ICT-UNPA-74-2013
ISSN: 1852 - 4516
Aprobado por Resolución Nro. 1053/13-R-UNPA

BALALU: ANTICIPAR LOS CAMBIOS EN UN SOFTWARE OA.

Análisis y evaluación de las operaciones de cambio y sus consecuencias.

Cecilia Fuentes y Sandra Casas
cfuentes277@gmail.com

Universidad Nacional de la Patagonia Austral
Instituto de Tecnología Aplicada

Campus Universitario, Av. P. Rivero S / N,
9400 Río Gallegos, Argentina

ICT-UNPA-74-2013
ISSN: 1852 - 4516
Aprobado por Resolución Nro. 1053/13-R-UNPA

1. INTRODUCCIÓN

El mantenimiento y evolución del software muchas veces se vuelve una tarea engorrosa, debido a que el software no se encuentra adecuadamente documentado, o bien las personas destinadas a realizar los cambios no son los propios desarrolladores del software, haciendo que la comprensión del código fuente sea un trabajo complejo. Esto se vuelve más difícil cuando se utilizan aspectos debido a que el código fuente base implícitamente invoca al código de los aspectos. La invocación implícita es especificada por los constructores específicos aspectuales (pointcuts) agregando un nivel extra de indirección, que hace más difícil comprender el comportamiento total del software. Diversos problemas como pointcuts frágiles e interacciones entre aspectos se producen fácil e inadvertidamente en la evolución del software orientado a aspectos, ya sea a partir de simples cambios en el código o bien por la aplicación de refactorings. Los desarrolladores deben analizar manualmente el código fuente (de aspectos y clases) para identificar estas ocurrencias, agregar más pruebas, en definitiva, se aumentan los costos y esfuerzos durante la evolución y mantenimiento.

Consecuentemente, existe la necesidad de plantear estrategias, técnicas y herramientas que permitan a los desarrolladores de software anticipar, analizar y evaluar los efectos e impactos en el software con aspectos, producto de la evolución del software.

En este trabajo planteamos como las características de la Programación Orientada a Aspectos (POA) mencionadas, inciden negativamente en la evolución y mantenimiento de aplicaciones OA, y proponemos la herramienta BaLaLu cuya finalidad es anticipar las consecuencias que un cambio podrá tener en una aplicación que usa aspectos.

La estructura de este informe se encuentra conformada en base a los siguientes elementos que se detallan: la Sección 2 corresponde a una breve introducción a la POA y al Lenguaje Orientado a Aspectos AspectJ. En la Sección 3 se hace referencia a la evolución y al mantenimiento de software OA. Luego en el apartado 4 se detalla un análisis de las operaciones de cambio. El modelo de anticipación se presenta y describe en la Sección 5, para luego mostrar la herramienta desarrollada y los ejemplos de los casos de estudio analizados en la Sección 6. En la Sección 7 se presentan los trabajos relacionados, finalizando en la Sección 8 con las conclusiones pertinentes.

2. PROGRAMACIÓN ORIENTADA A ASPECTOS

A lo largo de la historia del desarrollo de software los lenguajes de programación han evolucionado brindando cada vez mejores técnicas de modularización. La Programación Orientada a Objetos (POO) ha sido una de las técnicas más potentes en este tema, pero existen ciertos “concerns” o incumbencias de un software que aún son difíciles de modularizar de manera efectiva con la POO. A este problema se lo denomina incumbencias transversales o crosscutting concerns CCC, esto provoca la existencia de código disperso, es decir cuando existen invocaciones a un método esparcidas por toda la aplicación; y el código mezclado, que se

produce cuando una clase tiene código que nada tiene que ver con su esencia, lo cual disminuye su cohesión y por lo tanto viola uno de los principios básicos del diseño de objetos: el principio de única responsabilidad.

La Programación Orientada a Aspectos es un paradigma que tiene como objetivo facilitar la separación de crosscutting concerns (CCC) que entrecruzan varias clases y se extienden a lo largo de estas, logrando así un código menos enmarañado y menos disperso.

