Un proceso iterativo para la refactorización de aspectos

An iterative process for aspect refactoring

Santiago A. Vidal[1,2] Ing., Esteban S. Abait[1,3] Ing. y Claudia Marcos[1] PhD.
1. ISISTAN Research Institute, Facultad de Ciencias Exactas, UNICEN Campus Universitario, Buenos Aires, Argentina
2. CONICET, Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina.
3. CIC, Comisión de Investigaciones Científicas.
{svidal, eabait, cmarcos}@exa.unicen.edu.ar

Recibido para revisión 16 de mayo de 2009, aceptado 15 de junio de 2009, versión final 9 de julio de 2009

Resumen—El desarrollo de software orientado a aspectos permite encapsular *concerns* que cortan transversalmente las componentes funcionales de una aplicación, mejorando la modularización y como consecuencia el mantenimiento de la aplicación. Por esta razón, para aprovechar los beneficios de la orientación a aspectos surge la necesidad de migrar los sistemas orientados a objetos existentes a la orientación a aspectos mejorando el mantenimiento y evolución de los mismos. En este trabajo se presenta un proceso iterativo que asiste durante la tarea de migración de una aplicación orientada a objetos a una orientada a aspectos. Una vez que los aspectos han sido identificados por medio de alguna técnica de *aspect mining* se utilizan patrones de estructura que permiten identificar el *refactoring* a aplicar para generar el código orientado a aspectos resultante, *aspect refactoring*.

Palabras Clave—Desarrollo de Software Orientado a Aspectos, *Aspect Refactoring*, Mantenimiento de Aplicaciones Orientadas a Aspectos.

Abstract— Aspect-oriented software development allows the encapsulation of crosscutting concerns, achieving a better system modularization and, therefore, improving its maintenance. For this reason, in order to take advantage of the benefits of aspect-oriented programming, the legacy systems and applications have to be migrated. In this paper, an iterative process that assists during the migration process of an object-oriented system to an aspect-oriented one is presented. Once the aspects have been identified through some aspect mining technique, structural patterns are used to identify the refactorings that can be applied to aspectizable code, aspect refactoring.

Keywords—Aspect-Oriented Software Development, Aspect Refactoring, Aspect-oriented Applications Maintenance.

I. INTRODUCCIÓN

Una vez que los sistemas de software son desarrollados y entregados inevitablemente presentarán cambios motivados, por ejemplo, por nuevas exigencias del negocio que generarán cambios en el ambiente, correcciones a fallos encontrados en el funcionamiento, nuevos requerimientos, actualizaciones de los requerimientos existentes, adaptaciones a nuevas plataformas, mejoras de rendimiento u otras características no funcionales, entre otras [21] [3]. Estas necesidades han llevado al concepto de la evolución del software, la cual aborda la problemática de la funcionalidad de los sistemas cuando cambia su ambiente. En este sentido, Lehman estableció un conjunto de leyes (más bien hipótesis) concerniente a los cambios de los sistemas [16], de las cuales se desprende 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 ser usado y adaptado a un costo razonable [3].

La orientación a aspectos ha sido propuesta como un nuevo paradigma que permite mejorar la separación de concerns en el software [14]. De manera de capturar los concerns que cortan transversalmente las componentes funcionales de una aplicación, crosscutting concerns, un nuevo mecanismo de

abstracción denominado aspecto es incorporado al desarrollo de software. El objetivo principal del Desarrollo de Software Orientado a Aspectos (DSOA) es modularizar un sistema de manera tal de mejorar su evolución y mantenimiento. Por esta razón, para mejorar el proceso de mantenimiento de sistemas orientados a objetos es que surge la necesidad de migrar sistemas de software existentes hacia su equivalente orientado a aspectos y reestructurarlos de manera continua. Debido al gran tamaño de los sistemas legados, 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 [13]. Algunos de los síntomas que permiten identificar crosscutting concerns en un sistema son code tangling y code scattering [15]. El primero de estos se presenta cuando en un módulo existe más de un concern, en tanto que el segundo ocurre cuando un concern se encuentra diseminado en diferentes módulos.

El estudio y desarrollo de tales técnicas es el objetivo de aspect mining y aspect refactoring. Aspect mining [12] es la actividad de descubrir crosscuting concerns que potencialmente podrían convertirse en aspectos. Las técnicas de aspect mining toman como entrada el código de un sistema legado y de manera automática, o semiautomática, 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 [12]. El propósito de aspect refactoring es lograr un proceso similar al refactoring orientado a objetos que sea aplicable a la orientación a aspectos.

De manera tal de aprovechar los potenciales beneficios de la programación orientada a aspectos con respecto a su modularización y consecuente mantenimiento de aplicaciones, ha surgido la necesidad de migrar los sistemas y aplicaciones legadas orientadas a objetos. Dentro de este marco, en este trabajo se propone un proceso de migración de sistemas orientados a objetos a sistemas orientados a aspectos con el fin de mejorar el mantenimiento y evolución de las aplicaciones de software modularizando el sistema en concerns funcionales y crosscutting concerns. Adicionalmente, se presenta una herramienta que soporta el proceso y asiste al desarrollador durante la migración.

Para llevar a cabo el objetivo de migrar un sistema se combinaron distintas técnicas para lograr cubrir la amplia gama de posibles escenarios que se presentan al abordar esta tarea. El proceso de refactorización propuesto sigue un enfoque iterativo donde la entrada es código orientado a objetos con los aspectos candidatos identificados y su salida es una refactorización de este código a aspectos. Este enfoque busca obtener todas las evidencias de código aspectizable que se hayan hallado en un sistema (mediante el proceso de aspect mining). Una vez que los aspectos han sido identificados se analiza la posibilidad de aplicar uno o más aspect refactorings que permitan transformar o trasladar el código en consideración a un aspecto. Para ello se utilizan patrones de estructura que permiten identificar el refactoring a aplicar. El proceso propuesto soporta diferentes tipos de refactorings de manera tal de asistir al desarrollador durante el proceso de migración.

El resto de este trabajo se estructura de la siguiente manera. En la Sección 2, se describe el mecanismo de aspect refactoring y sus clasificaciones. En la Sección 3, se presenta el proceso de migración propuesto. Luego, en la Sección 4, se describe una herramienta llamada AspectRT que da soporte al proceso. En la Sección 5, se describe el proceso mediante su aplicación a un caso de estudio, donde varios refactorings han sido identificados automáticamente por la herramienta. Finalmente, en la Sección 6, se presentan las conclusiones y trabajos futuros identificados.

