es la tarea de descubrir aquellos crosscutting concerns que podrían potencialmente convertirse en aspectos, desde el código fuente y/o el comportamiento en tiempo de ejecución de un sistema de software. En este capítulo, se describen los trabajos existentes en el área, para luego mostrar un análisis y una clasificación de los mismos. Por cada trabajo analizado en este capítulo, se incluye el enfoque propuesto, se muestra un ejemplo de su aplicación y se describe la herramienta que da soporte al mismo.

1 Búsqueda de Patrones Recurrentes en las Trazas de Ejecución

Breu y Krinke [Breu y Krinke 2004] propusieron analizar las trazas de ejecución del sistema bajo análisis en búsqueda de patrones recurrentes de ejecución. En este enfoque, primero se obtienen las trazas de ejecución y luego son analizadas en búsqueda de patrones de ejecución recurrentes. Diferentes restricciones especifican cuándo un patrón de ejecución es recurrente (y por lo tanto, un aspecto potencial).

Este tipo de patrones describen ciertos aspectos del comportamiento del sistema de software. Los autores, esperan que los patrones recurrentes de ejecución sean potenciales *crosscuting concerns* que describen funcionalidad recurrente en el programa y por lo tanto posibles aspectos.

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

Fig. IV-1. Traza de ejemplo.

1.1 Enfoque Propuesto

El enfoque propuesto puede descomponerse en dos pasos bien definidos.

1. Investigar la traza del programa y abstraer relaciones de ejecución.
2. Obtener aspectos candidatos a partir de las relaciones que cumplan una serie de restricciones predefinidas.

A continuación, se desarrollan ambos pasos.

1.1.1 Clasificación de Relaciones de Ejecución

Para detectar los patrones presentes en las trazas del programa, es necesaria una clasificación de las posibles formas que estos pueden tomar. Por tal motivo, los autores introducen el concepto de relaciones de ejecución, las cuales describen la

relación entre la ejecución de dos métodos en la traza del programa.

La traza del programa es la secuencia de invocaciones de métodos durante la ejecución del mismo. Por cada método se considera la entrada y la salida del mismo. Formalmente, la traza de programa T_p de un programa P con N_p (signatura de los métodos) es definida como una lista $[t_1,..., t_n]$ de pares $t_i \in (N_p \; x \; \{ent, ext\})$, donde *ent* marca la entrada al método en ejecución, y *ext* marca la salida. La Fig. IV-1 muestra un ejemplo de traza de programa.

Para que las trazas de programas sean más fáciles de leer, los puntos *ent* y *ext* fueron reemplazados por '{' y '}' respectivamente (Fig. IV-1).

De esta manera se distinguen cuatro relaciones de ejecución:

- *externa-anterior*. Un método es ejecutado antes que el otro método sea ejecutado. Por ejemplo, B() en la línea 1 es ejecutado antes que A() (linea 7) sea ejecutado.
- *externa-posterior*. Un método es ejecutado después que el otro método sea ejecutado. Por ejemplo, A() en la línea 7 es ejecutado posteriormente a la ejecución de B() (linea 1).
- *interna-primera*. Un método es el primero en ejecutarse dentro de la invocación del método precedente. Por ejemplo, G() (linea 3) es el primero en llamarse dentro de la ejecución de C() (linea 2).
- *interna-última*. Un método es el último en ejecutarse dentro de la invocación del método precedente. Por ejemplo, H() (linea 4) es el último en llamarse dentro de la ejecución de C() (linea 2).

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

Fig. IV-2. Conjuntos de relaciones de ejecución. Primero se observan las relaciones de tipo externa-anterior, luego las de tipo interior-primera y finalmente las de interior-última.

En la Fig. IV-2 se muestran los conjuntos de relaciones de ejecución que se obtuvieron a partir de la traza de la Fig. IV-1. No se muestra el conjunto de relaciones externa-posterior, ya que el mismo se puede calcular revirtiendo el conjunto de relaciones externa-anterior.

1.1.2 Restricciones Sobre las Relaciones de Ejecución

Para decidir bajo qué circunstancias ciertas relaciones de ejecución son patrones recurrentes y por lo tanto potenciales *crosscuting concerns* en el sistema, se definen las siguientes restricciones:

Restricción de Uniformidad. Sea una relación exterior-anterior entre *u* y *v*, la misma será recurrente si cada ejecución de *v* es precedida por una ejecución de *u*. Análogamente se aplica a una relación exterior-posterior. A su vez, sea una relación interior (*u* invocado desde *v*) la misma será un patrón recurrente si *v* nunca ejecuta otro método más que *u* como primer (o último) método dentro de su cuerpo.

Restricción Crosscuting. Una relación de ejecución es denominada *crosscutting* si la misma ocurre en más de un contexto de llamada en la traza del programa. Para relaciones de ejecución interior (*u* invocado desde *v*) el contexto de llamada es el método que rodea la ejecución *v*. Para relaciones exteriores (*u* antes o después de *v*), el contexto de llamada es el método *v* invocado antes (o después) del cual el método *u* es siempre ejecutado.

1.2 Ejemplo

Continuando el ejemplo previamente introducido, en la Fig. IV-3 se observan los conjuntos de relaciones obtenidas luego de aplicar las restricciones de uniformidad y *crosscutting* sobre los conjuntos de relaciones de la Fig. IV-2. De la aplicación de la restricción *crosscutting* sobre el conjunto de relaciones uniformes, se obtienen tan sólo cuatro relaciones, las cuales son consideradas como aspectos candidatos.

Si se busca cada una de estas en la traza de la Fig. IV-1 se observará que las mismas no sólo son uniformes sino que también ocurren en diferentes contextos de ejecución. Por lo tanto, este tipo de relaciones permiten descubrir síntomas

de *tangling* ya que identifican métodos invocados de manera consistente desde diferentes lugares en el sistema.

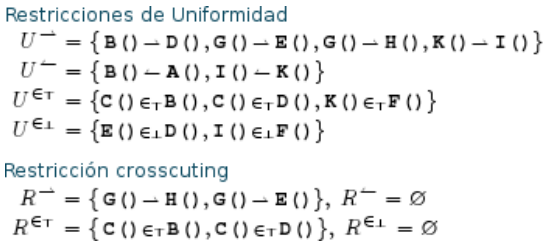


Fig. IV-3. Conjuntos de relaciones de ejecución resultantes de la aplicación de las restricciones de uniformidad y *crosscuting*.

1.3 Herramienta

Los autores hacen mención acerca de una herramienta llamada DynAMiT (*Dynamic Aspect Mining Tool*), la cual implementa el algoritmo propuesto. Aunque, la misma no se encuentra disponible para su utilización.

2 Enfoques Basados en *Formal Concept Analysis (FCA)*

Formal Concept Analysis [Ganter y Wille 1997] es una rama de *lattice theory* que provee una manera de identificar agrupaciones máximas de objetos que comparten los mismos atributos[3]. Dado un *contexto (O, A, R)*, donde *R* es una relación binaria entre objetos (del conjunto *O*) y atributos (del conjunto *A*), un *concepto c* se define como un par de conjuntos (*X*, *Y*) tal que:

$$X = \{o \in O \mid \forall a \in Y : (o, a) \in R\} . \qquad (1)$$
$$Y = \{a \in A \mid \forall o \in X : (o, a) \in R\} . \qquad (2)$$

donde *X* es la extensión (*extent*) (*Ext*[*c*]) del concepto *c* e *Y* es su intención (*intent)* (*Int*[*c*]). De esta manera, cada concepto consiste de un conjunto de objetos que poseen uno o más atributos en común tal que ningún otro objeto posea esos atributos o que no existan otros atributos declarados que tengan en común.

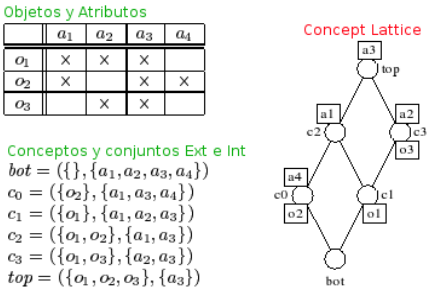


Fig. IV-4. Ejemplo de un *lattice* de conceptos.

La Fig. IV-4 muestra un ejemplo de contexto (top), en conjunto con los conceptos que pueden ser calculados para el mismo (bot). La relación de inclusión entre los conjuntos *extent* (o *intent*) de los diferentes conceptos define un orden parcial sobre el conjunto de todos los conceptos, la cual es mostrada como un *lattice* o entrelazado (Fig. IV-4 a la derecha). Dado un concepto c, los objetos y atributos que lo etiquetan indican las propiedades más específicas (objetos y atributos) que lo caracterizan.

A continuación, se describen dos técnicas de *aspect mining* que utilizan FCA para identificar *crosscutting concerns*.

3 Existen otras publicaciones [Tourwé y Mens 2004] que utilizan los términos *elemento* y *propiedad* para referirse a *objeto* y *atributo* respectivamente. Esta diferencia la realizan ya que consideran que objeto y atributo poseen significados muy específicos en el desarrollo de software orientado a objetos.

2.1 Aplicación de *Formal Concept Analysis* a las Trazas de Ejecución

Tonella y Ceccato [Tonella y Ceccato 2004] propusieron una técnica de análisis dinámico que se basa en la ejercitación de las unidades computacionales asociadas a las principales funcionalidades de la aplicación y su posterior análisis mediante FCA. La relación entre las trazas de ejecución asociadas a tales funcionalidades y las unidades computacionales (métodos de clases) invocadas durante cada ejecución es examinada mediante la exploración del *lattice* de conceptos producido por FCA. En tal estructura, es posible aislar los nodos (conceptos) que son específicos a una única funcionalidad y determinar qué clases están específicamente involucradas. Cuando una clase contribuye a múltiples funcionalidades, la presencia de un potencial *crosscuting concern* es detectada.

2.1.1 Feature Location

El objetivo de *feature location* [Eisenbarth et al. 2003] es identificar las unidades computacionales (por ejemplo, procedimientos o métodos de clases) que implementan específicamente una característica (por ejemplo, un requerimiento) de interés. Las trazas de ejecución obtenidas por correr el programa bajo escenarios dados proveen los datos de entrada (análisis dinámico). Luego, se aplica la técnica *formal concept analysis* a un contexto donde los atributos son unidades computacionales, los objetos son escenarios y la relación *R* contiene el par (*o*, *a*) si la unidad computacional *a* es ejecutada mientras el escenario *o* es llevado a cabo. La información sobre cual unidad computacional es ejecutada para cada escenario es obtenida recolectando las trazas de ejecución del programa.

Un concepto en el *lattice* de conceptos agrupa todas las unidades computacionales ejecutadas por todos los escenarios en su *extent*. Una unidad computacional etiqueta al mismo si es la unidad computacional más específica para los escenarios en el conjunto *extent* del concepto. Asumiendo que cada escenario se encuentra asociado a una única característica (o *feature*), el concepto específico para un escenario dado es aquel que posee sólo el escenario asociado en su *extent* (ver Fig. IV-4, concepto c_0 para el escenario o_2). Por lo tanto, las unidades computacionales específicas de una característica dada son aquellas que etiquetan dicho concepto. De esta manera, es posible determinar cuáles son las porciones de código más específicas que implementan la funcionalidad ejercitada por el escenario del conjunto *extent* de un concepto.

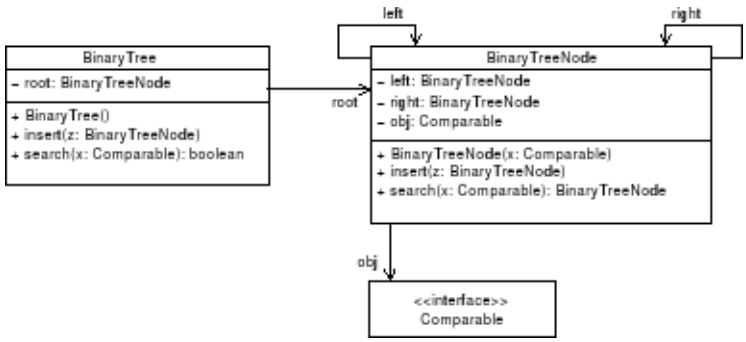


Fig. IV-5. Clases de la implementación del árbol de búsqueda binario.

2.1.2 Enfoque Propuesto

El primer paso en la migración de las aplicaciones existentes hacia POA consiste en identificar aquellos *crosscutting concerns* candidatos a ser implementados como aspectos. Este proceso, puede guiarse mediante la utilización de los casos de uso [Booch et al. 1999] de la aplicación, los cuales especifican la funcionalidad del sistema. Cuando tal funcionalidad es implementada por fragmentos de código distribuidos a través de diferentes unidades de modularización, entonces es posible transformarla en un aspecto.

El siguiente procedimiento detalla la utilización de *feature location* para *aspect mining*.

1) Obtener las trazas de ejecución mediante la ejecución del programa bajo análisis para un conjunto de escenarios (casos de uso).
2) La relación entre trazas de ejecución y las unidades computacionales ejecutadas es sometida a *conceptual analysis*. Donde las trazas de ejecución asociadas a los casos de uso son los objetos, mientras que los métodos de clase son los atributos.
3) En el *lattice* de conceptos resultante, los conceptos específicos a cada caso de uso o escenario son encontrados.
4) La reestructuración se realizará si las siguientes condiciones se cumplen:

(a) Un concepto específico de un caso de uso es etiquetado por unidades computacionales (métodos) que pertenecen a más de un módulo (clase).
(b) Diferentes unidades computacionales (métodos) de un mismo módulo (clase) etiquetan más de un concepto específico de un caso de uso.

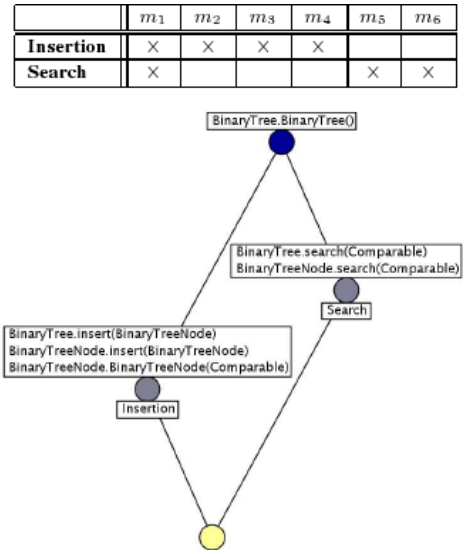


Fig. IV-6. *Lattice* de conceptos para el ejemplo analizado.

2.1.3 Ejemplo

Considerar una implementación de un árbol de búsqueda binario, la cual consiste de dos clases (Fig. IV-5). Las principales funcionalidades son la inserción de nodos y la búsqueda de un nodo en el árbol. Asumiendo que el caso de uso de búsqueda en el árbol es realizado sobre un árbol pre-cargado, la ejecución de los métodos en la Tabla IV-1 son recolectados por cada caso de uso.

Tabla IV-1. Relaciones entre casos de uso y métodos ejecutados.

	Inserción
m1	BinaryTree.BinaryTree()
m2	BinaryTree.insert(BinaryTreeNode)
m3	BinaryTreeNode.insert(BinaryTreeNode)
m4	BinaryTreeNode.BinaryTreeNode(Comparable)
	Búsqueda
m1	BinaryTree.BinaryTree()
m5	BinaryTree.search(Comparable)
m6	BinaryTreeNode.search(Comparable)

Mediante la aplicación de FCA a las relaciones de la Tabla IV-1, se obtiene el *lattice* de conceptos de la Fig. IV-6. El mismo indica que la inserción y la búsqueda sobre el árbol binario son *crosscutting concerns*. De hecho, los dos conceptos específicos a los casos de uso están etiquetados por métodos que pertenecen a más de una clase, donde cada clase contribuye a más de una funcionalidad.

2.1.4 Herramienta

Existe una herramienta denominada Dynamo [Dynamo], la cual permite obtener la traza de ejecución para programas Java. La misma es una modificación del compilador de Java *javac* version 1.4. Dynamo incluye facilidades para

habilitar y deshabilitar el *tracing* durante la ejecución, de esta manera se pueden obtener las trazas de las partes relevantes del sistema que se quiera analizar. La construcción y visualización del *lattice* de conceptos es realizada mediante la herramienta ToscanaJ [ToscanaJ].

2.2 Aplicación de *Formal Concept Analysis* al Código Fuente

Tourwé y Mens [Tourwé y Mens 2004] desarrollaron una técnica también basada en *Formal Concept Analysis*, pero a diferencia del enfoque anterior, FCA es ahora aplicada directamente sobre el código fuente en busca de lo que los autores denominaron *aspectual views* o vistas aspectuales. Una vista aspectual es un conjunto de entidades del código fuente que están estructuralmente relacionadas de alguna manera. Estas entidades pueden ser cualquier artefacto del código fuente tales como jerarquías de clases, clases, métodos, parámetros de un método o variables de instancia. Por lo tanto, una vista aspectual ofrece una vista sobre el código fuente que por lo general es transversal (*crosscut*) y que complementa las vistas estándar de los ambientes de desarrollo tradicionales.

2.2.1 Enfoque Propuesto

Los autores dividieron el proceso de descubrimiento de vistas aspectuales en las siguientes fases:

1. **Generar objetos (elementos) y atributos (propiedades).** En esta fase, se inspecciona el código fuente y se generan los objetos y los atributos a ser considerados por el algoritmo de FCA. El resultado de esta fase deberá estar en un formato procesable por el algoritmo utilizado.

2. **Generar el *lattice* de conceptos.** En la segunda fase, se aplica el algoritmo de FCA a los objetos y atributos generados por la fase anterior. El resultado de esta fase es el *lattice* de conceptos.

3. **Filtrar los conceptos no importantes.** Si el sistema de software es de tamaño significativo, el número de conceptos generados por el algoritmo de FCA puede ser muy alto. Para remediar esto, los autores definen un conjunto de heurísticas que permiten descartar automáticamente una gran cantidad de conceptos no importantes. Estas heurísticas pueden depender de los objetos y atributos que se estén analizando (específicos del domino), o pueden ser más generales.

4. **Analizar y clasificar los conceptos restantes.** Los conceptos remanentes del filtrado son analizados y clasificados acorde a algún criterio. Por ejemplo, cuando se trate con conceptos únicamente poseídos por métodos, será posible clasificarlos de acuerdo a las clases en que estén definidos. De esta manera, se podrá distinguir entre conceptos que contienen métodos que pertenecen a la misma clase, conceptos que contienen métodos que pertenecen a una misma jerarquía de clases, o conceptos que contienen métodos *crosscutting* definidos en diferentes clases no relacionadas por jerarquías.

5. **Mostrar los conceptos.** Permite que los desarrolladores inspeccionen los conceptos descubiertos.

2.2.2 Ejemplo

Como ejemplo se mostrará cómo fue aplicada esta técnica al sistema de software Soul [Wuyts 2001], un ambiente de programación lógica escrito en SmallTalk.

Los objetos que se darán como entrada al algoritmo de FCA son todas las clases que implementan el sistema, así como también todos los métodos que esas clases definen. Los atributos considerados para tales objetos están basados en la descomposición de sus nombres en subcadenas relevantes:

- Para una clase se considera su nombre y, acorde a las mayúsculas que ocurren en el mismo, se subdivide en subcadenas. Por ejemplo, los atributos asociados a la clase QuotedCodeConstant son las subcadenas 'quoted', 'code' y 'constant'.

- Para un método (en SmallTalk), también se considera como atributos que el mismo contiene ciertas subcadenas, y se generan esos atributos tomando el nombre del método y subdividiéndolo de acuerdo a sus palabras claves y las mayúsculas en las mismas. Por ejemplo, el método unifyWithDelayedVariable: inEnv:myIndex:hisIndex:inSource: posee 5 palabras claves, las cuales son divididas en las siguientes subcadenas: 'unify', 'delayed', 'variable', 'env', 'index' y 'source'. Notar que se desecharon cadenas con poco significado conceptual, tal como 'with' e 'in'. Además del nombre del método, también se tomaron en cuenta (los nombres de) sus parámetros formales.

La motivación detrás de este simple esquema para generar los atributos de los objetos (métodos y clases) es que los autores esperan que las convenciones de programación de SmallTalk puedan usarse para agrupar clases y métodos relacionados en vistas aspectuales.

Una vez filtrados, los conceptos restantes se clasifican automáticamente acorde al siguiente criterio:

conceptos con nombre de clase en palabra clave, contiene clases y métodos, donde el nombre de las clases forman una subcadena de los nombres de los métodos.

conceptos con nombre de clase en parámetro, contiene clases y métodos, donde los nombres de las clases ocurren en el nombre de uno o más parámetros definidos por los métodos.

conceptos sólo clases, contienen solo clases.

conceptos de accesores, contiene solo métodos, definidos en la misma clase que son nombrados luego de una variable de instancia de la clase.

conceptos de métodos en clase única, contiene métodos que están definidos en la misma clase, pero que no son accesores.

conceptos de métodos de jerarquía, contiene solo métodos, definidos en distintas clases, donde todas esas clases comparten una superclase en común diferente a Object (superclase de todo el sistema).

conceptos de métodos crosscutting, contiene solo métodos, definidos en clases distintas, donde la superclase en común de todas esas clases es Object y donde ninguno de los métodos en el concepto está definido en la clase Object.

El *lattice* de conceptos resultante fue calculado usando 1446 objetos diferentes y 516 atributos en total. El algoritmo de FCA creó 1212 conceptos en total, de los cuales 760 permanecieron luego de las operaciones de filtrado. A continuación se detallan los conceptos encontrados y clasificados bajo la categoría “conceptos de métodos de jerarquía”.

- Conceptos que agrupan métodos definidos por la misma jerarquía de clases. Un ejemplo, sería el concepto que contiene todos los métodos de la clase AbstractTerm los cuales son redefinidos por varias subclases.

- Conceptos que agrupan métodos no definidos en la misma jerarquía de clases. Un ejemplo, sería el concepto que contiene el método prettyListPrintOn:scope:, el cual es definido en las clases ListTerm, UnderScoreVariable y EmptyListConstant. La superclase común de las clases anteriores es AbstractTerm, quien no define ese método. Este es un caso de comportamiento que atraviesa (*crosscut*) una jerarquía de clases.

2.2.3 Limitaciones Encontradas

Según los autores, debido a que la técnica solo considera subcadenas de los nombre de las clases y métodos, no debería sorprender que el enfoque sufra de muchos falsos positivos. Un número importante de los conceptos encontrados no son vistas aspectuales significativas, y solamente son generados porque algunas subcadenas son compartidas por muchos objetos. Por una razón similar, los falsos negativos también ocurren. Algunos artefactos del código fuente que sí están relacionados son divididos en conceptos diferentes, por el solo hecho de que no comparten la misma subcadena en su nombre. Este problema se debe a que la técnica se basa en convenciones de programación y *programming idioms*. Si los mismos no son estrictamente seguidos, algunos conceptos importantes podrían perderse.

Ambos problemas podrían aliviarse si se considerara algún tipo de ontología de dominio. Esta ontología debería contener información acerca del dominio en el cual el algoritmo de FCA es aplicado, para filtrar atributos poco importantes y enlazar atributos relacionados de modo que terminen en el mismo concepto.

La limitación más obvia del enfoque es que la búsqueda no se realiza para encontrar aspectos reales, sino para encontrar vistas aspectuales. Aunque estas vistas agrupen entidades relacionadas esparcidas a través del código fuente, y que son por lo tanto *crosscuting*, la mayoría del tiempo no son aspectos en el verdadero sentido de la palabra.

2.2.4 Herramienta

Los autores no hacen mención acerca de la herramienta usada para soportar la técnica.

3 Análisis de Fan-In

Marin, Van Deursen y Moonen [Marin et al. 2007] proponen un enfoque de *aspect mining* basado en el cálculo de la métrica fan-in. El mismo, implica buscar métodos invocados desde diferentes lugares y cuya funcionalidad es necesaria a través de diferentes métodos, potencialmente dispersa sobre muchas clases y paquetes. Este enfoque calcula la métrica fan-in para cada método utilizando el grafo estático de llamadas del sistema. De esta manera, permite aplicar soluciones de aspectos cuando la misma funcionalidad es requerida desde muchos lugares en el código (síntoma de *scattering*).

3.1 Enfoque Propuesto

El análisis por fan-in consiste de tres pasos.

1. Calcular la métrica fan-in para todos los métodos.

2. Filtrar el conjunto de métodos de forma de obtener aquellos con mayores posibilidades de implementar comportamiento *crosscuting*.

3. Analizar los métodos resultantes del paso anterior para determinar cuáles de ellos son parte de la implementación de un *crosscuting concern*.

3.1.1 Calculo del Valor de Fan-In

Fan-in es una métrica que mide la cantidad de veces que un método es invocado desde otros métodos. Por lo tanto, se debe encontrar el conjunto de (potenciales) invocadores de cada método, siendo la cardinalidad de este conjunto el valor de fan-in para un método dado. En realidad, el fan-in de un método dependerá de la manera en que se tomen los métodos polimórficos.

El primer refinamiento, es contar la cantidad de *diferentes cuerpos de métodos* que llaman a otro método. De esta forma, si un único método abstracto es implementado en dos subclases concretas, es posible tratar ambas implementaciones como invocadores diferentes.

El segundo refinamiento tiene que ver con llamadas a métodos polimórficos. Si un método *m* pertenece a un *concern* en particular, es muy probable que las declaraciones de la superclase o subclases que redefinen *m* pertenezcan al mismo *concern*. Por esta razón, si el método *m'* aplica el método *m* a un objeto de tipo C, se agrega *m'* al conjunto de (potenciales) invocadores por cada *m* declarada en cualquier clase o subclase de C. Con esta definición, los métodos abstractos que estén arriba en la jerarquía de herencia actúan como acumuladores de fan-in.

De esta manera, si existen invocaciones a diferentes implementaciones de un método abstracto, se obtiene un alto valor de fan-in para el método de la superclase. Una reimplementación usando AspectJ posibilitaría capturar esas múltiples invocaciones en un *pointcut*, haciendo una invocación al método abstracto dentro del *advice*, y usar el mecanismo de *binding* dinámico para alcanzar la implementación especifica en cada caso.

3.1.2 Filtrado de Métodos

Obtenido el valor de fan-in para todos los métodos, se aplican los siguientes filtros.

Primero, se restringe el conjunto de métodos a aquellos que poseen un valor de fan-in por encima de un cierto mínimo. Este mínimo puede ser un valor absoluto (10 por ejemplo) o un valor relativo (el primer 5% de todos los métodos ordenados por su valor de fan-in, por ejemplo). Este mínimo actuará tanto como filtro de métodos, como indicador de la severidad del nivel de *scattering* en el código.

Segundo, se eliminan *setters* y *getters* de la lista de métodos.

Finalmente, se filtran métodos de utilidad, tales como toString(), clases como XMLDocumentsUtils que contiene “util” en su nombre, métodos para manipular colecciones, y así sucesivamente. Este paso, implica poseer familiaridad con el sistema bajo análisis.

3.1.3 Análisis de Aspectos Candidatos (Seeds)

El paso final, es conducir un análisis manual del conjunto resultante de métodos. El mismo puede realizarse de manera *bottom-up* o de manera *top-down*.

El enfoque *bottom-up*, implica buscar por invocaciones consistentes al método con alto fan-in desde lugares que podrían ser capturados a través de la definición de un *pointcut*. Ejemplo de tales invocaciones incluyen:

- Las llamadas ocurren siempre al principio o al final de un método.

- Las llamadas ocurren en métodos que son refinamientos de un único método abstracto, como, por ejemplo, contratos que abarcan varias jerarquías de clases.

- Las llamadas ocurren en métodos con nombres similares, como *handlers* del mouse o eventos del teclado.

- Todas las llamadas ocurren en métodos implementando un cierto rol, como, por ejemplo, objetos que se registran como observadores del estado de un tercer objeto.

La regularidad de esos sitios de invocación harán posible capturar todas las llamadas mediante un *pointcut*, y el cuerpo del método en un *advice*.

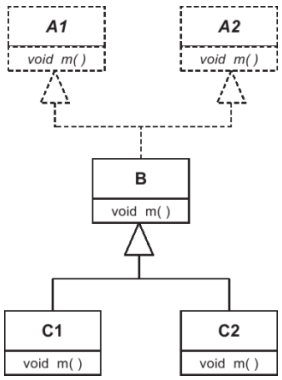


Fig. IV-7. Jerarquía de clases de ejemplo.

En el enfoque *top-down*, se toma en cuenta el conocimiento del dominio o el conocimiento de los típicos *crosscuting concerns*. En este sentido, algunos patrones de diseño definen roles (funcionalidad superimpuesta y *crosscutting*) y métodos específicos a esos roles que pueden aparecer en la lista de candidatos, por ejemplo, los métodos en una clase del patrón de diseño *Decorator* [Gamma et al. 1995] están caracterizados por redireccionar la funcionalidad que estos implementan.

3.2 Ejemplo

A continuación, se muestra un ejemplo de cálculo del valor de fan-in para un método *m*. La Fig. IV-7 muestra la jerarquía de clases y la Tabla IV-2 los valores correspondientes de fan-in.

Este ejemplo, ilustra los efectos de varias llamadas al método polimórfico *m* en diferentes lugares de la jerarquía. Notar que, según la definición de fan-in dada por los autores, el valor de fan-in para el método *m* en la clase C1 no es afectado por llamadas a *m* definidas en C2 y viceversa. Lo mismo ocurre en el caso de las clases hermanas A1 y A2.

Tabla IV-2. Valores de fan-in correspondientes.

	Contribución al fan-in				
Invocador	A1.*m*	A2.*m*	B.*m*	C1.*m*	C2.*m*
f1(A1 a1) {a1.*m()*;}	1	0	1	1	1
f2(A2 a2) {a2.*m()*;}	0	1	1	1	1
f3(B b) {b.*m()*;}	1	1	1	1	1
f4(C1 c1) {c1.*m()*;}	1	1	1	1	0
f5(C2 c2) {c2.*m()*;}	1	1	1	0	1
Fan-in total	4	4	5	4	4

3.3 Herramienta

FINT [FINT] (Fan-In Tool) es un plug-in para Eclipse que provee soporte automático para calcular la métrica, filtrar los

métodos, y realizar el análisis de los resultados.

Para calcular la métrica, la herramienta crea un árbol sintáctico de las fuentes seleccionadas por el desarrollador, y luego crea un grafo de invocaciones con los métodos declarados en la fuente seleccionada y sus invocadores. El cálculo de la métrica se deriva de este árbol, tal como fue explicado anteriormente. Los resultados son mostrados en la vista de análisis de fan-in, en conjunto con la lista de invocadores por cada método.

La misma vista puede ser utilizada para hacer el paso de filtrado. Por lo que, el usuario puede indicar un mínimo para el valor de fan-in. A su vez, el usuario puede filtrar métodos basándose en la signatura de los mismos, como por ejemplo "get*" o "set*".

4 Enfoques Basados en el Procesamiento de Lenguajes Naturales (PLN)

El Procesamiento del Lenguaje Natural -abreviado PLN, o NLP del idioma inglés *Natural Language Processing*- es una subdisciplina de la Inteligencia Artificial y la rama ingenieril de la lingüística computacional [PLN]. PLN estudia el problema de la generación automática y el entendimiento de los lenguajes naturales. Los sistemas de comprensión de lenguajes naturales convierten muestras del lenguaje humano en representaciones más formales y simples de procesar para una computadora.

A continuación, se describen dos técnicas que hacen uso de PLN para la localización de *crosscuting concerns*.

4.1 Aplicación *Lexical Chaining* para Identificar *Crosscutting Concerns*

Shepherd, Tourwé y Pollock [Shepherd et al. 2005] utilizan una técnica de PLN denominada Encadenado Léxico (o *Lexical Chaining*) para identificar grupos de entidades dentro del código fuente que estén "semánticamente" relacionadas. Los autores afirman que en ausencia de soporte para aspectos, los *crosscuting concerns* son implementados utilizando convenciones de nombrado y comentarios descriptivos en el código fuente.

Para poder relacionar entidades del código fuente (como métodos o atributos de clases) y de esta manera mejorar la comprensión del código, los desarrolladores tienden a utilizar convenciones de nombrado y de programación. Por lo tanto, la aplicación de técnicas de PLN permiten explotar este tipo de relaciones semánticas entre palabras.

Según los autores, esta técnica tiene como su mayor contribución el reconocimiento de relaciones de mayor complejidad entre palabras que otros enfoques. Basándose en la semántica subyacente de una palabra y no sólo su forma léxica, la técnica propuesta soluciona el problema de las variaciones en las convenciones de nombrado.

4.1.1 Encadenado Léxico

Encadenado Léxico [Morris y Hirst 1991] es el proceso de agrupar palabras semánticamente relacionadas en un documento (o parte de él) en cadenas. Un "Encadenador" (o *Chainer*) toma como entrada un texto y agrupa cada palabra del mismo en una cadena con palabras relacionadas que también aparecen en el texto. La salida es una lista de cadenas cada cual conteniendo palabras relacionadas entre sí. La Fig. IV-8 muestra un ejemplo de esta técnica sobre un texto de ejemplo.

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. IV-8. Párrafos con cadenas léxicas.

En este ejemplo, podría considerarse que **online auction** y **sold** corresponden a un "*concern*" dentro del texto y que *money*, *medical bill* y *funds* pertenecen a otro "*concern*".

Para poder generar las cadenas léxicas, es necesario calcular la distancia semántica (o fuerza de la relación) entre dos palabras dadas. Por lo general, se utiliza una base de datos que permita calcular esta distancia automáticamente.

4.1.2 Enfoque Propuesto

El código fuente de una aplicación consiste de muchas palabras de diferente tipo y en diferentes contextos. Muchas de estas no son útiles para deducir relaciones semánticas entre piezas del código. Es por este motivo, que sólo se consideran comentarios y campos (o variables de instancia) y nombres de tipos y de métodos, ya que los programadores suelen dejar importantes pistas semánticas entre esos constructores.

Cada uno de estos constructores se procesan de una manera particular:

- **Comentarios**. Se aplica un *speech tagger* para eliminar la ambigüedad de las palabras que componen el comentario. Por ejemplo, las palabras "dirección" y "destino" podrían relacionarse si ambas son usadas como sustantivos, pero no se deberían relacionar si las mismas actúan como verbos.

- **Nombres de métodos**. Se separa el nombre del método acorde a las mayúsculas que ocurren en el mismo. Por ejemplo, el método goAndDoThatThing se separa en cinco palabras: go, And, Do, That, Thing. Luego se aplica un *speech tagger* (el cual permite desambigüar las palabras) sobre la frase.

- **Campos y nombres de clases**. Se asume que los campos y los nombres de las clases son sustantivos. Luego, se separan los nombres de la misma manera que se hizo con los métodos.

El algoritmo de encadenado léxico primero obtiene todas las palabras de cada archivo del código fuente del programa y luego construye cadenas procesando cada palabra. Por cada palabra, encuentra la cadena más cercana semánticamente y agrega la misma a la cadena. De no existir una cadena suficientemente cercana a la palabra, entonces comienza una nueva cadena con esa palabra. Luego de procesar todas las palabras, el resultado es una lista de cadenas de palabras de diferentes longitudes.

Una vez encontradas las cadenas léxicas, las mismas son examinadas en busca de cadenas con un alto nivel de *scattering,* por ejemplo, las palabras miembro corresponden a diferentes archivos fuente. Los autores sospechan que esas cadenas podrían corresponderse con *concerns* de alto nivel diseminados por el código.