La POA encapsula y modulariza estos CCC en unidades denominadas aspectos, separada de la funcionalidad básica, 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, es decir que el comportamiento de un aspecto es implícitamente invocado en la implementación de otros módulos.

El soporte para este nuevo paradigma se logra a través de una clase especial de lenguajes, llamados Lenguajes Orientados a Aspectos (LOA), los cuales brindan mecanismos y constructores para capturar aquellos elementos que se diseminan por todo el sistema. Los Lenguajes de Programación Orientados a Aspectos son extensiones de lenguajes de programación existentes que incorporan los mecanismos necesarios para dar soporte a los aspectos. Un proceso de tejido (realizado por un nuevo tipo de compilador o intérprete) combina los componentes de funcionalidad básica con los aspectos, para generar la aplicación ejecutable [1]. El tejedor permite que en ciertos puntos de la ejecución de los componentes funcionales se inserte el código de los aspectos. Estos puntos se denominan puntos de unión (join-points) y podrían interpretarse como eventos que al ocurrir, activan la ejecución de un aspecto. Desde la perspectiva de los aspectos, éstos incluyen puntos de cortes (pointcuts), que representan el código de aplicación del aspecto y que se asocian a los puntos de unión del código funcional para adicionar cierto comportamiento (advice).
Así un aspecto puede consistir de múltiples y diferentes funcionalidades que necesitan que sean invocadas desde varios y distintos lugares (join-points) en el código, y la implementación de un aspecto puede estar conformada por varios pointcuts y advices.

2.1 Lenguaje de Programación AspectJ

El lenguaje AspectJ [2] es un lenguaje de propósito general que extiende al lenguaje Java para el soporte de aspects. Fue desarrollado en 1998 por Gregor Kickzales y el grupo de investigación dirigido por PARC (Palo Alto Research Center), una subsidiaria de Xerox. En diciembre del 2002 el proyecto AspectJ fue cedido a la comunidad open-source Eclispse.org, la cual continúa mejorando y dando soporte al proyecto.
Al ser una extensión del lenguaje Java es fácil de adoptar por programadores Java. Además, cualquier programa correcto en Java será un programa correcto en AspectJ, y todo programa correcto en AspectJ podrá correr sobre una máquina virtual de Java.

AspectJ al ser una extensión de Java para soportar el manejo de aspectos, agrega a la semántica de Java, las siguientes características:

Aspects (*Aspectos*): son los comúnmente llamados los "envoltorios" de código, son semejantes a las clases Java.

Join-point (*Puntos de enlace*): son puntos bien definidos en la ejecución de un programa, entre ellos podemos citar las llamadas a métodos y accesos a atributos, en resumen puntos en el código Java donde un aspecto puede interceptar a las clases.

Pointcuts (*Puntos de Corte*): son aquellos que logran concatenar lógicamente un conjunto de Join-points (puntos de enlace), permitiendo exponer el contexto en ejecución de dichos puntos. Existen cortes primitivos y también definidos por el usuario.

Advices (*Avisos*): son acciones que se ejecutan en cada Join-point (punto de enlace) incluido en un Pointcut (punto de corte). Los avisos tienen acceso a los valores expuestos por el Pointcut. En resumen los Advices son trozos de código asociados a Pointcuts, que injertan o añaden un nuevo comportamiento en todos los Join-points representados por el Pointcut (after, before o around).

Introduction – Inter Type Members (*introducciones y/o declaraciones*): son aquellas que permiten cambiar la estructura de clases de un programa agregando o extendiendo interfaces y clases con nuevos atributos, constructores o métodos. Esta entidad es estática porque afecta la signatura base del programa. En resumen permiten a un Aspect definir completamente los atributos y los métodos implementados para múltiples clases.