II. ASPECT REFACTORING

Para poder transformar código orientado a objetos en código orientado a aspectos se debe contar con mecanismos y estrategias de reestructuración de código, denominados aspect refactorings, los cuales deben tener en cuenta la presencia de crosscutting concerns en el sistema legado. Específicamente, para migrar sistemas orientados a objetos a sistemas orientados a aspectos será necesario estudiar y analizar tres tipos de aspect refactorings [8]:

1) Refactorings Aspect-Aware OO (orientados a objetos): este primer tipo engloba a aquellos refactorings orientados a objetos que se extendieron y adaptaron para poder ser utilizados en el paradigma orientado a aspectos. Cuando se reestructura software siguiendo técnicas de refactoring orientados a objetos es necesario también tener en cuenta las construcciones orientadas a aspectos para modificar, es decir, tener en cuenta los aspectos y su relación con las componentes funcionales de la aplicación. Los refactorings Aspect-Aware OO son el tema de muchas investigaciones como por ejemplo [7], [11] y [24].

2) Refactorings de construcciones AOP (aspect-oriented programing): los refactorings agrupados bajo este segundo tipo presentan la propiedad de estar orientados específicamente a elementos de la programación orientada a aspectos. Es posible encontrar una subdivisión con respecto a su procedencia. En primer lugar, aquellos refactorings que tienen un paralelo en la orientación a objetos, por ejemplo se pueden comparar los pointcuts con los métodos y los aspectos con las clases y a partir de esto intentar aplicar refactorings procedentes del paradigma orientado a objetos en construcciones de aspectos. En segundo lugar, hay un grupo de refactoring que son totalmente nuevos, es decir no tienen equivalentes en OO, debido a distintos elementos como los advices. Ejemplos de este

segundo tipo son abordados en las investigaciones [11], [19] y [20].

3) Refactorings de CCC (crosscutting concerns): por último, existe un tercer conjunto que surge de pequeñas transformaciones de código orientado a objetos hacia la orientación a aspectos. Este tercer grupo de refactorings tiene por objetivo transformar los crosscutting concerns en aspectos. Volviendo a la idea elemental del paradigma orientado a aspectos se puede decir que los refactorings agrupados bajo este tipo buscan agrupar los distintos concerns que se encuentran diseminados por todo el código al modularizarlos en un aspecto (Figura 1). Debido a su complejidad, estos refactorings generalmente suelen ser composiciones de otros refactorings de los tipos Aspect-Aware OO y de construcciones AOP. Es fácil intuir que este grupo de refactoring es útil en la tarea de migrar un sistema, por ejemplo orientado a objetos, a una orientación a aspectos. Ejemplos de este último grupo pueden encontrarse en [10], [18] y [19].

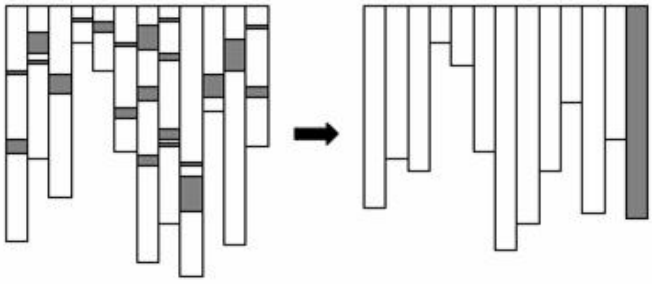


Figura 1. Esquema de refactorings CCC.

Ante lo expuesto anteriormente, es posible advertir que tanto el refactoring como la forma de identificar los crosscutting concerns estará determinado por la mejora que se desee obtener del código. Es decir, algunos de los objetivos que pueden promover la aplicación de aspect refactoring son:

- La migración de código orientado a objetos a uno orientado a aspectos.
- La reestructuración de sistemas orientados a aspectos ya sean los aspectos propiamente dichos y/o los objetos del código base.

III. PROCESO DE MIGRACIÓN

De la sección anterior se deriva que es necesario combinar distintas técnicas para lograr cubrir la amplia gama de posibles escenarios que se presentan al abordar la tarea de migrar sistemas orientados a objetos a sistemas orientados a aspectos. Es decir, las técnicas analizadas con anterioridad son en la mayoría de los casos complementarias ya que intentan resolver diferentes problemáticas del proceso de *refactoring*.

El hecho de tener que aplicar una amplia variedad de refactorings para migrar un sistema conduce a la necesidad de contar con herramientas que soporten los diferentes tipos de refactorings mencionados (Aspect-Aware OO, construcciones AOP y CCC) y que automaticen la mayor cantidad de tareas posibles con el fin de facilitar la tarea de migración al desarrollador.

Con este objetivo se propone un enfoque para la migración de sistemas legados orientados a objetos a sistemas orientados a aspectos. El mismo contempla los tres tipos de refactorings presentados (debido a la complementariedad entre estos para transformar código OO en código OA) y permite su aplicación en forma automática o semiautomática. Además, es posible su vinculación con el proceso de aspect mining para facilitar el descubrimiento de código aspectizable.

El proceso de refactorización propuesto sigue un enfoque iterativo donde la entrada es código orientado a objetos con los aspectos candidatos identificados, en una etapa anterior de *aspect mining*, y su salida es una refactorización de este código a aspectos en AspectJ [23]. Este enfoque busca obtener todas las evidencias de código aspectizable que se hayan hallado en un sistema y para cada una de ellas analiza la posibilidad de aplicar uno o más *aspect refactorings* que permitan transformar o trasladar el código en consideración a un aspecto.

El proceso de transformar código orientado a objetos en código orientado a aspectos consta de cinco pasos (Figura 2):

1) Obtener evidencias de código aspectizable: el objetivo de este paso es recuperar, proveniente del proceso de *aspect mining*, el código que ha sido identificado como aspectizable. Es decir, se vincula el proceso de refactorización con la actividad de aspect mining con el fin de identificar en el código orientado a objetos aquellos atributos, métodos, clases, etc. que deben ser refactorizados para encapsular los crosscutting concerns en aspectos. Este objetivo es llevado a cabo mediante un archivo XML que proviene de una herramienta externa de aspect mining que identifica los aspectos mediante análisis dinámico y reglas de asociación [1]. El mismo contiene una lista de aspectos candidatos e información relevante a cada uno de estos que puede ser recorrida iterativamente.

2) Analizar posibles refactorings CCC: este paso consiste en examinar la posibilidad de aplicar uno o un conjunto de refactorings CCC al código objetivo que se obtuvo en la etapa anterior. Es decir, en este paso se identifican los refactorings a aplicar. El hecho de que el tipo de refactoring a aplicar sea CCC se debe a que justamente los fragmentos de código que se han identificado en el paso anterior, provenientes del proceso de aspect mining, contienen crosscutting concerns que deben ser encapsulados en un aspecto.

Con el objetivo de reconocer de un conjunto de aspect refactorings CCC cuáles pueden ser aplicados a un determinado fragmento de código, se infieren patrones de estructura que delimiten un subconjunto de aspect refactorings entre los cuales el desarrollador pueda optar. Estos patrones varían según el refactoring en su estructura y complejidad y son identificados a partir de reglas simples como:

Si (el código objetivo es un método/field/clase/…)
entonces (Los refactorings aplicables son X,Y,Z,…)

Por ejemplo, el aspect refactoring *Move Method from Class to Inter-type* [19], el cual busca extraer a un aspecto un método relacionado con un concern de una clase determinada, es aplicable si el código objetivo seleccionado es un método que sólo contiene lógica inherente al crosscutting concern que se intenta encapsular. Es decir, mediante las reglas de inferencia podrá determinarse que el refactoring Move Method from Class to Inter-type es aplicable al código objetivo que se esta analizando, sin embargo, será tarea del desarrollador determinar si es útil su aplicación.

