Aplicaciones con Aspectos: de Cambios y sus Consecuencias

Sandra Casas, Héctor Reinaga, y Cecilia Fuentes

Universidad Nacional de la Patagonia Austral (UARG)
Campus Universitario, Av. Gregores y Piloto Lero Rivera
C.P. 9400, Río Gallegos, Argentina
scasas@unpa.edu.ar

Abstract. La AOP propone fundamentalmente una nueva clase de modularización llamada aspectos, cuyo mecanismo de invocación implícita, requiere que se especifiquen donde o cuando deben ser invocados. En particular, este mecanismo de invocación implícita introduce una capa adicional de complejidad en la construcción de un sistema. Esto puede hacer dura la comprensión de cómo y cuando el sistema base y los aspectos interactúan y así como el sistema se comportará. Además, cambios aparentemente inocentes del código base pueden conducir a comportamientos erróneos y no intencionales. Ya que es fácil perder el rastro de la característica global de cómo el código base interactúa con los aspectos. Puede volverse difícil identificar el código de tal comportamiento no anticipado. Este trabajo propone un modelo que permita anticipar las consecuencias ante eventuales operaciones de cambio.

Keywords: Programación Orientada a Aspectos, Separación de Concerns, Operaciones de Cambio, Evolución de Software, AspectJ.

1 Introducción

Las técnicas AOSD [1] proveen medios sistemáticos para identificar, modularizar, representar y componer crosscutting concerns (CCC) [2]. El término CCC refiere a aquellos factores o funcionalidades del software de calidad que no pueden ser efectivamente modularizados usando técnicas de desarrollo de software tradicionales y generan código mezclado y enmarañado.

La AOP [3] propone fundamentalmente una nueva clase de modularización que va más allá de los procedimientos generalizados: los aspectos. Un aspecto es un módulo que puede localizar la implementación de un CCC. La clave para esta técnica de modularización, radica en el mecanismo de composición del módulo. Las subrutinas explícitamente invocan el comportamiento implementado por otras subrutinas. En contraste, los aspectos tienen un mecanismo de invocación implícita. El comportamiento de un aspecto es implícitamente invocado en la implementación de otros módulos. Consecuentemente la implementación de estos módulos puede hacerse sin considerar a los CCC. Adicionalmente, la influencia de los aspectos respecto de

una clase base dada, no es visible en el código como consecuencia de la propiedad de obliviousness [4] de la AOP.

El mecanismo de invocación implícita requiere que el aspecto especifique donde o cuando debe ser invocado. La implementación de un aspecto consecuentemente consiste de dos partes conceptuales diferentes: el código funcional del aspecto (denominado advice) y el código de aplicación del aspecto (denominado pointcut).

La funcionalidad del aspecto no es esencialmente diferente del código "regular" y es ejecutado cuando el aspecto es invocado. Esta invocación del aspecto es determinada por el código de aplicabilidad del aspecto. Este código contiene enunciados que especifican donde o cuando el aspecto debe ser invocado. En la terminología estándar de AOSD [5], el código de aplicabilidad se refiere como expresión de "pointcut", y el código funcional del aspecto es referido como "advice". Así, un simple aspecto puede consistir de múltiples y diferentes funcionalidades que necesita que sean invocadas desde varios y distintos lugares (join-points) en el código, la implementación de un aspecto puede consistir de varios pointcuts y advices.

Los aspectos modularizan CCC mediante la encapsulación, no solo del comportamiento sino también de donde y cuando deben ser invocados. Como resultado, los otros módulos del sistema, llamados código base, realizan invocaciones implícitas al comportamiento de los aspectos. Primero, el flujo de ejecución de la aplicación base es reificado como una secuencia de join-points. Segundo, la especificación de las invocaciones implícitas es hecha a través de un pointcut que selecciona a cual join-point de los aspectos ejecutar. La especificación del comportamiento del aspecto es llamada advice. Un aspecto puede contener varios pointcuts y advice, donde cada advice es asociado con un pointcut. El concepto de pointcuts y advice, abre nuevas posibilidades entre el código base y el CCC.