A continuación se puede apreciar un ejemplo de un pointcut que selecciona los momentos en que se abra un archivo:

```
1  pointcut fileOpen() : call( FileInputStream.new(..) );
```

Luego la definición de un advice que se ejecutará antes de los join-points especificados por el pointcut fileOpen(). Este advice ejecutará un chequeo de privilegios implementado en la clase SecurityManager:

```
1  FileInputStream around() : fileOpen() {
2
3    if (SecurityManager.checkFileOpenPermission()){
4      return proceed();
5    }else{
6      System.err.println("Permiso negado para abrir archivo.");
7      return null;
8    }
9  }
```

Como el advice debe ser ejecutado alrededor de los join points de creación de un objeto de tipo FileInputStream, debe ser declarado su tipo de retorno con un tipo adecuado (en el ejemplo, FileInputStream). El advice realiza el chequeo de permisos, y en caso de que estos existan permite la creación del objeto. En caso contrario, imprime un mensaje y devuelve null en reemplazo del objeto.
El código que se presenta a continuación es un ejemplo de un aspecto simple compuesto por el pointcut y advice definidos anteriormente:

```
1  public aspect FileOpenPermissionEnforcer{
2
3    pointcut fileOpen() : call( FileInputStream.new(..) );
4
5    FileInputStream around() : fileOpen() {
6
7      if (SecurityManager.checkFileOpenPermission()){
8        return proceed();
9      }else{
10       System.out.println("Permiso negado para abrir archivo.");
11       return null;
12     }
13   }
14 }
```

Con el compilador de AspectJ, estos aspectos pueden ser incorporados a un sistema ya compilado en bytecode, lo que permite agregar funcionalidad a este sin tener acceso al código fuente original ni recompilar todo el sistema. Junto con el compilador, el proyecto AspectJ provee un depurador y plugins para algunos IDE.

3. EVOLUCIÓN Y MANTENIMIENTO DEL SOFTWARE OA

La fase de mantenimiento del software ha sido reconocida como la más difícil y costosa [3] [4], ya que requiere de los desarrolladores, primero comprender su estructura y su diseño para luego poder modificarlo. Con frecuencia, la documentación está incompleta y desactualizada, y los desarrolladores originales del software no son los que hacen el mantenimiento. Estos factores hacen más difícil esta tarea. Se ha determinado que cerca del 50-60% [5] [6] del tiempo que insume el mantenimiento, es usado en leer y comprender código, antes de poder realizar modificaciones.

En un software OA la separación de pointcuts y advices 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 cuándo el sistema base y los aspectos interactúan y cómo se comportará el sistema.

Cambios aparentemente inocentes del código base pueden conducir a comportamientos erróneos y no intencionales. Es fácil perder el rastro de la característica global de cómo el código base

interactúa con los aspectos y puede volverse difícil identificar el código de tal comportamiento no anticipado. Por ejemplo, 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, a esto se lo denomina Falsos Positivos; o donde ciertos join-points que se intentan capturar, no lo sean, a estos de los llama Falsos Negativos.

Un ejemplo de estos problemas es presentado en el trabajo de Coelho [7], en el que los autores investigan un número de aplicaciones que usan aspectos para determinar comportamientos erróneos relacionados al manejo de excepciones. Se determina que si bien los desarrolladores eran expertos en el uso de aspectos, los falsos positivos y los negativos ocurren. Específicamente "errores en las expresiones de pointcuts" fueron encontradas en aplicaciones Health Watcher [8] y Mobile Photo [9].

Una variante de este problema, es el llamado "fragile pointcuts" [10] [11]. El cual deja de manifiesto el alto acoplamiento que existe entre los aspectos y el código base. Por lo que se puede establecer que el punto más crítico se encuentra en los pointcuts, ya que ante un mínimo cambio en el dominio, se pueden alterar el conjunto de eventos y condiciones que fueron el objetivo al diseñar e implementar el mismo.

A los cambios en el dominio se los denominan operaciones de cambio, que representan la evolución del software. Son las acciones que llevan adelante los desarrolladores cuando modifican el código fuente.