3) Aplicar refactoring CCC: en esta etapa se ejecuta la planificación realizada anteriormente extrayendo los crosscutting concerns del código orientado a objetos e insertándolos en un aspecto. Es decir, en esta etapa se aplica la reestructuración al código. En general, debido a que los mecanismos de los refactorings se describen paso a paso, esta etapa se realiza automáticamente. Eventualmente, podría ser necesaria la intervención del desarrollador para algunas decisiones, como por ejemplo, la elección del aspecto en el cual se encapsulará un determinado fragmento de código, el nombre que se le dará a un pointcut, etc.

4) Aplicar refactorings OO o *Aspect-Aware OO*: en el caso que no se haya podido realizar ningún refactoring CCC este paso busca realizar refactorings orientados a objetos y aspect-aware OO sobre el código objetivo con el fin de reestructurarlo para así reintentar la actividad de analizar posibles refactorings CCC. Muchas veces sucede que el código identificado por la actividad de aspect mining no puede ser encapsulado directamente en un aspecto sino que previamente debe ser reestructurado el código orientado a objetos para que pueda adaptarse al patrón del aspect refactoring. Por ejemplo, si pretende aplicarse el aspect refactoring *Move Method from Class to Inter-type* y el método seleccionado contiene lógica que debe permanecer en la clase debería aplicarse el refactoring orientado a objetos *Extract Method* [5] al fragmento de código que contiene esa lógica. En cuanto a cómo reconocer de un conjunto de aspect refactorings OO o Aspect-Aware OO cuáles pueden ser aplicados a un determinado fragmento de código, nuevamente al igual que en el paso 2 se utilizan patrones de estructura. En este caso, se analiza el código objetivo con el fin de reconocer estructuras como métodos, llamadas a métodos, clases *inner*, interfaces, etc. para, de esta forma, sugerir posibles refactorings.

5) Aplicar refactoring AOP: este último paso, dentro de una iteración, busca aplicar refactorings AOP al código orientado a aspectos resultante. Al ser un proceso iterativo, en cada una de las iteraciones se genera código orientado a aspectos que es integrado al código de la iteración anterior, lo cual seguramente genera una necesidad de reestructuración del código orientado a aspectos resultante. Puede suceder que para extraer un crosscutting concern del código orientado a objetos se apliquen múltiples aspect refactorings. Ante esta situación la estructura interna del aspecto que encapsula el código aspectizable suele necesitar reestructuraciones para mejorar su legibilidad, modularidad, eliminar duplicación de código, etc. Por ejemplo, luego de aplicar repetidamente el aspect refactoring *Extract Fragment into Advice* [19], el cual tiene por finalidad encapsular un fragmento de código objetivo en un aspecto creando para ello un advice y su correspondiente pointcut, suele suceder que en el aspecto resultante haya pointcuts repetidos. Ante esta situación, se deben unificar los pointcuts y actualizar las referencias de los advices. Para reconocer de un conjunto de aspect refactorings AOP cuáles pueden ser aplicados a un determinado fragmento de código, se utiliza un enfoque de patrones similar al que se mencionó anteriormente. A diferencia de los casos anteriores en los que se analizaba el código objetivo aquí debe tenerse en cuenta todo el aspecto o incluso un grupo de estos ya que, por ejemplo, podrían generalizarse los pointcuts o crear una jerarquía de aspectos. Además, complementariamente se registra la historia de los refactorings aplicados con el fin de reconocer los problemas que podrían estar presentes en el

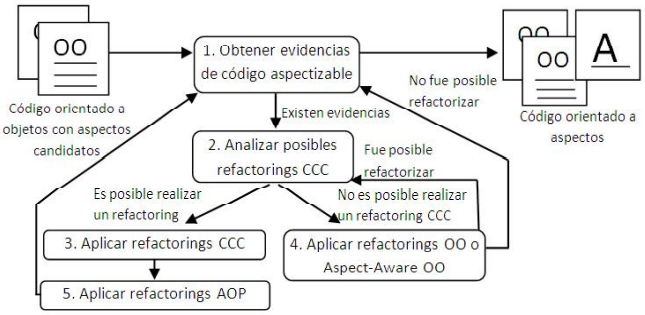


Figura 2. Proceso de migración de sistemas orientados a objetos a aspectos.

aspecto. Por ejemplo, sólo será útil buscar la existencia de pointcuts similares si estos fueron creados.

De esta forma, al utilizar el proceso no solo se encapsulan los crosscutting concerns, que el desarrollador elige en un aspecto, sino que además, se mejora la estructura interna de los aspectos del código resultante.

IV. ASPECTRT

De manera tal de soportar el enfoque propuesto se construyó una herramienta denominada AspectRT (Aspect Refactoring Tool), la cual fue implementada como un plugin para el entorno de desarrollo Eclipse [4] y se integra con AspectJ [23]. AspectRT es una herramienta que asiste a los arquitectos, diseñadores y desarrolladores a migrar sistemas orientados a objetos a una orientación a aspectos, esta provee un conjunto de aspect refactorings para lograr desarrollar sistemas bajo el paradigma

orientado a aspectos. Complementariamente, con el objetivo de proveer una mejor visualización del sistema que se esta migrando, se ha desarrollado una herramienta basada en el framework Vizz3D [17] la cual grafica en tres dimensiones los aspectos, clases y métodos involucrados en el proceso de migración.

Las características que presenta AspectRT son:

- Automatización de la mayor parte del proceso: esto implica, en el mejor de los casos, dado un código aspectizable la posibilidad de aplicar un refactoring de forma transparente al desarrollador.
- Soporte para los tres tipos de refactorings presentados: debido a la importancia que radica en la complementariedad entre estos es necesaria la utilización de diferentes tipos de reestructuraciones para obtener mejores soluciones.
- Vinculación con aspect mining: para facilitar la tarea del descubrimiento del código aspectizable se aprovecha la potencialidad que las técnicas de aspect mining ofrecen con este propósito.
- Extensión: el diseño de la herramienta es lo suficientemente flexible para ser extendida con nuevos refactorings.

A continuación se describe cómo se implementó el proceso de migración presentado en este trabajo.

A. Evidencias de código aspectizable

Para la vinculación de la herramienta con el proceso de aspect mining se utiliza un archivo XML. Dicho archivo contiene información sobre los aspectos candidatos, información relevante para analizar los refactorings a aplicar, como por ejemplo, indicador por el cual el código es aspectizable (scattering, tangling, etc.), clase y método en el que se encuentra, ubicación dentro del sistema, etc.