Este algoritmo es computacionalmente costoso, debido a que un programa típico consiste de muchas palabras, y el tiempo necesario crece en proporción al número de palabras únicas dadas como entrada.

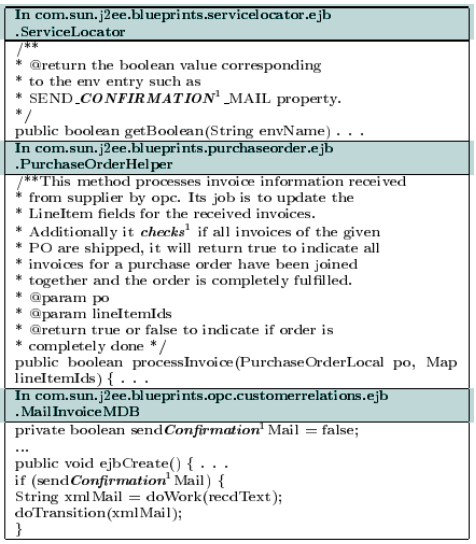


Fig. IV-9. Cadena léxica encontrada en el código fuente de PetSore.

4.1.3 Ejemplo

La ejecución de esta técnica sobre el código fuente de la aplicación PetStore [PetStore] generó aproximadamente 700 cadenas. PetStore posee 57,000 palabras para encadenar, y le tomó a la herramienta aproximadamente 7 horas para finalizar.

Una vez aplicada la técnica sobre la aplicación, se encontró la cadena (entre muchas otras) {SEND_CONFIRMATION_EMAIL, checks, confirmation, insure, check} (Fig. IV-9). Analizando las entidades del código fuente asociadas a la cadena, se descubrió que esta cadena corresponde a la característica "notificación a los

clientes". Esta funcionalidad corresponde al envío de *e-mails* a los clientes que realizan compras en el almacén.

4.1.4 Herramienta

Se implementó el algoritmo como un plug-in para Eclipse. La herramienta automatiza todas las partes del mismo, incluyendo la selección de palabras y el proceso de encadenado léxico. También posee un visualizador para una mejor navegación y exploración de las cadenas léxicas encontradas.

4.2 Aplicación de PLN para Localizar *Concerns* Orientados a Acciones

Shepherd et. al. [Shepherd et al. 2007], argumentan que muchos de los *crosscuting concern* son el resultado de la tensión existente entre acciones y objetos. Por ejemplo, en una aplicación para reproducir música, el código concerniente al reproductor de música podría residir en una clase abstracta Player (la cual incluiría funcionalidad básica para reproducir, pausar, detener, silenciar y buscar en un reproductor). Esta descomposición orientada a objetos hace que entender y mantener la clase Player sea sencillo pero puede provocar que conceptos orientados a acciones, tales como "reproducir tema", se comiencen a dispersar a través del código. En tal caso, los conceptos orientados a acciones están dispersos (*scattered*) ya que deben satisfacer la descomposición orientada a objetos. Luego, la programación orientada a aspectos permitiría modularizar estos *concerns* orientados a acciones implementando las acciones como aspectos.

4.2.1 Enfoque Propuesto

La localización de un *concern* en el código es realizada mediante un proceso de tres pasos: formulación inicial de la consulta, expansión de la consulta, y búsqueda e inspección del grafo resultante.

1. **Formulación de una consulta**. El usuario formula una consulta desde su concepto inicial. Esta consulta consiste en un verbo y un objeto directo (este último es opcional).

2. **Expansión de la consulta**. El usuario podrá aumentar su consulta explorando las recomendaciones dadas por el sistema. El sistema proporciona palabras relacionadas a las existentes en la consulta, utilizando su conocimiento del lenguaje natural así como también de cómo las palabras son utilizadas en el programa.

3. **Búsqueda sobre el grafo AOIG e inspección del grafo resultante**. El usuario podrá realizar búsquedas sobre el grafo AOIG (*action-oriented identifier graph* [Shepherd et al. 2006]) utilizando la consulta desarrollada. Luego, podrá observar los resultados en el grafo de resultados. El grafo de resultados consiste en nodos que representan métodos y arcos que representan relaciones estructurales.

El enfoque que proponen los autores es una técnica basada en búsquedas con consultas realizadas sobre un modelo del programa que captura las relaciones orientadas a acciones entre los identificadores del mismo. Para esto, se realiza un análisis PLN sobre el código fuente controlando la información sobre ocurrencias de verbos y sus objetos directos en el programa, representados mediante un grafo de identificadores orientados a acciones (AOIG). Este grafo representa las acciones en el código del programa en conjunto con sus objetos directos. Donde el objeto directo de un verbo es, por ejemplo, en la frase "eliminar atributo" la palabra "atributo".

Step	Set Updates
1	Abstract Initial-Query = "automatically finish the word"
2	Concrete Initial-Query = finish word
3	Verb-Query = finish Direct-Object-Query = word
4	Verb-Query = finish, finished Direct-Object-Query = the word, all words, complete word, first word
5 6	Verb-Recommendations = complete, end, stop, close, get, ...user chooses "complete" Verb-Query = finish, finished, complete, completed
5 6	Direct-Object-Recommendations = completions, text, line, paragraph, ...user chooses "completions" Direct-Object-Query = the word, all words, ... first word, completions
5 6	Verb-Recommendations = end, stop, close, get, ...user decides not to add any words

Fig. IV-10. Evolución de la consulta del ejemplo.

4.2.2 Ejemplo

En este ejemplo, se considera el problema de localizar el *concern* “finalizar la palabra automáticamente” (*automatically finish the word*) presente en un programa de edición de textos. Esta característica permite a los usuarios presionar una combinación de teclas y finalizar una palabra parcialmente completa.

En la Fig. IV-10, se observa la evolución de la consulta y las recomendaciones dadas por el sistema. El primer paso consiste en el ingreso de la consulta inicial por parte del usuario, en este caso, '*automatically finish the word*'. En el segundo paso, el usuario debe reformular su consulta para que contenga un verbo y un objeto directo, por lo que ingresa 'finish word'. A partir de allí, la herramienta realiza diferentes recomendaciones que permiten al usuario ir refinando su consulta inicial. Luego, la consulta inicial consiste ahora en una consulta de verbos {finish, finished} y una consulta de objetos directos {the word, all words, complete words, first word} (Fig. IV-10). Estas dos consultas serán expandidas mediante sugerencias realizadas por la herramienta hasta que el usuario se encuentre satisfecho con la consulta obtenida.

Una vez finalizada la expansión de la consulta, se dispara la búsqueda sobre el grafo AOIG utilizando la consulta resultante. Los resultados de la búsqueda se visualizan en el grafo de resultados (Fig. IV-11).

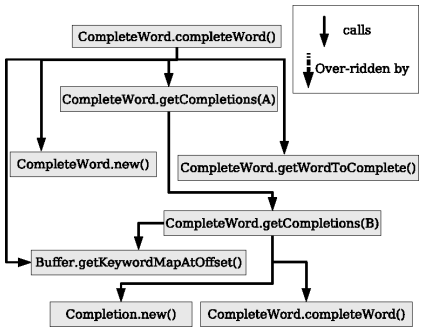


Fig. IV-11. Grafo de resultados para la consulta “complete word”.

4.2.3 Herramienta

FindConcept es la implementación de la técnica presentada como un plugin de Eclipse. En la misma fueron utilizados diversos componentes de PLN y de análisis estructural de programas. De PLN se utilizó el algoritmo de Porter para realizar *stemming* (reconoce diferentes formas de una misma palabra), Maxent de OpenNLP [OpenNLP] y una versión optimizada de WordNet para encontrar sinónimos. También se utilizaron las JDT de Eclipse para parsear el código fuente. Para visualizar el grafo de resultado se modificó Grappa [Grappa] para soportar el acceso al código desde el grafo visualizado.

5 Identificación de Métodos Únicos

Gybels y Kellens [Gybels y Kellens 2005] observaron que muchos *crosscuting concerns* son implementados como un único método que es invocado desde muchos lugares en el código. Por ejemplo, el *concern logging* puede ser encapsulado lo suficiente (en una única clase centralizada) como para evitar *tangling*, aunque no *scattering*. La Fig. IV-12, muestra cómo sería la implementación de este *concern* y el impacto que tiene sobre las demás clases en el sistema.

5.1 Enfoque Propuesto

Debido a que muchos *crosscuting concerns* son implementados como única entidad central, particularmente un método único, los autores proponen descubrir aspectos candidatos mediante la identificación de este tipo de métodos. Un método único se define como: “Un método sin valor de retorno el cual implementa un mensaje que no es implementado por ningún otro método”.

Luego de calcular todos los métodos únicos del sistema, estos se ordenan acorde al número de veces que son invocados, y se eliminan aquellos considerados irrelevantes (como *getters* y *setters*).

A continuación, el usuario podrá inspeccionar manualmente la lista de métodos resultantes de modo de encontrar aspectos candidatos.

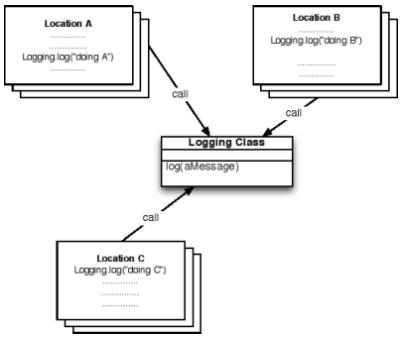


Fig. IV-12. Implementación del concern logging. La clase Logging Class define un único método encargado de ofrecer la funcionalidad de logging, el cual es invocado desde numerosos lugares en el código.

5.2 Ejemplo

Los autores reportan un experimento sobre una imagen de SmallTalk. En ese experimento, se seleccionaron todos los métodos únicos en una imagen estándar de VisualWorks SmallTalk conteniendo 3400 clases con un total de 66000 métodos. Fueron encontrados 6248 métodos únicos. De la lista resultante, se tomaron en cuenta aquellos métodos únicos que fueron invocados como mínimo cinco veces. El resultado es una lista de 228 métodos la cual puede ser manualmente analizada. La Tabla IV-3 muestra algunos de los métodos resultantes.

Tabla IV-3. Métodos únicos encontrados en el experimento desarrollado por los autores.

Clase	Método único (Signatura)
CodeComponent	#startLoad
Locale	#currentPolicy:
Menu	#addItem:value:
ScheduledWindow	#updateEvent:
UIFinderVW2	#showClasses:
ComposedText	#centered
UIBuilder	#wrapWith:
Text	#emphasizeAllWith:
Cursor	#show
Image	#pixelsDo:

Observando las signaturas de los métodos encontrados, la inclusión de las palabras “update” y “event” hacen que #updateEvent: se considere como un probable aspecto candidato. Por otro lado, métodos tales como #addItem:value de la clase Menu parecen menos probables que implementen un aspecto, ya que claramente se refiere a añadir items a un menú. Finalmente, se identificaron 16 aspectos candidatos. La Tabla IV-4 lista algunos de los aspectos identificados.

Tabla IV-4. Clases, signaturas y número de invocaciones de los aspectos identificados.

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

PundleModel	#updateAfterDo:	10

A pesar de haber reconocido algunos aspectos, la lista de métodos únicos no incluyó el método #changed: el cual implementa el aspecto de notificación de cambio de estado. Esto se debió a que existen otros dos métodos que redefinen el método central #changed: de la clase Object, en ambos casos estas implementaciones son modificaciones para hacer que los objetos se observen a sí mismos.

5.3 Herramienta

Los autores no hacen mención acerca de la herramienta usada para soportar la técnica.

6 Enfoques Basados en *Clustering*

Clustering, o más precisamente *cluster analysis*, es el estudio formal de algoritmos y métodos para agrupar, o clasificar objetos [Jain y Dubes 1988]. Un objeto es descripto, ya sea por un conjunto de medidas, o por las relaciones entre este y otros objetos. El objetivo de una tarea de *clustering* es encontrar una organización válida y conveniente de los datos en clusters o grupos. Cada *cluster* está compuesto por un conjunto de objetos "similares", de modo tal que cada *cluster* maximiza la "similitud" entre los objetos del mismo y minimiza la "similitud" entre objetos de diferentes *clusters*. El nivel de "similitud" entre dos objetos lo determina una función o métrica de distancia.

6.1 Aplicación de *Clustering* Jerárquico sobre el Nombre de los Métodos

Shepherd y Pollock [Shepherd y Pollock 2005] propusieron la aplicación de *clustering* sobre los métodos de un sistema con el objetivo de agrupar todo el código correspondiente a un *crosscuting concern*. De esta manera, es posible observar todo el código relacionado con un *crosscuting concern* en particular. Para esto, los autores desarrollaron una función de distancia basada en PLN que será utilizada para agrupar métodos con nombres relacionados. Por lo tanto, cada *cluster* contendrá métodos que poseen una pequeña distancia entre sí y esos *clusters* por lo general representarán un *crosscuting concern*.

6.1.1 Enfoque Propuesto

Primero se aplica *clustering* jerárquico acumulativo [Jain et al. 1999] (del inglés ALC ó *Agglomerative Hierachical Clustering*) para agrupar métodos relacionados, y colocar cada método en su *cluster*. Luego, se repiten los siguientes pasos:

1. Comparar todos los pares de *clusters* en base a una función de distancia, y marcar el par si posee la menor distancia hasta el momento.

2. Si la distancia del par marcado es menor que una cota mínima, combinar los dos *clusters*. Sino, detener el algoritmo.

Los pasos 1 y 2 son repetidos hasta que no existan *clusters* que estén lo suficientemente cercanos respecto del valor mínimo establecido. El algoritmo retornará todos los *clusters* cuya membresía sea mayor a 1.

Los *clusters* son almacenados como árboles, cuando dos *clusters* son combinados, se crea una nodo padre y ambos son colocados como hijos del mismo. Almacenar y combinar los *clusters* de esta manera genera un árbol donde los nodos hoja están más cercanos a sus vecinos que los nodos cercanos a la raíz.

La función de distancia utilizada, para dos métodos m y n, es 1 / largoDeLaSubcadenaEnComún(m.nombre, n.nombre). Para encontrar la distancia entre un *cluster* y un método, o un *cluster* y otro *cluster*, la etiqueta de la raíz del *cluster* es utilizada en vez del nombre del método. Una importante característica de esta función, es que la misma es reemplazable, en el sentido de que la técnica no depende exclusivamente de la misma y que diferentes funciones pueden ser utilizadas.

6.1.2 Ejemplo

Como caso de estudio se aplicó la técnica sobre el código fuente de JHotDraw [JHotDraw] 5.4b2, y se discutieron en particular tres casos representativos de *concerns* identificados. Cada uno de estos casos cae en una de las siguientes categorías.

1. *Crosscuting concerns* representados como una interfase con métodos implementados de manera

consistente.

2. *Crosscuting concerns* representados como una interfase con métodos que poseen duplicación de código y lógica dispersada entre ellos, y que están, aparentemente, implementados de manera inconsistente.

3. *Crosscuting concerns* representados por métodos con nombres similares a través de diferentes clases, sin interfases explícitas, un gran porcentaje de código duplicado, pero parecen ser consistentes.

El primero de los casos es el menos común de los tres. Usualmente, el tipo de método de interfase que cae en esta categoría es extremadamente específico de la clase. Por ejemplo, si existieran diferentes clases que realizan una interfaz Color, cada implementación del método changeColor sería específica de la clase.

Para la segunda de las categorías, el código que la técnica agrupó a partir de la subcadena en común "figureChanged". Donde "figureChanged" es el nodo padre y "figureChanged(PertFigure)", "figureChanged(TextFigure)", "figureChanged(HTMLTextAreaFigure)" y "figureChanged(GraphLayout)" son los métodos o nodos hojas.

La tercera de las categorías involucra métodos que el desarrollador debería haber implementado como una interfase o un aspecto.

6.1.3 Herramienta

AMAV (*Aspect Miner and Viewer*) integra los procedimientos de búsqueda y visualización. La herramienta permite llevar a cabo los procesos de *clustering* y presenta al usuario los resultados obtenidos relacionando los *clusters* encontrados con el código asociado a los mismos. AMAV consiste de tres paneles: el panel *crosscuting* (que muestra todos los métodos relacionados a un *cluster*), el panel de *clusters* (que muestra todos los *clusters* encontrados) y el panel de edición (permite ver el código de la clase a la cual pertenece un método en particular).

6.2 Aplicación y Comparación de Tres Técnicas de C*lustering*

Moldovan y Serban [Moldovan y Serban 2006a; Moldovan y Serban 2006b] presentan y comparan tres algoritmos de *clustering* en el contexto de *aspect mining*: *k-means* (KM) [Jain et al. 1999], *fuzzy c-means* (FCM) [Jain et al. 1999] e *hierachical agglomerative clustering* (HACA) [Jain et al. 1999].

6.2.1 Clustering en el contexto de aspect mining

Sea $M = \{ m_1, m_2, ..., m_n \}$ el sistema de software, donde m_i con $1 \leq i \leq n$ es un método del sistema y $n(|M|)$ es el número de métodos en el sistema.

Los autores consideran un *crosscuting concern* al conjunto de métodos $C = \{ c_1, c_2, ..., c_{cn} \}$ que implementan ese *concern*. El número de métodos en un *crosscuting concern* C es $c_n = |C|$. Sea $CCC = \{ C_1, C_2, ..., C_q \}$ el conjunto de todos los *crosscuting concerns* que existen en el sistema M, entonces, el número de *crosscuting concerns* en un sistema M es $q = |CCC|$.

El conjunto $K = \{ K_1, K_2, ..., K_p \}$ es denominado una partición del sistema M si sólo si $1 \leq p \leq n$, $K_i \subseteq M$, $K_i \neq \emptyset$, $\forall_i \in \{ 1, 2, ..., p \}$, $M = \coprod_{i=1}^{p} Ki$ y $K_i \cap K_j = \emptyset$, para todo i, j $\in \{ 1, 2, ..., p \}$, $i \neq j$. Donde K_i es el i-ésimo *cluster* de K siendo K el conjunto de *clusters*.

De hecho, según los autores, el problema de *aspect mining* puede ser visto como el problema de encontrar una partición K del sistema M.

6.2.2 Adaptación de los Algoritmos de Clustering para Aspect Mining

En este enfoque, los objetos a ser agrupados son los métodos del sistema. Los mismos pertenecen a clases de la aplicación o son llamados desde las clases del sistema. Los autores, consideran a cada método como un vector l-dimensional: $m_i = \{ m_{i1}, ..., m_{il} \}$. Se consideran dos modelos de espacios vectoriales:

- El vector asociado con el método m es $\{FIV, CC\}$, donde FIV es el valor de fan-in y CC es el número de clases invocantes. Este modelo se denota como M_1.

- El vector asociado con el método m es $\{FIV, B_1, B_2, ..., B_{l-1}\}$, donde FIV es el valor de fan-in y B_i es el valor del atributo correspondiente a la clase C_i. El valor de B_i es 1 si el método m es llamado desde un método perteneciente a C_i, y es 0 de lo contrario. Este modelo se denota como M_2.

1. ***Hard K-Means Clustering (KM)***: Este algoritmo particiona la colección de n métodos del sistema M en k

clusters distintos y no vacíos. El proceso de partición es iterativo y finaliza cuando una partición minimiza la suma de errores al cuadrado. El algoritmo comienza con *k* centroides iniciales, luego iterativamente recalcula los *clusters* y sus centroides hasta que converge. Cada objeto es asignado al *cluster* o centroide más cercano. Este algoritmo presenta dos claras desventajas:

a) La *performance* del algoritmo depende de los centroides iniciales.
b) Es necesario especificar el número de *clusters* por adelantado.

Los autores proponen la siguiente heurística para elegir el número de *clusters* y sus centroides iniciales:

(i) El número de *k clusters* es *n*.
(ii) El método elegido como centroide es el más lejano (el que maximize la suma de distancias a otros métodos).
(iii) El próximo centroide es elegido como el método que posee la mayor distancia del centroide más cercano ya elegido, y su distancia es positiva. De no existir tal método, el número *k* de *clusters* de disminuido.
(iv) El paso (iii) será repetido hasta que *k* centroides sean alcanzados.

2. ***Fuzzy C-Means Clustering***: FCM utiliza partición *fuzzy*, la cual permite que un método pertenezca a diferentes *clusters* con diferentes grados de membresía entre 0 y 1. Este algoritmo actualiza de manera iterativa los centros de los *clusters* y los grados de membresía para cada método. De esta manera FCM mueve el centro hacia la ubicación "correcta" dentro del conjunto de datos. FCM no asegura una convergencia hacia la solución óptima, debido a que los centroides iniciales se eligen de manera aleatoria. También es necesario especificar el número de *clusters* por adelantado.

3. ***Hierachical Agglomerative Clustering (HAC)***: Los métodos de clustering acumulativos (*bottom-up*) comienzan con *n singletons* (conjuntos de 1 elemento), combinándolos hasta que un único *cluster* es obtenido. En cada paso, los dos *clusters* más similares son elegidos para combinarse. Los autores proponen una versión modificada del algoritmo, el cual detendrá su ejecución una vez que se han alcanzado *k clusters*. Donde *k* es obtenido utilizando KM.

6.2.3 Enfoque Propuesto

Los pasos para identificar *crosscuting concerns* son los siguientes:

- **Filtrado**. Eliminar métodos pertenecientes a clases tipo ArrayList o Vector entre otras.

- **Generación**. Obtener el conjunto de métodos del código fuente seleccionado, y calcular el conjunto de valores de los atributos por cada método del conjunto.

- **Agrupación**. El conjunto de métodos restante es agrupado en *clusters* usando un algoritmo de *clustering* (KM, FCM o HAC). Los *clusters* se ordenan de manera descendente por la distancia promedio al punto O_l, donde O_l es el vector dimensional *l* con cada componente 0.

- **Análisis**. Los *clusters* obtenidos son analizados para determinar qué *clusters* contienen métodos pertenecientes a *crosscuting concerns*. Se analizan los *clusters* cuya distancia desde el punto O_l es mayor que una cierta cota.

```
public class A {
   private L l;
   public A(){l=new L();}
   public void methA(){ l.meth();}
   public void methB(){ l.meth();}
}
public class L {
   public L(){}
   public void meth(){}
}
public class B {
   public B(){}
   public void methC(L l){ l.meth();}
   public void methD(A a){a.methA();}
}
```

Fig. IV-13. Código del ejemplo.

6.2.4 Ejemplo

A continuación se presenta un ejemplo de la aplicación de técnicas de *clustering* según el modelo M_1. Para el código de ejemplo de la Fig. IV-13, la aplicación del modelo M_1 resulta en un conjunto de métodos y los valores asociados a los atributos del mismo (Tabla IV-5).

Tabla IV-5. Valores de los atributos para un modelo M_1.

Método	FIV	CC
A.A	0	0
A.methA	1	1
A.methB	0	0
B.B	0	0
B.methC	0	0
B.methD	0	0
L.L	1	1
L.meth	3	2

A partir de los vectores obtenidos y mediante la aplicación de los algoritmos de *clustering* se obtienen los *clusters* de métodos mostrados en la Tabla IV- 6. De los *clusters* resultantes, el primero corresponde a un *crosscutting concern*, ya que el método meth de la clase L es invocado desde diferentes métodos del ejemplo.

Tabla IV-6. *Clusters* obtenidos.

Cluster	**Métodos**
C1	{L.meth}
C2	{A.methA, L.L}
C3	{A.A, A.methB, B.B, B.methC, B-methD}

6.2.5 Herramienta

Los autores no hacen mención sobre la implementación de los algoritmos propuestos.

6.3 *Clustering* sobre las Trazas de Ejecución

En [He y Bai 2006], He y Bai, proponen la aplicación de *clustering* y de reglas de asociación sobre las trazas de ejecución de un sistema para identificar *crosscuting concerns* (*clusters* de escenarios con comportamiento similar) y para determinar la ubicación de los *join points* (relación entre el código base y el código aspectual) mediante asociaciones entre invocaciones de métodos.

6.3.1 Enfoque Propuesto

Este enfoque está compuesto por dos pasos bien definidos. Primero, se identifican los *crosscuting concerns* y luego se determinan los *join points* entre el código base y el código identificado como *crosscuting*. Aunque previo a ambos pasos, es necesario obtener la traza de ejecución del sistema bajo análisis.

1. **Identificación de *crosscuting concerns***. De existir un grupo de código que posee comportamiento similar, y que aparece frecuentemente en las trazas de ejecución, entonces es posible que implemente un *crosscuting concern*. Utilizando técnicas de *clustering* [Jain y Dubes 1988], se considera a cada traza como un objeto y los métodos ejecutados en la misma como los atributos del mismo. El número de veces que se invoca cada método representa el valor del atributo.

2. **Ubicación de los *join points***. Para esto, los autores proponen la utilización de reglas de asociación [Agrawal y Srikant 1994]. Sea M = (m1, m2, m3) una secuencia de invocaciones de métodos y m3 fue

reconocido como código *crosscuting* entonces la regla m2->m3 determina el punto en el cual el código *crosscuting* se relaciona con el código base. Cada traza obtenida mediante un escenario se considera como una transacción cuyo TID (TID: *Transaction Identifier* ó Identificador de Transacción) es el nombre del escenario. Cada método será un *ítem* por lo que los *itemsets* se compondrán de un conjunto de métodos. Las métricas soporte y confianza de cada regla aseguran que el aspecto candidato sea frecuente y que el *join point* sea estable. Sólo se generarán *itemsets* de tamaño 2, debido a que los autores pretenden obtener reglas del tipo "Si A fue invocado entonces B también". En base a este tipo de reglas es posible derivar los *poincuts* necesarios para implementar el código *crosscuting* como un aspecto.

6.3.2 Ejemplo

Como ejemplo se presenta una aplicación bancaria, en particular se analizan los módulos de gestión de cuentas y de balances. Los escenarios correspondientes a la gestión de cuentas son: crear cuentas nuevas (KH), depósitos (CK), y extracciones de dinero (QK). Para el módulo de balances, los escenarios considerados son: balance independiente (DLRJ) y balance central (ZJRJ). Luego, pueden existir variantes para cada escenario, por ejemplo, una cuenta nueva puede ser una cuenta corriente (KH1) o una cuenta fija (KH2).

Cada escenario (KH1, KH2, ...) es un objeto a ser agrupado en un *cluster*. En la Fig. IV-14 se observan todos los escenarios y sus atributos (o métodos invocados). La aplicación de técnicas de *clustering* da como resultado la existencia de dos *clusters*: { KH1, KH2, QK1, QK2, ... } y { DLRJ1, DLRJ2, ..., JZRJ1, JRZJ2, ... }. El primero de los *clusters* tiene como conjunto de atributos en común a los métodos { Trans.Accept, Encry.TsMac, Ans,Process, Trans.Send }, lo que implica la existencia de *crosscuting concerns* correspondientes a comunicaciones, encriptación y mensajería. Analizando el segundo *cluster* se encuentra evidencia de la existencia de *crosscuting concerns* correspondiente al acceso a la base de datos y manejo de *locks* (o cerrojos).

Para obtener las reglas de asociación, se obtienen las transacciones en base a los escenarios y los ítems de cada una serán los métodos de cada escenario. De esta manera, se descubren reglas como "Lock.UnlockTable" y "DB.Close" => "end()" la cual indica que los métodos "Lock.UnlockTable" y "DB.Close" son llamados al final de muchas funciones principales como DLRJ1 y DLRJ2.

Excutions	Executed Methods
KH_1	Trans.Accept(1), Encry.TsMac(1), Ans.Process(2), KH.hz_kh(1), .., Error.Lock(1), Trans.Send(1)
KH_2	Trans.Accept(1), Encry.TsMac(1), Ans.Process(2), KH.zl_kh(1), .., Trans.Send(1)
CK_1	Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.hz_ck(2), .., Trans.Send(1)
CK_2	Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.zl_ck (2), .., Trans.Send(1)
QK_1	Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.hz_qk(2), .., Trans.Send(1)
QK_2	Trans.Accept(1), Encry.TsMac(1), Ans.Process(3), KH.zl_qk(2), .., Trans.Send(1)
$DLRJ_1$	DB.Open(1), Lock.Table(5), RJ.rjemnllidr(1), Lock.UnlockTable(5), .., DB.Close(1)
$DLRJ_2$	DB.Open(1), Lock.Table(5), RJ.rjmemlcmsj(1), Lock.UnlockTable(5), .., DB.Close(1)
$JZRJ_1$	DB.Open(1), Lock.Table(8), RJ.rj1zzjzzo(1), Lock.UnlockTable(8), .., DB.Close(1)
$JZRJ_2$	DB.Open(1), Lock.Table(8), RJ.rj1hzlsz(1), Lock.UnlockTable(8), .., DB.Close(1)
	. . .

Fig. IV-14. Nombre del escenario y los métodos ejecutados en el mismo.

6.3.3 Herramienta

Los autores no mencionan el software utilizado.

7 Enfoques Basados en Detección de Código Duplicado

A continuación, se presentarán los enfoques basados en la detección de código duplicado. La clonación de código o el acto de copiar fragmentos de código y realizar pequeñas modificaciones no funcionales, es un conocido problema para la evolución de los sistemas de software que provoca la existencia de código duplicado o clonado [Rysselberghe y Demeyer 2003]. Por supuesto, el normal funcionamiento del sistema no es afectado, pero de no tomarse medidas para revertir el problema el futuro desarrollo del sistema podría ser prohibitivamente caro.

La detección de código clonado es un proceso de dos fases: una fase de transformación y otra de comparación [Rysselberghe y Demeyer 2003] . Durante la primera de las fases, se transforma el código fuente de la aplicación hacia un formato intermedio que permita el uso de un algoritmo de comparación más eficiente. Luego, durante la fase de comparación es cuando los clones de código son detectados.

7.1 Detección de Aspectos Candidatos Mediante Técnicas de Código Duplicado Basadas en PDGs

Shepherd, Gibson y Pollock [Shepherd et al. 2004], proponen explotar el grafo de dependencias (GDP ó PDG del inglés *Program Dependence Graph*) y el árbol sintáctico de un programa con el objetivo de automatizar la búsqueda de aspectos candidatos. En la representación GDP de un método, cada nodo representa una sentencia, y un arco entre dos nodos representa una dependencia de control o de datos entre las sentencias correspondientes.

La técnica propuesta se basa principalmente en dos algoritmos existentes que utilizan el grafo de dependencias del programa para detectar código duplicado [Krinke y Softwaresysteme 2001] y [Komondoor y Horwitz 2001]. La salida de la técnica es un conjunto de candidatos a refactorizar mediante *before advices* en AspectJ.

7.1.1 Enfoque Propuesto

Este enfoque se desarrolla en cuatro fases.

1. **Construir** GDPs para todos los métodos.
2. **Identificar** un conjunto de candidatos a refactorizar (dependencia de control).
3. **Filtrar** candidatos a refactorizar no deseables (dependencias de datos).
4. **Combinar** conjuntos relacionados de candidatos en clases.

La construcción de los GDP debe realizarse a nivel de código fuente y no al nivel de representación intermedia (RI). Por tal motivo, los autores construyen los GDP a nivel de código fuente basándose en la construcción previa de un GDP a nivel RI.

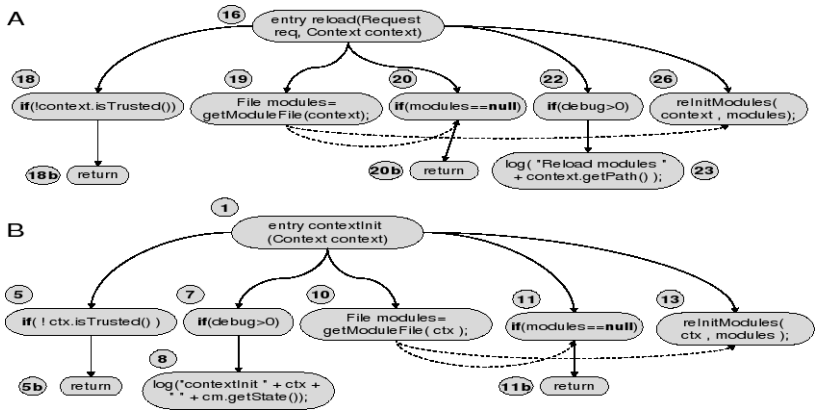


Fig. IV-15. Ejemplo de dos GDPs para dos métodos con código duplicado identificados en Tomcat.

Durante la fase de identificación del conjunto de candidatos iniciales, se comparan las representaciones GDP de cada método mediante la construcción de árboles sintácticos para cada sentencia, de esta manera se mejora la precisión de la comparación. La fase de filtrado implica filtrar los candidatos obtenidos en el paso anterior de acuerdo a un conjunto de filtros basados en la información sobre la dependencia de datos subyacente en cada clon. De los pasos anteriores, se obtiene un conjunto de pares candidatos, donde cada miembro del par pertenece a un método diferente.

Dado que es posible que existan pares similares, durante la fase de combinación, se identifican estos candidatos y se los unifica en conjuntos de pares similares.

7.1.2 Ejemplo

En la Fig. IV-15 se observan las representaciones GDPs de dos métodos con código duplicado que la técnica identificó en el código fuente del servidor de aplicaciones Tomcat [Tomcat]. Existen varias características de ambos métodos que hacen que sea dificultosa su identificación de manera léxica. Los nombres de las variables en la línea 5 y la correspondiente línea 18 son diferentes (Fig. IV-15). Esto demuestra que utilizar nombres de variables como semillas (o *seeds*) para una búsqueda léxica es indeseable. Las líneas 10 y 11 son semánticamente idénticas a las líneas 19 y 20, pero se encuentran en diferentes contextos dentro de cada método. Léxicamente, estos métodos parecen bastante diferentes, pero acorde a los GDPs los mismos son equivalentes. Como candidatos para refactorizar hacia aspectos, se observa en la línea 5 y 18 un buen ejemplo de política. La misma podría ser expresada como: "cuando un contexto no es fiable, no realizar un contextInit o una recarga con el mismo". Este tipo de políticas son buenos candidatos para su refactorización mediante POA.

7.1.3 Herramienta

Los autores implementaron la técnica como un plugin para Eclipse denominado Ophir. Ophir actualmente soporta la técnica antes descripta, y provee de una API que permite añadir nuevos tipos de análisis mediante el mecanismo de plug-ins de Eclipse [Eclipse].

7.2 Sobre la Aplicación de Técnicas de Código Duplicado para *Aspect Mining*

Bruntink et. al. [Bruntink et al. 2005], utilizan y evalúan la aplicabilidad y precisión de tres técnicas de código duplicado para detectar *crosscuting concerns*. A su vez, en [Bruntink 2004] se propone la aplicación de métricas que indiquen cuán relevante son los clones detectados respecto al proceso de *aspect mining*.

Según los autores, la incapacidad para modularizar apropiadamente un *crosscuting concern*, ya sea por falta de soporte en el lenguaje (no permite usar aspectos o capturar excepciones) o por un diseño incorrecto del sistema, provoca que los programadores se vean forzados a reutilizar código correspondiente a *crosscuting concerns* de una manera *ad-hoc* (por ejemplo, mediante *copy/paste*). A través del tiempo, esta práctica puede convertirse en parte del proceso de desarrollo donde fragmentos de código comunes encuentran su lugar en manuales como convenciones de programación o *idioms*.

Por lo tanto, los autores, en base a las observaciones anteriores plantean que la utilización de técnicas de código duplicado permitirían identificar algunas clases de *crosscuting concerns* ya que las mismas detectan automáticamente el código duplicado en el código fuente del sistema.