4. ANALISIS DE LAS OPERACIONES DE CAMBIO

Como se explicó en el punto anterior las operaciones de cambio representan la transición desde un estado de la evolución de una aplicación al próximo.
Estas operaciones de cambio se clasifican en atómicas, las cuales implican una única acción indivisible, como por ejemplo remover un método; y las operaciones de cambio compuestas, que son las resultantes de una secuencia de operaciones atómicas, como por ejemplo mover un método o un refactoring.

4.1 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 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. Las operaciones de cambio que se encuentran en esta tabla 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.

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

Operación de Cambio	Grupo						Falso Positivo	Falso Negativo	Interacción
	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 handler	X	X	X	-	-	-	-	+	-
Remove message	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	-	-	-	+	+	-

4.2 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 e interacciones (una interacción ocurre cuando 2 pointcuts atrapan el mismo join point (clase-método) con el mismo corte y con el mismo advice.).

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

Operación de Cambio	Código Base	Aspecto	Falso Positivo	Falso Negativo	Interacción
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	+	+	+
Add advice	X	X	-	-	+
Remove advice	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.

5. MODELO DE ANTICIPACIÓN

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 la identificación manual de los mismo 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.

El presente trabajo propone 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. 1, 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.

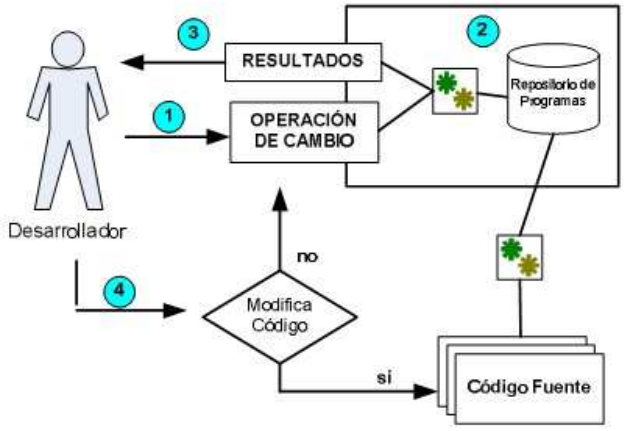


Figura 1. Flujo de trabajo del desarrollador

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.

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:

addMethod :: nuevo_metodo, identificador_clase → falsos_positivos
changeJoinPoint :: identificador_pointcut, nuevo_joinpoint → falsos_positivos, falsos_negativos.

La operación de cambio "Add method" requiere el identificador del nuevo método y la clase a la que se va a añadir. El resultado es un conjunto de falsos positivos. 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 está 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 field" es compuesta, ya que resulta de la secuencia de aplicar las operaciones de cambio atómicas "Remove field" y "Add field". Lo cual puede ser especificado de la siguiente manera:

moveField(id_field from_class to_class) ::
removeField(id_field from_class), addField(id_field to_class)
→ falsos_positivos,falsos_negativos

Operaciones de cambio atómicas identificadas:

removePackage :: identificador_package → falsos_negativos
removeClass :: identificador_class → falsos_negativos
removeMethod :: identificador_method, from_class → falsos_negativos
removeField :: identificador_field, from_class → falsos_negativos
removeMessage :: identificador_message, from_method, from_class → falsos_negativos
removeHandler :: identificador_exception, from_method, from_class → falsos_negativos

addPackage :: identificador_package → falsos_positivos
addClass :: identificador_class → falsos_positivos
addMethod :: identificador_method, to_class → falsos_positivos

addField :: identificador_field, to_class → falsos_positivos
addMessage :: identificador_message, to_method, to_class → falsos_positivos
addHandler :: identificador_exception, to_method, to_class → falsos_positivos

Operaciones compuestas identificadas:

moveClass(id_class from_pack to_pack) :: removeClass(id_class from_pack), addClass(id_class to_pack) → falsos_positivos, falsos_negativos

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

moveField(id_field from_class to_class) :: removeField(id_field from_class), addField(id_field to_class) → falsos_positivos, falsos_negativos

moveMessage(id_message from_method to_message) :: removeMessage(id_message from_method), addMessage(id_message to_methodclass) → falsos_positivos, falsos_negativos

Siguiendo el esquema, se han especificado las funciones para la operación de cambio "rename", aplicada a las distintas entidades: package, class, method, field.