Actualmente, este archivo es generado por una herramienta que analiza dinámicamente las trazas de ejecución utilizando reglas de asociación para el descubrimiento de potenciales aspectos [1].

Para cargar el archivo se debe acceder a “Iterative Tool” desde el menú principal de la aplicación (Figura 3).

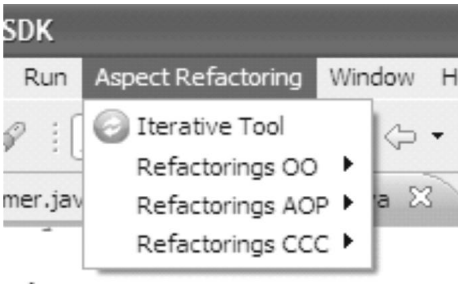


Figura 3. Menú principal de AspectRT.

Una vez hecho este procedimiento se carga toda la información presente en el archivo en la vista "Aspect candidates" (Figura 4). Mediante esta vista pueden recorrerse iterativamente las evidencias de código aspectizable para ser analizadas individualmente y aplicar los aspect refactorings necesarios. Como puede observarse en la Figura 4, los aspectos candidatos se cargan en una lista que puede ser recorrida utilizando los botones (flechas) de la parte superior derecha de la vista.

B. Análisis de posibles refactorings

En lo que respecta al análisis, en esta versión de la herramienta se identifican el o los refactorings que pueden ser aplicados sobre el código seleccionado utilizando patrones de estructura. Los mismos hacen uso de la información que presenta el código aspectizable sobre el cual se está iterando. Los datos más relevantes son especialmente aquellos que indican si el código aspectizable es un método, un field, código dentro de un método, una clase interna, etc. En la Tabla 1 se muestra una distribución de los posibles aspect refactorings según su estructura, esta información es representada por medio de patrones de estructura para su identificación automática. Por ejemplo, si el código aspectizable es código dentro de un método los aspect refactorings aplicables serán Extract Feature into Aspect y Extract Fragment into Advice. Como puede observarse para la mayoría de los atributos existe un reducido conjunto de aspect refactorings aplicables lo cual provoca que la técnica de identificación sea más precisa.

Los tres tipos de refactorings (CCC, AOP y Aspect-Aware OO), han sido representados con patrones de estructura. Cada

Tabla I. Patrones de estructura.

Atributo estructural	Aspect refactorings aplicables
Método	Move Method from Class to Inter-type, Extract Feature into Aspect
Clase abstracta	Change Abstract Class to Interface, Split Abstract Class between Aspect and Interface
Clase	Inline Class within Aspect
Clase interna	Extract Inner Class to Standalone
Field	Move Field from Class to Inter-type
Declaración implements	Encapsulate Implements with Declare Parents
Código dentro de un método	Extract Feature into Aspect, Extract Fragment into Advice
Interface	Inline Interface within Aspect,
Interface interna	Extend Marker Interface with Signature
Aspecto	Generalise Target Type with Marker Interface, Tidy Up Internal Aspect Structure, Extract Superaspect
Método Inter-Type	Replace Inter-type Method with Aspect Method, Pull Up Inter-type Declaration, Push Down Inter-type Declaration
Field Inter-Type	Replace Inter-type Field with Aspect Map, Pull Up Inter-type Declaration, Push Down Inter-type Declaration
Advice	Pull Up Advice, Push Down Advice
Declaración Declare Parents	Pull Up Declare Parents, Push Down Declare Parents
Pointcut	Pull Up Pointcut, Push Down Pointcut

uno de estos patrones representa la información para la identificación de los refactorings y las actividades a desarrollar para migrar el código. AspectRT, dependiendo de la información provista por aspect mining, identifica automáticamente el o los refactorings a aplicar los cuales son mostrados al desarrollador para que seleccione el que considere más adecuado. Por ejemplo,

como se observa en la Figura 5, se ha identificado como aspecto candidato al método drop y al intentar aplicar un refactoring CCC el único posible es *Move Method from Class to Inter-type*. Esto se debe a que el resto de los refactorings CCC no son aplicables a un método sino que, por ejemplo, *Move Field from Class to Inter-type* sólo es aplicable a un field, *Extract Fragment Into Advice* se aplica sobre un fragmento de un método, etc.

En esta versión de la herramienta se implementó un subconjunto de los refactorings presentados en [19], sin embargo la herramienta ha sido diseñada de manera tal de poder incorporar fácilmente nuevos catálogos. Para aplicar un determinado refactoring se debe seleccionar el código a

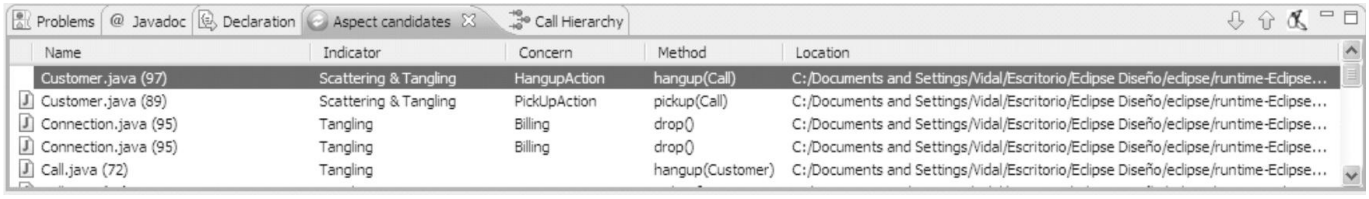


Figura 4. Vista de aspectos candidatos.

reestructurar y luego elegir el refactoring mediante el menú que se mostró en la Figura 3. Básicamente, para la interface de los distintos refactorings se siguió el enfoque utilizado por Eclipse para los refactorings orientados a objetos.

C. Visualización

La visualización en tres dimensiones (Figura 6) es una característica adicional con la que se ha dotado a la herramienta para la exploración del código. Su ventaja principal es la posibilidad de visualizar en un ambiente 3D las relaciones que existen entre paquetes, clases, aspectos, métodos y crosscutting concerns. De esta forma, el desarrollador puede contar con una visión de la arquitectura actual del sistema que se esta refactorizando desde otro punto de vista, contando con más recursos con los cuales decidir qué aspectos crear, resolver dónde encapsular los aspectos candidatos, decidir cuáles de estos últimos son espurios o si su encapsulamiento no es conveniente. La visualización puede ser utilizada en diferentes etapas del proceso de refactoring:

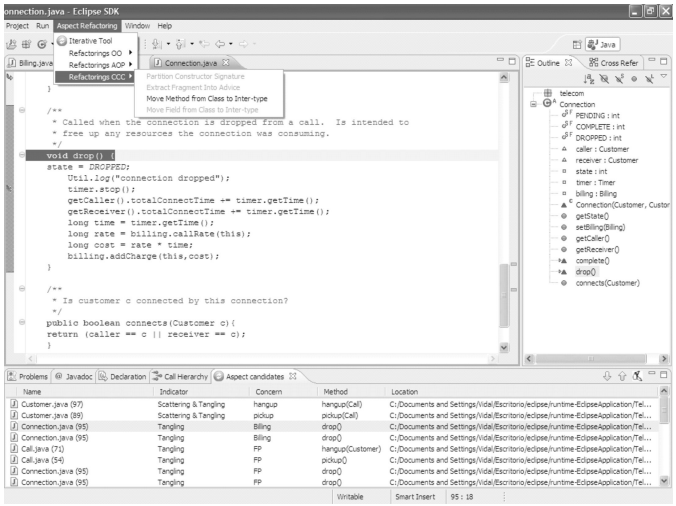


Figura 5. Análisis sobre posibles refactorings.