7.2.1 Enfoque Propuesto

El enfoque propuesto es básicamente aplicar las técnicas de detección de código duplicado para luego filtrar los resultados y obtener indicadores de *scattering*. Este enfoque esta compuesto por los siguientes pasos.

1. **Configurar las herramientas de detección de código duplicado**. Esta configuración afecta los tipos de clones detectados. Por ejemplo, se podría configurar el tamaño mínimo de los clones.

2. **Abstraer los clones detectados en clases de clones**. Estas clases de clones son grupos de fragmentos de código los cuales son todos clones entre sí.

3. **Aplicación de métricas sobre las clases de clones**. Existen dos tipos de métricas (donde *C* es una clase de clon):
   (a) Métricas que capturan la severidad del código duplicado y que son relevantes desde el punto de vista del mantenimiento del código.
      i. *Número de Clones (NC)*: Número de clones incluidos en *C*.
      ii. *Número de Líneas (NL)*: El número de líneas (diferentes) del código en *C*.
      iii. *Tamaño Promedio del Clon (ACS)*: El número de líneas (NL) dividido por el número de clones (NC).
   (b) Métricas que capturan la noción de *scattering* para una clase de clon.
      i. *Número de Archivos (NFI)*: La cantidad de archivos diferentes donde aparecen clones de tipo *C*.
      ii. *Número de Funciones (NFU)*: La cantidad de funciones distintas donde aparecen clones de tipo *C*.

4. **Grading**. Usando las métricas anteriores es posible rotular cada clase de clon con un "grado". Este grado debería indicar la mejora que se obtendría si la clase de clon es refactorizada utilizando aspectos. El grado de una clase de clon *C* se calcula como:

$$Grade(C) = NL(C) * NFU(C)$$

Consecuentemente, las clases de clones que sean grandes (NL) y estén más esparcidas (NFU) tendrán grados mayores.

7.2.2 Ejemplo

Los autores reportan la aplicación de la técnica sobre un componente de software denominado CC el cual consiste de 16,406 líneas de código C. Este componente es responsable de convertir datos entre diferentes estructuras de datos. Dentro de este componente los autores han identificado cinco *crosscuting concerns*: manejo de errores de memoria, verificación de valores *null*, verificación de rangos, manejo de errores y *tracing*.

Una vez aplicadas las técnicas de detección de código duplicado, los autores evaluaron la precisión de las técnicas de detección de código duplicado para identificar *crosscuting concerns* [Bruntink et al. 2005]. Para esto midieron la precisión y el *recall* de las clases de clones obtenidos respecto a los *crosscuting concerns* existentes.

Por otra parte, en [Bruntink 2004] los autores rotularon las clases de clones con su grado correspondiente y observaron que un pequeño número de clases de clones poseen los grados más altos, mientras que el resto se mantiene cerca del promedio. Los autores afirman que desde el punto de vista del mantenimiento esta es una propiedad deseable, ya que grandes mejoras pueden ser obtenidas mediante la aplicación de aspectos para un pequeño grupo de clases de clones.

7.2.3 Herramientas

La detección de código duplicado fue realizada utilizando las siguientes herramientas:

- "ccdiml" la cual es una implementación de una variante del enfoque de Baxter [Baxter et al. 1998], la cual cae en la categoría de técnicas basadas en árboles sintácticos.
- CCFinder [Kamiya et al. 2002] es una técnica basada en representaciones tokenizadas del código fuente.
- Por último se utilizó una técnica basada en grafos de dependencia de programas (ver sección 10.2).

Los autores no mencionan implementaciones que den soporte al resto del proceso.

8 *Aspect Mining* Mediante *Random Walks*

Zhang y Jacobsen [Zhang y Jacobsen 2007] afirman que una de las precondiciones necesarias para las herramientas de *aspect mining* es su habilidad para caracterizar adecuadamente los rastros sintácticos pertenecientes a *crosscuting concerns*. Semejante caracterización es dificultosa, sino completamente imposible de obtener. Es por esto que los autores proponen una solución que modela el proceso manual de descubrimiento de *crosscuting concerns*. La misma define un Modelo de Markov y fue derivado del algoritmo *page-rank* [Page et al. 1998] para reflejar la naturaleza de la investigación del código fuente de un programa. Un modelo de markov es un proceso estocástico y discreto donde el próximo estado depende solamente del estado actual y no de los estados anteriores. Por otra parte, *random walk* se refiere a la formalización de la idea de ir tomando sucesivos pasos cada uno en una dirección aleatoria. Por lo tanto, podría afirmarce que los autores consideran la búsqueda manual de *crosscuting concerns* en el código fuente de una aplicación como un *random walk*.

8.1 Enfoque Propuesto

El enfoque podría subdividirse en los siguientes pasos:

1. **Construcción del grafo de acoplamiento**. Donde los nodos del grafo representan diferentes elementos del programa como componentes, paquetes, clases, métodos, o colecciones de estos elementos. El mismo, registra todas las relaciones aferentes (de entrada) y todas las relaciones eferentes (de salida) de cada uno de estos elementos.

2. **Recorrido del grafo**. El grafo se recorre en dos direcciones: desde las raíces a las hojas (dirección eferente) y

desde las hojas a las raíces (dirección aferente). Durante estos recorridos se actualizan dos medidas:

a) *Popularidad*: La popularidad de cada elemento se calcula como la probabilidad acumulada de ser visitado a través de la relación saliente. Por lo tanto, un elemento puede ser popular porque es visitado muy frecuentemente por otros elementos, o porque es visitado por elementos populares.

b) *Significancia*: Medida de la probabilidad del elemento siendo visitado siguiendo la relación eferente. Por lo tanto, si un elemento referencia a un gran número de elementos distintos se dice que es un elemento significante. Inclusive, si un elemento no referencia a un gran número de elementos, el mismo puede ser significante si los elementos que referencia son significantes.

3. **Generación de *crosscuting concerns* candidatos**. En base a las medidas tomadas en el paso anterior, se generan candidatos de dos maneras:

a) *Crosscuting homogéneos*: son *crosscuting concerns* utilizados de manera uniforme en el código (como es el caso del *concern logging*). La detección de este tipo de *crosscuting concern* se realiza verificando si la popularidad de un elemento es mayor a su significancia acorde a una cota mínima para la diferencia.

b) *Crosscuting heterogéneos*: son *crosscuting concerns* utilizados de forma no uniforme en el código, consistiendo de diferentes piezas de código esparcidas por todo el código fuente. La detección de este tipo de *concern* se realiza definiendo un conjunto de elementos fundamentales (*core elements*) los cuales poseen mayores niveles de significancia que de popularidad. Luego, se listan los elementos conocidos por los del conjunto como candidatos de *crosscuting heterogéneos*.

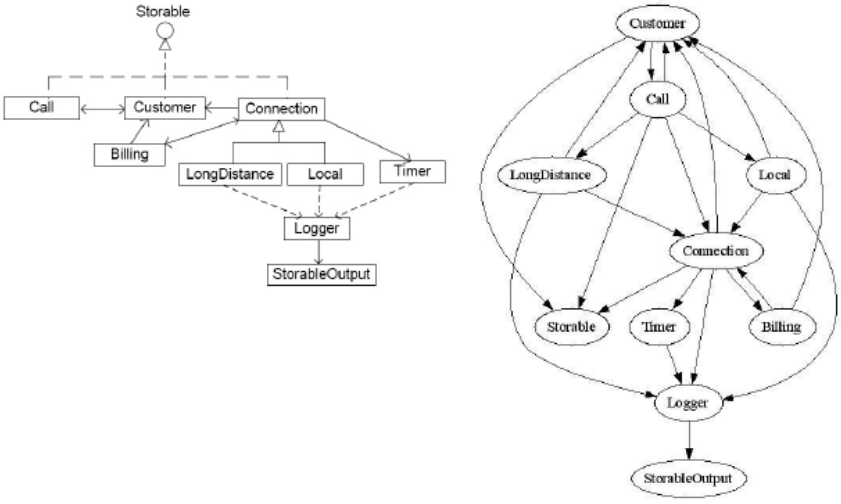


Fig. IV-16 Diagrama UML (*izq.*) y grafo de acoplamiento (*der.*).

8.2 Ejemplo

A continuación, se describe la aplicación de la técnica sobre una versión adaptada de la aplicación de ejemplo "telecom" cuyo código fuente se distribuye en conjunto con el compilador de AspectJ. En este ejemplo, "Billing" y "Timing" son dos *crosscuting concerns* originalmente implementados como aspectos. En la versión adaptada, los autores, implementaron estos *crosscuting concerns* directamente en Java por lo que se convirtieron en el objetivo de la búsqueda. En la Fig. IV-16 se observa un diagrama UML del ejemplo y el grafo de acoplamiento obtenido.

Como aspectos candidatos se encuentran los tipos Billing, Timer, Storable, Logger y StorableOutput. Si bien Customer y Connection poseen un alto fan-in, ambas clases implementan funcionalidad del sistema por lo que no pueden ser clasificadas como *concerns* candidatos.

En la Tabla IV-7, se presentan los valores de las probabilidades y los *rankings* respectivos para los tipos que conforman el grafo de acoplamiento. Se observa que el algoritmo puso a StorableOutput como el tipo de mayor *crosscuting* aunque sea conocido solamente por Logger. Los tipos principales: Call, Connection, y Customer poseen altos *rankings* de *crosscuting*. Sin embargo, también poseen altos valores de *ranking* aferente por lo que no se consideran como *crosscuting concerns* candidatos.

Tabla IV-7. Valores de las probabilidades y *rankings* para los tipos del grafo de acoplamiento.

Tipo	Probabilidad	*Ranking*	Probabilidad	*Ranking*
	Aferentes		*Eferentes*	
Local	0,05227	1	0,0943	6
LongDistance	0,05227	2	0,0943	7
Billing	0,0550	3	0,0920	5
Timer	0,0550	4	0,0073	3
Call	0,09690	5	0,2727	10
Connection	0,1115	6	0,1537	8
Customer	0,1327	7	0,2657	9
Storable	0,1365	8	0,0050	2
Logger	0,1404	9	0,0098	4
StorableOutput	0,1673	10	0,0050	1

8.3 Herramienta

Los autores implementaron su enfoque en una herramienta denominada *Prism Aspect Miner* (PAM) la cual está basada en PQL [Prism] (*Prism Query Language*) un lenguaje declarativo de consultas que permite expresar complejos criterios de búsqueda de patrones sobre código Java/AspectJ.

9 Clasificación y Comparación

Para poder realizar la clasificación de las técnicas antes presentadas, se tomarán como base los criterios utilizados en [Kellens et al. 2007]. Según Kellens et al. [Kellens et al. 2007], estos criterios forman un *framework* comparativo inicial que aún puede evolucionar, acorde nuevos desarrollos surjan en el campo de *aspect mining*.

A continuación, se describen los criterios de comparación:

- **Datos estáticos versus datos dinámicos** *¿Qué tipo de datos analiza la técnica?* ¿Analiza datos de entrada que pueden ser obtenidos mediante un análisis estático del código, o información dinámica la cual es obtenida mediante ejecuciones del programa, o ambos?

- **Análisis basado en tokens versus análisis estructural/comportamiento** *¿Qué tipo de análisis realiza la técnica?* Se distinguen dos tipos:
  - **Basado en tokens.** Simple análisis léxico del programa: secuencias de caracteres, expresiones regulares, etc.
  - **Estructural/comportamiento.** El análisis estructural o de comportamiento implica analizar árboles de parsing, información de tipos, envío de mensajes, etc.

- **Granularidad** *¿Cuál es el nivel de granularidad de los aspectos candidatos (seeds) encontrados?* Algunas técnicas descubren aspectos candidatos al nivel de métodos, otras trabajan a un nivel más fino considerando sentencias o fragmentos de código.

- **Tangling y Scattering** *¿Qué síntomas de mala modularización detecta la técnica?* ¿Busca por síntomas de scattering, tangling o ambos?

- **Participación del usuario** *¿Qué tipo de participación es necesaria por parte del usuario?* ¿Cuanto esfuerzo requiere la técnica por parte de quien la usa? ¿El usuario debe recorrer manualmente los resultados para poder encontrar aspectos candidatos viables? ¿Debe el usuario ingresar datos adicionales durante el proceso de búsqueda?

- **Mayor evaluación empírica** *¿Sobre qué tamaño de sistema se aplicó la técnica?* Ya que pueden existir problemas al aplicar la técnica sobre grandes sistemas de software, como por ejemplo excesiva complejidad computacional o participación del usuario, caída en la precisión de la técnica, etc. La evaluación sobre grandes sistemas puede ser usado como indicador de escalabilidad.

- **Validación empírica** *¿Hasta que punto las técnicas existentes han sido validadas sobre casos de la vida real?* ¿Para las validaciones realizadas, se han reportado cuántos de los aspectos conocidos fueron encontrados y cuántos de los candidatos reportados eran falsos positivos?

- **Precondiciones** *¿Qué condiciones (explícitas o implícitas) deben ser satisfechas por los concerns en el programa bajo investigación para que una técnica de aspect mining pueda encontrar aspectos candidatos convenientes?*

Tabla IV-8. Lista de las técnicas a comparar.

Nombre abreviado	Descripción	Sección
Relaciones de ejecución	Analiza patrones recurrentes presentes en las trazas de ejecución	4.1
Análisis dinámico	Aplica FCA sobre las trazas de ejecución	4.2.1
Análisis de identificadores	Aplica FCA sobre identificadores	4.2.2
Análisis Fan-in	Análisis mediante métrica fan-in	4.3
PLN (Cadenas léxicas)	Genera cadenas léxicas desde el código fuente	4.4.1
PLN (*Concern* de acciones)	Búsqueda de *concerns* orientados a acciones	4.4.2
Métodos únicos	Detecta métodos únicos del sistema	4.5
Clustering (Nom. métodos)	*Clustering* basado en el nombre de los métodos	4.6.1
Clustering (Comparación)	Comparación de tres algoritmos de *clustering*	4.6.2
Clustering (Trazas)	*Clustering* sobre las trazas de ejecución	4.6.3
Clone detection (PDG)	Aplica detección de clones de código basada en PDGs	4.7.1
Clone detection (Tokens)	Aplica detección de clones de código basada en tokens	4.7.2
Clone detection (AST)	Aplica detección de clones de código basada en AST	4.7.2
Random walks	Aplica algoritmo de *random walks*	4.8

Luego, a partir de los criterios anteriores, se comparan las técnicas introducidas anteriormente. Para ahorrar espacio, se abreviaron los nombres de las técnicas usadas tal como se muestra en la Tabla IV-8.

Para cada una de las técnicas comparadas, la Tabla IV-9 muestra el tipo de datos (estático o dinámico) analizado por cada técnica, así como también el tipo de análisis realizado (token o estructural/comportamiento).

Tabla IV-9. Tipo de datos analizado y tipo de análisis realizado.

	Tipo de dato de entrada		Tipo de análisis	
	estático	dinámico	token	estructural/comportamiento
Relaciones de ejecución	-	x	-	x
Análisis dinámico	-	x	-	x
Análisis de identificadores	x	-	x	-
Análisis Fan-in	x	-	-	x
PLN (Cadenas léxicas)	x	-	x	-
PLN (*Concern* de acciones)	x	-	-	x
Métodos únicos	x	-	-	x
Clustering (Nom. métodos)	x	-	x	-

Clustering (Comparación)	x	-	-	x
Clustering (Trazas)	-	x	-	x
Clone detection (PDG)	x	-	-	x
Clone detection (Tokens)	x	-	x	-
Clone detection (AST)	x	-	-	x
Random walks	x	-	-	x

Las técnicas de *aspect mining* pueden subdividirse en dos grandes categorías, de análisis estático y de análisis dinámico. Las técnicas de análisis estático analizan las frecuencias de los elementos de un programa y explotan la homogeneidad sintáctica de los *crosscutting concerns* [Marin et al. 2007], como puede observarse en la Tabla IV-9, la mayoría de las técnicas son de este tipo. En cambio, las técnicas de análisis dinámico razonan sobre las trazas de ejecución y por lo tanto requieren la ejecución del código bajo análisis [Kellens et al. 2007].

En cuanto al tipo de razonamiento, hay cuatro enfoques que realizan un análisis basado en tokens. 'Análisis de identificadores' y '*Clustering* (Nom. métodos)' razonan solamente acerca de los nombres de los métodos del sistema, mientras que el enfoque 'PLN (Cadenas léxicas)' analiza cada palabra individual que aparece en el código fuente del programa. Las cuatro técnicas anteriores se basan en el hecho de que los *crosscutting concerns* son generalmente implementados mediante la utilización de rigurosas convenciones de nombrado. El resto de las técnicas basan su razonamiento en heurísticas que consideran las características estructurales o de comportamiento del programa analizado (Tabla IV-9). Por ejemplo, la técnica 'Relaciones de ejecución' analiza el comportamiento del sistema ya que intenta encontrar patrones recurrentes de métodos invocados durante la ejecución del mismo. Por otra parte, el análisis realizado por la técnica 'Analisis Fan-in' es estructural ya que toma en cuenta el grafo estático de invocaciones para poder calcular el valor de la métrica para los métodos.

Tabla IV-10. Granularidad y síntomas.

	Granularidad		Síntomas	
	método	fragmento de código	scattering	tangling
Relaciones de ejecución	x	-	x	-
Análisis dinámico	x	-	x	x
Análisis de identificadores	x	-	x	-
Análisis Fan-in	x	-	x	-
PLN (Cadenas léxicas)	x	-	x	-
PLN (*Concern* de acciones)	x	-	x	-
Métodos únicos	x	-	x	-
Clustering (Nom. métodos)	x	-	x	-
Clustering (Comparación)	x	-	x	-
Clustering (Trazas)	x	-	x	-
Clone detection (PDG)	-	x	x	-
Clone detection (Tokens)	-	x	x	-
Clone detection (AST)	-	x	x	-
Random walks	x	-	x	-

A excepción de las técnicas basadas en búsqueda de código duplicado, el resto trabajan con una granularidad a nivel método (Tabla IV-10). Esto quiere decir que la salida de las técnicas son métodos que han sido identificados como parte de algún *crosscutting concern*. La ventaja de las técnicas '*Clone detection*' es que pueden detectar código aspectual al nivel de fragmentos de código, proveyendo al desarrollador un *feedback* más preciso sobre el código que debe ser refactorizado.

Todas las técnicas buscan por síntomas de código *scattering* como indicador de la presencia de un *crosscutting concern* (Tabla IV-10). Sólo 'Análisis dinámico' toma en cuenta ambos síntomas (*scattering* y *tangling*).

Tabla IV-11. Información sobre las aplicaciones usadas para evaluar las técnicas.

Técnica	Mayor caso de estudio	Tamaño del caso	Validado empíricamente
Relaciones de ejecución	Graffiti	3,100 métodos / 82KLOC	-
Análisis dinámico	JHotDraw	2,800 métodos / 18KLOC	-
Análisis de identificadores	Soul	2,800 métodos / 18KLOC	-
Análisis Fan-in	JHotDraw	2,800 métodos / 18KLOC	-
PLN (Cadenas léxicas)	PetStore	10 KLOC	-
PLN (*Concern* de acciones)	iReport	74 KLOC	-
Métodos únicos	Imágen Smalltalk	3,400 clases / 66,000 mét.	-
Clustering (Nom. métodos)	JHotDraw	2,800 métodos / 18KLOC	-
Clustering (Comparación)	JHotDraw	2,800 métodos / 18KLOC	-
Clustering (Trazas)	Ejemplo de un banco	12 métodos	-
Clone detection (PDG)	Tomcat	38 KLOC	-
Clone detection (Tokens)	ASML (código C)	20 KLOC	x
Clone detection (AST)	ASML (código C)	20 KLOC	x
Random walks	Websphere Application Server	2,000 KLOC	-

Mientras que el tamaño del mayor sistema analizado es significativo para la mayoría de las técnicas descriptas (a excepción de '*Clustering* (Trazas)'), la validación empírica de los resultados ha sido por lo general menospreciada (Tabla IV-11). Esta validación sólo fue realizada para las técnicas '*Clone detection* (Tokens)' y '*Clone detection* (AST)' debido a que las mismas fueron aplicadas sobre un sistema (ASML) del cual se conocían sus *crosscutting concerns* previamente. Esto permitió analizar cuáles *seeds* candidatas correspondían a verdaderas *seeds*, cuáles eran falsos positivos y qué falsos negativos tuvo la técnica. Cabe destacar que en cinco de catorce trabajos el enfoque propuesto fue evaluado sobre la aplicación *open-source* JHotDraw, haciendo del mismo un *benchmark* común para la evaluación de técnicas de *aspect mining* [Ceccatto et al. 2005].

Tabla IV-12. Precondiciones necesarias sobre los crosscutting concerns para su detección.

Técnica	Precondiciones sobre los *crosscutting concerns*
Relaciones de ejecución	El orden de las llamadas es siempre el mismo
Análisis dinámico	Existe al menos un caso de uso que expone al *crosscutting concern* y uno que no
Análisis de identificadores	Los nombres de los métodos que implementan el *concern* son parecidos
Análisis Fan-in	El *concern* es implementado en un método separado el cual es invocado un gran número de veces, o muchos métodos que implementan el *concern* llaman al mismo método
PLN (Cadenas léxicas)	El contexto del *concern* contiene palabras que son sinónimos del *crosscutting concern*
PLN (*Concern* de acciones)	Existen métodos que contienen verbos los cuales representan acciones
Métodos únicos	El *concern* es implementado por exactamente un método
Clustering (Nom. métodos)	Los nombres de los métodos que implementan el *concern* son parecidos
Clustering (Comparación)	El *concern* es implementado mediante llamadas a los mismos métodos desde diferentes módulos
Clustering (Trazas)	El *concern* es implementado mediante llamadas a los mismos métodos desde diferentes módulos
Clone detection (PDG)	El *concern* es implementado por fragmentos de código similares
Clone detection (Tokens)	El *concern* es implementado por fragmentos de código similares
Clone detection (AST)	El *concern* es implementado por fragmentos de código similares

Random walks	Los elementos que implementan el *concern* son usados desde muchas partes del sistema

Un importante criterio que ayuda a seleccionar una técnica apropiada para buscar aspectos candidatos en un sistema dado, es qué condiciones explícitas o implícitas el enfoque realiza acerca de cómo los *crosscutting concerns* están implementados (Tabla IV-12). Por ejemplo, las técnicas 'Análisis de identificadores', '*Clustering* (Nom. métodos)', 'PLN (Cadenas léxicas)' y '*Clone detection* (Tokens)' se basan en que los desarrolladores utilizan rigurosas convenciones de nombrado cuando implementan *crosscutting concerns*. Por otra parte, 'Relaciones de ejecución' y '*Clustering* (Comparación)' asumen que aquellos métodos que son invocados en conjunto desde diferentes contextos son aspectos candidatos. La técnica 'Análisis Fan-in', asume que los *crosscutting concerns* son implementados por métodos que son llamados muchas veces, o por métodos que invocan a tales métodos.

Tabla IV-13. Tipo de participación del usuario.

Técnica	Participación del usuario
Relaciones de ejecución	Obtención de la traza de ejecución del programa e inspección de los patrones resultantes
Análisis dinámico	Selección de los casos de uso, obtención de las trazas e inspección del *lattice* resultante
Análisis de identificadores	Navegación de los aspectos candidatos obtenidos
Análisis Fan-in	Selección de los aspectos candidatos desde la lista métodos ordenados por valor de fan-in.
PLN (Cadenas léxicas)	Interpretación manual de las cadenas léxicas obtenidas
PLN (*Concern* de acciones)	Generación de la consulta a realizar sobre el grafo, y análisis de los resultados obtenidos
Métodos únicos	Inspección de los métodos únicos encontrados
Clustering (Nom. métodos)	Navegación y análisis manual de los *clusters* obtenidos
Clustering (Comparación)	Navegación y análisis manual de los *clusters* obtenidos
Clustering (Trazas)	Navegación y análisis manual de los *clusters* obtenidos
Clone detection (PDG)	Navegación e interpretación manual de los clones descubiertos
Clone detection (Tokens)	Navegación e interpretación manual de los clones descubiertos
Clone detection (AST)	Navegación e interpretación manual de los clones descubiertos
Random walks	Interpretación de la lista de métodos obtenida a partir de los valores aferentes y eferentes.

La Tabla IV-13 resume el tipo de participación necesaria por parte del usuario para cada una de las técnicas. Ninguna trabaja de manera automática, todas las técnicas requieren que el usuario recorra y analice los resultados obtenidos. Las técnicas dinámicas, además, requieren que el usuario obtenga una representación correcta como entrada a la misma (las trazas).

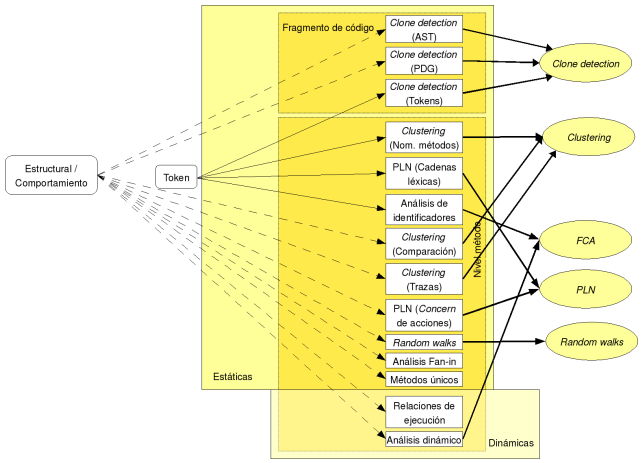


Fig. IV-17. Clasificación resultante de las técnicas de *aspect mining*.

En base a los criterios descriptos anteriormente y a las tablas obtenidas, se desarrolló la taxonomía mostrada en la Fig. IV-17. Cada una de las técnicas consideradas está representada como un rectángulo blanco, mientras que los cuatro rectángulos más grandes distinguen las técnicas 'Estáticas' de las 'Dinámicas' y a las técnicas que trabajan a 'Nivel método' de las que trabajan a nivel 'Fragmento de código'. Los dos rectángulos redondeados de la izquierda separan a las técnicas acorde al razonamiento utilizado: 'Estructural/Comportamiento' o basado en 'Token'. Por otra parte, las elipses a la derecha muestran que algoritmos están detrás de cada enfoque.

10 Conclusiones

Podría decirse que, la investigación en *aspect mining* puede ser clasificada en dos categorías: técnicas de análisis estático y técnicas de análisis dinámico.

Las técnicas de análisis estático analizan las frecuencias de los elementos de un programa y explotan la homogeneidad sintáctica de los *crosscutting concerns*. Ejemplos de estas técnicas, son las basadas en métricas de software (como análisis de fan-in), en la aplicación de alguna técnica de análisis de datos (como FCA, *clustering*, PLN o *random walks*) sobre el código fuente, en heurísticas (identificación de métodos únicos) o en la búsqueda de código duplicado.

Las técnicas de análisis dinámico tienen como precondición la obtención de una traza del sistema, donde esta traza representa el comportamiento de la aplicación en tiempo de ejecución. Las técnicas de análisis dinámico se diferencian en dos puntos fundamentales: el tipo de información que registran en la traza, y la técnica o algoritmo utilizado para analizar la misma. Algunas técnicas, necesitan una única traza para poder identificar aspectos candidatos, en cambio, otras técnicas deben derivar un conjunto de trazas correspondientes a posibles escenarios de uso del sistema analizado. Diferentes enfoques propuestos han utilizado desde FCA o *clustering* hasta heurísticas (identificación de patrones recurrentes) para identificar aspectos candidatos en las trazas.

Todos los enfoques propuestos discutidos en este capítulo han sido validados empíricamente por sus respectivos autores. De todos los sistemas utilizados para realizar esta validación, la mayoría ha optado por la utilización del *framework* de dibujado JHotdraw. En consecuencia, existe un conjunto estable y consensuado de *crosscutting concerns* presentes en JHotdraw, que lo convierten en un *benchmark* de referencia para la validación de futuras técnicas de *aspect mining*.

Capítulo V

Aspect Mining Mediante Análisis Dinámico y Reglas de Asociación

La presencia de crosscutting concerns en los sistemas orientados a objetos actuales dificulta la correcta comprensión, modularización y evolución del software. Aspect mining es la actividad que identifica estos crosscutting concerns en sistemas de software legados. En este capítulo, se presenta un proceso semiautomático de cinco pasos para aspect mining basado en análisis dinámico y reglas de asociación. Este proceso comienza con la obtención de las trazas de ejecución de un sistema y su posterior análisis mediante algoritmos de reglas de asociación. Las reglas obtenidas permitirán al desarrollador identificar potenciales crosscutting concerns en el código legado. Como ejemplo de su aplicación se analizó una implementación del patrón Observer, descubriendo la totalidad de los crosscutting concerns previamente identificados por otros autores. Como resultado del enfoque propuesto se presenta YAAM (Yet Another Aspect Miner), una herramienta que permite obtener y analizar aspectos candidatos a partir de información dinámica previamente suministrada. Por último, se comparan las características del enfoque propuesto con dos técnicas de aspect mining basadas en análisis dinámico.

1 Análisis Dinámico

En el contexto de la ingeniería de software, el análisis dinámico es la investigación de las propiedades de un sistema de software utilizando información reunida a medida que el sistema se ejecuta [Ball 1999]. Opuesto al análisis dinámico se encuentra el concepto de análisis estático, el cual recoge su información desde artefactos tales como código fuente, documentos de diseño, archivos de configuración, etc. de forma de investigar las propiedades del sistema [Zaidman 2006]. Mientras que el análisis estático examina el texto de un programa para derivar propiedades válidas para todas las ejecuciones, el análisis dinámico deriva propiedades que son válidas para una o más ejecuciones mediante el análisis del programa en ejecución.

Permitir el análisis dinámico en un contexto de reingeniería de software requiere la generación de un conjunto de trazas de ejecución del sistema de software bajo estudio; siendo cada traza de ejecución la estructura en la cual la información acumulada es almacenada. Para obtener esta traza de ejecución, es necesario ejecutar el sistema de software acorde a un escenario de ejecución bien definido. Donde, un escenario de ejecución es una instancia de uno o más casos de uso [Booch et al. 1999].

1.1 Características del Análisis Dinámico

Existen dos factores principales por los cuales el análisis dinámico se diferencia del análisis estático [Zaidman 2006]. Primero, este tipo de análisis permite un enfoque orientado al objetivo, lo que significa que se analizarán aquellas partes de la aplicación consideradas interesantes. Segundo, el análisis dinámico es mucho más sucinto para tratar con el polimorfismo, el cual se hace presente en todos los sistemas modernos orientados a objetos. A continuación, se desarrollan brevemente ambos factores.

1.1.1 Estrategia Orientada al Objetivo

El análisis dinámico posibilita seguir una estrategia orientada al objetivo, este tipo de estrategia facilita el acercamiento del desarrollador a sistemas de software legado. La misma se basa en el conocimiento de las capacidades funcionales del sistema y su utilización para generar trazas específicas a determinados escenarios que se desean analizar.

Por ejemplo, un desarrollador debe modificar el mecanismo utilizado por un procesador de texto para cambiar de formato un texto que ha sido seleccionado. Aplicar análisis dinámico, implica ejercitar el sistema mediante los escenarios relacionados a la selección del texto y los cambios de las propiedades del mismo (poner negrita, subrayar, etc.). El resultado es la obtención de una traza que registra las clases participantes y la interacción entre las mismas para satisfacer el escenario ejercitado. Si se utiliza una técnica de análisis estático, el desarrollador necesitaría conocer gran parte de la aplicación antes de saber exactamente qué partes están relacionadas a la funcionalidad que debe modificar.

1.1.2 Polimorfismo

El mecanismo de polimorfismo permite escribir programas de manera más eficiente, y también favorece la evolución del software al permitir el desarrollo de sistemas más flexibles. Sin embargo, para los propósitos de la comprensión de programas, el polimorfismo puede generar complicaciones ya que es más difícil entender el comportamiento preciso de una aplicación sin observar el sistema de software en ejecución. Esto se debe a que una llamada polimórfica es un punto de variación que puede dar lugar a un gran número de comportamientos diferentes.

En contraste, cuando se observa al sistema de software con ayuda del análisis dinámico, la vista obtenida del software es precisa respecto al escenario ejecutado.

1.2 Tecnologías de Extracción de Trazas

A continuación se describen las tecnologías más utilizadas para extraer las trazas de ejecución de un sistema de software [Zaidman 2006].

- **Basadas en *profilers* o *debuggers***. Un *profiler* es típicamente usado para investigar la *performance* o los requerimientos de memoria de una aplicación. Un *debugger*, por otra parte, es frecuentemente usado para recorrer paso a paso la ejecución de un sistema de software con la intención de encontrar las razones de un comportamiento no anticipado.
  Por lo general, los *profilers* y *debuggers* poseen una infraestructura basada en el envío de eventos durante ciertas etapas de la ejecución [Zaidman 2006]. Por lo que, es posible escribir un *plugin* que capture estos eventos y los registre en una traza de ejecución. Estos eventos incluyen la activación de métodos/procedimientos, acceso a variables o campos, entre otros.

- **Basadas en POA**. Mediante POA es posible desarrollar un conjunto de aspectos que inserten el código de *tracing* al principio y/o final de cada método/procedimiento. De esta manera, el aspecto de *tracing* genera una entrada en la traza cada vez que se invoca o se retorna de un método.

- **Basadas en modificaciones sobre el AST**. Durante el proceso de *parsing* del código fuente de un programa, es posible realizar alteraciones sobre árbol sintáctico abstracto (del inglés *Abstract Syntax Tree* – AST) antes de la fase de generación de código [Akers 2005]. Estas modificaciones corresponderían a la incorporación del mecanismo de *tracing*.

- **Basadas en *wrappers* de métodos**. Los *wrappers* de métodos posibilitan interceptar y aumentar el comportamiento de los métodos existentes de forma de incorporar nuevo comportamiento antes o después del método en sí [Brant et al. 1998; Greevy y Ducasse 2005]. Este comportamiento podría ser el correspondiente al mecanismo de *tracing*.

- **Extracción Ad-hoc**. Los mecanismos previos prescriben una manera estructurada de realizar la extracción de la traza. Sin embargo, algunas veces, cuando sólo existe un pequeño conjunto de puntos de interés dentro del sistema de software, la instrumentación manual o basada en *scripts* puede ser una solución (a corto plazo).

1.3 Problemas en el Análisis Dinámico

Durante la realización de análisis dinámico, se desea generar una traza de ejecución de calidad a partir del escenario de ejecución. Que sea de calidad, significa que la traza obtenida es un reflejo real de lo sucedido durante la ejecución del software. Sin embargo, existe un número de situaciones que son problemáticas cuando se realiza análisis dinámico. A continuación, se describen estas situaciones [Zaidman 2006].

- **El efecto observador**. En muchas disciplinas de las ciencias exactas, el efecto observador se refiere a cambios que el efecto de observar produce sobre el fenómeno que se está observando. Un ejemplo clásico de esta situación se da en la disciplina de la física cuántica, donde la observación de un electrón provocará un cambio en la trayectoria de este debido a la luz o radiación que introduce el mecanismo de observación. En el campo de la ingeniería de software, un efecto similar ha sido reportado denominándose el efecto de la prueba (*probe effect*) [Andrews 1998]. Este efecto puede manifestarse de las siguientes maneras en un contexto de análisis dinámico:
  - Debido a que el software bajo análisis posee un tiempo de respuesta menor durante su ejecución, el usuario podría modificar su comportamiento y, por ejemplo, presionar sobre un botón múltiples veces. Por lo tanto, el escenario que se ejecuta difiere del escenario predefinido.