Sin embargo esta separación (pointcuts y advice) hace más difícil al desarrollador evaluar el comportamiento del sistema. En particular, el mecanismo de invocación implícita introduce una capa adicional de complejidad en la construcción de un sistema. Esto puede hacer dura la comprensión de cómo y cuando el sistema base y los aspectos interactúan y así como el sistema se comportará.

Además, cambios aparentemente inocentes del código base pueden conducir a comportamientos erróneos y no intencionales. Ya que es fácil perder el rastro de la característica global de cómo el código base interactúa con los aspectos. Puede volverse difícil identificar el código de tal comportamiento no anticipado.

En este trabajo planteamos como las características de la AOP mencionadas, inciden negativamente en la evolución y mantenimiento de aplicaciones AOP (Sección 2); analizamos los factores de AOP que obstaculizan en términos de operaciones de cambio y sus potenciales efectos como falsos positivos y negativos (Sección 3); y proponemos un modelo que permita anticipar las consecuencias ante eventuales operaciones de cambio. Este trabajo se enfoca en el análisis de elementos de código fuente, es decir, se centra en gran medida al nivel de implementación, por lo que se circunscribe específicamente al lenguaje AOP AspectJ [6].

2 Motivaciones

Cuando se escribe la definición de un pointcut, no siempre es claro para el desarrollador, donde el aspecto va a intervenir en el código base. Esto puede llevar a situaciones en las cuales el pointcut captura demasiados join-points (falsos positivos) o donde ciertos join-points que se intentan capturar, no lo sean (falsos negativos). Un ejemplo claro es aportado en el trabajo de Coelho [7]. Los autores investigan un número de aplicaciones que usan aspectos para determinar comportamientos erróneos relacionados al manejo de excepciones. Encuentran que aunque los desarrolladores eran expertos en el uso de aspectos, los falsos positivos y falsos negativos ocurren. Específicamente "errores en la expresiones de pointcuts" fueron encontradas en las aplicaciones Health Watcher [8] y Mobile Photo [9], los cuales son ambas casos de estudio bien conocidos en AOSD.

Por otro lado, se advierte que cambios aparentemente inocentes del código base pueden conducir a comportamientos erróneos y no intencionales. Esto se debe a que es relativamente fácil perder el rastro de la característica global de cómo el código base interactúa con los aspectos, y así puede volverse difícil identificar el código de tal comportamiento no anticipado. Una variante de este problema, es el problema llamado "fragile pointcuts" [10] [11], el cual deja de manifiesto el alto acoplamiento que existe entre los aspectos y el código base. Varios lenguajes AOP, como AspectJ [6], ofrecen constructores llamados "comodines", para reducir el acoplamiento. Sin embargo este mecanismo introduce nuevos problemas. Al utilizar estos constructores, un pointcut queda obligado a sostener y aplicar un uso de convenciones de nombres. Como tales convenciones no son chequeadas por el compilador, nunca está garantizado su cumplimiento. Como resultado, los programadores deben tener mucho cuidado con los pointcuts para evitar emparejar (interceptar) join-points espurios o lo contrario (falsos positivos), perder la intercepción de los join-points necesarios (falsos negativos).

En la Fig. 1, el aspecto LogCambiosPosicion, implementa el mecanismo de log para registrar los cambios de posición de los objetos de tipo Punto y Linea.

```
public aspect LogCambiosPosicion {
   pointcut cambioPosicionPunto(Punto p): call(* Punto.set*(int))&&
target(p);
   after(Punto p): cambioPosicionPunto(p) {
      Logger.writeLog("Cambio posición Punto: "+p.toString());
   }
   pointcut cambioPosicionLinea(Linea l): call( * Linea.set*(Punto) ) &&
target(l);
   after(Linea l): cambioPosicionLinea(l) {
      Logger.writeLog("Cambio posición Linea: "+l.toString());
   }
}
```

Fig. 1. Aspecto LogCambiosPosicion.

Tomando este código de ejemplo se comprueba que:

- Si se renombra el tipo Punto por MiPunto, entonces la intercepción de {call (void Punto.set*(int) } es vacío y se producen falsos negativos.