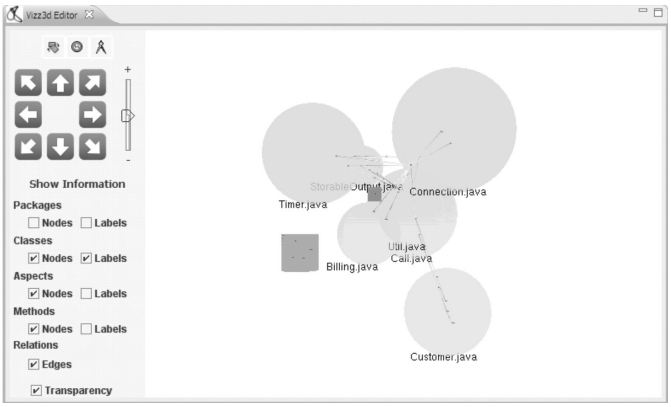


Figura 6. Visualización en 3D.

- Antes del proceso de migración para identificar la ubicación de los aspectos candidatos y las relaciones entre ellos.
- Durante el proceso de migración para visualizar cómo quedaría el sistema en caso de aplicar un determinado refactoring.
- Durante el proceso de migración para visualizar cómo se encuentra el sistema luego de haber aplicado un conjunto de refactorings e identificar aquellos aspectos candidatos que aún no se han encapsulado en aspectos.
- Luego del proceso de migración para visualizar el estado final del sistema orientado a aspectos.

Además, otra característica de la visualización es que ésta es navegable de forma tal que al pulsar sobre algún elemento del gráfico se accede al elemento en el código.

El gráfico 3D que se crea refleja la estructura de paquetes clases y aspectos conteniendo los siguientes elementos:

- Paquetes: estos se representan mediante cubos los cuales pueden contener otros paquetes, clases o aspectos.
- Clases: se representan mediante esferas. Dentro de una clase se presentan aquellos métodos en los cuales se han identificado aspectos candidatos.
- Aspectos: se grafican como cilindros, dentro de los mismos se incluyen los métodos, pointcuts y advices.
- Métodos, pointcuts y advices: se representan como conos dentro de las clases y aspectos.
- Relaciones: las relaciones se visualizan como rectas entre métodos. Existe una relación entre dos métodos cuando se ha especificado un aspecto candidato que es una invocación a un método de una clase. Este tipo de asociaciones se presentan especialmente cuando hay tangling [1].

Por ejemplo, en la Figura 6 se puede observar una visualización realizada durante el proceso de migración de un sistema con el fin de visualizar el estado del sistema. En la figura se observa un conjunto de clases que corresponden a un mismo paquete (por simplicidad los paquetes no se muestran en esta figura). Dentro de estas clases se encuentran uno o más métodos que han sido marcados como aspectizables por el proceso de aspect mining. Las relaciones en forma de flechas representan las invocaciones entre estos métodos. Además, se pueden observar dos aspectos los cuales contienen en su interior los métodos, pointcuts y advices que presentan.

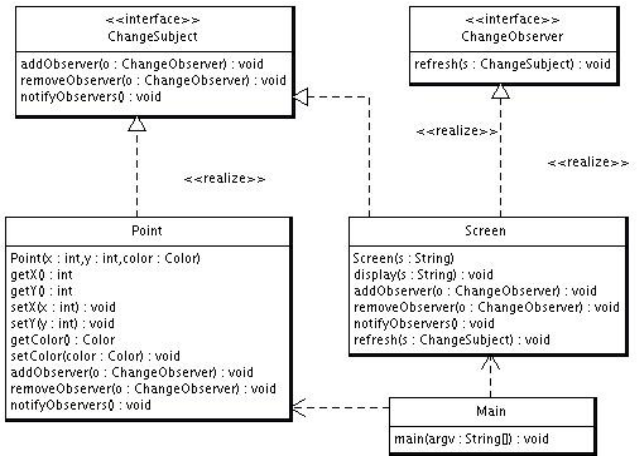


Figura 7. Diagrama de clases de UML para el caso de estudio

V. CASO DE ESTUDIO

En esta sección se presenta un caso de estudio (inicialmente presentado en [9]) en el cual se aplicará el enfoque propuesto para migrar sistemas orientados a objetos a sistemas orientados a aspectos. Esto ayudará a demostrar cómo se utiliza el proceso iterativo de refactorización.

El caso de estudio es el uso del patrón de diseño Observer [6] en una aplicación GUI sencilla. En esta implementación del Observer (Figura 7) la clase Point juega el rol del sujeto y la clase Screen juega ambos roles de sujeto y observer (de la clase Point y de si mismo).

La Figura 8 exhibe un fragmento de la clase Point en la cual existe un crosscutting concern debido a la inclusión de un arreglo de observers y la notificación a estos.

```
import java.awt.Color;
import java.util.HashSet;
import java.util.Iterator;

public class Point implements ChangeSubject {
    private HashSet observers;
    private int x;
    private int y;
    private Color color;

    public Point(int x, int y, Color color) {
         this.x = x;
         this.y = y;
         this.color=color;
         this.observers = new HashSet();
    }
    public void setX(int x) {
         this.x = x;
         notifyObservers();
    }
    public void setY(int y) {
         this.y = y;
         notifyObservers();
    }
    public void setColor(Color color) {
         this.color = color;
         notifyObservers();
    }
    public void addObserver(ChangeObserver o){
         this.observers.add(o);
    }
 public void removeObserver(ChangeObserver o){
         this.observers.remove(o);
    }
    public void notifyObservers() {
      for (Iterator e = observers.iterator() ;
            e.hasNext() ;) {
       (ChangeObserver)e.next()).refresh(this);
        }
    }
}
```

Figura 8. Clase Point.

A. *Obtener evidencias de código aspectizable*

Siguiendo el enfoque, se comienza obteniendo las evidencias del código aspectizable (Paso 1). Para esto se importa el archivo XML correspondiente al proyecto que se está analizando. Una vez hecho esto los aspectos candidatos son señalados en AspectRT como se muestra en la Figura 9.