- Otro efecto, más serio aún, es la influencia que podría tener el mecanismo de *tracing* sobre la interacción de los hilos de ejecución que ocurren dentro del programa que está siendo analizado.

Por lo tanto, se debe generar el mínimo de *overhead* posible durante la extracción de la traza de un sistema en ejecución de forma de minimizar las consecuencias de este efecto.

- **Presencia de múltiples hilos de ejecución**. Cuando existen múltiples hilos de ejecución dentro de un sistema de software, se debe tener la precaución de generar una traza por cada hilo. Ya que almacenar todas las acciones de todos los hilos en una única traza podría generar una imagen confusa acerca de lo ocurrido durante la ejecución de la aplicación.

- **Utilización de mecanismos reflexivos**. La presencia de mecanismos de reflexión [Maes y Nardi 1988] provocará que la traza resultante contenga entradas correspondientes a los mismos cada vez que se realicen llamadas a métodos pertenecientes a las clases cargadas dinámicamente. Esto puede provocar una situación donde las llamadas de los métodos no son correctamente registradas, apareciendo una invocación al mecanismo de reflexión y no al método que realmente se invocó.

2 Reglas de Asociación

La búsqueda de reglas de asociación [Agrawal y Srikant 1994], es una de las técnicas más importantes e investigadas dentro del campo de *data mining* [Han y Kamber 2000]. Este tipo de técnica tiene como objetivo extraer interesantes asociaciones, correlaciones, patrones frecuentes o estructuras casuales entre un conjunto de elementos en un dominio dado [Kotsiantis y Kanellopoulos 2006].

2.1 Definición

Sea $I = \{i_1, i_2, ..., i_m\}$ un conjunto de m elementos distintos, T una transacción que contiene un conjunto de elementos tal que $T \subseteq I$ y D una base de datos constituidas por diferentes transacciones *Ts*. Una regla de asociación es una implicación de la forma $A \Rightarrow B$, donde $A, B \subset I$ son conjuntos de elementos denominados *itemsets*, y $A \cap B = \emptyset$ [Agrawal y Srikant 1994]. A su vez, A se denomina el antecedente de la regla y B se denomina su consecuente, siendo el significado de la misma "A implica a B".

Existen dos medidas básicas consideradas como más importantes para las reglas de asociación, el soporte (s) y la confianza (c). El soporte de una regla de asociación se define como el porcentaje o fracción de transacciones que contienen $A \cup B$ sobre el total de transacciones en la base de datos. La confianza de una regla de asociación se define como el porcentaje o fracción del número de transacciones que contienen $A \cup B$ por sobre el número total de transacciones que contienen A. La confianza es una medida de la fortaleza de una regla de asociación, si la confianza de una regla $A \Rightarrow B$ es del 80%, esto significa que el 80% de las transacciones que contienen A también contienen B. La confianza también puede considerarse como un indicador de la probabilidad condicional entre A y B.

En resumen, soporte($A \Rightarrow B$) = P($A \cup B$) y confianza($A \Rightarrow B$) = P($B \mid A$), donde aquellas reglas que satisfacen un mínimo valor de soporte (*min_sop*) y un mínimo valor de confianza (*min_conf*) se denominan *fuertes* [Han y Kamber 2000], y son la salida de los algoritmos de reglas de asociación.

2.2 Generación de Reglas de Asociación

Un conjunto de elementos (tales como el antecedente o el consecuente de una regla) se denomina *itemset*. El número de elementos dentro de un *itemset* determina el tamaño del *itemset*, donde *itemsets* de tamaño k son referidos como *k-itemsets*.

Por lo general, un algoritmo de búsqueda de reglas de asociación contiene los siguientes pasos [Kotsiantis y Kanellopoulos 2006]:

1. El conjunto de *k-itemsets* candidatos es generado realizando extensiones de un elemento a partir del *(k – 1) -itemset* frecuente obtenido en la iteración anterior.

2. Calcular, a partir de la base de datos, el soporte de los nuevos *k-itemsets*.

3. Aquellos *itemsets* que no posean el mínimo soporte son descartados y los restantes *itemsets* son denominados *itemsets* frecuentes.

Este proceso es repetido hasta que no se puedan generar más *itemsets*.

Una vez que se obtuvieron todos los *itemsets* frecuentes, se generan a partir de estos las reglas de asociación cuyo valor de confianza excedan *min_conf* [Han y Kamber 2000]. Sea un *itemset* frecuente $L_k = \{I_1, ..., I_k\}$, las reglas de asociación son generadas de la siguiente manera [Kotsiantis y Kanellopoulos 2006]: la primer regla propuesta es $\{I_1, I_2, ..., I_{k-1}\} => \{I_k\}$, luego se controla que la confianza de la regla de asociación sea mayor a *min_conf*, caso contrario la regla es descartada. Luego, las demás reglas son generadas eliminando el último elemento del antecedente y agregándolo al consecuente, siempre controlando que la regla de asociación cumpla con el valor mínimo de confianza. El proceso continúa hasta que el antecedente queda vacío.

2.3 Algoritmo Apriori

En el presente trabajo, para obtener las reglas de asociación se utilizó el algoritmo Apriori [Agrawal y Srikant 1994]. Este algoritmo posee un funcionamiento similar al descripto anteriormente para los algoritmos de generación de reglas de asociación. Pero, para aumentar la eficiencia del proceso de generación de *itemsets* frecuentes, este algoritmo aplica lo que se conoce como la *propiedad Apriori* [Han y Kamber 2000]. Esta propiedad permite reducir el espacio de búsqueda y la cantidad de recorridas de la base de datos, mejorando de esta manera la *performance* del algoritmo.

2.3.1 Propiedad Apriori

Esta propiedad se basa en la siguiente observación. Por definición, si un *itemset I* no satisface el mínimo valor de soporte (*min_sop*), entonces *I* no es frecuente ($P(I) < min\_sop$). Si un elemento *X* es agregado al *itemset I*, entonces el *itemset* resultante ($I \cup X$) no puede ser más frecuente que *I*, por lo tanto, $I \cup X$ tampoco es frecuente ($P(I \cup X) < min\_sop$). La idea detrás de esta propiedad, es que cualquier subconjunto de un *itemset* frecuente debe ser, a su vez, frecuente.

El algoritmo Apriori, aprovecha esta propiedad durante la generación de *itemsets* candidatos haciendo que los *k-itemsets* candidatos se generen a partir de los *(k-1)-itemsets* frecuentes obtenidos en la iteración anterior, y eliminando aquellos *itemsets* generados que contengan algún subconjunto que no sea frecuente. Este procedimiento tiene como resultado la generación de un conjunto mucho menor de *itemsets* candidatos. En consecuencia, se reduce la cantidad de recorridas de la base de datos para determinar el soporte de estos nuevos *itemsets*.

Algoritmo: Apriori (Generación de *itemset* frecuentes).

Entrada: *D* (Base de datos); *min_sop* (mínimo valor de soporte).

Salida: *L*, conjunto de *itemsets* frecuentes en *D*.

Pasos:

```
1)   L1 = obtener_1_itemsets_frec (D);
2)   for (k = 2; Lk-1 ≠ Ø; k++) {
3)       Ck = apriori_gen(Lk-1, min_sop);
4)           for each transacción t ∈ D {
5)               Ct = subset(Ck, t);
6)               for each candidato c ∈ Ct
7)                   c.soporte++;
8)           }
9)       Lk = { c ∈ Ck | c.soporte >= min_sop };
10)  }
11)  retornar L = ∪k Lk;
```

Fig. V-1. Pseudo código del algoritmo Apriori.

2.3.2 Funcionamiento del Algoritmo

En la Fig. V-1 se presenta el algoritmo Apriori, donde L_k es un conjunto de *k-itemsets* frecuentes y C_k es un conjunto de *k-itemsets* candidatos. El algoritmo comienza obteniendo L_1, los *itemsets* de tamaño 1 frecuentes (paso 1). Luego, en los

pasos 2 a 10, L_{k-1} es usado para generar los candidatos C_k y de esa forma encontrar L_k. El procedimiento apriori_gen genera los candidatos y luego aplica la propiedad Apriori para eliminar aquellos que posean un subconjunto que no sea frecuente (paso 3). Una vez que los candidatos han sido generados, se recorre la base de datos (paso 4), y por cada transacción *t* se utiliza una función subset que retorna todos los subconjuntos de la transacción que son candidatos (paso 5). En los paso 6 a 7 se aumenta el soporte correspondiente a cada candidato presente en la transacción *t*. Finalmente, todos los candidatos cuyo valor de soporte es mayor a *min_sop* son considerados como *itemsets* frecuentes (paso 9). La salida del algoritmo es la unión de todos los *itemsets* frecuentes encontrados.

El procedimiento apriori_gen, realiza dos tipos de acciones: combinación (*join*) y poda (*prone*). Durante la combinación, el conjunto L_{k-1} es combinado consigo mismo para generar nuevos candidatos potenciales. La poda, en cambio, toma cada subconjunto de tamaño k - 1 de un *k-itemset* y verifica que el mismo esté incluido en L_{k-1}, caso contrario es borrado de C_k.

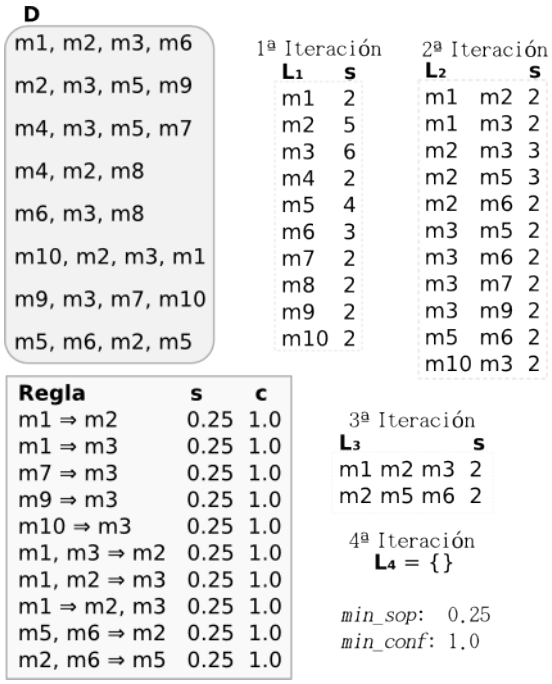


Fig. V-2. Ejemplo de generación de reglas de asociación a partir de una base de datos con 8 transacciones y *min_sop* = 0.25 y *min_conf* = 1.0.

En la Fig. V-2, se muestra un ejemplo de la utilización del algoritmo Apriori para obtener un conjunto de reglas de asociación a partir de una base de datos *D*. En este ejemplo, la base de datos posee 8 transacciones, las cuales están compuestas por elementos del conjunto $I = \{m1, m2, m3, m4, m5, m6, m7, m8, m9, m10\}$. Además, los valores de las cotas *min_sop* y *min_conf* son 0.25 y 1.0 respectivamente. Durante la primera iteración del algoritmo se obtuvieron 10 *itemsets* de tamaño 1 que satisfacen el mínimo soporte necesario (0.25 * 8 = 2). En la segunda iteración, los *itemsets* candidatos (C_2) se generaron combinando entre sí los *itemsets* de L_1, obteniéndose 11 *itemsets* frecuentes (L_2) de tamaño 2. La tercera iteración tiene como resultado la obtención de 2 *itemsets* frecuentes (L_3) y la cuarta iteración no genera ningún *itemset* frecuente ($L_4 = \emptyset$), por lo que el algoritmo se detiene.

Las reglas de asociación se obtienen a partir de los *itemset* frecuentes obtenidos, tal como se explicó anteriormente.

2.3.3 Cuestiones de Rendimiento

La eficiencia de los algoritmos de búsqueda de *itemsets*, está determinada principalmente por tres factores: la forma en que los candidatos son generados, las estructuras de datos utilizadas y los detalles de implementación [Bodon 2003]. La generación de los candidatos, para el caso del algoritmo Apriori, se ve acotada por la propiedad Apriori. Las estructuras de datos, tienen un papel principal debido a que durante los procesos de generación de candidatos, cada nuevo candidato debe ser almacenado y encontrado de forma eficiente en dicha estructura.

Existen dos estructuras de datos que han sido utilizadas previamente para almacenar los candidatos, a continuación se describen brevemente.

- ***Hashtree*** [Agrawal et al. 1996]. Un *hashtree* es un árbol n-ario donde cada nodo contiene una lista de *itemsets* (nodo hoja) o una tabla *hash* (nodo interno). Las tablas *hash* de los nodos internos apuntan a otros nodos (hojas o internos).

- ***Trie*** [Borgelt y Kruse 2002; Amir et al. 1997]. En un *trie*, cada *k-itemset* posee un nodo asociado al mismo, al igual que su prefijo (k-1)-*itemset*. La raíz del *trie* es el *itemset* sin elementos, los 1-*itemset* están enlazados al nodo raíz. Los demás k-*itemsets* están enlazados a su *itemset* prefijo de tamaño k – 1. Cada nodo almacena el último elemento del *itemset* que representa, su soporte y sus hijos.

Según Bodon [Bodon 2003], la estructura de datos *trie* ha demostrado superar a los *hashtrees* en velocidad, uso de memoria y sensibilidad de los parámetros.

2.4 Post-procesamiento de las Reglas de Asociación

La principal ventaja de las reglas de asociación es la posibilidad de obtener eficientemente el conjunto total de asociaciones que existen en los datos. Estas asociaciones proveen el panorama completo de las regularidades subyacentes en el dominio. Sin embargo, esta ventaja conlleva un gran inconveniente, el número de reglas de asociación descubiertas puede ser muy grande [Liu et al. 1999]; para una tarea de *data mining* este número podría ser del orden de los miles o cientos de miles.

Es evidente que dicho número de reglas de asociación es muy difícil (o imposible) de analizar por un ser humano. Por lo tanto, luego de la etapa de generación o descubrimiento de las reglas de asociación debe realizarse una etapa de filtrado o post-procesamiento de las mismas. Este filtrado puede realizarse con el objetivo de encontrar reglas que el usuario considere interesantes [Silberschatz y Tuzhilin 1995], eliminar reglas redundantes [Shah et al. 1999], o generar una representación resumida de las mismas.

La búsqueda de reglas de asociación interesantes implica encontrar aquellas reglas que el usuario considere útiles u originales. Existen dos tipos de medidas de interés aplicables a un patrón: medidas objetivas y medidas subjetivas [Silberschatz y Tuzhilin 1995]. Las medidas objetivas dependen únicamente de la estructura del patrón y de los datos utilizados durante el proceso de descubrimiento. Por ejemplo, según Piatetsky-Shapiro [Piatetsky-Shapiro 1991] el nivel de interés de una regla $A \Rightarrow B$ se puede definir como una función de P(A), P(B) y P($A \wedge B$). Las medidas subjetivas, por otra parte, dependen del conocimiento experto del usuario que las examina. En [Liu y Hsu 1996] se propuso una técnica basada en lógica difusa [Zimmermann 2001], que permite comparar reglas de asociación con reglas basadas en conocimiento que el usuario posee del dominio. En [Klementinen et al. 1994], se propone representar el conocimiento experto del usuario como un conjunto de reglas *template* cuya definición restringe la cantidad y tipo de elementos que una regla puede poseer sobre su antecedente o consecuente. De esta manera, todas las reglas que satisfacen las condiciones impuestas por las reglas *template* serán las más interesantes desde el punto de vista del experto.

Según Shah [Shah et al. 1999], las técnicas utilizadas para definir el interés poseen una gran desventaja: además de obtener los patrones deseados también generan una gran cantidad de patrones redundantes. Un patrón es considerado redundante si la misma información semántica es capturada por múltiples de ellos. Por ejemplo, sean las siguientes reglas de asociación: m1, m3 $\Rightarrow$ m2 (c: 91%) y m1 $\Rightarrow$ m2 (c: 90%) ambas reglas poseen un porcentaje de confianza similar y el mismo consecuente (m2). Se observa que el antecedente de la primer regla está lógicamente subsumido por el antecedente de la segunda regla, utilizando lógica de primer orden se determina que la segunda regla implica a la primera. Luego, la primer regla se denominará redundante, mientras que sólo la segunda será considerada como interesante.

Además de filtrar las reglas de asociación acorde a su nivel de interés o de eliminar aquellas consideradas como redundantes, algunos autores han propuesto técnicas para generar representaciones más compactas o resumidas del conjunto final de reglas de asociación obtenidas. En [Liu et al. 1999], se presenta una técnica que identifica un subconjunto de las reglas de asociación para formar un resumen de las asociaciones descubiertas. Este subconjunto de reglas de asociación representa las relaciones esenciales subyacentes en los datos, el resto de las reglas sólo agregan detalles adicionales a estas. Usando este subconjunto de reglas de asociación como base, el usuario podrá concentrarse en los aspectos claves del dominio y profundizar selectivamente en los detalles que el mismo desee.

Una vez que las reglas de asociación han sido generadas y filtradas, el usuario debe poder explorar y navegar entre las mismas. Diferentes técnicas de visualización han sido propuestas [Klementinen et al. 1994; Wong et al. 1999; Hetzler et al. 1998] para este propósito.

Por lo tanto, en base a las técnicas descriptas anteriormente, un posible escenario genérico de descubrimiento de reglas de asociación (Fig. V-3) estaría formado por las siguientes actividades: (1) Generación de las reglas de asociación, (2) Post-procesamiento de las reglas de asociación y, finalmente, (3) Visualización de las reglas de asociación.

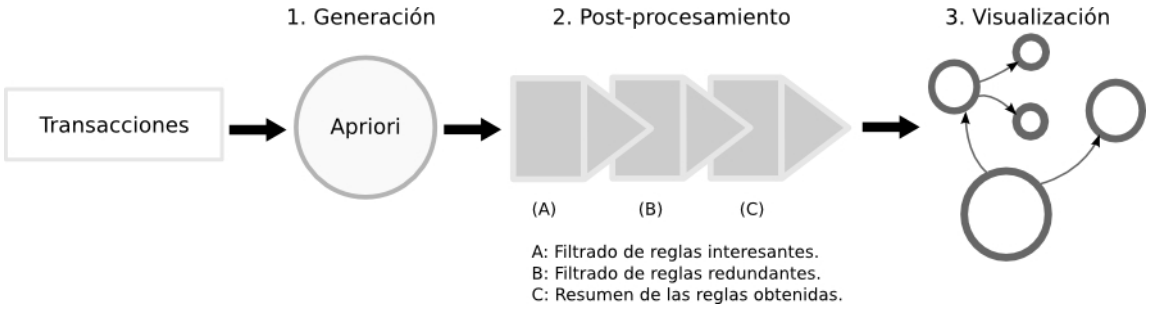


Fig. V-3. Proceso general utilizado en la obtención de reglas de asociación.

3 Identificación de Aspectos Candidatos como Reglas de Asociación

Dado que una traza de ejecución se compone de los métodos invocados durante una ejecución particular del sistema, diferentes ejecuciones del mismo programa deberían generar diferentes trazas, ya que diferentes partes del sistema son ejercitadas y corresponden a diferentes casos de uso o escenarios.

Sin embargo, muchas veces la misma clase podría colaborar en la realización de múltiples casos de uso, por lo que un método perteneciente a una clase podría estar presente en diferentes trazas obtenidas para diferentes escenarios. En base a esta observación, es posible identificar los patrones recurrentes presentes en las trazas aplicando reglas de asociación a las mismas, de manera de encontrar las asociaciones más relevantes entre los métodos ejecutados. Este tipo de reglas de asociación podrían revelar las asociaciones más frecuentes entre métodos permitiendo descubrir potenciales *crosscutting concerns*. En particular, las reglas de asociación obtenidas otorgan a los desarrolladores valiosa información sobre el comportamiento dinámico del sistema y permitirán la identificación de síntomas de código *scattering*.

En esta sección se describirá la técnica de *aspect mining* propuesta en detalle. Primero, se presenta el *workflow* del enfoque propuesto (sección 5.3.1). Luego, se describe el mecanismo de *tracing* desarrollado para realizar el análisis dinámico de un sistema (sección 5.3.2). La obtención de reglas de asociación es discutida (sección 5.3.3) y los filtros utilizados durante la fase de post-procesamiento son presentados y analizados (sección 5.3.4).

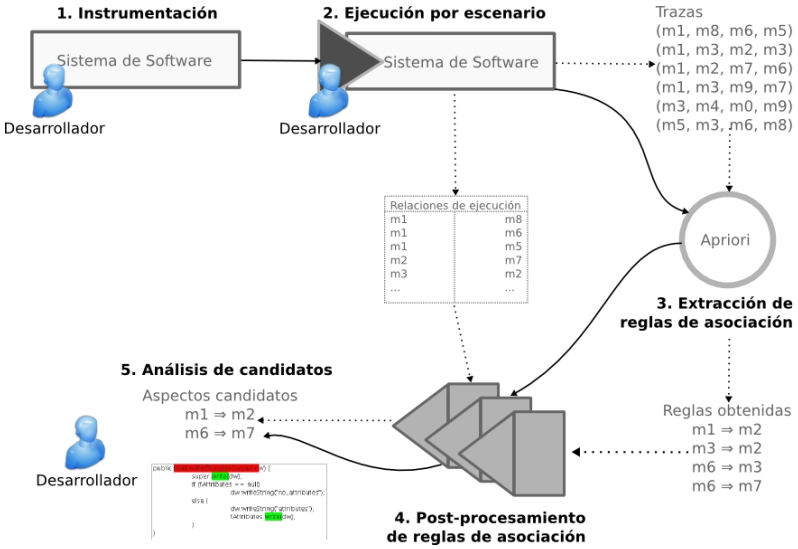


Fig. V-4. *Workflow* de la técnica propuesta. Las flechas punteadas indican datos de entrada o salida, mientras que las flechas continuas indican flujos de control entre las actividades.

3.1 *Workflow* de la Técnica Propuesta

En la Fig. V-4, los principales pasos del enfoque de *aspect mining* propuesto son mostrados. La participación del desarrollador dentro del *workflow* propuesto está dada principalmente en las primeras actividades y en la última de ellas. Podría considerarse la participación del usuario para establecer los valores mínimos de soporte y confianza durante la generación de las reglas de asociación (paso 3), pero la generación propiamente dicha está automatizada por el algoritmo correspondiente y su implementación.

La instrumentación del sistema (paso 1) implica modificar el sistema analizado para establecer la infraestructura necesaria que permita el análisis dinámico del mismo. En particular, el mecanismo de *tracing* es añadido a la aplicación como un nuevo paquete que contiene el aspecto de *tracing* y las clases de soporte. Si la aplicación posee una interfase gráfica, la misma es modificada para reflejar la funcionalidad correspondiente a la habilitación e inhabilitación del mecanismo de *tracing*.

Durante el segundo paso del proceso, el sistema es ejecutado a partir de un conjunto de escenarios de ejecución. Cada escenario representa una funcionalidad de alto nivel desde el punto de vista del usuario. La elección del conjunto de escenarios a probar en el sistema depende del tipo de análisis a realizar:

- *Estrategia orientada al objetivo*. De adoptarse este tipo de estrategia, los escenarios a elegir dependerán del objetivo que se desee alcanzar. Por ejemplo, si se deseara analizar el *concern* de persistencia en la aplicación JHotdraw, los escenarios que se deben probar son los correspondientes al almacenamiento y recuperación de un gráfico.

- *Análisis general*. Si lo que se desea es realizar un análisis general del sistema con el objetivo de ver qué *crosscutting concerns* pueden estar presentes en el mismo, los escenarios se deben elegir de forma tal que cada uno represente una funcionalidad de la aplicación visible externamente. Se deben evitar escenarios que sean equivalentes funcionalmente, ya que estos podrían ejercitar las mismas secciones de código del sistema induciendo un sesgo en el análisis posterior mediante reglas de asociación.

La salida de este segundo paso es un conjunto de trazas (una por escenario ejecutado) y las relaciones de ejecución que se dieron durante los escenarios ejecutados.

El tercer paso se corresponde con la ejecución del algoritmo Apriori de reglas de asociación. Este algoritmo toma como entrada un conjunto de transacciones (cada transacción se corresponde con una traza) y el mínimo valor de soporte y confianza. La salida de este paso es un conjunto de reglas de asociación.

En el cuarto paso, algunas reglas de asociación son descartadas y otras clasificadas mediante diferentes filtros de post-procesamiento. Estos filtros deben eliminar las reglas redundantes y eliminar reglas que no sean de interés para el desarrollador. A su vez, otros filtros implementan heurísticas para reconocer qué reglas de asociación representan aspectos candidatos.

El quinto paso corresponde al análisis de los aspectos candidatos obtenidos por parte del desarrollador. Para esto, el desarrollador puede valerse de técnicas que visualicen las reglas de asociación sobre el código fuente del sistema, o de otras técnicas de navegación y exploración que tomen como entrada alguno de los métodos dentro de las reglas.

3.2 Obtención de Trazas de Ejecución

El enfoque propuesto utiliza dos tipos de información dinámica: trazas de ejecución y relaciones de ejecución. La traza de ejecución de un programa está compuesta por la secuencia de métodos invocados durante la ejecución del mismo, mientras que, las relaciones de ejecución registran qué métodos son invocados por qué otros métodos.

La traza es una lista de los métodos que son invocados durante la ejecución de la aplicación, por lo tanto, el mecanismo de *tracing* debe observar qué métodos son invocados y por cada uno de ellos agregarlos a la traza de la ejecución.

Las relaciones de ejecución, se pueden representar como una tabla de dos columnas, la primera mostrando el método invocador y la segunda mostrando el método que fue invocado. Por lo tanto, el mecanismo de *tracing* no sólo debe observar qué método es invocado, sino también desde cuál otro método fue invocado. Por ejemplo, si un método m1 posee una llamada a un método m2, durante la ejecución del sistema la tabla de relaciones de ejecución poseerá una entrada con m1 en la columna invocador y m2 en la columna invocado.

En base a estas restricciones, el mecanismo de *tracing* (Fig. V-5) fue desarrollado utilizando tecnología POA, en particular se utilizó el lenguaje AspectJ para implementar el aspecto de *tracing*. La utilización de aspectos para obtener información dinámica permitió que el mecanismo de *tracing* sea reutilizable para diferentes aplicaciones, esto se debe a que el código del aspecto no depende del código de la aplicación. La única dependencia entre el aspecto de *tracing* y la aplicación está dada en la definición de sus *pointcuts*, por lo que la adaptación del mecanismo está restringida a la definición de los *pointcuts*.


```
1  package tracing;
2
3  public aspect Tracing {
4
5      private TraceBase trace = new TraceBase();
6
7      pointcut thePublicMethods(Object t) :
8              target(t) &&
9              execution(* CH.ifa.draw..*(..)) &&
10             !within(Tracing);
11
12     pointcut theConstructors() :
13             call(CH.ifa.draw..*.new(..)) && !within(Tracing);
14
15     pointcut theStaticMethods() :
16             execution(static * CH.ifa.draw..*(..)) && !within(Tracing);
17
18     private void beforeMethodAction(String s) {
19             if (trace.getEnabled()) {
20                     Element e = trace.addMethodTrace(s);
21                     if (e != null)
22                             trace.registrar_before(e.toStringFull());
23             }
24     }
25
26     private void beforeConstructorAction(String s) {
27             if (trace.getEnabled()) {
28                     Element e = trace.addConstructorTrace(s);
29                     if (e != null)
30                             trace.registrar_before(e.toStringFull());
31             }
32     }
33
34     private void afterAction() {
35             if (trace.getEnabled());
36                     trace.registrar_after();
37     }
38
39     before():theConstructors() {
40            beforeConstructorAction(thisJoinPoint.getSignature().toString());
41     }
42
43     before (Object t):thePublicMethods(t) {
44             beforeMethodAction(thisJoinPoint.getSignature().toString());
45     }
46
47     before ():theStaticMethods() {
48             beforeMethodAction(thisJoinPoint.getSignature().toString());
49     }
50
51     after():theConstructors() {
52             afterAction();
53     }
54
55     after(Object t):thePublicMethods(t) {
56             afterAction();
57     }
58
59     after():theStaticMethods() {
60             afterAction();
61     }
62 }
```

Fig. V-5. Aspecto que implementa el mecanismo de *tracing* necesario llevar a cabo el análisis dinámico de un sistema.

El aspecto de *tracing* posee un conjunto de *pointcuts* (líneas 7 a 16 de la Fig. V-5) que permiten entrelazar el código aspectual con el código base del sistema analizado, estos *pointcuts* deberán ser modificados manualmente por cada nuevo sistema que se desee analizar. Los *advice* asociados a los *pointcuts* (líneas 39 a 61 de la Fig. V-5) delegan en la clase TraceBase la gestión de la información dinámica. Esta clase es la encargada de actualizar la traza y las relaciones de ejecución durante la ejecución del sistema, a su vez posee métodos para pausar y reiniciar el registro de los datos. Las relaciones de ejecución se registran en una tabla de una base de datos en memoria, para esto se utilizó la base de datos HSQL[HSQL]. HSQL es una base de datos que puede ser mantenida totalmente en memoria permitiendo un acceso más rápido a los datos almacenados.

Una vez que el aspecto de *tracing* intercepta la llamada a un método definido dentro de alguno de los paquetes declarados en los *pointcuts*, el código del *advice* es ejecutado. El *advice*, a su vez, utiliza los métodos addMethodTrace y registrar_before o registrar_after de la clase TraceBase (líneas 20-22, 28-30, y 36 de la Fig. V-5). El método addMethodTrace registra un nuevo método en la traza siempre y cuando no haya sido insertado anteriormente (no se

permiten métodos repetidos). El método registrar_before tienen como responsabilidad actualizar las relaciones de ejecución, para esto, basan su funcionamiento en una pila. De esta manera, puede saber qué método está activo (el que está en el tope de la pila) al momento de realizarse la llamada al método actual, quedando formada la relación entre ambos métodos. La pila, por su parte, es actualizada agregando el método al principio de su invocación (responsabilidad del método registrar_before) y eliminando el mismo al final de su invocación (responsabilidad del método registrar_after).

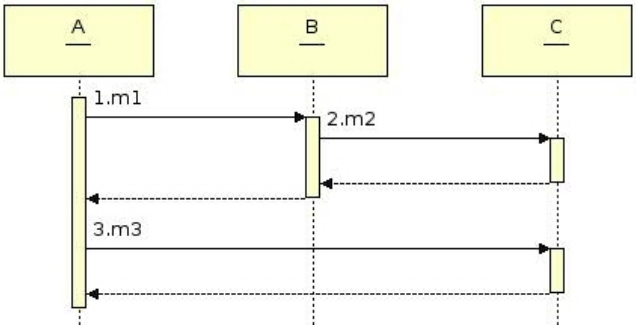


Fig. V-6. Ejemplo de un sistema en ejecución.

Sea el ejemplo de la Fig. V-6, la Tabla V-1 muestra el funcionamiento del mecanismo descripto anteriormente. Por cada evento, se muestra la situación inicial de la traza, la pila y las relaciones de ejecución y el estado final luego de tomar las acciones correspondientes. Cada vez que se invoca un método, éste es agregado a la traza, a la pila, y a una relación de ejecución si existe otro método en el tope de la pila (caso de la invocación m2). Cada vez que termina un método, se elimina un elemento de la pila (el tope).

Tabla V-1. Obtención de información dinámica a partir del ejemplo de la Fig. V-6.

	Inicio			**Final**		
Evento	**Traza**	**Pila**	**Relac. Ejec.**	**Traza**	**Pila**	**Relac. Ejec.**
Invocación m1	{}	Vacía	{}	{m1}	(m1)	{}
Invocación m2	{m1}	(m1)	{}	{m1, m2}	(m2, m1)	{(m1, m2)}
Fin m2	{m1, m2}	(m2, m1)	{(m1, m2)}	{m1, m2}	(m1)	{(m1, m2)}
Fin m1	{m1, m2}	(m1)	{(m1, m2)}	{m1, m2}	Vacía	{(m1, m2)}
Invocación m3	{m1, m2}	Vacía	{(m1, m2)}	{m1, m2, m3}	(m3)	{(m1, m2)}
Fin m3	{m1, m2, m3}	(m3)	{(m1, m2)}	{m1, m2, m3}	Vacía	{(m1, m2)}

3.3 Generación de las Reglas de Asociación

Si cada traza obtenida del sistema bajo análisis es considerada como una transacción *T* y los métodos contenidos en todas las trazas como el conjunto de elementos *I*, es posible obtener una base de datos *D* desde la cual generar un conjunto de reglas de asociación. Por ejemplo, las reglas de asociación para el ejemplo de la Fig. V-6 tendrán la siguiente forma:

$$B.m1 \Rightarrow C.m2 \text{ (s: 1.0; c: 1.0)}$$

Tanto los antecedentes como los consecuentes de las reglas resultantes estarán formados por los métodos ejercitados durante la etapa de análisis dinámico; estos métodos poseen como prefijo el nombre de la clase a la que pertenecen.

El valor de soporte de la regla indica el número de trazas (transacciones) en las cuales ambos métodos se encuentran presentes. En el ejemplo anterior, el valor de soporte indica que ambos métodos están presentes en todas las trazas. Por otra parte, el valor de confianza indica la estabilidad de la relación entre ambos métodos, entonces un valor de confianza de 1.0 significa que cada vez que el método *m1* de B es llamado también es llamado el método *m2* de C.

Adicionalmente, durante la generación de las reglas de asociación, el tamaño de los *itemsets* es acotado a *itemsets* de tamaño 2. Esto se realizó para permitir una mayor escalabilidad del proceso de generación de reglas de asociación. Se

debe considerar que el tamaño de las trazas puede alcanzar fácilmente los 150 métodos[4] y que la complejidad computacional de la generación de los *itemsets* depende de la cantidad de elementos en las transacciones. En consecuencia, las reglas de asociación obtenidas poseerán un único elemento en su antecedente y un único elemento en su consecuente.

3.4 Filtros de Identificación de Aspectos Candidatos

La identificación de posibles aspectos candidatos es realizado mediante la aplicación de dos filtros sobre las reglas de asociación obtenidas. El primero de ellos busca reglas de asociación cuyos antecedentes y consecuentes están compuestos por métodos que poseen el mismo nombre (Filtro Conceptual). Mientras que el segundo de ellos, busca reglas de asociación que compartan el mismo método en el consecuente (Filtro de Consecuente Recurrente). A continuación, se detallan ambos filtros.

3.4.1 Filtro Conceptual

Sea una regla de asociación A.*m* ⇒ B.*m* (soporte: *s* y confianza: *c*), se pueden hacer las siguientes observaciones a partir de ella:

1. Las clases A y B implementan el mismo método (*m*).

2. La ejecución de ambos métodos (A.*m* y B.*m*) fue consistente para tantos escenarios como indique el valor de soporte *s* de la regla. Si *s* es igual a 1.0, entonces ambos métodos fueron invocados durante la ejecución de todos los escenarios usados durante la fase de análisis dinámico.

3. Si el valor de confianza de la regla es alto (cercano a 1.0), entonces cada vez que se ejecutó el método *m* de la clase A también se ejecutó el método *m* de la clase B. Por lo que podría considerarse que ambos métodos trabajan en conjunto para satisfacer alguna funcionalidad o concepto de alto nivel.