- Si se cambia la signatura de void setX(int) por void setX(double) en la clase Punto, entonces {call (void Punto.set*(int)} se rompe y se producen falsos negativos.
- Suponiendo que originalmente todas las clases están en un paquete llamado example, y un designador de pointcut es usado (example.*). Si Punto se mueve a otro paquete, este designador se rompe y se producen falsos negativos.
- Si se agrega cualquier atributo a las clases Punto y/o Linea, que no refiera a la posición y luego se agrega el consecuente método set para modificar su estado, las llamadas a este método serán interceptadas por alguno de los pointcuts del aspecto LogCambiosPosicion, y se producen falsos positivos.

El problema de los pointcuts frágiles obstaculiza la evolución del software que usa aspectos.

3 Análisis de las Operaciones de Cambio

Las operaciones de cambio representan la evolución del software. Son las acciones que llevan adelante los desarrolladores cuando modifican el código fuente. Las operaciones de cambio representan la transición desde un estado de evolución de un sistema, al próximo. Ejemplos de operaciones de cambio son: agregar una clase, renombrar un método, o aplicar algún refactoring [12]. Las operaciones de cambio se clasifican en "atómicas" y "compuestas". Las operaciones atómicas implican una única acción indivisible, como por ejemplo, "remover un método". Las operaciones de cambio compuestas resultan ser una secuencia de operaciones de cambio atómicas, como por ejemplo, un refactoring. También han sido distinguidas en sintácticas y semánticas [13]. En este caso consideramos una categorización de las mismas, según el lugar en que se hacen, siendo en el código base y aquellas que se hacen en los aspectos. Nos importan las operaciones de cambio en tanto son potencialmente generadoras de falsos positivos y/o falsos negativos en las aplicaciones que usan aspectos, y por ende su identificación resulta imprescindible. A continuación planteamos que es posible establecer relaciones entre la operación de cambio, las expresiones de pointcuts de los aspectos y la potencial ocurrencia de falsos positivos y/o negativos.

3.1 Operaciones de Cambio en el Código Base

AspectJ proporciona una serie de descriptores de pointcuts que permiten identificar grupos de join-points que cumplen diferentes criterios. Estos descriptores se clasifican en diferentes grupos:

Basados en las categorías de join-points (G1): capturan los join-points según la categoría a la que pertenecen: call(methodSignature), Llamada a método - execution(methodSignature), Ejecución de método - call(constructorSignature), Llamada a constructor - execute(constructorSignature), Ejecución de constructor - get(fieldSignature), Lectura de atributo - set(fieldSignature), Asignación de atributo - handler(typeSignature), Ejecución de manejador - staticinitialization(typeSignature), Inicialización de clase - initialization(constructorSignature), Inicialización de objeto

preinitialization(constructorSignature), Pre-inicialización de objeto - adviceexecution(), Ejecución de un punto de ejecución.

Basados en el flujo de control (G2): capturan join-points de cualquier categoría siempre y cuando ocurran en el contexto de otro punto de corte. Estos descriptores son cflow y cflowbelow.

Basados en la localización de código (G3): capturan join-points de cualquier categoría que se localizan en ciertos fragmentos de código, por ejemplo, dentro de una clase o dentro del cuerpo de un método. Estos descriptores son within y withincode.

Basados en los objetos en tiempo de ejecución (G4): capturan los join-points cuyo objeto actual (this) u objeto destino (target) son de un cierto tipo. Además de capturar los join-points asociados con los objetos referenciados, permite exponer el contexto de los join-points.

Basados en los argumentos del punto de enlace (G5): capturan los join-points cuyos argumentos son de un cierto tipo mediante el descriptor args. También puede ser usado para exponer el contexto.

Basados en condiciones (G6): capturan join-points basándose en alguna condición usando el descriptor if(expresionBooleana).

Otra cuestión del planteo a tener en cuenta son los comodines. En las declaraciones de pointcuts pueden expresarse los join-points mediante un conjunto de comodines que permiten identificar join-points que tienen características comunes. El significado dependerá del contexto en que aparezcan:

- ***** : el asterisco en algunos contextos significa cualquier número de caracteres excepto un punto y en otros representa cualquier tipo (paquete, clase, interfaz, tipo primitivo o aspecto)
- .. : el carácter dos puntos representa cualquier número de caracteres, incluido el punto. Cuando se usa para indicar los parámetros de un método, significa que el método puede tener un número y tipo de parámetros arbitrario.
- **+** : el operador adición representa una clase y todos sus descendientes, tanto directos como indirectos.