Una vez importados los aspectos candidatos se comienza a iterar sobre los mismos comenzando por el primero de la lista o con el que el desarrollador desee. Aquí se encuentra, en un primer momento, el uso de la variable observers dentro del método addObserver. Con el fin de facilitar el análisis cuando

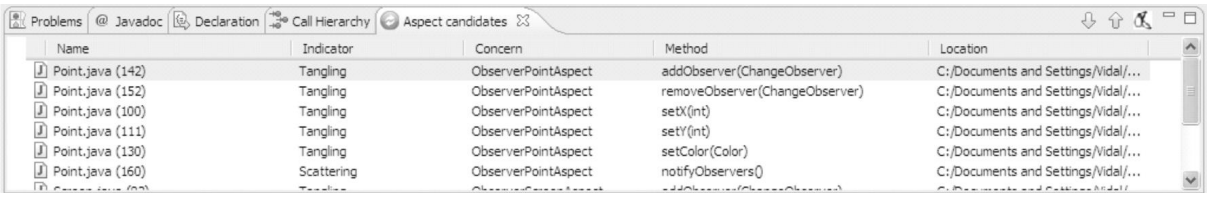
se selecciona el aspecto candidato a refactorizar la herramienta abre la clase donde se encuentra el código objetivo y señala el mismo resaltándolo.

B. Análisis de posibles refactorings CCC sobre el método addObserver

Al analizar los posibles refactorings CCC (Paso 2) la herramienta identifica el conjunto de refactorings que puede ser aplicado al aspecto candidato utilizando para ello los patrones de estructura. En este caso serán seleccionados los aspect refactorings relevantes a un método (Move Method from Class to Inter-type y Extract Feature into Aspect). Es aquí cuando el desarrollador debe seleccionar cuál de estos refactorings es más beneficioso; por ejemplo, en este caso la conveniencia o no de encapsular el código en un aspecto. Como puede observarse el código del método correspondiente a la variable observers es efectivamente un crosscutting concerns que debe ser encapsulado en un aspecto ya que esta no debe ser una responsabilidad de la clase Point. Por esta razón, se selecciona el aspect refactoring Extract Feature into Aspect [19] el cual tiene como objetivo general trasladar a un aspecto el código de una clase que presenta síntomas de scattering. Este aspect refactoring utiliza un conjunto de refactorings los cuales se deben aplicar dependiendo de ciertas condiciones, en este caso los que aplican son Move Field from Class to Inter-type y Move Method from Class to Inter-type [19].

C. Aplicación de refactorings CCC sobre el método addObserver

La aplicación del conjunto de refactorings identificados en el paso anterior corresponde al Paso 3 del proceso. El primero de estos refactorings (Move Field from Class to Inter-type) traslada la declaración de un field perteneciente a un concern a un aspecto como una declaración inter-type, en el ejemplo es la variable observers. El desarrollador también debe especificar el nombre del aspecto (se provee uno por defecto) y el paquete en el cuál se guardará ya sea creando uno nuevo o utilizando uno ya existente en este caso se crea un nuevo aspecto al cual se denomina ObserverPointAspect.



Name	Indicator	Concern	Method	Location
Point.java (142)	Tangling	ObserverPointAspect	addObserver(ChangeObserver)	C:/Documents and Settings/Vidal/...
Point.java (152)	Tangling	ObserverPointAspect	removeObserver(ChangeObserver)	C:/Documents and Settings/Vidal/...
Point.java (100)	Tangling	ObserverPointAspect	setX(int)	C:/Documents and Settings/Vidal/...
Point.java (111)	Tangling	ObserverPointAspect	setY(int)	C:/Documents and Settings/Vidal/...
Point.java (130)	Tangling	ObserverPointAspect	setColor(Color)	C:/Documents and Settings/Vidal/...
Point.java (160)	Scattering	ObserverPointAspect	notifyObservers()	C:/Documents and Settings/Vidal/...

Figura 9. Aspectos candidatos para el caso de estudio

Una vez trasladada la variable observers la herramienta aplica Move Method from Class to Inter-type para extraer al aspecto aquellos métodos que están relacionados con el concern. En el ejemplo, se trata del método addObserver el cual hace uso de la variable observers y es el que se encuentra bajo evaluación.

Adicionalmente, debido a que el método addObserver es implementado a partir de la interface ChangeSubject la herramienta propone la aplicación del refactoring Encapsulate Implements with Declare Parents buscando encapsular el rol que cumple la implementación de la interfaz mediante un aspecto, para esto se elimina la declaración implements de la clase y se la reemplaza por declare parents dentro del aspecto. El código resultante de estas reestructuraciones es mostrado en Fig. 11. Wizard para refactoring Extract Fragment Into Advice.

```
import java.util.HashSet;

public aspect ObserverPointAspect {

  declare parents : Point implements ChangeSubject;

  public HashSet<ChangeObserver> Point.observers;


  public void Point.addObserver(ChangeObserver o) {
        this.observers.add(o);
  }

}
```

Figura 10. Aplicación de los refactorings Move Field from Class to Inter-type, Move Method from Class to Inter-type y Encapsulate Implements with Declare Parents

D. Análisis de posibles refactorings AOP

Una vez que el código orientado a aspectos ha sido creado se analiza la posibilidad de aplicar uno o más refactorings AOP al aspecto resultante (Paso 5). Sin embargo, en este punto del desarrollo no se cuenta con información sobre los posibles fragmentos de código a refactorizar (como si ocurría en el Paso 2 debido al proceso de aspect mining) debido a que el código al que se quiere aplicar refactoring es el código orientado a aspectos resultante y no el original que se está migrando. Por esta razón, el análisis de los fragmentos de código en los cuales aplicar reestructuraciones será una tarea que deba realizar el desarrollador. La herramienta asistirá indicando qué aspect refactoring AOP son aplicables a un fragmento de código señalado y realizando automáticamente el aspect refactoring elegido por el desarrollador. En este caso no es necesario hacer reestructuraciones ya que la complejidad que presenta el aspecto ObserverPointAspect es muy básica. Por esta razón, se continúa con el proceso de migración retornando al Paso 1.

E. Iteración sobre la lista de aspectos candidatos

Al iterar sobre la lista de aspectos candidatos se obtiene el método setX (el método removeObserver es muy similar al caso de addObserver por eso no se detalla) el cual ha sido señalado como aspectizable debido a que presenta una llamada al método notifyObservers.