Dado que diferentes clases implementan métodos con la misma signatura (o similar[5]) y que ambos métodos son ejercitados por más de un escenario de forma consistente, puede afirmarse que ambas clases están colaborando en la realización de un mismo *concern* y de que ese *concern* es, a su vez, *crosscutting*.

Si bien puede parecer que este filtro está basado puramente en una propiedad sintáctica (el nombre de los métodos – punto 1), la utilización de los valores de soporte y confianza aseguran que exista una relación semántica entre los elementos que componen una regla de asociación (por los puntos 2 y 3). La confianza de una regla, al ser un indicador de la probabilidad condicional entre el antecedente y el consecuente permite verificar estadísticamente que, por ejemplo, cada vez que A.*m* fue invocado también lo fue B.*m*. El valor de confianza de una regla de asociación para este tipo de filtro debería ser lo más alto posible.

De esta manera, el filtro conceptual se define como sigue: sea una regla de asociación $X \Rightarrow Y$, donde X e Y son métodos, el nombre de X debe ser igual al nombre de Y.

3.4.2 Filtro de Consecuente Recurrente

Cuando dos o más reglas comparten el mismo consecuente (A.*m2* ⇒ A.*m1* y B.*m3* ⇒ A.*m1*), es posible asumir que el método incluido en este ha sido invocado de manera consistente desde los métodos incluidos en los antecedentes de las reglas. Por lo tanto, el método incluido en el consecuente podría estar implementando funcionalidad que es requerida desde diferentes partes del sistema, indicando la presencia de un síntoma de *scattering*.

Este filtro se basa en la presencia de múltiples reglas de asociación que comparten el mismo consecuente, el método contenido en los consecuentes provee, por lo tanto, funcionalidad o comportamiento *crosscutting* siempre y cuando los métodos y clases que aparecen en los antecedentes sean parte de otros *concerns*. De lo contrario, se estaría en presencia de un método que ha sido llamado muchas veces durante la ejecución del sistema, pero no un indicador de código *scattering*.

La definición de este filtro es la siguiente: sea una regla de asociación $A \Rightarrow B$, donde A y B son métodos, las siguientes condiciones deben ser verdaderas.

1. A y B deben estar presentes en una relación de ejecución, donde A debe ser el método invocador y B el método que fue invocado.

4 Para sistemas de tamaño mediano (aproximadamente 5-10KLOC).

5 Los métodos se comparan por su nombre, pero no por su tipo de retorno o tipo y cantidad de parámetros.

2. *B* debe estar incluido en el consecuente de otra regla de asociación $C \Rightarrow B$, tal que *C* y *B* cumplen con la primera condición.

La primera condición permite asegurar que el método del antecedente invocó en tiempo de ejecución al método incluido en el consecuente, mientras que, la segunda condición verifica que el consecuente aparezca de forma recurrente en más de una regla de asociación.

4 Ejemplo del Enfoque Propuesto

Esta sección presenta un ejemplo de la utilización del enfoque propuesto para identificar aspectos candidatos en una implementación del patrón de diseño *Observer* [Gamma et al. 1995] (Fig. V-X).

El primer paso es la instrumentación del sistema, la misma se realizó habilitando y deshabilitando de forma manual el mecanismo de *tracing* directamente en el código del ejemplo analizado. Esto se debe a que el sistema es muy simple y carece de interfase gráfica.

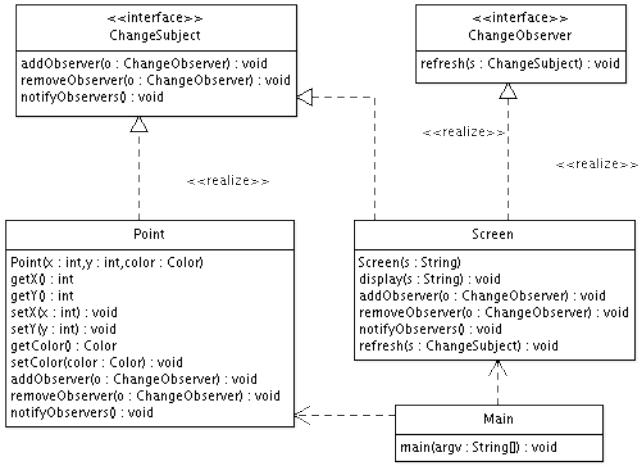


Fig. V-7. Diagrama de clases del ejemplo analizado.

El segundo paso corresponde a la ejercitación del código mediante la prueba del sistema bajo un conjunto de escenarios de ejecución, para esto se utilizaron dos escenarios: "un punto cambio su color" y "un punto cambia su posición". La ejecución del ejemplo de la Fig. V-7 bajo el escenario *un punto cambia su color* generará la traza y las relaciones de ejecución mostradas en la Fig. V-8. La traza resultante contiene la secuencia de métodos invocados (addObserver de la clase Point, addObserver de la clase Screen, setColor de la clase Point y así sucesivamente). Las relaciones de ejecución, por su parte, poseen entradas con las “relaciones de uso” que se dieron en tiempo de ejecución entre los métodos del sistema (setColor invocó a notifyObservers, notifyObservers a refresh, etc).

Traza del escenario “un punto cambia su color”

```
void observer.Point.addObserver(ChangeObserver);
void observer.Screen.addObserver(ChangeObserver);
void observer.Point.setColor(Color);void observer.Point.notifyObservers();
void observer.Screen.refresh(ChangeSubject);void observer.Screen.display(String);
void observer.Screen.notifyObservers();
```

Relaciones de ejecución para el escenario “un punto cambia su color”

Invocador	Invocado
void observer.Point.setColor(Color)	void observer.Point.notifyObservers()
void observer.Point.notifyObservers()	void observer.Screen.refresh(ChangeSubject)
...	...
void observer.Screen.refresh(ChangeSubject)	void observer.Screen.display(String)
void observer.Screen.display(String)	void observer.Screen.notifyObservers()

Fig. V-8. Ejemplo de información dinámica generada para el ejemplo de la Fig. II-5.

Durante el tercer paso, la información dinámica obtenida es sometida a un análisis mediante reglas de asociación. En

este caso, la base de datos *D* estará compuesta por dos transacciones (una por cada traza) y el conjunto de elementos *I* estará compuesto por nueve métodos. Estos nueve métodos son los que fueron ejercitados por los escenarios utilizados para ejecutar el código del ejemplo.

Tabla V-2. Reglas de asociación obtenidas para el ejemplo del patrón *Observer*.

	Regla de asociación	Soporte	Confianza
1	Main.main ⇒ Point.addObserver	1.0	1.0
2	Main.main ⇒ Point.setColor	0.5	0.5
3	Point.setColor ⇒ Point.notifyObservers	0.5	1.0
4	Point.setX ⇒ Point.notifyObservers	0.5	1.0
5	Screen.addObserver ⇒ Point.addObserver	1.0	1.0
6	Point.addObserver ⇒ Screen.addObserver	1.0	1.0

Mediante la aplicación del algoritmo Apriori sobre las trazas obtenidas con mínimo valor de soporte de 0.1 y mínimo valor de confianza de 0.1[6], se obtuvieron 70 reglas de asociación (parcialmente mostradas en la Tabla V-2).

El conjunto de reglas de asociación resultante demuestra la importancia del paso de post-procesamiento de las reglas. La sola aplicación del algoritmo de reglas de asociación es insuficiente para obtener resultados satisfactorios en la identificación de potenciales aspectos candidatos. Esto se debe principalmente a dos factores: primero, la cantidad de reglas de asociación obtenidas es tan alta que sin el filtrado de las mismas resultaría imposible su análisis, segundo, los filtros utilizados para identificar aspectos candidatos permiten aumentar el significado semántico de las mismas asociando a éstas la razón del por qué fueron consideradas como aspectos candidatos.

Por ejemplo, reglas de asociación en cuyos antecedentes o consecuentes se encuentran métodos de tipo "utilitarios" como main, hashCode o toString, no son interesantes respecto a los propósitos de *aspect mining*. Las reglas 1 y 2 sólo existen debido a que durante la ejecución de ambos escenarios el punto de partida es el método main. Por lo tanto, el primero de los filtros de post-procesamiento debe eliminar las reglas de asociación que contengan ya sea en su antecedente o consecuente este tipo de métodos no deseados.

Las reglas de asociación redundantes también deben ser eliminadas del conjunto final de reglas de asociación. Este caso se da para las reglas 5 y 6, ya que ambas reglas poseen mismo valor de confianza y proveen la misma información (en este caso se eliminaría la regla 6).

De la aplicación de los filtros de post-procesamiento (paso 4) sobre las reglas de asociación obtenidas inicialmente (mostradas parcialmente en la Tabla V-3), se obtuvieron las reglas de asociación mostradas en la Tabla V-3.

Tabla V-3. Conjunto final de reglas de asociación para el ejemplo del patrón *Observer*.

Concern	**Regla**	**Filtro**	**Sop.**	**Conf.**
Administración de observadores	Screen.addObserver ⇒ Point.addObserver	Conceptual	1.0	1.0
Mecanismo de notificación	Screen.notifyObservers ⇒ Point.notifyObservers	Conceptual	1.0	1.0
Mecanismo de notificación	Point.setColor ⇒ Point.notifyObservers	Consecuente recurrente	0.5	1.0
Mecanismo de notificación	Point.setX ⇒ Point.notifyObservers	Consecuente recurrente	0.5	1.0
Lógica de actualización	Point.notifyObservers ⇒ Screen.refresh	Consecuente recurrente	1.0	1.0
Lógica de actualización	Screen.notifyObservers ⇒ Screen.refresh	Consecuente recurrente	1.0	1.0

El quinto paso del proceso es un análisis subjetivo por parte del desarrollador, este análisis estará guiado por las reglas de asociación obtenidas y deberá resultar en la confirmación de algunas de estas reglas como indicadoras de

6 Los valores mínimos de soporte y confianza se eligieron de tal manera que el algoritmo de reglas de asociación retorne la máxima cantidad de reglas de asociación.

crosscutting concerns. Por ejemplo, observando las reglas (Point.setColor ⇒ Point.notifyObservers y Point.setX ⇒ Point.notifyObservers) el desarrollador podría concluir que existe un método de notificación que está siendo usado en los métodos que cambian el estado de la clase Point. El resultado de este análisis se refleja en la columna *Concern* de la Tabla V-3, ya que el desarrollador a medida que evalúa las reglas las va clasificando según el *concern* al que pertenecen.

En [Hannemann y Kiczales 2002], la misma implementación del patrón es analizada y refactorizada hacia una solución orientada a aspectos. En particular, los autores determinaron que el patrón *Observer* superpone funcionalidad correspondiente a los roles Subject y Observer sobre las clases Point y Screen. Esta funcionalidad es inherentemente *crosscutting* y se da en los siguientes mecanismos del patrón: administración de observadores (como mantener el mapeo entre el sujeto y sus observadores), mecanismo de notificación (lógica para avisar a los observadores que el estado del sujeto cambió), y lógica de actualización (el medio especifico para actualizar cada observador). Todos estos mecanismos son considerados *crosscutting concerns* y forman parte de la solución orientada a aspectos propuesta por Hannemann y Kiczales.

La aplicación de la técnica propuesta permitió detectar de manera semi-automática los mismos *crosscutting concerns* que los autores identificaron analizando manualmente el código del ejemplo.

5 YAAM (*Yet Another Aspect Miner*)

YAAM (*Yet Another Aspect Miner*) es una herramienta prototípica implementada en Java que permite la extracción de reglas de asociación a partir de trazas y relaciones de ejecución, y su posterior post-procesamiento. Además, YAAM brinda soporte para la visualización de las reglas de asociación sobre el código fuente de la aplicación bajo análisis. De esta manera, la herramienta permite automatizar, en gran parte, los pasos tres (extracción de reglas de asociación), cuatro (post-procesamiento de las reglas de asociación) y cinco (análisis de los candidatos) del proceso de *aspect mining* propuesto.

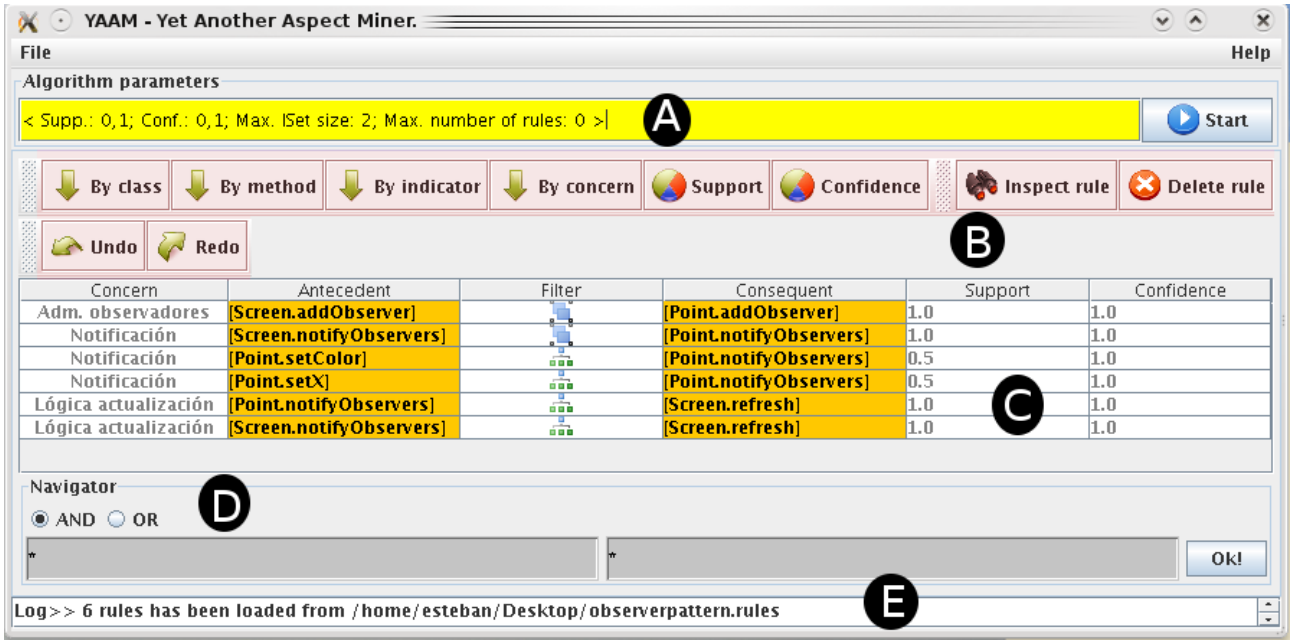


Fig. V-9. Vista principal de YAAM.

En la Fig. V-9 se muestra la ventana principal de la herramienta YAAM, los componentes más importantes de la misma son presentados a continuación.

A) Muestra los últimos parámetros utilizados para generar las reglas de asociación (soporte, confianza, tamaño máximo de *itemsets* y máximo número de reglas de asociación).

B) La barra de herramientas provee un conjunto de opciones para operar sobre las reglas de asociación generadas o cargadas desde un archivo. Los tres primeros botones permiten ordenar las reglas acorde a algún criterio (por clase, método, filtro o *concern*). Los botones Support y Confidence permiten al usuario filtrar las

reglas obtenidas según el soporte o la confianza. El botón 'Inspect rule' permite visualizar la regla de asociación seleccionada sobre el código fuente de la aplicación bajo análisis. Por otra parte, el botón 'Delete rule' permite eliminar una o más reglas de asociación. Los botones Undo y Redo proveen soporte para deshacer y rehacer las últimas acciones.

C) Por cada regla de asociación se muestra su soporte y confianza (a derecha de la Fig. V-9), su *concern* (a izquierda de la Fig. V-9), el filtro que la clasificó como aspecto candidato (centro de la Fig. V-9) y su antecedente y consecuente.

D) El navegador permite refinar el conjunto de reglas de asociación obtenido. Este componente permite expresar determinadas condiciones que pueden ser evaluadas sobre el antecedente o el consecuente de una regla o sobre ambos. Por ejemplo, escribir sobre la izquierda del navegador la expresión "**m**notifyObservers" descartaría aquellas reglas que no posean el método notifyObserver en su antecedente.

E) En la parte inferior de la Fig. V-9, está presente el *log* de acciones realizadas por el usuario. Este *log* muestra información que surge de la utilización de las diferentes operaciones y los resultados de las mismas.

Además de la funcionalidad descripta, la herramienta es capaz de almacenar y cargar las reglas de asociación a partir de un archivo y de exportar las reglas en formato html y xml.

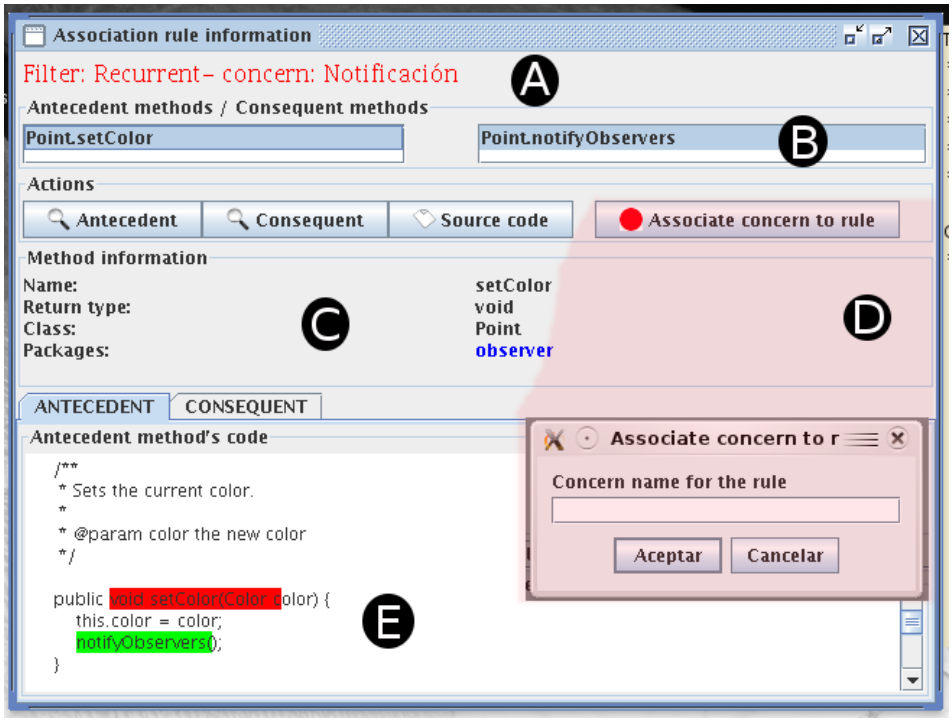


Fig. V-10. Visualización de un aspecto candidato. En rojo, YAAM marca el método del antecedente, y en verde, el método del consecuente.

YAAM permite visualizar una regla de asociación sobre el código fuente de la aplicación (Fig. V-10). Para esto, el usuario debe especificar el *path* al código fuente y la herramienta se encargará posteriormente de cargar los fuentes correspondientes.

Cada vez que una regla de asociación es visualizada, se puede observar:

A) El *concern* que esa regla de asociación representa (en caso de que el usuario haya especificado alguno), y el filtro mediante el cual esa regla fue clasificada.

B) Los métodos correspondientes tanto al antecedente como al consecuente.

C) Información correspondiente a cada uno de esos métodos, tal como su nombre, la clase y paquete al que pertenece y su tipo de retorno.

D) El botón 'Associate concern to rule' permite al desarrollador especificar o asociar un *concern* a la regla de asociación que está siendo visualizada.

E) A su vez, el código fuente de las clases del método del antecedente y del método del consecuente es cargado y puede ser navegado por el desarrollador. Luego, dependiendo de la pestaña elegida (ANTECEDENT o CONSEQUENT), YAAM resalta en rojo y verde el nombre del método correspondiente.

6 Comparación con Enfoque Previos de *Aspect Mining* Dinámicos

La investigación en *aspect mining*, puede ser clasificada en dos categorías: técnicas basadas en análisis estático y técnicas basadas en análisis dinámico. A continuación, se compara el enfoque propuesto respecto a las técnicas de *aspect mining* basadas en análisis dinámico.

En [Breu y Krinke 2004] analizan las trazas de ejecución en búsqueda de patrones recurrentes en la ejecución de los métodos del sistema. Un patrón es considerado como aspecto candidato si ocurre más de una vez de manera uniforme, además, para asegurarse que estos patrones sean suficientemente *crosscutting*, los mismos deben aparecer en diferentes 'contextos de ejecución'. Las condiciones que se imponen sobre los métodos en esta técnica, son similares a los impuestos por el filtro de consecuente recurrente del enfoque propuesto. La técnica propuesta supera a la propuesta en [Breu y Krinke 2004] en el tipo de candidatos generados (una regla de asociación es más expresiva que un método recurrente) y en la flexibilidad del enfoque propuesto (es posible ser extendido con nuevos tipos de reglas de asociación y tipos de post-procesamiento).

Otro enfoque dinámico [Tonella y Ceccato 2004], analiza las trazas del sistema mediante *formal concept analysis*, e identifica aspectos candidatos en el *lattice* de conceptos resultante. La principal ventaja de esta técnica respecto a la propuesta en este trabajo, reside en que es capaz de reconocer indicadores de código *tangling* y de código *scattering,* en contraste con el enfoque propuesto que sólo reconoce indicadores de código *scattering*. Sin embargo, el *lattice* de conceptos resultante de esta técnica reviste una complejidad mayor para su análisis respecto a una lista de reglas de asociación.

No existen aún estudios cuantitativos que permitan comparar dos o más técnicas de *aspect mining* respecto a un sistema de referencia en común. Por lo tanto, las comparaciones que se realizaron son de índole cualitativa y no cuantitativa.

7 Conclusiones

En el presente capítulo, se presentó y desarrolló un enfoque para la identificación de *crosscutting concerns* en sistemas legados orientados a objetos. Este enfoque está basado en la utilización de análisis dinámico como medio para obtener una representación del comportamiento del sistema bajo un conjunto de escenarios de ejecución predefinidos, y en la aplicación de algoritmos de reglas de asociación como técnica para el análisis de la información obtenida dinámicamente.

Las principales ventajas de este enfoque radican en la identificación automática de síntomas de *scattering*, en la obtención de aspectos candidatos en forma de reglas de asociación, las cuales poseen un gran poder expresivo y en la posibilidad de definir nuevos y mejores filtros de post-procesamiento. Además, el enfoque es extensible en mucho de sus pasos, por ejemplo, es posible registrar más información dinámica (como la cantidad de veces que un método fue ejecutado) o cambiar el tipo de reglas de asociación a extraer (como reglas de asociación generalizadas [Srikant y Agrawal 1995]).

Los principales problemas asociados al enfoque, son los mismos que poseen el análisis dinámico y las reglas de asociación: el análisis es parcial (solo se analiza lo que se ejecutó) y la cantidad de reglas de asociación muchas veces es excesiva. Ambos problemas han sido reconocidos anteriormente y muchos autores han propuesto diferentes técnicas para solucionar o disminuir el impacto de los mismos.

Capítulo VI

Resultados Experimentales

El presente capítulo describe los resultados obtenidos a partir del análisis de la aplicación JHotDraw en su versión 5.4b1 mediante el enfoque de aspect mining propuesto. Para esto, se relevaron de la literatura y trabajos previos en el área un conjunto de crosscutting concerns previamente identificados en la aplicación. Este conjunto de crosscutting concerns permitirán evaluar hasta qué punto el enfoque propuesto es capaz de identificar crosscutting concerns en un sistema legado. Posteriormente, se analizan los resultados obtenidos para determinar el porcentaje de falsos positivos y la precisión en general de la técnica propuesta y de los diferentes filtros de post-procesamiento utilizados.

1 Evaluación de Técnicas de *Aspect Mining*

Acorde a Marin et al. [Marin et al. 2007], cuando se evalúa la calidad de una técnica de *aspect mining* hay dos cuestiones principales que deben ser tenidas en cuenta. La primer cuestión es la falta de un sistema de referencia (o *benchmark*) donde los *crosscutting concerns* hayan sido previamente identificados y aceptados. Hasta el momento, dicho *benchmark* no existe. Sin embargo, un número creciente de investigaciones en *aspect mining* [Marin et al. 2007; Ceccato et al. 2005; Shepherd y Pollock 2005; Moldovan y Serban 2006a; Moldovan y Serban 2006b; Zhang y Jacobsen 2007] han utilizado alguna versión de la aplicación JHotDraw [JHotDraw] como su caso de estudio, la cual, por lo tanto, se está convirtiendo en dicho sistema de referencia.

La segunda cuestión, corresponde a que la decisión de si un *concern* es *crosscutting* y si el mismo posee una implementación adecuada mediante el uso de aspectos es una elección de diseño, la cual es un *trade-off* entre diferente alternativas. Por lo tanto, no existe una respuesta "sí/no" a la pregunta sobre si un *concern* identificado puede ser encapsulado como un aspecto. En consecuencia, la recolección de datos cuantitativos sobre el número de falsos negativos (cuantos *crosscutting concern* no fueron detectados) o falsos positivos (cuantos de los *concerns* identificados no son de hecho *crosscutting*) es una actividad subjetiva. Por lo que, la evaluación de una técnica de *aspect mining* en términos de, por ejemplo, porcentajes de falsos positivos y negativos, o en términos de precisión y *recall*, es una sobre-simplificación.

En el presente capítulo, se analizará el sistema JHotDraw a través de un análisis cualitativo y un análisis cuantitativo. El análisis cualitativo, pretende analizar las reglas de asociación obtenidas y dar razones de por qué estas dan indicios de la presencia de un *concern* en particular. El análisis cuantitativo, en cambio, analiza todas las reglas de asociación obtenidas con el objetivo de evaluar la precisión de la técnica según el valor de soporte utilizado, los porcentajes de *concerns* identificados, los *concerns* identificados según el filtro de post-procesamiento, entre otros análisis.

2 JHotDraw 5.4b1

JHotDraw es un *framework* de dibujado en dos dimensiones desarrollado originalmente como un ejercicio para ilustrar el uso correcto de los patrones de diseño orientados a objetos [Gamma et al. 1995]. El mismo esta implementado en Java y posee aproximadamente 18000 líneas no comentadas de código y 2800 métodos. Estas características lo recomiendan como un caso de estudio bien diseñado, un requisito para demostrar las mejoras que se pueden obtener a partir de las técnicas orientadas a aspectos. Además, demuestra que inclusive en sistemas (legados) bien diseñados existen limitaciones de modularización [Ceccato et al. 2005].

Desde su adopción original, JHotDraw ha sido utilizado en diferentes estudios sobre técnicas de *aspect mining*. Por ejemplo, en [Ceccato et al. 2005] se analiza JHotDraw mediante tres técnicas de *aspect mining* y se comparan sus resultados de manera cualitativa. Mientras que en [Marin et al. 2007], se analiza JHotDraw a través de la métrica fan-in y se discuten en profundidad los diferentes *crosscutting concerns* descubiertos por el autor.

Por lo tanto, para comparar y validar los resultados obtenidos sobre JHotDraw, se comparará la unión de los *crosscutting concerns* identificados en [Ceccato et al. 2005] y [Marin et al. 2007] (Tabla VI-1) con aquellos descubiertos por el enfoque propuesto.

Tabla VI-1. *Crosscutting concerns* objetivo.

Crosscutting concern	**Descripción**
Adapter	Instancias del patrón de diseño *Adapter* [Gamma et al. 1995].
Command	Instancias del patrón de diseño *Command* [Gamma et al. 1995].
Composite	Instancias del patrón de diseño *Composite* [Gamma et al. 1995].
Consistent Behaviour (CB)	Funcionalidad requerida desde, o impuesta sobre un conjunto de participantes en un contexto dado. Por ejemplo, un método log agregado al final de todos los métodos que se deseen registrar.
Contract Enforcement (CE)	Condiciones al inicio o al final de un conjunto de métodos. Puede verse como una especialización del *concern* anterior, donde la funcionalidad impuesta es la verificación de una condición.
Decorator	Instancias del patrón de diseño *Decorator* [Gamma et al. 1995].
Handle	Funcionalidad relacionada a la gestión de los "handles" asociados con los elementos gráficos. Estos "handles" permiten la edición interactiva de las figuras mediante operaciones tales como el cambio de dimensión de una figura.
Observer	Instancias del patrón de diseño *Observer* [Gamma et al. 1995].
Persistence	Funcionalidad relacionada a la persistencia y resurrección del estado de un objeto.
State	Instancias del patrón de diseño *State* [Gamma et al. 1995].
Undo	Funcionalidad que permite deshacer o rehacer los cambios provocados por una operación.
Bring to Front / Send to Back	La funcionalidad asociada a este *concern* consiste en la posibilidad de mover las figuras hacia arriba o abajo de otra figura o imágen.

2.1 Preparación del Experimento

El experimento comienza mediante la realización de los pasos 1 y 2 del proceso de *aspect mining* propuesto. Estos dos pasos consisten en la etapa de análisis dinámico del enfoque propuesto.

La instrumentación (paso 1) es realizada añadiendo el aspecto de *tracing* a la aplicación y modificando la interfase gráfica de la misma de forma de permitir la habilitación e inhabilitación del mismo. En la Fig. VI-1, se observa cómo el mecanismo de *tracing* fue añadido como un nuevo paquete en la aplicación (parte superior de la figura) y las modificaciones realizadas a la interfase gráfica (parte inferior de la figura).

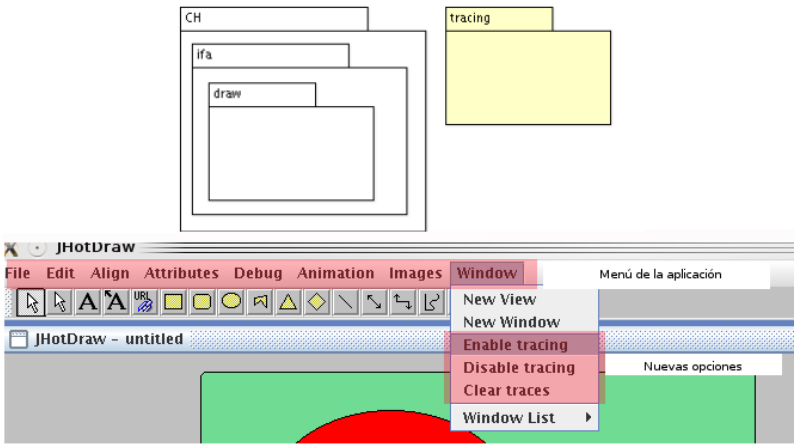


Fig. VI-1. Instrumentación de JHotDraw para permitir el análisis dinámico de la aplicación.

El segundo paso del proceso consiste en la ejercitación del sistema a través de un conjunto de escenarios de

ejecución. Para el caso de JHotDraw, se definieron 21 escenarios (Tabla VI-2) representativos de las funcionalidades principales descriptas en la documentación disponible [JHotDraw]. Cuando los escenarios fueron probados, estos ejercitaron 610 métodos de un total de 2800 métodos presentes en JHotDraw.

Tabla VI-2. Listado de los escenarios de ejecución utilizados en el experimento.

	Escenarios de ejecución utilizados
1	Inicio de la aplicación.
2	Crear un nuevo documento.
3	Agregar un rectángulo al documento activo.
4	Seleccionar un rectángulo.
5	Cambiar el color de un rectángulo.
6	Seleccionar la operación Undo.
7	Seleccionar la operación Redo.
8	Seleccionar la operación Duplicate.
9	Seleccionar la operación Delete.
10	Agregar una imagen.
11	Guardar el documento activo.
12	Abrir un documento previamente guardado.
13	Cerrar la ventana del documento activo.
14	Cambiar el tamaño de la ventana activa.
15	Seleccionar la herramienta Text Tool y agregar una al documento activo.
16	Agregar un elemento URL a una figura del documento activo.
17	Mover un rectángulo.
18	Cambiar de tamaño un rectángulo.
19	Seleccionar todos los elementos de un documento y cortarlos (*cut*).
20	Pegar un conjunto de elementos previamente cortados.
21	Seleccionar la opción Toggle Snap to Grid.

El hecho de que sólo el 21% (610 sobre 2800) de los métodos presentes en JHotDraw 5.4b1 hayan sido ejercitados demuestra que el análisis dinámico sólo puede arrojar resultados parciales, debido a que no es posible alcanzar una cobertura total de los métodos de la aplicación. En consecuencia, la posible existencia de *crosscutting concerns* en los métodos no ejecutados podría aumentar la cantidad de falsos negativos de la técnica propuesta.

2.2 Obtención de las Reglas de Asociación

Los pasos 3 y 4 del proceso de *aspect mining* propuesto corresponden a la generación y post-procesamiento de las reglas de asociación obtenidas a partir de las trazas y relaciones de ejecución. La herramienta YAAM permite automatizar estos dos pasos, aunque existe una pequeña participación del desarrollador ya sea especificando el archivo de las trazas y relaciones de ejecución o estableciendo los parámetros del algoritmo de búsqueda de reglas de asociación.

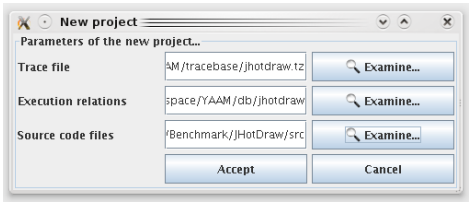


Fig. VI-2. Crear un nuevo proyecto en YAAM.

En la Fig. VI-2, se observa la creación de un nuevo proyecto para el análisis de la aplicación JHotDraw. En la figura se observa que el desarrollador debió especificar diferentes *path* correspondientes al archivo que contiene las trazas de ejecución (jhotdraw.tz), el archivo que contiene las relaciones de ejecución (jhotdraw) y la raíz del código fuente (esto último para poder visualizar las reglas sobre el código fuente).

A continuación, el desarrollador debe especificar los valores de entrada para el algoritmo Apriori. En la Fig. VI-3, se observan los valores utilizados para generar las reglas de asociación. El valor de soporte es el mínimo necesario para que cada regla de asociación sea válida en dos transacciones (o trazas de ejecución), mientras que se utilizó un valor alto de confianza de manera de reducir el conjunto final de reglas de asociación.

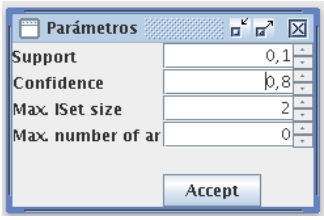


Fig. VI-3. Valores de entrada para el algoritmo de reglas de asociación.

Una vez que los filtros de post-procesamiento fueron aplicados, el conjunto de aspectos candidatos alcanzó las 482 reglas de asociación. Posteriormente, cada regla de asociación fue analizada de forma manual por parte del desarrollador para decidir si esta es indicadora de un *crosscuting concern* o no. El tiempo necesario para la realización de este análisis depende principalmente de dos factores:

- El conocimiento que el desarrollador posee sobre el sistema analizado.
- El soporte que la herramienta provee para navegar y explorar las reglas de asociación.

Para este caso en particular, el desarrollador posee conocimiento sobre el diseño y la implementación de la aplicación, y además, el código posee una documentación acabada sobre el propósito de cada clase dentro del sistema y sobre los patrones de diseño utilizados. Por otra parte, YAAM provee soporte para visualizar una regla de asociación sobre el código fuente lo que permite un análisis cercano al código para determinar si la regla de asociación es indicadora de un *crosscutting concern* o no.