En la Fig. 1 se presentó un simple ejemplo de cómo posibles operaciones de cambio simples generan falsos positivos y negativos cuando se usan aspectos. De manera más completa, en la Tabla 1, se han identificado un conjunto de operaciones de cambio típicas, el grupo de descriptor de pointcuts en el que pueden incidir y su potencial consecuencia en términos de falsos positivos y falsos negativos. Por ejemplo, la operación “Remove method” incide en el grupo G1, en particular en los designadores que refieren a los métodos (call – execution, etc.), y se indica que se producirán falsos negativos si un determinado método que se pretende remover es alcanzado por algún pointcuts existente.

Tabla 1. Análisis de operaciones de cambio en el código base.

Operación de Cambio	Grupo						Falso Positivo	Falso Negativo
	G1	G2	G3	G4	G5	G6		
Add package	X	X	X	X	X	X	+	
Add class	X	X	X	X	X	X	+	
Add method	X	X	X	-	-	-	+	
Add field	X	X	X	-	-	-	+	
Add handler	X	X	X	-	-	-	+	
Add message	X	X	X	-	-	-	+	
Remove package	X	X	X	X	X	X		+
Remove class	X	X	X	X	X	X		+
Remove method	X	X	X	-	-	-		+
Remove field	X	X	X	-	-	-		+
Remove message	X	X	X	-	-	-		+
Remove handler	X	X	X	-	-	-		+
Move package	X	X	X	X	X	X	+	+
Move class	X	X	X	X	X	X	+	+
Move method	X	X	X	-	-	-	+	+
Move field	X	X	X	-	-	-	+	+
Move handler	X	X	X	-	-	-	+	+
Move message	X	X	X	-	-	-	+	+
Rename Package	X	X	X	X	X	X	+	+
Rename class	X	X	X	X	X	X	+	+
Rename method	X	X	X	-	-	-	+	+
Rename field	X	X	X	-	-	-	+	+

Las operaciones de cambio citadas tienen en el grupo G1 su principal impacto, considerando que por su semántica siempre refieren a elementos de programa como identificador de clase, método u atributo. Sumado al hecho que prácticamente todo pointcut debe definirse en torno a algún designador de este grupo. Por el contrario, en comparación el grupo G6, resulta ser mucho menos usado y aunque pudiera hacer mención en su definición a algún elemento de programa, dicho uso no es obligatorio.

3.2 Operaciones de Cambio en los Aspectos

En los aspectos pueden ocurrir las mismas operaciones de cambio atómicas citadas en las Tabla 1, pero además aquellas que operen sobre los mecanismos de composición que le son propios a los aspectos. En la Tabla 2, se cita la operación de cambio, en que módulo (código base – aspecto) puede incidir y si en tal caso puede producir potenciales falsos positivos y/o negativos.

Tabla 2. Análisis de operaciones de cambio en los aspectos.

Operación de cambio	Código Base	Aspecto	Falso Positivo	Falso Negativo
Add pointcut	X	X	+	-
Remove pointcut	X	X	-	+
Rename pointcut	-	X	-	-
Add declare parents	X	X	+	+
Remove declare parents	X	X	-	+
Change declare parents	X	X	+	+
Change pointcut designator	X	X	+	+
Change join-point	X	X	+	+
Change advice	X	X	-	-

Las consecuencias de la operación “Change pointcut designator” son realmente inesperadas en términos de falsos positivos/negativos, si se considera que puede consistir en cambiar el designador (por ejemplo “call” por “execution” o viceversa) o agregar/remover una restricción de localización (“whithin-withincode”). Similar situación ocurre con la operación “Change join-point”, si esta se da tanto al reemplazar un comodín por una definición explícita o viceversa,