Al analizar los posibles refactorings CCC a ser aplicados sobre el código aspectizable (Paso 2) la herramienta identifica que el código aspectizable es código dentro de un método por lo que en el subconjunto de refactorings aplicables incluye Extract Feature into Aspect y Extract Fragment Into Advice [19]. El desarrollador selecciona el segundo de estos ya que sólo se debe trasladar al aspecto la llamada a notifyObservers en el método setX. Cuando la herramienta aplica las reestructuraciones crea un nuevo pointcut, al que el desarrollador denomina subjectChange, y un advice que lo referencia (Figura 11). En aquellos casos en los que ya exista

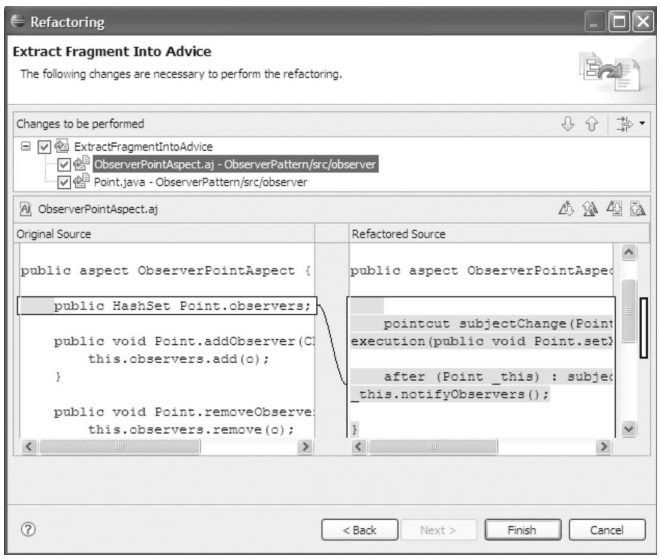


Figura 11. Wizard para refactoring Extract Fragment Into Advice.

Al analizar los posibles refactorings CCC a ser aplicados sobre el código aspectizable (Paso 2) la herramienta identifica que el código aspectizable es código dentro de un método por lo que en el subconjunto de refactorings aplicables incluye Extract Feature into Aspect y Extract Fragment Into Advice [19]. El desarrollador selecciona el segundo de estos ya que sólo se debe trasladar al aspecto la llamada a notifyObservers en el método setX. Cuando la herramienta aplica las reestructuraciones crea un nuevo pointcut, al que el desarrollador denomina subjectChange, y un advice que lo referencia (Figura 11). En aquellos casos en los que ya exista un pointcut en el aspecto con los mismos parámetros que se necesitan (por ejemplo, en este caso una variable de tipo Point) la herramienta sugiere que se reutilice el pointcut existente.

F. Análisis y aplicación de refactorings AOP

Finalmente, una vez aplicados los refactorings correspondientes al Paso 3 se pueden aplicar refactorings AOP sobre el código reestructurado como lo indica el Paso 5. Nuevamente, no son necesarios en este caso ya que la complejidad del aspecto no lo hace necesario, el aspecto sólo presenta un pointcut y un advice.

La refactorización continúa a partir del siguiente aspecto candidato de la lista. En este ejemplo, una vez que son recorridos todos los aspectos candidatos de la lista, el resultado es la creación de dos aspectos los cuales encapsulan los concerns ObserverPointAspect y ObserverScreenAspect de las clases Point y Screen respectivamente. Sin embargo, debido a la similitud que estos presentan se realiza la aplicación de refactorings AOP que generalizan estos en uno solo.

G Obtener evidencias de código aspectizable

Si bien este es un caso de estudio pequeño permite observar la complejidad del proceso de migración. Se refactorizaron dos crosscutting concerns que se encontraban en dos clases orientadas a objetos a un aspecto. En la Tabla 2 se presenta un resumen de la actividad de migración realizada.

De la Tabla 2 se puede inferir lo tediosa y compleja que resulta la tarea de refactoring. Dadas sólo dos clase y dos crosscutting concerns se han aplicado 27 refactorings. A pesar que el desarrollador tiene un rol importante en el proceso, la identificación de los posibles refactorings a aplicar es una tarea automática.

Otra métrica que puede mencionarse es, por ejemplo, CBO (Coupling between object classes) [2]. Esta métrica permite conocer el número de clases acopladas a una clase. Una clase esta acoplada a otra si utiliza métodos o variables de instancia de otra clase. La hipótesis de la métrica es que un valor alto disminuye el diseño modular y dificulta el reuso por lo que el acoplamiento debe mantenerse al mínimo para mejorar la modularidad y el encapsulamiento. En el caso de estudio se ha disminuido la CBO ya que se ha eliminó el acoplamiento con la interface ChangeSubject al encapsular el rol que esta representaba en un aspecto.

Tabla II. Aspect refactorings utilizados en la migración.

Aspect Refactoring	Tipo	N° de veces aplicado	Código aspectizable obtenido automáticamente
Extract feature into aspect	CCC	2	Si
Extract Fragment into advice	CCC	6	Si
Generalise Target Type with Marker Interface	AOP	2	No
Move field from class to inter-type	CCC	2	Si
Move Method	Aspect-Aware OO	6	Si
Pull Up Advice	AOP	2	No
Pull Up Field	Aspect-Aware OO	5	Si
Pull Up Pointcut	AOP	2	No
Total		27	78%

Además, es útil analizar el LoC (Line of code) [22]. Esta métrica permite conocer cuán complejo y propenso a errores es un programa según la cantidad de líneas de código que presenta. Generalmente, cuanto más grande sea el tamaño del código de un componente, más complejo y propenso a errores será. En el caso de estudio, el número total de líneas de código de todo el

sistema ha disminuido en un 6% debido a la presencia de código duplicado a pesar de la creación de los aspectos. Además, se ha disminuido en promedio un 40% el tamaño de la clase Employee disminuyendo su complejidad. Es decir, la cantidad de líneas de código total se ha disminuido al igual que la de los componente que se refactorizó.

VI. CONCLUSIONES

Al examinar la problemática del mantenimiento de sistemas orientados a objetos se observa que este es muy costoso. Con el fin de solucionar dichas problemáticas se ha planteado la necesidad de contar con sistemas que puedan ser adaptados a los cambios que surgirán durante la evolución del sistema y con técnicas que los realicen como el refactoring. Uno de los atributos arquitectónicos que permiten una mejora en el mantenimiento es la modularidad de los componentes, sin embargo, con las técnicas tradicionales de la ingeniería de software no es posible modularizar todos los concerns, principalmente aquellos que cortan transversalmente los componentes funcionales del sistema. Con este objetivo se presenta el desarrollo de software orientado a aspectos (DSOA) como un paradigma de programación que modulariza los sistemas de manera tal de mejorar su evolución y mantenimiento. Por esta razón, con el fin de aprovechar las ventajas de este paradigma, este trabajo presenta un proceso de migración de sistemas orientados a objetos (OO) existentes a la orientación a aspectos. Para realizar esta tarea se debe contar con mecanismos y estrategias de reestructuración de código, denominados aspect refactorings los cuales presentan un proceso similar al refactoring orientado a objetos que sea aplicable en la orientación a aspectos.

Este trabajo, presenta una propuesta de refactorización de sistemas orientados a objetos a sistemas orientados a aspectos el cual trata de automatizar el proceso de migración. Con este fin utiliza tres tipos de aspect refactoring (Refactorings Aspect-Aware OO, Refactorings de construcciones AOP y Refactorings de CCC) los cuales pueden ser utilizados en diferentes etapas de la migración de sistemas.