2.3 Análisis Cualitativo de las Reglas de Asociación Obtenidas

En la Tabla VI-3 se muestra un subconjunto de reglas de asociación correspondiente a cada *crosscutting concern* detectado mediante el enfoque propuesto. Por cada regla de asociación, se incluye el *concern* al que pertenece, el filtro que la identificó, el valor de soporte y el valor de confianza.

La técnica de *aspect mining* propuesta fue capaz de detectar la mayoría de los *crosscutting concerns* de la Tabla VI-1 más una instancia del patrón *Mediator* [Gamma et al. 1995]. El *concern* “Bring to front/Send to back” es el único que la técnica no fue capaz de detectar.

Tabla VI-3. Reglas de asociación por crosscutting concern identificado.

Concern	Regla	Filtro	Sop.	Conf.
Adapter	UndoableCommand.execute ⇒ UndoableCommand.getWrappedCommand	Cons. rec.	0.28	1.0
Adapter	UndoableCommand.isExecutable ⇒ UndoableCommand.getWrappedCommand	Cons. rec.	0.66	1.0
Consistent Behaviour	AlignCommand.isExecutableWithView ⇒ StandardDrawingView.selectionCount	Cons. rec.	0.66	1.0
Consistent Behaviour	BringToFrontCommand.isExecutableWithView ⇒ StandardDrawingView.selectionCount	Cons. rec.	0.66	1.0
Contract Enforcement	AbstractTool.deactivate ⇒ AbstractTool.isActive	Cons. rec.	0.38	1.0

Contract Enforcement	DragNDropTool.viewCreated ⇒ AbstractTool.isActive	Cons. rec.	0.09	1.0
Command	BringToFrontCommand.isExecutableWithView ⇒ AlignCommand.isExecutableWithView	Concepto	0.66	1.0
Command	UngroupCommand.isExecutableWithView ⇒ UndoCommand.isExecutableWithView	Concepto	0.66	1.0
Composite	CompositeFigure.draw ⇒ AttributeFigure.draw	Concepto	0.76	0.84
Composite	CompositeFigure.findFigureInside ⇒ AbstractFigure.findFigureInside	Concepto	0.09	1.0
Decorator	DecoratorFigure.findFigureInside ⇒ AbstractFigure.findFigureInside	Concepto	0.09	1.0
Decorator	DecoratorFigure.basicDisplayBox ⇒ DecoratorFigure.getDecoratedFigure	Cons. rec.	0.09	1.0
Decorator	DecoratorFigure.containsPoint ⇒ DecoratorFigure.getDecoratedFigure	Cons. rec.	0.19	1.0
Handle	RelativeLocator.south ⇒ BoxHandleKit.south	Concepto	0.14	1.0
Handle	RelativeLocator.west ⇒ BoxHandleKit.west	Concepto	0.14	1.0
Mediator	UndoableCommand.execute ⇒ DrawApplication.figureSelectionChanged	Cons. rec.	0.28	1.0
Mediator	UndoableTool.deactivate ⇒ DrawApplication.figureSelectionChanged	Cons. rec.	0.14	1.0
Observer	AbstractFigure.displayBox ⇒ AbstractFigure.willChange	Cons. rec.	0.14	1.0
Observer	AbstractFigure.moveBy ⇒ AbstractFigure.willChange	Cons. rec.	0.14	1.0
Persistence	AttributeFigure.read ⇒ AbstractFigure.read	Concepto	0.14	1.0
Persistence	RectangleFigure.write ⇒ AnimationDecorator.write	Concepto	0.14	1.0
State	CreationTool.activate ⇒ AbstractTool.activate	Cons. rec.	0.09	1.0
State	DrawApplication.setTool ⇒ AbstractTool.activate	Cons. rec.	0.33	1.0
State	CreationTool.activate ⇒ UndoableTool.activate	Concepto	0.09	1.0
Undo	DragTracker.activate ⇒ AbstractTool.getUndoActivity	Cons. rec.	0.09	1.0
Undo	UndoableTool.deactivate ⇒ AbstractTool.getUndoActivity	Cons. rec.	0.14	1.0

A continuación, se analizan los *crosscutting concerns* descubiertos en mayor detalle.

2.3.1 Patrón Adapter

Las reglas de asociación para el patrón *Adapter* muestran una instancia donde este patrón fue utilizado para adaptar objetos de un patrón *Command* a elementos de la interfase gráfica. La técnica identificó este patrón ya que para acceder a la instancia de la clase *adaptada* se utiliza de manera consistente una llamada al método getWrappedCommand.

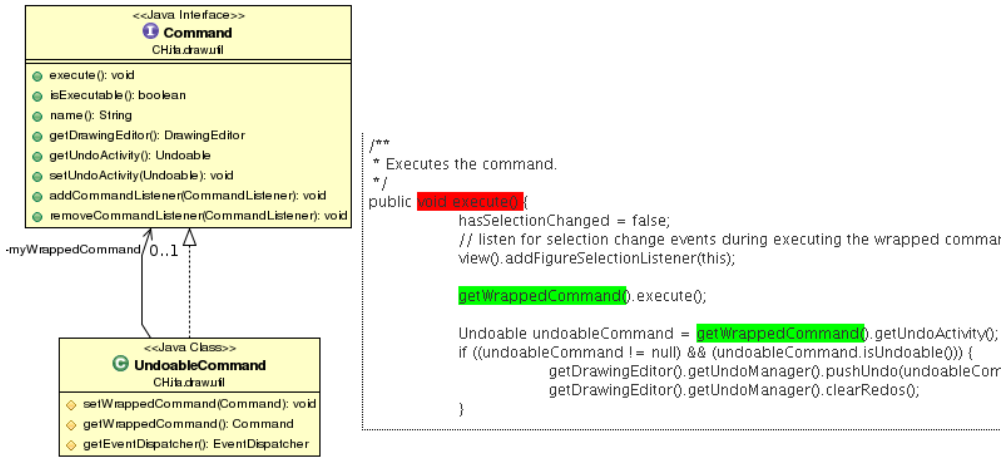


Fig. VI-4. Patrón *Adapter* en JHotDraw.

En la Fig. VI-4, se presenta el diagrama de clases correspondiente a este patrón (simplificado por cuestiones de espacio) y la visualización de la regla de asociación UndoableCommand.execute ⇒ UndoableCommand.getWrappedCommand.

La refactorización de este patrón hacia una solución orientada a aspectos, consiste en eliminar las clases correspondientes a los adaptadores y utilizar aspectos que entretejan el código del patrón sobre las clases adaptadas [Hannemann y Kiczales 2002; Lesiecki 2005].

2.3.2 Consistent Behaviour

Las reglas de asociación presentadas muestran una de las instancias descubiertas de este *concern* donde se utiliza de forma consistente el método selectionCount de la clase StandarDrawingView. La invocación a este método es realizada desde diferentes métodos en diferentes clases.

```
public boolean isExecutableWithView() {
        return view().selectionCount() > 0;
}

protected boolean isExecutableWithView() {
        return view().selectionCount() > 1;
}
```

Fig. VI-5. Código del *concern Consistent Behaviour*.

En la Fig. VI-5, se muestra la visualización de las reglas de asociación correspondientes a este *concern*. El filtro de consecuentes recurrentes fue el que identificó todas las instancias de este *concern*.

Para refactorizar este *concern* hacia una solución orientada a aspectos se debe mover el código duplicado desde las clases hacia un *advice* en un aspecto y definir el *pointcut* correspondiente.

2.3.3 Contract Enforcement

Las reglas de asociación correspondientes a este *concern* relacionan métodos que son parte de pre/pos-condiciones, siendo por lo general el método incluido en el consecuente el que forma parte de la condición a verificar. Este tipo de *concern* es identificado principalmente mediante el filtro de consecuente recurrente.

En la Fig. VI-6, se observan las visualizaciones correspondientes a las dos reglas presentadas para este *concern*. El mismo se refactoriza utilizando la misma solución que el *concern* anterior.

```
public void viewCreated(DrawingView view) {
        super.viewCreated(view);
        if(Component.class.isInstance( view )/* && DNDInterface.class.isInstance( view
                Component c = (Component) view;
                try {
                        DropTarget dt = new DropTarget(c ,Dnl
                        fDropTargets.add( dt );
                        //System.out.println("View " + view.getID
                }
                catch (java.lang.NullPointerException npe) {
                        System.err.println("View Failed to initialize to DND.");
                        System.err.println("Container likely did not have peer
                        System.err.println(npe);
                }
        }
        if(isActive()) {
                createDragGestureRecognizer(view, this);
        }
}
public void deactivate() {
        if (isActive()) {
                if (view() != null) {
                        view().setCursor(Cursor.getDefaultCursor());
                }
                getEventDispatcher().fireToolDeactivatedEvent();
        }
```

Fig. VI-6. *Contract enforcement* en JHotDraw.

2.3.4 Patrón Command

Las reglas de asociación que indican la presencia del patrón *Command* fueron clasificadas como aspectos candidatos por el filtro conceptual. Esto se debe a que este patrón se encuentra disperso en muchas de las clases que fueron ejercitadas durante la etapa de análisis dinámico, las cuales comparten los mismos métodos ya que heredan de la misma clase (AbstractCommand) o realizan la misma interfase (Command) (Fig. VI-7).

Debido a que JHotDraw esta organizado a partir de una arquitectura Model-View-Controller [Buschmann et al.

1996], la utilización del patrón *Command* implica que todas las funcionalidades de alto nivel ofrecidas desde la interfaz gráfica de la aplicación estén encapsuladas como un objeto *Command*. En consecuencia, las trazas de ejecución registran un alto contenido de métodos relacionados a este patrón y las reglas de asociación obtenidas a partir de estas también.

Este es el *concern* que posee mayor cantidad de reglas de asociación (106), seguido del *concern* de persistencia, para el cual se identificaron 35 reglas.

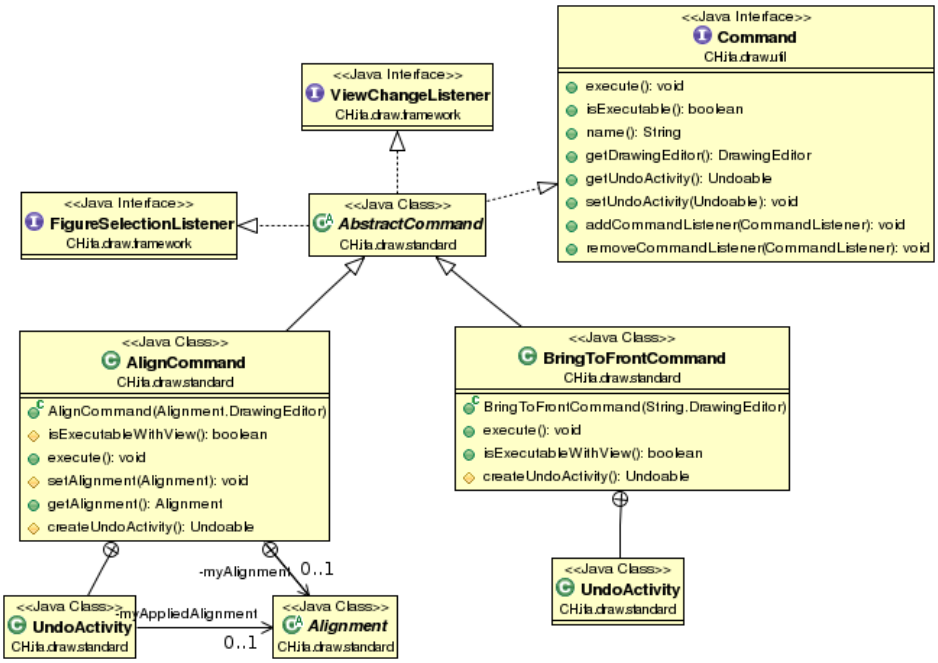


Fig. VI-7. Diagrama de clases que muestra parcialmente la jerarquía de clases descendientes de Command.

En [Hannemann y Kiczales 2002], se propone una posible solución para la refactorización hacia aspectos de este *concern*.

2.3.5 Patrón Composite

El patrón *Composite* superpone responsabilidades adicionales a la funcionalidad de una clase, como por ejemplo la gestión y el acceso a sus hijos.

En la Fig. VI-8, se observa cómo el método draw de un elemento gráfico compuesto posee código correspondiente para el acceso a sus hijos. Otras reglas para este *concern* muestran la utilización de los métodos figures o figuresReverse de la clase CompositeFigure, los cuales retornan una colección con los hijos del objeto compuesto.

```
/**
 * Draws all the contained figures
 * @see Figure#draw
 */
public void draw(Graphics g) {
        FigureEnumeration fe = figures();
        while (fe.hasMoreElements()) {
                fe.nextFigure().draw(g);
        }
}
```

Fig. VI-8. Código correspondiente al patrón *Composite*.

La refactorización del patrón *Composite* utilizando el paradigma orientado a aspectos es descripta en [Hannemann y

Kiczales 2002].

2.3.6 Patrón Decorator

El patrón *Decorator* se caracteriza (al igual que el patrón *Adapter*) por hacer uso de re-direccionado consistente (Fig. VI-9). Las reglas encontradas para este *concern*, demuestran la presencia del re-direccionado consistente y de la utilización del método getDecoratedFigure para acceder a la instancia decorada.

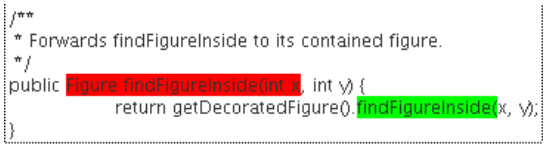


Fig. VI-9. Instancia identificada del patrón *Decorator*.

Este patrón posee una refactorización orientada a aspectos similar a la del patrón *Adapter*, la cual se describe en [Hannemann y Kiczales 2002] y en [Lesiecki 2005].

2.3.7 Handle

Este *concern* representa funcionalidad correspondiente a la gestión de los "handles" (Fig. VI-10) asociados a las figuras gráficas; estos "handles" permiten cambiar el tamaño de una figura, realizar operaciones *drag&drop*, etc. Estos "handles" tienen como objetivo la manipulación de las diferentes figuras a través de una interfase común.

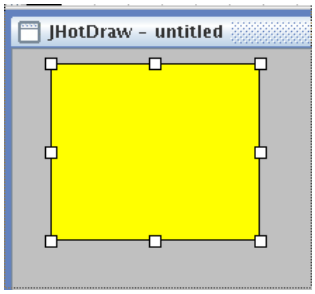


Fig. VI-10. *Handles* asociados a un rectángulo.

Las reglas de asociación para este *concern* no relacionan métodos directamente relacionados a la interfase Handle, sino que identifican la presencia de este *concern* en las clases BoxHandleKit y en la clase RelativeLocator.

Las reglas correspondientes a este *concern* fueron clasificadas por el filtro conceptual, identificando la utilización de los métodos south, north, east y west, los cuales están presentes en clases con responsabilidades diferentes.

La refactorización de este *concern* mediante aspectos no ha sido descripta en ningún trabajo previo. Aunque, debido a que los métodos south, north, east y west son *factory methods* [Gamma et al. 1995], la refactorización mediante aspectos es aplicable y ya ha sido realizada en [Hannemann y Kiczales 2002].

2.3.8 Patrón Mediator

El patrón *Mediator* es utilizado en JHotDraw para coordinar los diferentes objetos que conforman la interfase gráfica de alguna aplicación derivada del *framework*. El protocolo para este patrón está definido en la interfase DrawingEditor, la cual es realizada por dos clases concretas, una para editores basados en *applets* (DrawApplet) y otra para editores basados en Swing (DrawApplication) (Fig. VI-11).

Las reglas correspondientes a este patrón fueron clasificadas por el filtro de consecuentes recurrentes, el cual identificó la utilización consistente del método fireSelectionChanged. Este método es responsable de informar que la selección actual ha cambiado.

Una implementación orientada a aspectos para este patrón es discutida en [Hannemann y Kiczales 2002]. Dicha implementación permite evitar las dependencias cíclicas típicas de la aplicación de este patrón.

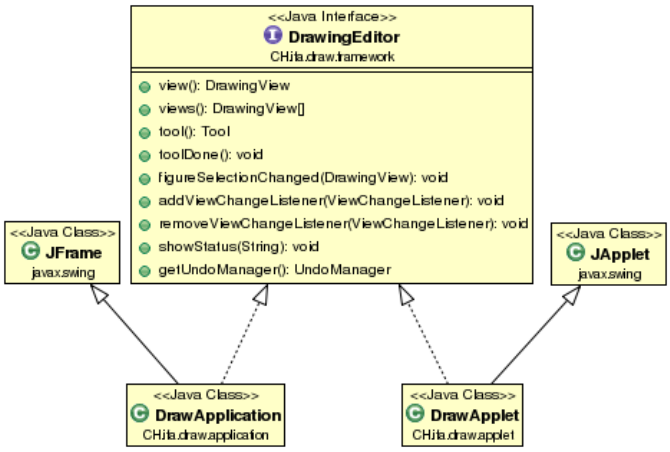


Fig. VI-11. Patrón *Mediator* en JHotDraw.

2.3.9 *Patrón Observer*

El patrón *Observer* es utilizado en JHotDraw para desacoplar la vista de las figuras, permitiendo que las figuras comuniquen cambios en su estado de manera flexible sin que estas conozcan los interesados. Este patrón superpone diferentes funcionalidades *crosscutting* sobre sus participantes, tales como: el mecanismo de notificación, el mecanismo de actualización y la gestión de los objetos observadores. Las reglas de asociación para este patrón identificaron diferentes métodos correspondientes a alguno de los mecanismos *crosscutting* anteriores (Fig. VI-12).

La implementación mediante aspectos de este patrón es tratada en profundidad en [Hannemann y Kiczales 2002] y en [Lesiecki 2005].

```
/**
 * Moves the figure by the given offset.
 */
public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
}
/**
 * Moves the figure by the given offset.
 */
public void moveBy(int dx, int dy) {
        willChange();
        basicMoveBy(dx, dy);
        changed();
}
```

Fig. VI-12. Mecanismo de notificación en JHotDraw.

2.3.10 *Persistencia*

La presencia de reglas de asociación con métodos read, readInt o write permitió una rápida identificación de este *concern* prácticamente sin explorar el código relacionado al mismo.

La característica principal respecto a las reglas que guiaron el descubrimiento de este *concern*, es que las mismas poseen un valor de soporte igual o menor a 0.14. Esto se debe a que durante la fase de análisis dinámico se ejercitaron únicamente dos escenarios (sobre 21) relacionados directamente a este *concern* (abrir y almacenar un documento).

En la Fig. VI-13, se presenta la visualización de la regla AttributeFigure.read ⇒ AbstractFigure.read clasificada como aspecto candidato por el filtro conceptual (debido a que ambas clases implementan la operación read).

Una posible refactorización hacia aspectos para este *concern* es propuesta en [Marin et al. 2007]. Tal refactorización

se basa en el uso del mecanismo de introducción de miembros de AspectJ, de forma de superponer la interfaz Storable sobre aquellas clases que deben ser persistidas.

```
/**
 * Reads the Figure from a StorableInput.
 */
public void read(StorableInput dr) throws IOException {
        super.read(dr);
        String s = dr.readString();
        if (s.toLowerCase().equals("attributes")) {
                fAttributes = new FigureAttributes();
                fAttributes.read(dr);
        }
}
```

Fig. VI-13. Persistencia en JHotDraw.

2.3.11 State

El patrón *State* es utilizado en JHotDraw para modelar el comportamiento de las operaciones en la barra de herramienta de la interfase gráfica, ya que este comportamiento depende del contexto en que se usa dicha operación. La clase StandarDrawingView (Fig. VI-14) es la responsable de gestionar la entrada de datos (los *clicks* del usuario) redirigiendo los eventos generados a la herramienta correspondiente. La interfase Tool (Fig. VI-14) juega el papel de Estado dentro del patrón, mientras que la clase StandarDrawingView juega el rol de Contexto.

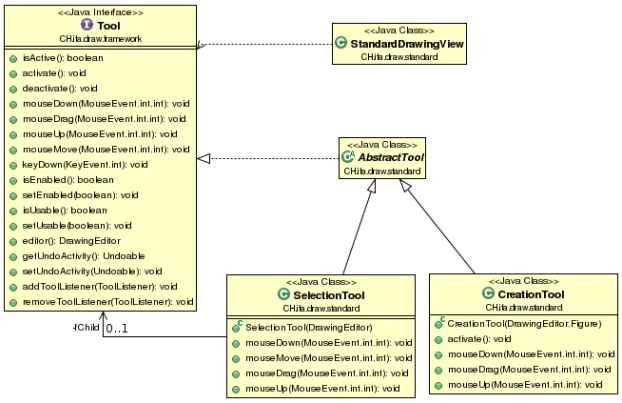


Fig. VI-14. Jerarquía parcial de Tools en JHotDraw.

Cada herramienta puede estar en alguno de los siguientes estados: habilitada, deshabilitada, utilizable, no utilizable, activa, e inactiva. La interfase Tool define operaciones tales como activate o deactivate, utilizadas para realizar las transiciones entre los estados de un objeto de este tipo. Las reglas de asociación para este *concern,* muestran la presencia de los métodos correspondientes a las transiciones dispersos en diferentes clases (ya sean como implementación de la operación o como una llamada al método en sí).

Una implementación orientada a aspectos para este patrón es discutida en [Hannemann y Kiczales 2002].

2.3.12 Undo

Este *crosscutting concern* superpone código sobre diversas clases que definen actividades que pueden ser revertidas en sus acciones. En JHotDraw, la interfase Undoable encapsula la noción de "deshacer" una acción, para lo cual provee una operación undo. Cada clase que implementa una actividad que puede ser "deshecha" define una clase estática anidada que conforma esta interfase. Cada vez que la actividad modifica su estado, también actualiza un campo que mantiene la actividad necesaria (de tipo Undoable) para deshacer los cambios.

Las reglas de asociación presentadas para este *concern* muestran la utilización de métodos para obtener la actividad que deshace los cambios (getUndoActivity) (Fig. VI-15).

La refactorización de este *concern* hacia una solución orientada a aspectos se discute en [Marin 2004]. La misma consiste en un conjunto de pasos:

- Primero, las actividades existentes son extendidas con una asociación al objeto encargado de "deshacerla".

- Segundo, las operaciones existentes son extendidas con funcionalidad para llevar registro del estado anterior, de forma de poder deshacer los cambios.

- Finalmente, se introducen, mediante el mecanismo de introducción de miembros de AspectJ, las diferentes clases anidadas que contienen las actividades encargadas de deshacer los cambios.

```
public void deactivate() {
        getWrappedTool().deactivate();
        Undoable undoActivity = getWrappedTool().getUndoActivity();
        if ((undoActivity != null) && (undoActivity.isUndoable())) {
                editor().getUndoManager().pushUndo(undoActivity);
                editor().getUndoManager().clearRedos();
                // update menus
                editor().figureSelectionChanged(view());
        }
}
```

Fig. VI-15. Código correspondiente al *concern Undo*.

3 Análisis Cuantitativo de los Resultados Obtenidos

En esta sección, se presentan un conjunto de métricas y análisis asociados a éstas obtenidos a partir de las reglas de asociación resultantes del enfoque propuesto. El análisis de los candidatos a partir de métricas permitirá detectar los posibles problemas o inconvenientes en la técnica propuesta, así como sus puntos fuertes o ventajas.

El conjunto final de aspectos candidatos contiene 482 reglas de asociación, por lo que, el primer análisis a realizar es determinar cuántas de estas corresponden a un *crosscutting concern*, cuántas no, y cuántas fueron clasificadas por qué filtro (Tabla VI-4, Fig. VI-16).

Tabla VI-4. Análisis general de los resultados obtenidos.

Candidatos	Falsos positivos (FP)	Confirmados	Filtro conceptual	Filtro de cons. rec.
482	199	283	220	262

Sobre un total de 482 reglas de asociación, 283 de ellas corresponden a candidatos confirmados, por lo que el 58% de las reglas obtenidas son indicadoras de *crosscutting concern* presentes en el código fuente de la aplicación.

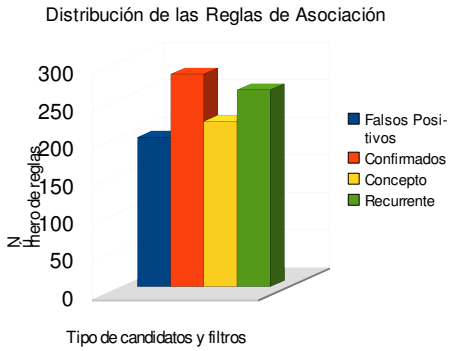


Fig. VI-16. Distribución de las reglas de asociación.

La cantidad de falsos positivos (199 o el 42% sobre el total de candidatos) puede ser considerado como un valor bajo, sobre todo considerando que FINT[7] [Marin et al. 2007] alcanzó un 49% de falsos positivos sobre el mismo sistema analizado.

Por otra parte, la cantidad de candidatos obtenidos por cada tipo de filtro es aproximadamente equivalente, mientras

7 La comparación se realiza con FINT ya que, hasta el momento, es la única técnica de la que se disponen los datos.

que el filtro conceptual clasificó el 45% de los candidatos, el filtro de consecuente recurrente clasificó el 55% restante.

Tabla VI-5. Número de reglas de asociación por *concern* identificado.

Concern	**Número de reglas**
Adapter	17
Command	106
Composite	13
Consistent Behaviour (CB)	14
Contract Enforcement (CE)	17
Decorator	23
Handle	9
Mediator	3
Observer	27
Persistencia	35
State	5
Undo	14

En la Tabla VI-5 y Fig. VI-17, se presenta la distribución de las reglas de asociación resultantes por *concern* identificado. El *concern* con mayor cantidad de reglas de asociación es el correspondiente al patrón *Command* (37%), mientras que el *concern* con menor cantidad de reglas de asociación es el del patrón *Mediator* (1%). La diferencia reside en la importancia que cada patrón tienen dentro del diseño de JHotDraw. Mientras que el patrón *Command* tiene un papel relevante como mecanismo para realizar la arquitectura MVC, el patrón *Mediator* es utilizado para solucionar un problema específico en la interfase de usuario.

En promedio, por cada *concern* identificado hay 23 reglas de asociación (283 reglas confirmadas / 12 *concerns*), aunque, si se elimina el *concern command* de la evaluación el promedio por *concern* sería de 16 reglas de asociación por *concern* descubierto ((283 reglas confirmadas – 106 reglas del *concern command*) / 11 *concerns*).

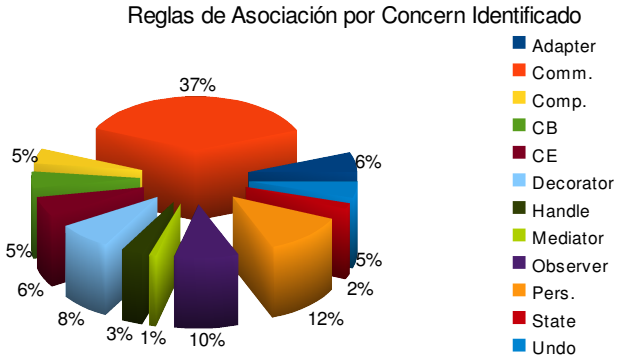


Fig. VI-17. Reglas de asociación por *concern* identificado.

El soporte de las reglas de asociación es uno de los parámetros que deben ser considerados a la hora de analizar la aplicación de la técnica propuesta sobre un caso de estudio. Este conocimiento es de importancia para futuros experimentos, ya que permitirá refinar los parámetros utilizados y disminuir el tiempo necesario para encontrar el valor justo durante la extracción de reglas de asociación.

Tabla VI-6. Candidatos según su valor de soporte.

Rango de soporte	Número de reglas	Falsos positivos (FP)	Confirmados
(0.0, 0.1]	88	53	35
(0.1, 0.5]	209	98	111
(0.5, 0.8]	178	42	136
(0.8, 1.0]	7	6	1

Para realizar este análisis, se dividió el rango de los posibles valores que puede tomar el soporte en cuatro rangos discretos (Tabla VI-6). Por cada rango, se determinó: la cantidad de reglas de asociación que poseen un valor de soporte dentro de ese rango y cuántas de esas reglas son falsos positivos o candidatos confirmados.

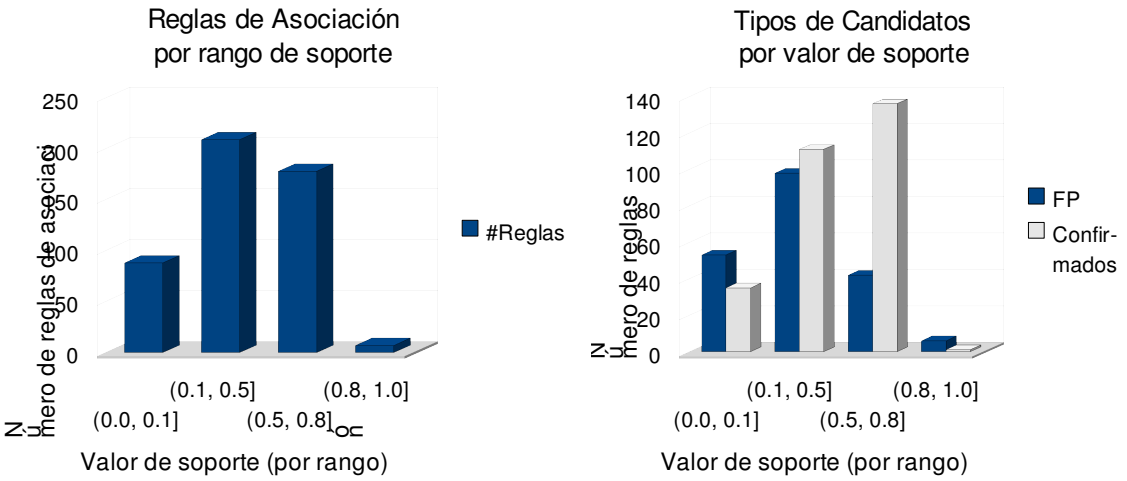


Fig. VI-18. Análisis de las reglas de asociación por valor de soporte.

Los rangos intermedios son los que mayor número de reglas de asociación congregan (Fig. VI-18 izquierda). El rango de valores de soporte que va de 0.8 a 1.0 posee la menor cantidad de reglas de asociación, sólo 7, de las cuales tan sólo 1 regla es un candidato confirmado. Analizando las reglas correspondientes a los falsos positivos para ese rango, se observa que todas ellas fueron descartadas debido a que corresponden a métodos utilitarios (por ejemplo, hasMoreElements de la clase FigureEnumerator), los cuales tienen un alto nivel de reuso pero no por ello son *crosscutting concerns*.

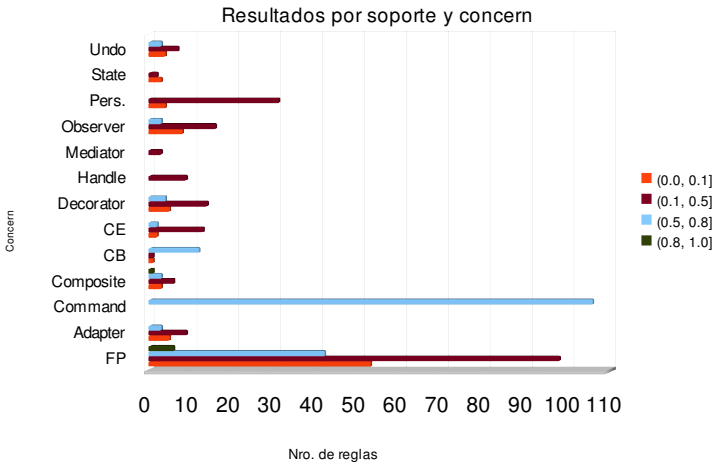


Fig. VI-19. Análisis de las reglas por *concern* y rango de soporte.

El rango de valores que va de 0.0 a 0.1, posee el mayor número de falsos positivos, esto puede significar que los candidatos encontrados no son lo suficientemente válidos estadísticamente. En cambio, el rango (0.5, 0.8] posee el mayor porcentaje de candidatos confirmados (76% aproximadamente) (Fig. VI-18 - derecha).

Como refinamiento del análisis anterior, se analizaron los diferentes *concern* identificados respecto al valor de soporte de las reglas de asociación correspondiente a cada uno. El resultado es el gráfico de la Fig. VI-19.

Se observa que dentro del rango (0.5, 0.8], el cual posee el mayor porcentaje de candidatos confirmados, se encuentran las reglas correspondientes al patrón *Command*. Si se analiza este rango sin considerar las reglas de asociación correspondientes al *concern Command*, el porcentaje de candidatos confirmados se reduce al 42% [(136 candidatos confirmados – 106 reglas del *Command*) / (178 reglas en el rango – 106 reglas del *Command*)].

Por otra parte, las reglas de asociación para los *concerns Undo*, Persistencia, *Observer*, *Mediator*, *Handle*, *Decorator*, *Composite* y *Adapter* poseen valores de soporte únicamente dentro del rango (0.1, 0.5], el cual posee un 53% de candidatos confirmados (111 candidatos sobre 209 reglas en el rango).

Por lo tanto, es muy difícil sacar conclusiones acerca de cuál es el valor óptimo para el parámetro de soporte, aunque deberían evitarse valores muy altos o muy bajos.

Tabla VI-7. Filtros de post-procesamiento y concerns identificados.

Concern	**Concepto**	**Consecuente recurrente**
Falsos positivos (FP)	58	141
Adapter	6	11
Command	105	1
Composite	3	10
Consistent Behaviour (CB)	0	14
Contract Enforcement (CE)	0	17
Decorator	11	12
Handle	7	2
Mediator	0	3
Observer	9	18
Persistencia	20	15
State	1	4
Undo	0	14

En la Tabla VI-7 y en el gráfico de la Fig. VI-20 se presenta el número de reglas de asociación por *concern* identificado según el tipo de filtro de post-procesamiento.

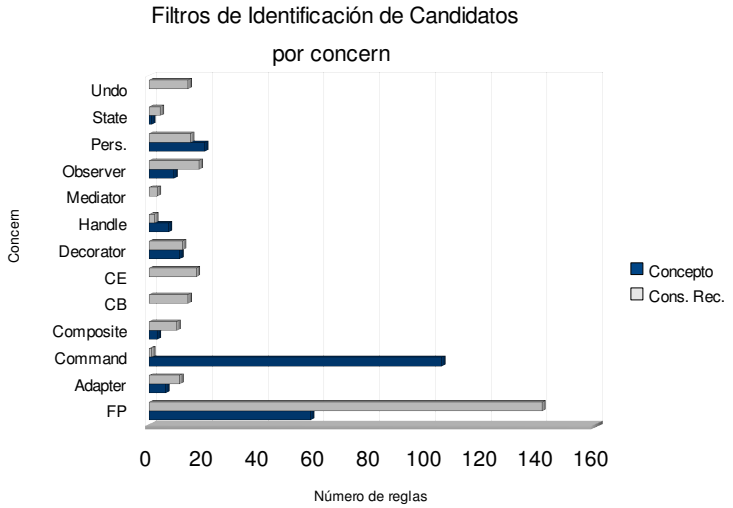


Fig. VI-20. Reglas por *concern* y filtro de post-procesamiento.

Este tipo de análisis, permite evaluar la cantidad de falsos positivos generados por cada filtro y obtener una idea de que tipo de *concern* es más fácil de detectar según el filtro aplicado.