4 Modelo de Anticipación de Cambios

Como se ha indicado en las Tablas 1 y 2, es factible que se produzcan falsos positivos y/o negativos ante cambios aparentemente inocentes en algún programa base o la modificación de pointcuts. En estas situaciones el desarrollar debe identificar estos falsos positivos y/o negativos y resolver el problema. Sin embargo esta identificación manual de falsos positivos y/o negativos o su causa, no resulta una tarea trivial, si se trata de una aplicación de mediana a gran envergadura, y si además se hace a posteriori del cambio introducido. Requiere que los desarrolladores procedan a realizar exhaustivos análisis e inspecciones del código fuente, e intensas y sendas ejecuciones de casos de prueba. Estas tareas afectan fuertemente el tiempo que requiere el mantenimiento, el cual se incrementa considerablemente. En estos escenarios, plantear técnicas y herramientas que reduzcan los esfuerzos y costos del

mantenimiento de aplicaciones que usan aspectos, parecen conducentes para mejorar la productividad.

Proponemos un modelo de anticipación que cumpla con los siguientes objetivos:

1) **Identificar** las consecuencias de realizar un cambio en una aplicación con aspectos.
2) **Cuantificar** estas consecuencias mediante métricas que faciliten el análisis al desarrollador.
3) **Cualificar** las consecuencias y relacionar estas descripciones con la información cuantitativa proporcionada.

En la Fig. 2, se establece el flujo de trabajo que plantea el modelo que proponemos. Este consiste de algunos pasos básicos: primero el desarrollador identifica la operación de cambio que debe realizar, luego somete la misma al análisis de un sistema que da soporte al modelo. Como resultado del análisis, el sistema le proporciona al desarrollador información sobre las consecuencias que dicho cambio tendrá. Por último, y de acuerdo a los resultados obtenidos, el desarrollador continúa analizando los cambios a realizar o procede a efectuar esos u otros cambios en el código fuente que corresponde al software real. Dichos cambios generan la actualización del repositorio de programas del sistema que da soporte al modelo.

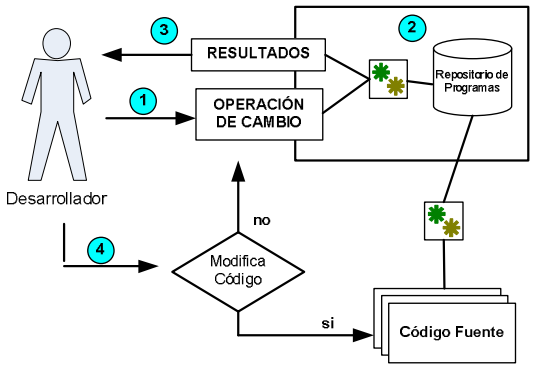

Fig. 2. Flujo de trabajo del desarrollador.

El modelo de anticipación se compone en principio de los siguientes componentes básicos:

Repositorio de programas. La estructura de los programas, sus relaciones, sus elementos constituyentes, etc., son representados y almacenados en un repositorio. Este es actualizado con la última versión de la aplicación, mediante un proceso automático que se puede hacer a través del IDE o un parser. En el caso particular que nos ocupa (Java y ApsectJ) las principales entidades resultan ser package, class, interface, aspect, method, atributte, pointcut y advice. Sobre estos elementos de programa es posible definir múltiples y distintos tipos de relaciones, como ser:

Relaciones de propiedad. Las que existen entre package, class, interface y aspect; como entre class, method y atributte; entre otras entidades.

Relaciones de reuso. las dadas por la herencia e implementación entre interfaces, class, y aspects.

Relaciones de composición. Donde se distinguen el envío de mensajes, y las establecidas por los pointcuts-advice y los métodos de las clases, de precedencia entre aspectos.

Una estructura que refleja estas relaciones se presenta en el esquema de la Fig. 3.

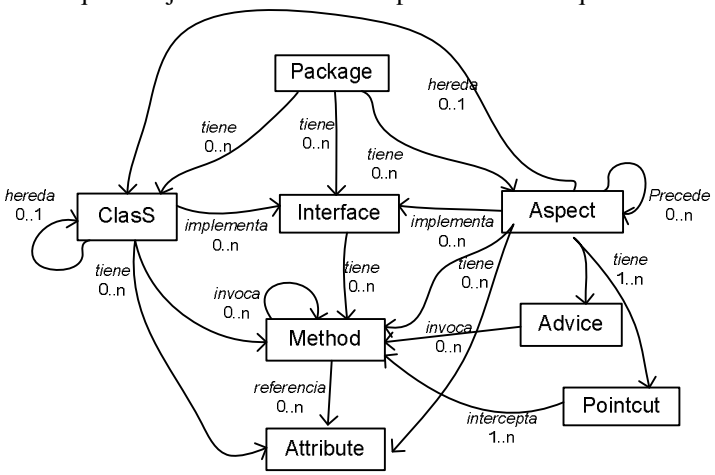

Fig. 3. Estructura de programas. Entidades y relaciones.