Con el fin de soportar el proceso de migración se desarrolló una herramienta llamada AspectRT la cual es un plugin para Eclipse que se integra con AspectJ. Esta herramienta toma como entrada un archivo XML el cual contiene la información de los aspectos candidatos y su ubicación en el código de la aplicación orientada a objetos. Luego, con la información de los aspectos candidatos se identifican automáticamente el o los posibles refactorings y con la intervención del desarrollador los refactorings son aplicados generando código AspectJ. En cualquier momento del desarrollo es posible observar en tres dimensiones el código de la aplicación (clases, aspectos, métodos).

Las principales ventajas del proceso son:

- Integración de los tres tipos de refactorings: Al complementar los distintos tipos de refactorings existentes no solo se encapsulan los crosscutting concerns (mediante refactorings CCC), que el desarrollador elige en un aspecto, sino que además, se mejora la estructura interna de los aspectos (mediante refactorings AOP) y de las clases (mediante refactorings Aspect-Aware OO y refactorings OO).
- Automatización: La migración asistida por herramientas automáticas es una gran ventaja ya que como se mostró en el caso de estudio el proceso de migración es muy complejo por lo que es una tarea muy dificultosa si no se cuenta con herramientas que asistan al desarrollador durante el mismo.
- Identificación automática de los posibles refactoring a aplicar: La herramienta realiza un análisis automático de los posibles refactorings a aplicar dados los aspectos candidatos identificados por el aspect mining. El desarrollador solo interviene para seleccionar refactorings a aplicar (de un conjunto identificado por AspectRT) y proporcionar nombres para los nuevos aspectos, pointcuts, métodos, etc.

Hasta el momento, el proceso ha sido testeado con tres casos de estudios pequeños (uno de ellos el presentado en este paper referente al patrón Observer) de alrededor de 100 líneas de código (LoC) cada uno. Los resultados demuestran las potencialidades del proceso reduciendo el acoplamiento entre clases y el LoC de las clases involucradas (en aproximadamente 40%), e icrementando la modularidad. Sin embargo, todavía existen algunos problemas y temas abiertos. Por ejemplo, como incluir identificación automática del código aspectual a ser refactorizado (Paso 5) y como asistir al desarrollador para decidir con respecto a que refactoring debe ser seleccionado (Paso 2).

Como trabajo futuro se prevé testear el proceso con sistemas más complejos. Además, se definirán estrategias y mecanismos que soporten dinámicamente la identificación de refactorings AOP.

REFERENCIAS

[1]E.S. Abait, S.A. Vidal and C.A. Marcos. Dynamic Analysis and Association Rules for Aspects Identification. II Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2008), Campinas, Brasil, 2008.

[2]S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng., 1994.

[3]S. Demeyer, P. Ducasse and O. Nierstrasz. Object Oriented Reengineering Patterns. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2002.

[4]Eclipse home page, www.eclipse.org.

[5]M. Fowler. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[6]E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design patterns - Elements of reusable object-oriented software. Professional Computing Series. Addison Wesley, 1995.

[7]S. Hanenberg, C. Oberschulte and R. Unland. Refactoring of aspect-oriented software. In 4th International Conf. on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World, pages 19-35, Erfurt, Germany, 2003.

[8]J. Hannemann. Aspect-Oriented Refactoring: Classification and Challenges. Workshop on Linking Aspect Technology and Evolution (LATE'06). 5th International Conference on Aspect-Oriented Software Development (AOSD'06), Bonn, Germany, 2006.

[9]J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, pages 161-173. ACM Press, 2002.

[10]J. Hannemann, G.C. Murphy and G. Kiczales. Role-based refactoring of crosscutting concerns. In Proceedings of the 4th international conference on Aspect-oriented software development, pages 135-146, Chicago, Illinois. ACM Press, 2005.

[11]M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In Proc. of 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, USA, 2003.

[12] A. Kellens and K. Mens. A survey of aspect mining tools and techniques. Technical Report 2005-08, INGI, UCL, Belgium, 2005.

[13]A. Kellens, K. Mens and P. Tonella. A Survey of Automated Code-Level Aspect Mining Techniques. Transactions on Aspect-Oriented Software Development IV, LNCS 4640, pages 143-162. Springer Verlag, 2007.

[14]G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin. Aspect-Oriented Programming. In Proceedings of the 11th European Conference on Object-Oriented Programming, pages 220-242, 1997.

[15]R. Laddad. I want my AOP. Part 1, 2, 3 from JavaWorld, Enero-Marzo-Abril, 2002.

[16]M.M. Lehman and L.A. Belady. Program evolution: processes of software change. Academic Press Professional, Inc., 1985.

[17]W. Löwe and T. Panas. Rapid Construction of Software Comprehension Tools. International Journal of Software Engineering and Knowledge Engineering. Special Issue on Maturing the Practice of Software Artefacts Comprehension. 2005.

[18]M. Marin, L. Moonen and A. Van Deursen. An approach to aspect refactoring based on crosscutting concern types. In Proceedings of the 2005 workshop on Modeling and analysis of concerns in software, pages 1-5, St. Louis, Missouri. ACM Press, 2005.

[19]M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.

[20]M.P. Monteiro and J.M. Fernandes. Towards a catalog of aspect-oriented refactorings. In Proceedings of the 4th international conference on Aspect-oriented software development, pages 111-122, Chicago, Illinois. ACM Press, 2005.

[21]I. Sommerville. Software Engineering, 7th ed. Addison Wesley, 2005.

[22]D.P. Tegarden, S.D. Sheetz and D.E. Monarchi. Effectiveness of Traditional Metrics for Object-Oriented Systems. In Proceedings 25th Hawaii International Conference on System Sciences, pp. 359-368, Kauai, Hawaii, 1992.

[23]The AspectJ Team. The AspectJ Programming Guide. 2004.

[24]J. Wloka. Refactoring in the Presence of Aspects. 13th Workshop for PhD Students in Object-Oriented Systems (PhDOOS), at ECOOP '03, Darmstadt, Germany, 2003.

Universidad Nacional de Colombia Sede Medellín
Facultad de Minas
Escuela de Ingeniería de Sistemas

Grupos de Investigación

Grupo de Investigación en Sistemas e Informática

Categoría A de Excelencia Colciencias
2004 - 2006 y 2000.

GIDIA: Grupo de Investigación y Desarrollo en Inteligencia Artificial

Categoría A de Excelencia Colciencias
2006 – 2009.

Grupo de Ingeniería de Software

Categoría C Colciencias 2006.

Grupo de Finanzas Computacionales

Categoría C Colciencias 2006.

Centro de Excelencia en Complejidad

Colciencias 2006

Escuela de Ingeniería de Sistemas
Dirección Postal:
Carrera 80 No. 65 - 223 Bloque M8A
Facultad de Minas. Medellín - Colombia
Tel: (574) 4255350 Fax: (574) 4255365
Email: esistema@unalmed.edu.co
http://pisis.unalmed.edu.co/