En cuanto a la cantidad de falsos positivos, se observa que el filtro de consecuente recurrentes genera un 53% de falsos positivos (esto es 141 falsos positivos sobre 262 reglas clasificadas). En tanto que el filtro conceptual genera un 26% de falsos positivos. Evidentemente, la precisión del filtro conceptual es mucho mayor que la del filtro de consecuentes recurrentes.

Por otra parte, existen *concerns* que han sido identificados únicamente por el filtro de consecuentes recurrentes: *Undo*, *Mediator*, *Consistent Behaviour* y *Contract Enforcement*. En cambio, *concerns* como *Command* o *Handle* fueron identificados principalmente a través del filtro conceptual.

Por lo tanto, si bien los resultados obtenidos mediante el filtro conceptual son mejores que los obtenidos mediante el filtro de consecuentes recurrentes ambos filtros deben considerarse complementarios.

4 Conclusiones

El análisis de la aplicación JHotDraw versión 5.4b1 a través del enfoque propuesto se realizó desde dos punto de vista diferentes: el cualitativo y el cuantitativo. El análisis cualitativo, fue realizado para justificar la evidencia que dan las reglas de asociación sobre la presencia de *crosscutting concerns* en el código fuente del sistema. El análisis cuantitativo, consistió en la recolección de medidas y métricas con el objetivo de evaluar el desempeño de la técnica en la identificación de *crosscutting cocnerns*.

Del análisis cualitativo, se concluye que la técnica propuesta identificó 11 de los 12 *crosscutting concerns* previamente identificados por otros autores. También, que determinados *concerns* pueden tener mayor influencia dentro del diseño de una aplicación (como el patrón *Command* en JHotDraw) y que esa influencia puede provocar que el análisis dinámico genere mayor cantidad de información respecto a ese *concern*. Además, por cada *concern* descubierto se incluyeron pautas o referencias para su refactorización.

Del análisis cuantitativo, la primera conclusión alcanzada es que la técnica obtuvo un 58% de candidatos confirmados. Este porcentaje, se basa en un juicio subjetivo sobre si una regla de asociación es indicadora de la presencia de un *crosscutting concern* o no. Por lo tanto, no debe ser tomado como algo absoluto o definitorio sino, más bien, como un buen indicador sobre la aptitud de la técnica propuesta para identificar *crosscutting concerns* sobre código legado. Por otra parte, mediante este análisis se pudo concluir que las reglas de asociación con valores muy altos o muy bajos de soporte exhiben el mayor porcentaje de falsos positivos. Respecto a los filtros de post-procesamiento, las medidas utilizadas demostraron que el filtro conceptual posee un número mayor de candidatos confirmados, en cambio, el filtro de consecuente recurrente posee un mayor número de falsos positivos. Pero, analizando los tipos de *concerns* identificados, existen determinados *concerns* que sólo fueron identificados por uno de los dos filtros utilizados. Por lo que, ambos tipos de filtros son complementarios en su aplicación.

Capítulo VII

Conclusiones

A continuación, se presentan las conclusiones alcanzadas luego de desarrollar el enfoque propuesto y un caso de estudio para el mismo. El capítulo comienza con un análisis de los puntos fuertes y débiles del enfoque respecto a las técnicas que lo componen, y un resumen de los resultados alcanzados luego de aplicar la técnica propuesta a un caso de estudio. Luego, se compara el enfoque propuesto con técnicas de aspect mining existentes basadas en análisis dinámico y se presentan los trabajos futuros a realizar.

1 Análisis del Enfoque Propuesto

Aspect mining [Kellens et al. 2007] es la actividad de descubrir *crosscuting concerns* que potencialmente podrían convertirse en aspectos. A partir de allí, estos *concerns* pueden, ser encapsulados en nuevos aspectos del sistema (mediante la utilización de técnicas conocidas como *aspect refactoring* [Kellens et al. 2007]) o podrían documentarse con el objetivo de mejorar la comprensión y documentación del programa.

El enfoque propuesto para *aspect mining*, se basa en la ejercitación de las funcionalidad de alto nivel de un sistema para observar qué métodos son invocados y saber cuáles de ellos participan en un único escenario de ejecución, y cuáles participan en la realización de más de un escenario o función de alto nivel. Mediante la aplicación de técnicas de descubrimiento de reglas de asociación es posible descubrir interesantes asociaciones, correlaciones, patrones frecuentes o estructuras casuales entre un conjunto de métodos pertenecientes a las trazas de ejecución. La aplicación de reglas de asociación sobre las trazas de ejecución generadas durante la fase de análisis dinámico facilitan la identificación de este tipo de métodos transversales. Este tipo de métodos son, por lo tanto, posibles indicadores de la presencia de un *crosscutting concern* ya que son utilizados desde diferentes partes del sistema en tiempo de ejecución.

YAAM (*Yet Another Aspect Miner*) es una herramienta prototípica implementada en Java que permite la extracción de reglas de asociación a partir de trazas y relaciones de ejecución, y su posterior post-procesamiento. A su vez, YAAM brinda soporte para la visualización de las reglas de asociación sobre el código fuente de la aplicación bajo análisis. De esta manera, la herramienta permite automatizar, en gran parte, los pasos tres (extracción de reglas de asociación), cuatro (post-procesamiento de las reglas de asociación) y cinco (análisis de los candidatos) del proceso de *aspect mining* propuesto.

El enfoque propuesto fue analizado sobre JHotDraw 5.4b1 [JHotDraw], un sistema *open-source* que ha sido sujeto de análisis previos mediante técnicas de *aspect mining* por otros autores.

JHotDraw es un *framework* de dibujado en dos dimensiones desarrollado originalmente como un ejercicio para ilustrar el uso correcto de los patrones de diseño orientados a objetos [Gamma et al. 1995]. El mismo esta implementado en Java y posee aproximadamente 18000 líneas no comentadas de código y 2800 métodos. Estas características lo recomiendan como un caso de estudio bien diseñado, un requisito para demostrar las mejoras que se pueden obtener a partir de las técnicas orientadas a aspectos.

El análisis de JHotDraw a través del enfoque propuesto se realizó desde dos punto de vista diferentes: el cualitativo y el cuantitativo. El análisis cualitativo, fue realizado para justificar la evidencia que dan las reglas de asociación sobre la presencia de *crosscutting concerns* en el código fuente del sistema. El análisis cuantitativo, consistió en la recolección de medidas y métricas con el objetivo de evaluar el desempeño de la técnica en la identificación de *crosscutting cocnerns*.

Del análisis cualitativo, se concluye que la técnica propuesta identificó 11 de los 12 *crosscutting concerns* previamente identificados por otros autores, principalmente los descubiertos por [Marin et al. 2007] y [Ceccato et al. 2005]. Además, se observó que algunos *concerns* pueden tener mayor influencia dentro del diseño de una aplicación (como el patrón *Command* en JHotDraw) y que esa influencia puede provocar que el análisis dinámico genere mayor cantidad de información respecto a ese *concern*.

Del análisis cuantitativo, la primera conclusión alcanzada es que la técnica obtuvo un 58% de candidatos confirmados. Por otra parte, mediante este análisis se pudo concluir que las reglas de asociación con valores muy altos o muy bajos de soporte exhiben el mayor porcentaje de falsos positivos. Respecto a los filtros de post-procesamiento, las

medidas utilizadas demostraron que el filtro conceptual posee un número mayor de candidatos confirmados, en cambio, el filtro de consecuente recurrente posee un mayor número de falsos positivos. Pero, analizando los tipos de *concerns* identificados, existen determinados *concerns* que sólo fueron identificados por uno de los dos filtros utilizados. Por lo que, ambos tipos de filtros son complementarios en su aplicación.

Debido a que la técnica propuesta se construye sobre técnicas de análisis dinámico y reglas de asociación, muchas de sus ventajas e inconvenientes son heredados de estos dos tipos de técnicas. En la Tabla VII-1 se muestran las ventajas e inconvenientes heredados y se explica el impacto que poseen sobre el enfoque propuesto.

Tabla VII-1. Ventajas e inconvenientes inherentes al enfoque propuesto.

	Descripción	**Impacto en el enfoque**
Ventajas	El uso de análisis dinámico permite un enfoque orientado al objetivo.	Es posible utilizar la técnica utilizando un paradigma *top-down*. Donde se busca encontrar evidencia sobre la presencia de un *corsscutting concern* previamente conocido (o que se sospecha que está presente).
	El análisis dinámico es mucho más sucinto (respecto a técnicas de análisis estático) para tratar con el polimorfismo.	Los aspectos candidatos (o *seeds*) son mucho más precisos y reflejan exactamente las relaciones y eventos que se dan en tiempo de ejecución.
	Las reglas de asociación generan patrones que son fáciles de entender.	Los aspectos candidatos resultantes son muy expresivos ya que asocian dos métodos según el tipo de filtro que los haya clasificado.
	Existen gran diversidad de algoritmos y técnicas de post-procesamiento relacionados a las reglas de asociación.	El enfoque propuesto posee varios puntos de mejora y variación debido a que las reglas de asociación poseen muchas variantes (como las reglas de asociación generalizadas) y algoritmos asociados.
Inconvenientes	El efecto observador. Inherente al mecanismo de observación utilizado para realizar el análisis dinámico.	Durante la ejercitación del sistema mediante escenarios de ejecución, el desarrollador podría cambiar su comportamiento en caso de que el mecanismo encargado de registrar la información dinámica modifique sustancialmente el tiempo de respuesta de la aplicación bajo análisis.
	Aplicaciones con múltiples hilos de ejecución.	El soporte utilizado para registrar las trazas de ejecución debería considerar la presencia de múltiples hilos de ejecución y llevar una traza por cada hilo.
	Presencia de mecanismos reflexivos.	La presencia de mecanismos de reflexión podrían provocar que la traza de ejecución obtenida no sea un fiel reflejo de lo sucedido durante la prueba del sistema.
	Generación de muchas reglas de asociación, gran parte de las cuales no son interesantes o son redundantes.	La cantidad de aspectos candidatos generados es muy alta, sobre todo comparándose con técnicas derivadas del análisis estático.

2 Trabajos Futuros

A continuación, se detallan trabajos futuros respecto a la técnica propuesta:

- Implementación del prototipo YAAM como un *plugin* de Eclipse [Eclipse]. De esta forma, será posible integrar el reconocimiento de *crosscutting concern* con otros *plugins* para el modelado de *concerns* [Robillard y Murphy 2007] o de exploración del código fuente de una aplicación [JQuery].

- Exploración de nuevos tipos de reglas de asociación, principalmente el uso de reglas de asociación generalizadas [Srikant y Agrawal 1995].

- Creación de nuevos tipos de filtros de post-procesamiento, los cuales podrían utilizar funciones de procesamiento del lenguaje natural [PLN] para comparar dos métodos.

- Mejoras en el soporte para la realización del análisis dinámico. Debería ser tan fácil de ejecutar y gestionar las trazas de ejecución como es hacer *debugging* en un IDE como Eclipse.

Capítulo VIII

Bibliografía

[Agrawal y Srikant 1994] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487-499.

[Akers 2005] R. L. Akers, "Using build process intervention to accommodate dynamic instrumentation of complex systems," Department of Mathematics & Computer Science, Tech. Rep., 2005.

[Amir et al. 1997] A. Amir, R. Feldman, and R. Kashi, "A new and versatile method for association generation," *Inf. Syst.*, vol. 22, no. 6-7, pp. 333-347, 1997.

[Andrews 1998] J. H. Andrews, "Testing using log file analysis: Tools, methods and issues," in *In Proceedings of the 1998 International Conference on Automated Software Engineering (ASE'98*, 1998, pp. 157-166.

[Araújo et al. 2005] J. Araujo, E. Baniassad, P. Clements, A. Moreira, A. Rashid, and B. Tekinerdoğan, "Early aspects: The current landscape," Lancaster University, Tech. Rep., February 2005.

[Ball 1999] Ball, T. (1999). The concept of dynamic analysis. In: ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pages 216–234. Springer-Verlag.

[Baxter et al. 1998] I. D. Baxter, A. Yahin, L. Moura, M. Sant'anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM '98: Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998.

[Biggerstaff et al. 1994] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assigment problem," *Commun. ACM*, vol. 37, no. 5, pp. 72-82, 1994.

[Booch et al. 1999] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2nd Edition) (The Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, May 2005.

[Borgelt y Kruse 2002] C. Borgelt and R. Kruse, "Induction of association rules: Apriori implementation," in *15th Conference on Computational Statistics (COMPSTAT 2002)*. Physica Verlag, Heidelberg, Germany, 2002.

[Brant et al. 1998] J. Brant, B. Foote, R. E. Johnson, and D. Roberts, "Wrappers to the rescue," in *In Proceedings ECOOP '98, volume 1445 of LNCS*, 1998, pp. 396-417.

[Breu y Krinke 2004] S. Breu and J. Krinke, "Aspect mining using event traces," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp.

310-315.

[Bruntink 2004] M. Bruntink, "Aspect mining using clone class metrics," in *Proceedings of the 2004 Workshop on Aspect Reverse Engineering (co-located with WCRE 2004)*, November 2004.

[Bruntink et al. 2005] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 804-818, October 2005.

[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[Chikofsky y Cross 1990] E. J. Chikofsky, E. J. Chikofsky, J. H. Cross, and J. H. Cross, "Reverse engineering and design recovery: a taxonomy reverse engineering and design recovery: a taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13-17, 1990.

[Chitchyan et al. 2005] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. P. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson, "Survey of analysis and design approaches," AOSD-Europe, Tech. Rep., May 2005.

[Demeyer et al. 2002] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object Oriented Reengineering Patterns (The Morgan Kaufmann Series in Software Engineering and Programming)*. Morgan Kaufmann, July 2002.

[Dijkstra 1982] E. W. Dijkstra, "Ewd 447: On the role of scientific thought," *Selected Writings on Computing: A Personal Perspective*, pp. 60-66, 1982.

[Dynamo] http://star.itc.it/dynamo/

[Eclipse] www.eclipse.org

[Eisenbarth et al. 2003] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210-224, March 2003.

[Ferenc et al. 2005] R. Ferenc, A. Beszedes, L. Fulop, y J. Lele, "Design pattern mining enhanced by machine learning," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 295-304, 2005.

[FINT] http://swerl.tudelft.nl/bin/view/AMR/FINT. Versión 0,6

[Fowler 2004] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern". 2004. *Disponible en: http://martinfowler.com/articles/injection.html*

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley Professional, January 1994.

[Ganter y Wille 1997] B. Ganter and R. Wille, "Applied lattice theory: Formal concept analysis," in *In General Lattice Theory, G. Grätzer editor, Birkhäuser*, 1997.

[Gong et al. 2006] M. Gong, V. Muthusamy, H. Jacobsen. AspeCt-oriented C Tutorial. University of Toronto. September 2006. http://research.msrg.utoronto.ca/ACC/Tutorial

[Grappa http://www.research.att.com/~john/Grappa/

[Greevy y Ducasse 2005] O. Greevy and S. Ducasse, "Correlating features and code using a compact two-

sided trace analysis approach," in *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 314-323.

[Gybels y Kellens 2005] K. Gybels and A. Kellens, "Experiences with identifying aspects in smalltalk using 'unique methods'," in *Linking Aspect Technology and Evolution (LATE), collocated with Aspect-Oriented Software Development*, 2005.

[Hannemann y Kiczales 2002] J. Hannemann and G. Kiczales, "Design pattern implementation in java and aspectj," in *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. New York, NY, USA: ACM Press, 2002, pp. 161-173.

[He y Bai 2006] L. He and H. Bai, "Aspect mining using clustering and association rule method," *International Journal of Computer Science and Network Security*, vol. 6, no. 2, pp. 247-251.

[Hetzler et al. 1998] B. Hetzler, W. Harris, S. Havre, and P. Whitney, "Visualizing the full spectrum of document relationships," in *Fifth International Society for Knowledge Organization (ISKO) Conference*, 1998.

[HSQL] HSQL Database engine. http://hsqldb.org/

[Jain et al. 1999] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264-323, September 1999.

[Jain y Dubes 1988] A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data (Prentice Hall Advanced Reference Series : Computer Science)*. Prentice Hall College Div, March 1988.

[JBoss] JBoss Aplication Server. http://www.jboss.org/jbossas/

[JBossAOP] JBoss AOP - User Guide. The Case For Aspects. http://www.jboss.org/jbossaop/

[JHotDraw] http://www.jhotdraw.org/

[JQuery] JQuery, a query based browser. http://jquery.cs.ubc.ca/

[Kamiya et al. 2002] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654-670, July 2002.

[Kellens et al. 2007] A. Kellens, K. Mens, and P. Tonella, "A survey of automated code-level aspect mining techniques," in *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, November 2007, vol. 4640/2007, pp. 143-162.

[Kiczales et al. 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220-242.

[Kiczales et al. 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP*, ser. Lecture Notes in Computer Science, J. L. Knudsen and J. L. Knudsen, Eds., vol. 2072. Springer, 2001, pp. 327-353.

[Klementinen et al. 1994] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and I. A. Verkamo, "Finding interesting rules from large sets of discovered association rules," in *CIKM '94: Proceedings of the third international conference on Information and knowledge management*. New York, NY, USA: ACM, 1994, pp. 401-407.

[Komondoor y Horwitz 2001] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *In Proceedings of the 8th International Symposium on Static Analysis*, vol. 2126, 2001, pp. 40-56.

[Koschke y Quante 2005] R. Koschke and J. Quante, "On dynamic feature location," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM Press, 2005, pp. 86-95.

[Krinke y Softwaresysteme 2001] J. Krinke and L. Softwaresysteme, "Identifying similar code with program dependence graphs," in *In Proc. Eigth Working Conference on Reverse Engineering*, 2001, pp. 301-309.

[Laddad 2003] R. Laddad, *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, July 2003.

[Lesiecki 2005] N. Lesiecki, "Enhance design patterns with aspectj, part 1 & 2," May 2005. http://www.ibm.com/developerworks/library/j-aopwork5/

[Lientz y Swanson 1980] B. P. Lientz and B. E. Swanson, *Software Maintenance Management*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1980.

[Maes y Nardi 1988] P. Maes and D. Nardi, Eds., *Meta-Level Architectures and Reflection*. New York, NY, USA: Elsevier Science Inc., 1988.

[Marin 2004] M. Marin, "Refactoring jhotdraw's undo concern to aspectj," in *Proceedings Workshop on Aspect Reverse Engineering (WARE) at WCRE*, 2004.

[Marin et al. 2007] 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.

[Moldovan y Serban 2006a] G. S. Moldovan and G. Serban, "Aspect mining using a vector-space model based clustering approach," in *In: Proceedings of Linking Aspect Technology and Evolution (LATE) Workshop, 2006*.

[Moldovan y Serban 2006b] G. Serban and G. S. Moldovan, "A comparison of clustering techniques in aspect mining," *INFORMATICA*, vol. L1, no. 1.

[Morris y Hirst 1991] J. Morris and G. Hirst, "Lexical cohesion computed by thesaural relations as an indicator of the structure of text," *Comput. Linguist.*, vol. 17, no. 1, pp. 21-48, March 1991.

[Nosek y Palvia 1990] J. T. Nosek and P. Palvia, "Software maintenance management: changes in the last decade," *Journal of Software Maintenance*, vol. 2, no. 3, pp. 157-174, 1990.

[Opdyke 1992] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Urbana-Champaign, IL, USA, 1992.

[OpenNLP] http://opennlp.sourceforge.net/

[Page et al. 1998] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.

[Parnas 1972] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

[PetStore] http://developer.java.sun.com/developer/releases/petstore/

[Piatetsky-Shapiro 1991] G. Piatetsky-Shapiro, *Discovery, Analysis, and Presentation of Strong Rules*. Cambridge, MA: AAAI/MIT Press, 1991.

[PLN] Wikipedia: http://en.wikipedia.org/wiki/Natural_language_processing

[Prism] Prism Query Language. http:\\www.eecg.utoronto.ca\~zhang\pql

[Rich y Wills 1990] Charles Rich, Linda M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," IEEE Software, vol. 7, no. 1, pp. 82-89, Jan/Feb., 1990.

[Robillard y Murphy 2007] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 1, February 2007.

[Rysselberghe y Demeyer 2003] F. Van Rysselberghe and S. Demeyer, "Evaluating clone detection techniques from a refactoring perspective," in *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 336-339.

[Sant' Anna et al. 2003] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, and A. v. von Staa, "On the reuse and maintenance of aspect-oriented software: An assessment framework," in *Proceedings XVII Brazilian Symposium on Software Engineering*, 2003.

[Schult et al. 2003] W. Schult, P. Troeger, and A. Polze, "Loom.net - an aspect weaving tool," in *Workshop on Aspect-Oriented Programming, ECOOP*, Darmstadt, 2003.

[Shah et al. 1999] D. Shah, L. V. S. Lakshmanan, K. Ramamritham, and S. Sudarshan, "Interestingness and pruning of mined patterns," in *ACMSIGMOD Workshop on Research Issues in Data Mining*, 1999.

[Shepherd et al. 2004] D. Shepherd, E. Gibson, and L. L. Pollock, "Design and evaluation of an automated aspect mining tool," in *Software Engineering Research and Practice*, H. R. Arabnia, H. Reza, H. R. Arabnia, and H. Reza, Eds. CSREA Press, 2004, pp. 601-607.

[Shepherd et al. 2005] D. Shepherd, L. L. Pollock, and T. Tourwé, "Using language clues to discover crosscutting concerns," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1-6, 2005.

[Shepherd et al. 2006] D. Shepherd, L. Pollock, and K. Vijay-Shanker, "Towards supporting on-demand virtual remodularization using program graphs," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2006, pp. 3-14.

[Shepherd et al. 2007] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2007, pp. 212-224.

[Shepherd y Pollock 2005] D. Shepherd and L. Pollock, "Interfaces, aspects and views," in *Workshop on Linking Aspect Technology and Evolution (LATE 2005), co-located with International Conference on Aspect Oriented Software Development (AOSD 2005)*, March 2005.

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

[SpringAOP] Spring Framework. “Chapter 6. Spring AOP: Aspect Oriented Programming

with Spring". http://static.springframework.org/spring/docs/2.5.x/reference/aop.html

[Srikant y Agrawal 1995] R. Srikant and R. Agrawal, "Mining generalized association rules," in *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 407-419.

[Sutton y Rouvellou 2002] S. M. Sutton and I. Rouvellou, "Modeling of software concerns in cosmos," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2002, pp. 127-133.

[Tarr et al. 1999] P. Tarr, H. Ossher, W. Harrison, and Jr, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE '99: Proceedings of the 21st international conference on Software engineering*. New York, NY, USA: ACM, 1999, pp. 107-119.

[Tomcat] http://tomcat.apache.org/

[Tonella y Ceccato 2004] P. Tonella and M. Ceccato, "Aspect mining through the formal concept analysis of execution traces," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 112-121.

[ToscanaJ] http://toscanaj.sourceforge.net/

[Tourwé y Mens 2004] T. Tourwé and K. Mens, "Mining aspectual views using formal concept analysis," in *In Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004*, 2004, pp. 97-106.

[van Deursen et al. 2000] A. Van Deursen, A. Quilici, and S. Woods, "Program plan recognition for year 2000 tools," in *In Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*, 1997, pp. 124-133.

[Visser 2001] E. Visser, "A survey of strategies in program transformation systems," *Electronic Notes in Theoretical Computer Science*, vol. 57, 2001.

[Wilde y Huit 1992] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 12, pp. 1038-1044, 1992.

[Wilde y Scully 1995] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49-62, 1995.

[Wills 1990] L. M. Wills, "Automated program recognition: a feasibility demonstration," *Artif. Intell.*, vol. 45, no. 1-2, pp. 113-171, 1990.

[Wuyts 2001] R. Wuyts, "A logic meta programming approach to support the co-evolution of object-oriented design and implementation," Ph.D. dissertation, Vrije Universiteit Brussel, 2001.

[Xie et al. 2006] X. Xie, D. Poshyvanyk, and A. Marcus, "3d visualization for concept location in source code," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 839-842.

[Zaidman 2006] A. Zaidman, "Scalability solutions for program comprehension through dynamic analysis," Ph.D. dissertation, September 2006.

[Zhang y Jacobsen 2007] C. Zhang and H.-A. Jacobsen, "Efficiently mining crosscutting concerns through random walks," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2007, pp. 226-238.

[Zimmermann 2001] H.-J. Zimmermann, *Fuzzy Set Theory and its Applications*, 4th ed. Springer, October 2001.

Anexo I

Dynamic Analysis and Association Rules for Aspects Identification

Anexo II

Aspect Mining Mediante Análisis Dinámico y Reglas de Asociación

Aspect Mining Mediante Análisis Dinámico y Reglas de Asociación

Esteban S. Abait

Facultad de Ciencias Exactas – Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA)
Campus Universitario (CP 7000) – Tandil – Bs. As. – Argentina
eabait@alumnos.exa.unicen.edu.ar

Resumen. La presencia de crosscutting concerns en los sistemas orientados a objetos actuales es uno de los mayores impedimentos para la correcta comprensión, modularización y evolución del software. Aspect mining es la tarea de identificar crosscutting concerns presentes en sistemas de software legados. El presente trabajo, propone un proceso semiautomático de cuatro pasos para aspect mining basado en análisis dinámico y reglas de asociación, donde las trazas de ejecución de un sistema son analizadas en búsqueda de síntomas que revelen la presencia de crosscutting concerns. La técnica propuesta fue evaluada en dos sistemas existentes y se determinó que la misma fue capaz de descubrir los crosscutting concerns previamente identificados en estos sistemas por los enfoques existentes.

Palabras clave: Programación orientada a aspectos, crosscutting concerns, aspect mining, análisis dinámico, reglas de asociación.

1 Introducción

La Programación Orientada a Aspectos (POA) [1] es un nuevo paradigma que permite alcanzar una mejor separación de concerns en sistemas de software. Algunos de estos concerns atraviesan los módulos correspondientes a la descomposición principal de la aplicación y son denominados crosscutting concerns [26]. Como ejemplos de dichos concerns se pueden incluir a los mecanismos de sincronización, la persistencia o el logging, entre otros. El código relacionado a estos concerns puede exhibir dos tipos de síntomas de mala modularización: código scattering (o disperso) y código tangling (o entremezclado) [2]. El código scattering sucede cuando el código correspondiente al mismo concern no se encuentra localizado en una única unidad del sistema, en cambio, el código tangling ocurre cuando en una misma unidad existe código correspondiente a diferentes concerns. La POA introduce una nueva unidad de modularización, denominada aspecto, la cual es utilizada para encapsular crosscutting concerns.

Debido a que los sistemas actuales se encuentran codificados utilizando lenguajes orientados a objetos, uno de los principales objetivos es desarrollar mecanismos o técnicas que permitan migrar estos sistemas orientados a objetos a sistemas orientados a aspectos [3]. La tarea de identificar los crosscutting concerns adecuados para una implementación orientada a aspectos se denomina *aspect mining*. Luego, *aspect refactoring* es la actividad que transforma esos aspectos potenciales en aspectos reales del sistema de software.

El presente trabajo se enfoca en el desarrollo de una técnica de aspect mining, la cual se basa en el análisis de las trazas de ejecución de un sistema mediante la utilización de algoritmos de reglas de asociación. Las reglas generadas permiten identificar, tanto síntomas de código scattering como síntomas de código tangling y ayudan a los desarrolladores a encontrar aspectos potenciales en su código base.

El enfoque de aspect mining propuesto es un proceso de cuatro pasos, los primeros dos pasos corresponden a la fase de análisis dinámico del sistema, y los dos restantes corresponden a la generación y post-procesamiento de las reglas de asociación. El análisis dinámico del sistema (sección 3.1) es utilizado para obtener los datos que servirán de entrada al algoritmo de reglas de asociación (sección 3.2). Luego, el paso de identificación de aspectos consiste en un conjunto de filtros que clasifican las reglas obtenidas como posibles indicadoras de tangling o scattering o ambos. Los dos últimos pasos del proceso están soportados por la herramienta YAAM (Yet Another Aspect Miner) (sección 4). YAAM fue aplicada a diferentes aplicaciones orientadas a objetos, tales como una implementación del patrón de diseño Observer (sección 5) y JHotDraw (sección 6) el benchmark de facto para técnicas de aspect mining [4]. Finalmente, se presentan los trabajos relacionados en el área y las conclusiones alcanzadas (secciones 7 y 8).

2 Evolución del Software y Aspect Mining

El mantenimiento, reestructuración, y evolución de los sistemas de software se ha convertido en una cuestión de vital importancia. La ley de entropía del software dictamina que la mayoría de los sistemas pasado un tiempo tienden a decaer gradualmente en su calidad, a menos que estos sean mantenidos y adaptados a los requerimientos cambiantes [24]. En este sentido, Lehman [28] estableció un conjunto de leyes (más bien hipótesis) concernientes a los cambios de los sistemas, las siguientes son las más importantes:

- Ley del cambio continuado: Un programa que se usa en un entorno real necesariamente debe cambiar o se volverá progresivamente menos útil en ese entorno.
- Ley de la complejidad creciente: A medida que un programa en evolución cambia, su estructura tiende a ser cada vez más compleja. Se deben dedicar recursos extras para preservar y simplificar la estructura.

Por lo tanto, el tiempo de vida de un sistema de software puede ser extendido manteniéndolo o reestructurándolo. Pero, un sistema legado no puede ser ni reemplazado ni actualizado excepto a un alto costo. Por lo que el objetivo de la

reestructuración es reducir la complejidad del sistema legado lo suficiente como para poder ser usado y adaptado a un costo razonable [27].

Dentro de la investigación de trasformación de programas, se distinguen dos enfoques de reestructuración diferentes [25]. El término *rephrasing* es utilizado para referirse a técnicas que mejoran la estructura del software sin cambiar el lenguaje de implementación. El ejemplo típico es software refactoring [29], una técnica que intenta mejorar la estructura interna de un programa sin cambiar su comportamiento externo. El término *translation* se refiere a técnicas que reestructuran el software a través de lenguajes de programación. Un ejemplo típico es la migración de código legado a un nuevo paradigma que permita mejoras en términos de calidad interna del mismo.

Uno de los problemas esenciales en el desarrollo del software es la "tiranía de la descomposición dominante" [2], no importa cuan bien un sistema de software se descomponga en unidades modulares, siempre existirán concerns que cortan transversalmente la descomposición elegida. El código de esos crosscutting concerns estará necesariamente presente sobre diferentes módulos, lo que tiene un impacto negativo sobre la calidad del software en términos de comprensión, adaptación y evolución.

La POA ha sido propuesta como una solución a este problema [30]. De manera de capturar dichos crosscutting concerns de forma localizada, un nuevo mecanismo de abstracción (llamado aspecto) es agregado a los lenguajes de programación existentes (por ejemplo, AspectJ [26] para Java).

Para que la POA sea verdaderamente exitosa, es necesario migrar los sistemas de software existentes hacia su equivalente orientado a aspectos y reestructurarlos de manera continua. Sin embargo, aplicar manualmente técnicas orientadas a aspectos a un sistema legado es un proceso difícil y tendiente al error [3]. Debido al gran tamaño de estos sistemas, la complejidad de su implementación, la falta de documentación y conocimiento sobre el sistema, es que existe la necesidad de herramientas y técnicas que ayuden a los desarrolladores a localizar y documentar los crosscutting concerns presentes en esos sistemas.

La tarea de identificar los crosscutting concerns adecuados para una implementación orientada a aspectos se denomina *aspect mining* [3]. Varios trabajos han sido propuestos con el objetivo de proveer mecanismos automáticos o semiautomáticos para identificar crosscutting concerns en sistemas existentes. Sin embargo, aún carecen de precisión y confiabilidad, o los resultados arrojados por los mismos son difíciles de interpretar por parte de los desarrolladores.

En este trabajo, se propone una técnica que combina el análisis dinámico de sistemas con reglas de asociación. La principal ventaja de la misma reside en la cantidad de información que es capaz de proveer al desarrollador, ya que cada aspecto candidato es una regla que relaciona dos o más métodos del sistema analizado y que además se encuentra clasificada acorde al síntoma que representa (tangling y scattering).

3 Análisis Dinámico y Reglas de Asociación

Por medio de las reglas de asociación es posible obtener las asociaciones más relevantes existentes en las trazas de ejecución obtenidas utilizando análisis dinámico. Este tipo de asociaciones otorgan a los desarrolladores valiosa información sobre el comportamiento dinámico del sistema y permiten la identificación de síntomas de tangling y scattering.

A continuación, se introducen los conceptos básicos de análisis dinámico y reglas de asociación.

3.1 Análisis Dinámico

La idea básica detrás de los algoritmos de análisis dinámico es la de observar el comportamiento de los sistemas de software en tiempo de ejecución para extraer información sobre la ejecución de los mismos [5].

Nuestro enfoque utiliza dos tipos de información: trazas de ejecución y relaciones de ejecución. Las trazas y relaciones de ejecución son obtenidas corriendo el programa bajo un conjunto de escenarios, donde cada escenario es similar a una instancia de un caso de uso [6]. La traza de ejecución de un programa está compuesta por la secuencia de métodos invocados durante la ejecución del programa, y las relaciones de ejecución registran qué métodos son invocados por qué otros métodos.

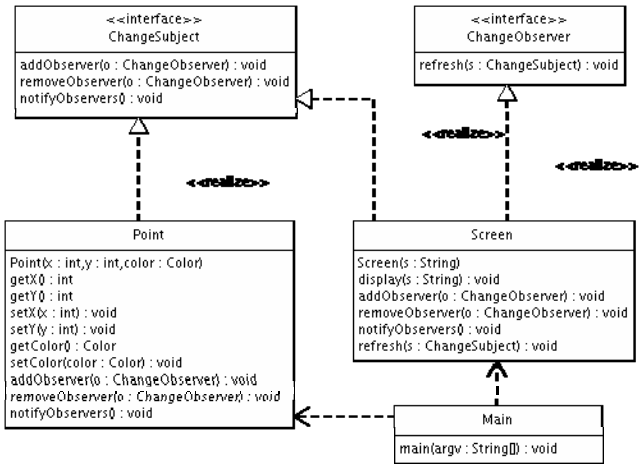


Fig. 1. Diagrama de clases de UML para el programa analizado.

En la Fig. 1, se muestra el diagrama de clases correspondiente a una implementación del patrón de diseño Observer [10]. El objetivo del patrón Observer es la de "definir una dependencia uno-a-muchos entre objetos de forma que cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente" [7]. Esta implementación muestra una instancia del patrón donde la clase Point participa bajo el rol de observable (o sujeto) y la clase Display juega los roles de observable y observador (de Point y de sí misma).

La ejecución de esta aplicación bajo el escenario "un punto cambia su color" genera la traza y las relaciones de ejecución mostradas en la Fig. 2. Entonces, la traza contiene la secuencia de métodos invocados (addObserver de la clase Point luego addObserver de la clase Screen luego setColor de la clase Point y así sucesivamente) y las relaciones de ejecución, son representadas como una tabla de dos columnas, la primera mostrando el método invocador (setColor, por ejemplo) y la segunda mostrando al método invocado (notifyObservers, por ejemplo).

Traza del escenario "cambiar color"

```
void observer.Point.addObserver(ChangeObserver);
void observer.Screen.addObserver(ChangeObserver);
void observer.Point.setColor(Color);void observer.Point.notifyObservers();
void observer.Screen.refresh(ChangeSubject);void observer.Screen.display(String);
void observer.Screen.notifyObservers();
```

Relaciones de ejecución para el escenario "cambiar color"

Invocador	Invocado
void observer.Point.setColor(Color)	void observer.Point.notifyObservers()
void observer.Point.notifyObservers()	void observer.Screen.refresh(ChangeSubject)
...	...
void observer.Screen.refresh(ChangeSubject)	void observer.Screen.display(String)
void observer.Screen.display(String)	void observer.Screen.notifyObservers()

Fig. 2. Traza (*arriba*) y relaciones de ejecución (*abajo*) para el sistema analizado.