4.1 Operaciones de Cambio.

Desde nuestra perspectiva, una operación de cambio es una función que se aplica a una representación del código fuente, con el objeto de conocer anticipadamente sus potenciales consecuencias. Independientemente de la naturaleza del mantenimiento (correctivo, adaptativo, perfectivo, preventivo) se sabe que un conjunto de operaciones de cambio tendrá lugar en el código fuente, lo que conformará una nueva versión de la aplicación software. La cantidad de operaciones de cambio aplicadas en cada versión, dependerá de múltiples factores como cantidad y tipo de requerimientos, experiencia del desarrollador, lenguaje de programación, etc.

Desde el enfoque que proponemos la operación de cambio se aplicará al repositorio R_n. Las distintas instancias de $R = \{R_1, R_2, \ldots, R_n\}$ se presentan coincidentemente con las versiones de programa, $V = \{V_1, V_2, \ldots, V_n\}$. Esto produce que una determinada operación si es aplicada a diferentes instancias de R pueda generar distintas consecuencias.

Al definir una operación de cambio como una función es necesario especificar en cada caso el conjunto de entrada y de salida, de manera general:

```
<operación_de_cambio> :: <entradas> → <consecuencias>
```

Así por ejemplo:

```
addClass :: nueva_clase, identificador_paquete → falsos_positivos
changeJoinPoint ::
identificador_pointcut,nuevo_joinpoint→falsos_positivos,falsos_negativos
```

La operación de cambio “Add class” requiere el identificador de la nueva clase y el paquete al que se va a añadir. El resultado es un conjunto de falsos positivos que se producen. Para la operación “Change join-points”, se requiere el identificador del

pointcut y la nueva expresión de join-point, de igual forma el conjunto de falsos positivos y negativos que puedan producir. Pero la mayor complejidad esta dada en las operaciones de cambio compuestas, principalmente aquellas que devienen de la aplicación de refactorings. Aquí importa identificar la secuencia de operaciones de cambio que las componen en su preciso orden de aplicación. Lo que posibilita que una operación de cambio compuesta sea una secuencia de operaciones de cambio atómicas y/o compuestas, es que sus entradas y salidas están determinadas por las operaciones de cambio que pertenecen al conjunto definido. Las definiciones previas se pueden completar con:

```
<operación_de_cambio> ::= <operación_de_cambio_atomica> |
                          <operación_de_cambio_compuesta>
                          → <consecuencias>
<operación_de_cambio_compuesta> ::= {<operación_de_cambio_atomica> |
                                     <operación_de_cambio_compuesta>}
                                     → <consecuencias>
```

Siguiendo este esquema, la operación de cambio "Move method" es compuesta, ya que resulta de la secuencia de aplicar las operaciones de cambio atómicas "Remove method" y "Add method". Lo cual puede ser especificado de la siguiente manera:

```
moveMethod(id_method from_class to_class) ::
                                  removeMethod(id_method from_class),
                                  addMethod(id_method to_class)
                                  → falsos_positivos,falsos_negativos
```

4.2 Visualización Anticipada de Resultados.

Tal como se ha indicado en las Tablas 1 y 2, una operación de cambio tiene consecuencias. Interesa al modelo cuantificar estas consecuencias mediante métricas que faciliten el análisis al desarrollador y a la vez cualificar las consecuencias y relacionar estas descripciones con la información cuantitativa proporcionada. Por cada operación de cambio y de acuerdo a sus potenciales consecuencias, deberá presentarse información clara y bien organizada que informe:

Para las operaciones de cambio localizadas en el código base:

- Falsos negativos: se informará la cantidad y el detalle de cada uno de estos (identificador de aspecto, pointcut y advice desactivado).
- Falsos positivos: se informará la cantidad y el detalle de cada uno de estos (identificador de aspecto, pointcut y advice activado).

Para las operaciones de cambio localizadas en los aspectos:

- Falsos negativos: se informará la cantidad y el detalle de cada uno de estos (identificador de elemento de código base desactivado).
- Falsos positivos: se informará la cantidad y el detalle de cada uno de estos (identificador de elemento de código base activado).
- Inalterables: elementos de código base que eran interceptados antes del cambio y continuarían siendo interceptados luego del cambio.

5 Trabajos Relacionados.

Nuestra propuesta guarda estrecha relación con el enfoque de evolución de software basado en cambios (CBSE) [13, 14, 15, 16, 17], en el que nos hemos inspirado. CBSE surge en la última década, a partir de los trabajos de Robbes y Lanza, como enfoque contrapuesto y superador del Software para la Gestión de la Configuración como [18] y [19]. El objetivo de este nuevo enfoque es modelar con mayor precisión, como el software evoluciona tratando a los cambios como entidades de primera clase. Una diferencia que se observa con nuestra propuesta es el propósito del modelo CBSE, el cual define la historia de un programa como la secuencia de cambios que el programa atravesó. A partir de esta historia de cambios se puede reconstruir cada estado sucesivo del código fuente de los programas. Encontramos otras diferencias como ser: mientras que para CBSE las operaciones de cambio se establecen como entidades de primera clase, para la anticipación se definen como funciones; el éxito del modelo CBSE requiere que pueda ser implementado en los IDES o herramienta de desarrollo, mientras que el modelo de anticipación puede implantarse en estos IDES o en herramientas específicas. El trabajo abarcó solo aplicaciones OO (Java y Smalltalk) y no fue considerado aún el uso de aspectos, aunque suponemos que la extensión del modelo a aspectos es pausible.

La tesis de Störzer [20] trata en extenso el análisis de impacto de AspectJ. El abordaje es planteado en dos fases. Primero aplica una combinación de métodos de análisis de impacto dinámicos y estáticos a nivel sentencia para tratar los problemas específicos introducidos por la AOP. Basado en el grafo de llamadas e información "delta" estructurada, en particular se determina el impacto de la operación "Add aspect". Se efectúa una comparación estructurada de la versión original de un programa y su versión editada de la cual se deriva el conjunto de cambios atómicos que las diferencian (análisis estático). A partir de este conjunto de cambios identificados se hacen determinados juegos de test cuya ejecución constituyen el análisis dinámico. La segunda fase, cuya idea básica es que los programadores puedan ser alertados de los cambios en la intercepción del comportamiento de los advices (el esperado vs. el obtenido) si el sistema base subyacente cambia. Para ello emplea una técnica denominada "Análisis delta pointcuts". Para derivar los deltas pointcuts se calcula el conjunto de join-points de las dos versiones del programa (versión original y versión editada) y se comparan los conjuntos resultado, produciendo la información delta por los intercepciones de los pointcuts. Siguiendo el enfoque de la fase uno, se calculan los grafos de dependencias para cada versión, lo que constituye el análisis estático. Se completa con el análisis dinámico mediante ABC test Suite.

Este trabajo tiene una apoyatura formal fuerte y el problema que trata es perfectamente analizado en su descripción, lo que constituye una excelente base de partida para cualquier estudio. El objetivo del autor coincide con el nuestro, en cuanto a que pretende lograr la identificación de las consecuencias de un cambio y que los programadores puedan ser alertados de los cambios en la intercepción del comportamiento de los advices. La principales diferencias radican en: a) se plantea un proceso que se basa totalmente en la comparación de versiones de programas, lo que

implica que el impacto se analiza y detecta a posterior del cambio, mientras que nosotros proponemos analizar los cambios sobre un repositorio que represente a los programas "antes" de efectuar los cambios en el código; b) dado que trabaja con versiones de programas, solo puede hallar diferencias y consecuencias en términos de operaciones atómicas, no pudiendo definir si estas devienen de operaciones compuestas como son los refactoring. Todas las operaciones da cambio atómicas halladas se ubican así al mismo nivel, resultando realmente complejo analizarlas de manera mas relacionadas y que esto además tenga mayor significado para el desarrollador.