Dado que una traza de ejecución se compone de los métodos llamados durante una ejecución particular del sistema, diferentes ejecuciones del mismo programa deberían generar diferentes trazas, ya que diferentes partes del mismo son ejercitadas y corresponden a diferentes casos de uso o escenarios. Por lo tanto, muchas veces la misma clase podría colaborar en la realización de múltiples casos de uso, por lo que un método perteneciente a una clase podría estar presente en diferentes trazas obtenidas para diferentes escenarios. En base a esta observación, es posible identificar los patrones recurrentes presentes en las trazas aplicando reglas de asociación a las mismas de manera de encontrar las asociaciones más relevantes entre los métodos ejecutados. Este tipo de reglas de asociación podrían revelar las asociaciones más frecuentes entre métodos permitiendo descubrir potenciales crosscutting concerns.

3.2 Conceptos de Reglas de Asociación

La búsqueda de reglas de asociación [8] permite obtener asociaciones o correlaciones interesantes entre un conjunto de ítems en un dominio dado. Por lo tanto, la aplicación de un algoritmo de reglas de asociación sobre un conjunto de trazas resultará en el descubrimiento de interesantes patrones de ejecución entre métodos.

Sea $I = \{i_1, i_2, \ldots, i_m\}$ un conjunto de ítems y $D = \{T_1, T_2, \ldots, T_n\}$ un conjunto de transacciones, donde cada transacción T es un conjunto de ítems tal que $T \subseteq I$. Una regla de asociación es una implicación de la forma $A \Rightarrow B$, donde $A \subset I$, $B \subset I$ y $A \cap$

$B = \varnothing$ [8]. A su vez, A se denomina el antecedente de la regla y B se denomina su consecuente. La regla $A \Rightarrow B$ es válida en el conjunto de transacciones D con soporte s si s es el porcentaje de transacciones en D que contienen $A \cup B$. La regla $A \Rightarrow B$ posee confianza c en el conjunto de transacciones D si el porcentaje c es el porcentaje de transacciones en D que contienen tanto a A como a B. Esto es, soporte($A \Rightarrow B$) = $P(A \cup B)$ y confianza($A \Rightarrow B$) = $P(A \mid B)$. Aquellas reglas que satisfacen un mínimo valor de soporte (*min_supp*) y un mínimo valor de confianza (*min_conf*) se denominan *fuertes* [9], y son la salida de los algoritmos de reglas de asociación.

En el presente trabajo, para obtener las reglas de asociación a partir de un conjunto de trazas se utilizó el algoritmo Apriori [8]. Este algoritmo consiste en dos pasos bien definidos: primero encuentra todos los itemsets (conjunto de ítems que cumplen con el mínimo valor de soporte *min_supp*), y luego, a partir de estos itemsets genera reglas de asociación cuya confianza supera el mínimo valor de confianza (*min_conf*).

En la Fig. 3 se observa un ejemplo de obtención de reglas de asociación a partir de un conjunto de ítems $I = \{m_1, m_2, m_3, m_4\}$ y un conjunto de transacciones $D = \{T_1, T_2, T_3, T_4\}$. Además, se fijó *min_supp* = 0.5 y *min_conf* = 0.8 de forma que todas las reglas deben ser válidas en al menos dos transacciones.

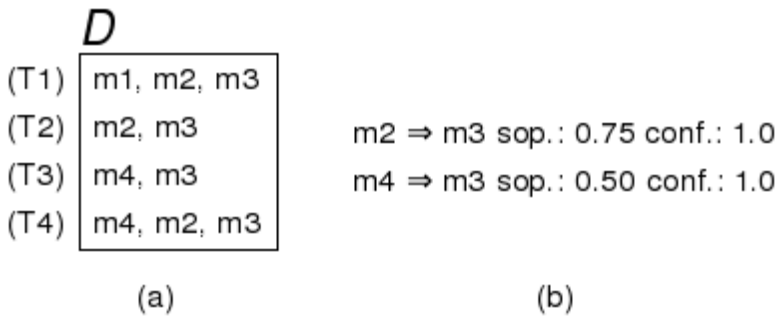


Fig. 3. Base de datos con cuatro transacciones (*izq.*) y reglas de asociación obtenidas (*der.*).

Del ejemplo anterior se observa que la primera de las reglas obtenidas posee un valor de soporte de 0.75 ya que los ítems m2 y m3 aparecen en tres de las cuatro transacciones. Además, su valor de confianza es 1.0 debido a que cada vez que el ítem m2 aparece en una transacción también lo hace m3. El mismo análisis puede hacerse para la segunda regla obtenida. El resto de las reglas de asociación fueron descartadas ya sea porque poseen un valor de soporte menor a 0.5 o un valor de confianza menor a 0.8.

4 Identificando Aspectos como Reglas de Asociación

A continuación, se introducirá la técnica de aspect mining propuesta para luego presentar YAAM (Yet Another Aspect Miner) una herramienta que automatiza la

extracción y procesamiento de las reglas de asociación a partir de la información obtenida mediante análisis dinámico.

Si cada traza del sistema bajo análisis es considerada como una transacción *T* y los métodos contenidos en todas las trazas como el conjunto de ítems *I*, es posible obtener una base de datos *D* a partir de la cual generar un conjunto de reglas de asociación. Por ejemplo, si fuese posible generar dicho conjunto *D* para el ejemplo mostrado en la Fig. 1, las reglas resultantes tendrían la siguiente forma:

Point.notifyObservers ⇒ Screen.refresh (soporte: 1.0 y confianza: 1.0).

Donde, el valor de soporte de la regla indica el número de trazas (transacciones) en las cuales la asociación está presente, debido a que cada traza corresponde a un escenario la misma debe estar presente como mínimo en dos transacciones para que la regla sea considerada como indicadora de un crosscutting concern. En tanto, el valor de confianza de la regla indica la estabilidad de la relación entre los métodos del antecedente y del consecuente.

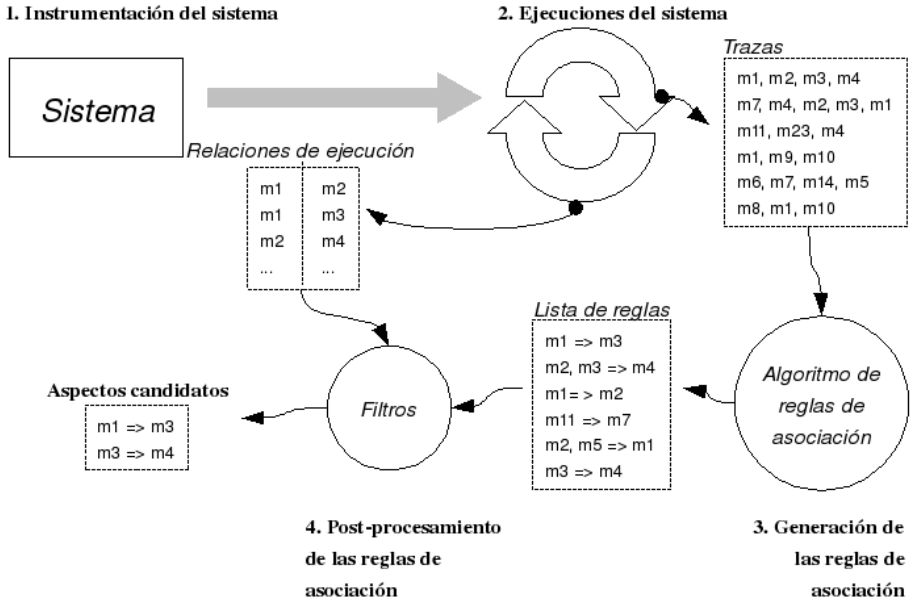


Fig. 4. Workflow de la técnica de aspect mining propuesta.

En la Fig. 4, los principales pasos de nuestro proceso de aspect mining son mostrados. El primer (Instrumentación del sistema) y segundo (Ejecuciones del sistema) pasos corresponden al análisis dinámico del sistema, es decir se obtienen las trazas y relaciones de ejecución. El tercer paso (Generación de las reglas de asociación), toma como entrada el conjunto de trazas obtenidas del paso anterior y aplica un algoritmo de reglas de asociación para encontrar relaciones entre los métodos de las trazas. El cuarto paso (Post-procesamiento de las reglas de asociación) toma como entrada las reglas obtenidas y las relaciones de ejecución y clasifica las reglas como indicadoras de scattering, indicadoras de tangling o indicadoras de

scattering y tangling, las reglas que no pueden ser clasificadas son descartadas. A su vez, durante el cuarto paso se filtran reglas redundantes (por ejemplo, $A \Rightarrow B$ y $B \Rightarrow A$ con mismos valores de soporte y confianza) así como también reglas que incluyan métodos utilitarios (como 'main' o 'run').

Observando la regla Screen.addObserver ⇒ Point.addObserver, se podría afirmar que la misma es indicadora de scattering, ya que muestra cómo dos clases (Screen y Point) definen el mismo método (addObserver), por lo tanto el mismo concern (mapeo sujeto-observador) está presente en dos clases. En general, para clasificar una regla $A \Rightarrow B$ como indicadora de scattering, A y B deben ser de tamaño uno y la signatura del método contenido en A debe ser igual a la signatura del método contenido en B. Luego, el filtro utilizado para clasificar las reglas generadas como indicadoras de scattering se define en base a las condiciones anteriores.

Por otra parte, la regla Point.notifyObservers ⇒ Screen.refresh indica que cada vez que el método notifyObservers de la clase Point es llamado también lo es el método refresh de la clase Screen. Esta regla podría mostrar un síntoma de tangling si la misma fuese verdadera en dos o más trazas y si los métodos del antecedente (notifyObserver en este caso) representaran funcionalidad diferente a la que representa el método del consecuente (refresh). En general, para clasificar una regla $A \Rightarrow B$ como indicadora de tangling, B debe ser de tamaño uno y los métodos del antecedente (A) deben existir en una relación de ejecución con el método contenido en el consecuente (B). La última condición es necesaria para evitar reglas surgidas por pura combinatorias de los métodos (ítems) dados como entrada al algoritmo de reglas de asociación; registrar en una tabla las relaciones de ejecución entre los métodos permite el filtrado de ese tipo de reglas espurias. Luego, el filtro utilizado para clasificar las reglas generadas como indicadoras de tangling se define en base a las condiciones anteriores.

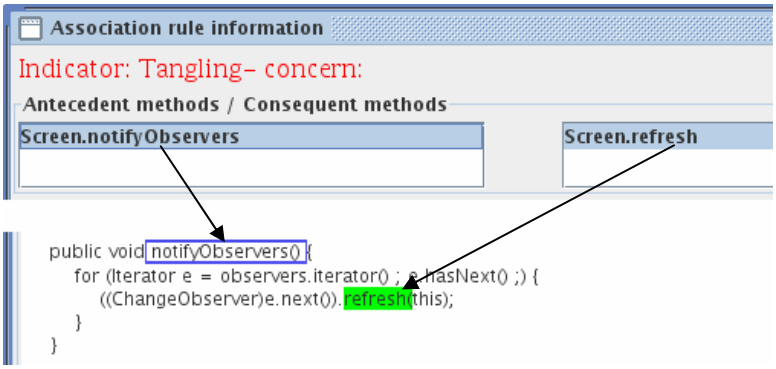


Fig. 5. Visualización en YAAM de la regla Point.notifyObservers ⇒ Screen.refresh.

Para ambos filtros cabe la misma aclaración, es el desarrollador quien debe determinar finalmente si la regla representa o no un síntoma de tangling o scattering. Los filtros sólo señalan aquellas reglas con mayor posibilidad de representar estos síntomas, pero no son capaces de determinar cuáles métodos representan que funcionalidad, es decir no pueden inferir la semántica de los métodos y por lo tanto no pueden determinar fehacientemente si existen síntomas de tangling o scattering.

YAAM (Yet Another Aspect Miner) es una herramienta implementada en Java que da soporte a los pasos tres y cuatro del proceso propuesto.

YAAM permite la visualización de las reglas en el código fuente del sistema. Cuando una regla es visualizada, la clase del antecedente es cargada y la signatura del método del antecedente es marcado sobre el texto al igual que el nombre del método del consecuente. En la Fig. 5, se observa cómo la regla Point.notifyObservers ⇒ Screen.refresh es visualizada por YAAM. Además, YAAM da soporte para diferentes operadores de filtrado y ordenamiento sobre las reglas de asociación.

5 Caso de Estudio: Patrón Observer

En esta sección se muestra el proceso anteriormente introducido aplicado a la implementación del patrón Observer analizado en secciones anteriores.

La instrumentación del sistema fue realizada agregando un aspecto de tracing al sistema original. Luego, se utilizaron dos escenarios para ejercitar las funcionalidades del sistema y obtener las trazas y relaciones de ejecución: "un punto cambia su color" y "un punto cambia su posición". En la Fig. 2 se observa la traza obtenida para el primero de los escenarios. Para generar las reglas se utilizaron valores de soporte y confianza iguales a 1.0, la lista de aspectos candidatos alcanzada se muestra en la Tabla 1. En la Tabla 1, por cada aspecto candidato obtenido se muestra el indicador (tangling, scattering o ambos), el concern al que corresponde (introducido manualmente al analizar la regla), la regla, el soporte y la confianza.

Por ejemplo, la regla Screen.notifyObservers ⇒ Point.notifyObservers fue clasificada como indicador de scattering ya que hay dos clases que implementan una misma operación asociada al concern "mecanismo de notificación". Además, la regla Screen.display ⇒ Screen.notifyObservers confirma la naturaleza crosscutting del mecanismo de notificación impuesto por el patrón.

Tabla 1. Aspectos candidatos para el caso de estudio patrón Observer.

Indicador	Concern	Regla	Sop.	Conf.
Tangling	Lógica de actualización	Point.notifyObservers ⇒ Screen.refresh	1.0	1.0
Tangling	Falso positivo	Screen.refresh ⇒ Screen.display	1.0	1.0
Tangling	Lógica de actualización	Screen.notifyObservers ⇒ Screen.refresh	1.0	1.0
Tangling	Mecanismo de notificación	Screen.display ⇒ Screen.notifyObservers	1.0	1.0
Scattering	Mapeo Sujeto-Observador	Screen.addObserver ⇒ Point.addObserver	1.0	1.0
Scattering	Mecanismo de notificación	Screen.notifyObservers ⇒ Point.notifyObservers	1.0	1.0

El concern “Mapeo Sujeto-Observador” corresponde al mantenimiento del mapeo entre los sujetos y observadores, por lo que cada sujeto poseerá métodos para agregar (addObserver) y eliminar observadores (removeObserver). La regla Screen.addObserver ⇒ Point.addObserver muestra cómo este concern provoca síntomas de scattering ya que dos clases deben añadir funcionalidad adicional a su funcionalidad original.

Luego, el concern “Lógica de actualización” corresponde a las acciones que los observadores realizan cuando un sujeto cambia su estado. Por ejemplo, la regla Point.notifyObservers ⇒ Screen.refresh indica que cuando un objeto Point cambia su estado, una actualización (refresh) de la pantalla (Screen) es llevada a cabo. A su vez, la misma acción es tomada por los objetos Screen cada vez que ellos mismos cambian de estado (regla Screen.notifyObservers ⇒ Screen.refresh).

La regla Screen.refresh ⇒ Screen.display clasificada como indicadora de tangling existe debido a que el método refresh posee una invocación al método display, pero la misma no corresponde a un crosscutting concern. Por lo tanto, esta regla es un falso positivo en el sentido que indica un posible aspecto candidato aunque esta posibilidad es desechada luego de una inspección manual del código.

En [10] los autores analizan y comparan los resultados obtenidos luego de implementar los patrones de diseño [7] en un lenguaje orientado a aspectos (AspectJ [26]) y un lenguaje orientado a objetos (Java). En particular se utiliza al patrón Observer como caso de estudio y se discute cómo este patrón introduce crosscutting concerns sobre las clases que participan en el mismo. En nuestro caso, la aplicación de la técnica propuesta sobre el caso de estudio nos permitió comprobar que la misma fue capaz de descubrir los mismos concerns que los autores habían identificado empíricamente como crosscutting concerns.

6 Caso de Estudio: JHotDraw

Esta sección presenta los resultados de aplicar la técnica propuesta a la versión 5.4b1 de JHotDraw [11], un programa Java con aproximadamente 18000 líneas no comentadas de código y alrededor de 2800 métodos. JHotDraw es un framework de dibujado en dos dimensiones y fue desarrollado originalmente como un ejercicio para ilustrar el uso correcto de los patrones de diseño orientados a objetos [7]. Estas características lo recomiendan como un caso de estudio bien diseñado, un pre-requisito para demostrar las mejoras que se pueden obtener a partir de las técnicas orientadas a aspectos. Además, demuestra que inclusive en sistemas (legados) bien diseñados existen limitaciones de modularización [4].

Desde su adopción original, JHotDraw ha sido utilizado en diferentes estudios sobre técnicas de aspect mining. Por ejemplo, en [4] se analiza JHotDraw mediante tres técnicas de aspect mining y se comparan sus resultados de manera cualitativa. Mientras que en [12], se analiza JHotDraw a través de la métrica fan-in y se discuten en profundidad los diferentes crosscutting concerns descubiertos por el autor. Por lo tanto, para comparar y validar nuestros resultados sobre JHotDraw, comparamos la unión de los crosscutting concerns identificados en [4] y [12] con aquellos descubiertos por nuestra técnica.

Tabla 2. Resumen de las reglas de asociación obtenidas para JHotDraw 5.4b1.

Concern	Regla/s	Sop.	Conf.	Indicador
Adapter	LocatorHandle.locate ⇒ AbstractHandle.owner	0.42	1.0	Tangling
Command	UndoableCommand.execute ⇒ AbstractCommand.execute	0.28	1.0	Scattering
	UndoableCommand.isExecutable ⇒ AbstractCommand.isExecutable	0.66	1.0	Scatt. & Tang.
Composite	CompositeFigure.includes ⇒ AbstractFigure.includes	0.19	1.0	Scatt. & Tang.
	CompositeFigure.add ⇒ AbstractFigure.addToContainer	0.28	1.0	Tangling
Consistent behaviour y Contract enforcement	AbstractCommand.execute ⇒ AbstractCommand.view	0.42	1.0	Tangling
	CreationTool.activate ⇒ AbstractTool.activate	0.09	1.0	Scatt. & Tang.
Decorator	DecoratorFigure.containsPoint ⇒ AbstractFigure.containsPoint	0.19	1.0	Scatt. & Tang.
	DecoratorFigure.moveBy ⇒ AbstractFigure.moveBy	0.14	1.0	Scatt. & Tang.
Observer	AbstractFigure.addToContainer ⇒ AbstractFigure.addFigureChangeListener	0.28	1.0	Tangling
	AbstractFigure.moveBy ⇒ AbstractFigure.changed	0.14	1.0	Tangling
Persistencia	AttributeFigure.write ⇒ AbstractFigure.write	0.14	1.0	Scatt. & Tang.
	RectangleFigure.read ⇒ Attribute.read	0.14	1.0	Scatt. & Tang.
Undo	UndoableCommand.execute ⇒ UndoableAdapter.isUndoable	0.28	1.0	Tangling
	UndoableCommand.execute ⇒ AbstractCommand.getUndoActivity	0.28	1.0	Tangling
Manage handles	RelativeLocator.south ⇒ BoxHandleKit.south	0.14	1.0	Scattering
	RelativeLocator.west ⇒ BoxHandleKit.west	0.14	1.0	Scattering

El primer paso para analizar JHotDraw fue realizar una versión instrumentada del sistema que permita obtener las trazas y las relaciones de ejecución. Para esto, se

agregó un aspecto que registra este tipo de información interceptando llamadas a métodos y constructores durante la ejecución de un escenario. A continuación, se definieron 21 escenarios acorde a las principales funcionalidades descriptas en la documentación de la aplicación, por ejemplo, se crearon escenarios para dibujar una figura, para pintar una figura, para abrir un documento guardado, etc. La ejecución de estos escenarios provocó la ejercitación de 610 métodos.

Luego, la aplicación de la técnica sobre las 21 trazas arrojó un resultado de 634 reglas utilizando un valor de soporte mínimo de 0.1, y una confianza mínima de 0.8. También se fijó el tamaño de los itemsets a dos, ya que nos interesaba obtener reglas del tipo "si m1 entonces m2". La Tabla 2 presenta un resumen de las reglas obtenidas, en la misma se muestra el indicador (tangling, scattering o ambos), el concern al que corresponde (introducido manualmente al analizar la regla), la regla, el soporte y la confianza.

Para el patrón Adapter, se encontraron reglas que muestran un método (owner) el cual manipula la referencia a la clase adaptada.

Las reglas para el patrón Command muestran cómo este patrón provoca síntomas de scattering y tangling para el caso de métodos como execute o isExecutable.

En JHotDraw, la clase CompositeFigure es parte de una instancia del patrón Composite. En este caso, las reglas dan evidencia sobre la existencia de síntomas de scattering y tangling, ya que la clase debe implementar no sólo funcionalidad correspondiente a una figura sino también la funcionalidad impuesta por el patrón (como por ejemplo, gestión de las figuras contenidas).

Los concerns "Consistent behaviour y Contract enforcement" generalmente describen funcionalidad común requerida desde, o impuesta sobre, los participantes en un contexto dado. Este es el caso para la jerarquía de Commands, para la cual los métodos execute contienen código para asegurarse que la pre-condición "existe una vista activa" es válida [4]. Para este caso, la primera de las reglas hace referencia al crosscutting concern anterior, y la segunda muestra un crosscutting concern relacionado a las actividades de inicialización de las clases participantes.

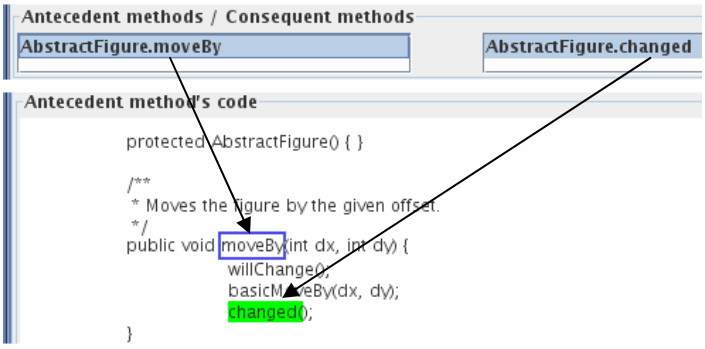


Fig. 6. Visualización de la regla AbstractFigure.moveBy ⇒ AbstractFigure.changed en JHotDraw.

Muchas de las reglas obtenidas y clasificadas por la técnica como indicadoras de scattering y tangling corresponden al patrón Decorador. Estas reglas demuestran el mecanismo de re-direccionamiento consistente que caracteriza al patrón.

En cuanto al patrón Observer, las reglas encontradas muestran el código tangling que introduce el concern de mapeo entre sujetos y observadores (método addFigureChangeListener), y el mecanismo de notificación (método changed) (Fig. 6).

En la Fig. 7 se presenta con mayor detalle la estructura del patrón Observer en JHotDraw. Como se puede observar, una figura (representada por la interfaz Figure) no solo debe implementar su funcionalidad básica (como saber dibujarse) sino también debe implementar un conjunto de operaciones para poder notificar cambios en su estado (operaciones changed y willChange) y para gestionar los observadores (addChangeListener y removeChangeListener). Por lo tanto, el código correspondiente a esos concerns se encontrará esparcido a lo largo de toda la jerarquía de figuras.

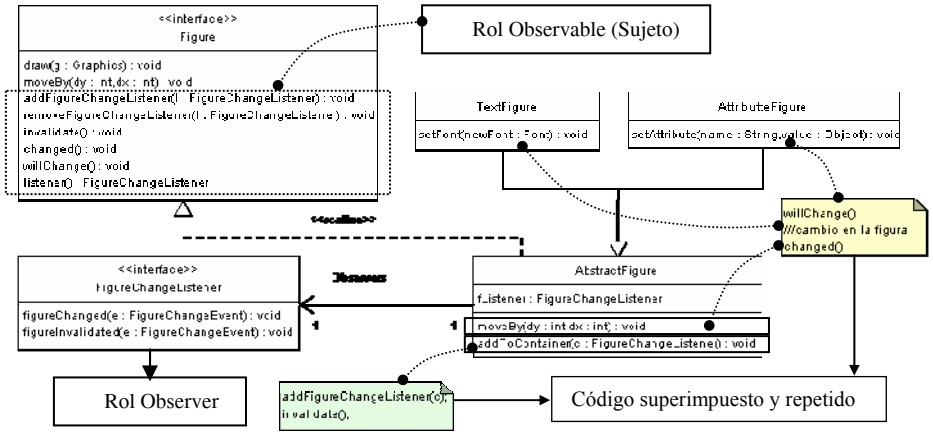


Fig. 7. El patrón Observer en JHotDraw. No se incluyeron todas las clases y relaciones existentes para simplificar la presentación del diagrama.

Las reglas relacionadas con el concern de persistencia muestran la presencia de los métodos write y read dispersos a través de diferentes clases.

El concern Undo está esparcido sobre varias clases entremezclando su funcionalidad con la funcionalidad básica de cada clase, dicha relación es explícita en las reglas encontradas para este concern.

El concern "Manage handles" representa una funcionalidad crosscutting responsable del manejo de handles asociados a elementos gráficos (de manera de soportar cambios de dimensión, drag & drop, etc.). Algunos de los métodos que implementan esta funcionalidad son north(), south(), east() y west(), los cuales se encuentran en las reglas para este concern.

En [4] se reporta a la funcionalidad "bring to front/send to back" como un crosscutting concern, aunque la misma no fue identificada por nuestra técnica como un posible aspecto. Esto se debe a que los escenarios utilizados para generar las trazas no incluyeron la utilización de esa característica, por lo que la misma nunca fue ejercitada.

La técnica propuesta fue capaz de identificar casi todos los crosscutting concerns previamente descubiertos en [4] y [12]. El único crosscutting concern no identificado,

muestra uno de los principales problemas asociados a las técnicas basadas en análisis dinámico: los resultados dependen de los escenarios utilizados para ejercitar al sistema.

7 Trabajos Relacionados

La investigación en aspect mining puede ser clasificada en dos categorías: técnicas de análisis estático y técnicas de análisis dinámico.

Las técnicas de análisis estático analizan las frecuencias de los elementos de un programa y explotan la homogeneidad sintáctica de los crosscutting concerns. En [12], se propone un enfoque de aspect mining basado en determinar métodos con un alto valor de fan-in, los cuales pueden ser vistos como un síntoma de funcionalidad crosscutting. En [13], los autores utilizan formal concept analysis sobre el código fuente de manera de agrupar elementos del mismo basados en sus nombres. Enfoques basados en el uso de algoritmos de procesamiento del lenguaje natural ([14] y [15]), intentan identificar elementos del código fuente relacionados semánticamente, los cuales se podrían corresponder con la presencia de crosscutting concerns. Por otra parte, en [16], los autores proponen identificar "métodos únicos" los cuales son métodos sin retorno que implementan un mensaje que ningún otro método implementa. Otros enfoques, se basan en la aplicación de técnicas de clustering sobre el código fuente ([17] y [18]) o en la detección de código duplicado ([19], [20] y [21]). En cambio, [31] reporta sobre una técnica basada en random walks, la cual imita el proceso manual de búsqueda de crosscutting concerns para la identificación de potenciales aspectos.

Similar al presente trabajo, en [22] analizan las trazas de ejecución en búsqueda de patrones recurrentes en la ejecución de los métodos del sistema. Un patrón es considerado como aspecto candidato si ocurre más de una vez de manera uniforme, además, para asegurarse que estos patrones sean suficientemente crosscutting, los mismos deben aparecer en diferentes 'contextos de ejecución'. La principal diferencia con nuestra técnica es que la misma permite identificar síntomas de scattering y tangling, mientras que la técnica anterior sólo identifica síntomas de scattering. Otro enfoque dinámico [23], analiza las trazas del sistema mediante formal concept analysis, e identifica aspectos candidatos en el lattice de conceptos resultante. Si bien esta técnica, al igual que la nuestra, permite identificar síntomas de scattering y tangling, ambas se diferencias en el tipo de análisis realizados sobre las trazas de ejecución. Sin embargo, es necesario más análisis para comparar ambos enfoques en términos de precisión.

8 Conclusiones y Trabajo Futuro

El presente trabajo, detalla los resultados obtenidos en el desarrollo de una nueva técnica para aspect mining la cual está basada en análisis dinámico y reglas de asociación. La misma es un proceso de cuatro pasos, cuya salida es una lista de reglas de asociación clasificadas automáticamente como indicadoras de scattering, tangling

o scatttering y tangling. A su vez, se implementó una herramienta denominada YAAM (Yet Another Aspect Miner) la cual automatiza los últimos dos pasos del proceso propuesto correspondientes a la extracción y filtrado de las reglas de asociación.

Las principales ventajas de la técnica es la posibilidad de (semi) automáticamente identificar síntomas de scattering y tangling, y la posibilidad de obtener aspectos candidatos en forma de reglas, las cuales poseen un gran poder expresivo. Ya que las mismas pueden relacionar dos o más métodos, es posible saber cuáles métodos corresponden a cuál concern (scattering) o dónde el crosscutting concern está entremezclado (tangling) con las clases principales.

Por otra parte, una de las desventajas de la técnica es la generación de muchos falsos positivos, en particular la generación de falsos indicadores de tangling (tal como fue discutido para el caso de estudio Observer). Otras de las desventajas de la misma son aplicables a todas las técnicas basadas en análisis dinámico:

- Las mismas son parciales, ya que no todos los métodos involucrados en un crosscutting concern son obtenidos. Por ejemplo, para el caso de estudio JHotDraw, se detectó la presencia del patrón Observer aunque no se reconocieron todas las instancias presentes en el mismo.
- Los aspectos identificados dependen de los escenarios utilizados, lo que significa que algunos aspectos candidatos podrían perderse (tal como sucedió con la funcionalidad “bring to front/send to back” en el caso de estudio JHotDraw).

Los casos de estudio analizados nos permitieron validar los resultados de la técnica e identificar sus posibles problemas.

Como trabajo futuro, estamos analizando diferentes filtros de post-procesamiento para las reglas de asociación de manera de mejorar la precisión a la hora de identificar síntomas de tangling. Por ejemplo, un nuevo filtro podría considerar los contextos de llamada del método incluido en el consecuente de una regla. Entonces, dadas dos reglas $A \Rightarrow B$ y $C \Rightarrow B$ ambas reglas serían clasificadas como indicadoras de tangling ya que el método B es el mismo en ambos consecuentes estando su funcionalidad incluida tanto en A como en B. Pero, si la regla $A \Rightarrow B$ no existiese entonces la regla $C \Rightarrow B$ no sería clasificada como indicador de tangling, y viceversa.

Además, como trabajo futuro, se comparará la técnica propuesta con las existentes técnicas de análisis dinámico de manera de mejorar el conocimiento existente en este tipo de técnicas.

Agradecimientos

Este trabajo fue dirigido por la Dra. Caludia Marcos, ISISTAN Research Institute, Facultad de Ciencias Exactas, Universidad Nacional del Centro de la Provincia de Buenos Aires (UNCPBA), Tandil, Bs. Argentina.

Referencias

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., Irwin, J.: Aspect-oriented programming. In: ECOOP, pp. 220--242 (1997)
2. 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)
3. Kellens, A., Mens, K., Tonella, P.: A Survey of Automated Code-Level Aspect Mining Techniques. pp. 143--162 (2007)
4. Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T.: A Qualitative Comparison of three Aspect Mining Techniques. In: 13th International Workshop on Program Comprehension, pp. 13--22. IEEE Computer Society, Washington DC, USA (2005)
5. 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)
6. Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide (2nd Edition). Addison-Wesley Professional (2005)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (1995)
8. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: 20th International Conference on Very Large Data Bases, pp. 487--499. Morgan Kaufmann Publishers Inc., San Francisco, USA (1994)
9. Han, J., Kamber, M.: Data Mining: Concepts and Techniques (2nd Edition). Morgan Kaufmann (2006)
10. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and Aspectj. In: 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 161--173. ACM Press, New York, USA (2002)
11. JHotDraw Start Page, www.jhotdraw.org
12. Marin, M., Van Deursen, A., Moonen, L.: Identifying Crosscutting Concerns Using Fan-in Analysis. ACM Trans. Softw. Eng. Methodol. 17 (1), 1--37 (2007)
13. 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)
14. Shepherd, D., Pollock, L. L., Tourwé, T.: Using Language Clues to Discover Crosscutting Concerns. ACM SIGSOFT Software Engineering Notes 30 (4), 1--6 (2005)
15. Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., Vijay-Shanker, K.: Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In: 6th International Conference on Aspect-oriented software development, pp. 212--224. ACM Press, New York, USA (2007)
16. 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)
17. Shepherd, D., Pollock, L.: Interfaces, Aspects, and Views. In: International Conference on Aspect Oriented Software Development. Amsterdam, The Netherlands (2005)
18. Serban, G., Moldovan, G. S.: A New K-Means Based Clustering Algorithm in Aspect Mining. In: Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 69--74. IEEE Computer Society, Washington DC, USA (2006)
19. Shepherd, D., Gibson, E., Pollock, L. L.: Design and Evaluation of an Automated Aspect Mining Tool. In: Arabnia, H. R., Reza, H., Arabnia, H. R., Reza, H. (eds.), Software Engineering Research and Practice. CSREA Press, 601--607 (2004)

20. Bruntink, M., van Deursen, A., van Engelen, R., Tourwe, T.: On the Use of Clone Detection for Identifying Crosscutting Concern Code. IEEE Trans. Softw. Eng. 31 (10), 804--818 (2005)
21. M. Bruntink: Aspect Mining Using Clone Class Metrics. In: Working Conference on Reverse Engineering (2004)
22. 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)
23. 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)
24. Lanza, M.: Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization. Ph.D. Thesis, University of Berne, Switzerland (2003)
25. Visser, E.: A Survey of Strategies in Program Transformation Systems. Electronic Notes in Theoretical Computer Science (2001)
26. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G.: An Overview of Aspectj. In: Knudsen, J. L., Knudsen, J. L. (eds.) ECOOP 2001. LNCS, vol. 2072, pp. 327--353, Springer (2001)
27. Demeyer, S., Ducasse, S., Nierstrasz, O.: Object Oriented Reengineering Patterns. Morgan Kaufmann (2002)
28. Lehman, M. M., Belady, L. A. (eds.): Program Evolution: Processes of Software Change. Academic Press Professional, Inc., San Diego, CA, USA (1985)
29. Opdyke, W. F.: Refactoring Object-Oriented Frameworks. Ph.D. Thesis, Urbana-Champaign, IL, USA (1992)
30. T. Mens, K. Mens, T. Tourwé. Aspect-Oriented Software Evolution. ERCIM News 58: 36--37 (2004)
31. Zhang, C., Jacobsen, H.-A.: Efficiently Mining Crosscutting Concerns through Random Walks. In: 6th international Conference on Aspect-oriented Software Development, pp. 226--238. ACM Press, New York, USA (2007)