6 Conclusiones.

CBSE busca poder fácilmente volver atrás en el tiempo, al mantener la historia de la evolución a partir de los cambios, y así que el "undo - reundo" mantenga información semántica de los cambios realizados y resulte por ende menos costoso para el desarrollador este rastreo. La pregunta que nos planteamos es porque fue necesario deshacer un cambio, porque fue necesario volver a una versión anterior. Aunque las respuestas son muchas, se puede coincidir en que algo no esta funcionando como se esperaba cuando se introdujo dicho cambio, que se desea deshacer. Nuestra propuesta tiene aquí su razón de ser, ya que el objetivo radica en evitar o al menos disminuir la cantidad de veces que sea necesario deshacer cambios o volver a versiones anteriores de los programas. Y finalmente, hacer las tareas de mantenimiento y evolución del software que usan aspectos, mas predecibles y menos costosas.

Nuestro trabajo actual se enfoca a dar implementación a los cambios de las Tablas 1 y 2 y luego continuar con operaciones más complejas como refactorings.

Referencias.

[1] Aspect-Oriented Software Development (2004), http://aosd.net

[2] Rashid A., Moreira A., Araujo J.: Modularisation and Composition of Aspectual Requirements. In: 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston, USA (2003)

[3] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J.: Aspect-oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag (1997)

[4] Filman R., Friedman D.: Aspect-Oriented Programming is Quantification and Obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA (2000)

[5] Berg K., Conejero J., Chitchyan R.: AOSD Ontology 1.0-Public Ontology of Aspect Orientation. Report of the EU Network of Excellence on AOSD (2005)

[6] Kiczales G.: Tutorial on Aspect-Oriented Programming with AspectJ, FSE (2000)

[7] Coelho R., Rashid A., Garcia A., Ferrari F., Cacho N., Kulesza U., von Staa A., Lucena C.: Assessing the Impact of Aspects on Exception Owns: An Exploratory Study. In: European Conference on Object-Oriented Programming (ECOOP), pp. 207-234 (2008)

[8] Soares S., Borba P., Laureano E.: Distribution and Persistence as Aspects. Softw., Pract. Exper., 36(7):711-759 (2006)

[9] Figueiredo E., Cacho N., Sant Anna C., Monteiro M., Kulesza U., Garcia A., Soares S., Ferrari F., Khan S., Castor Filho F., Dantas F.: Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability. In: ICSE 08: Proceedings of the 30th International Conference on Software Engineering, pp. 261-270, New York, NY, USA (2008)

[10] Kellens A., Mens K., Brichau J., Gybels K.: Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In: European Conference on Object-Oriented Programming (ECOOP), number 4067 in LNCS, pp. 501-525 (2006)

[11] Koppen C., Stoerzer M., Pcdiff: Attacking the Fragile Pointcut Problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany (2004)

[12] Fowler M.: Refactoring: Improving the Design of Existing Code. Addison Wesley (1999)

[13] Robbes R., Lanza M.: An Approach to Software Evolution Based on Semantic Change. In: Proceedings of Fase 2007, pp. 27, 41 (2007)

[14] Robbes R., Lanza M.: Change-Based Software Evolution. EVOL 2006, pp. 159-164 (2006)

[15] Robbes R., Lanza M.: A Change-Based Approach to Software Evolution. In: ENTCS, volume 166, issue 1, pp. 93-109 (2007)

[16] Robbes R., Lanza M.: Towards Change-Aware Development Tools. Technical Report at USI, 25 pages (2007)

[17] Robbes R.: Of Change and Software. Ph.D. Thesis, University of Lugano, 210 pages (2008)

[18] Concurrent Versions System (CVS), http://www.nongnu.org/cvs/

[19] Open Source Software Engineering Tools, SVN, http://subversion.tigris.org/

[20] Störzer M.: Impact Analysis for AspectJ: A Critical Analysis and Tool-Based Approach to AOP, PhD, Dissertation. In: Eingereicht an der Fakultät für Informatik und Mathematik der Universität Passau (2007)