Visualización Anticipada de Resultados.

De acuerdo a lo plasmado 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.

• **Interacciones:** dos o más aspectos pueden cortar transversalmente el mismo módulo de funcionalidad básica.

6. BALALU Y CASOS DE ESTUDIO

Como se mencionó anteriormente, la dependencia y acoplamiento existente entre aspectos y clases generan un alto impacto cuando se modifica el dominio. Esta situación es propia de aplicaciones que usan aspectos implementados en lenguajes del estilo de AspectJ [12]. En estos casos el desarrollador se enfrenta a nuevos dilemas puesto que el comportamiento introducido por los aspectos se ejecuta o deja de hacerlo sin que dicho aspecto haya sido modificado.
La mejor manera de resolver un problema es anticiparse, por lo que este trabajo pretende encontrar una posible solución al problema planteado. Consiste en la herramienta BaLaLu, cuya finalidad es anticipar las consecuencias que un cambio podrá tener en una aplicación que usa aspectos.

En BaLaLu un repositorio administra el modelo de datos de manera tal que representa a los programas (clases, aspectos, interfaces, etc.) de una aplicación que usa aspectos. En el repositorio el código fuente es representado en términos de entidades, elementos, relaciones e interacciones, como se grafica en la Figura 2. En esta representación, la entidad Pointcut asume una relevancia particular dado que son exhaustivamente analizadas ante la ejecución de cualquier función de operaciones de cambio.

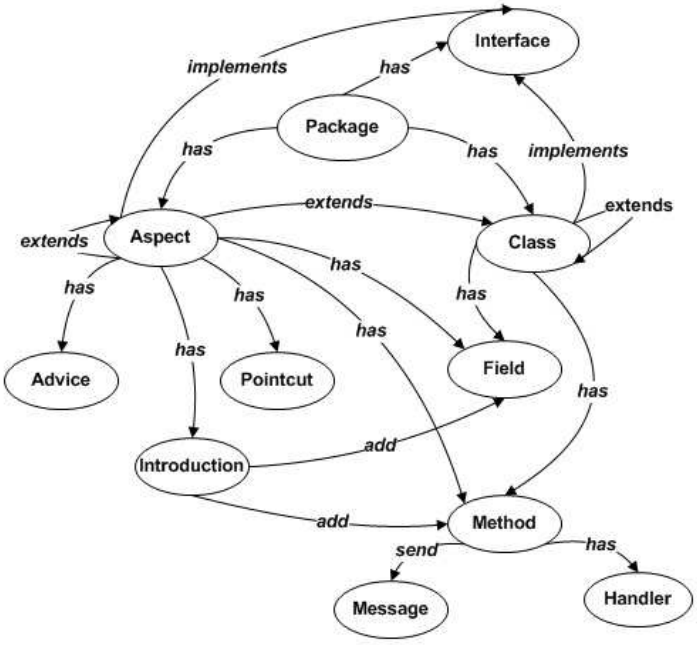


Figura 2. Entidades principales del repositorio de BaLaLu

Las operaciones de cambio comprenden una jerarquía en la cual las clasificaciones atómicas y compuestas son las principales subclases. AddClass, AddPackage, AddMethod, AddField, AddAdvice, addMessage, removeClass, RemoveMethod, RemoveField, RemoveMessage, RemoveAdvice, y así sucesivamente son subclases de operaciones de cambio atómicas (AtomicChange).
MoveClass, MoveMethod, MoveField, MoveMessage, RenameClass, RenameMethod, RenameField, y así sucesivamente son subclases de las operaciones de cambio compuestas (CompositeChange). Cada cambio atómico tiene consecuencias (posibles falsos positivos/negativos y/o interacción).
Los objetos CompositeChange se componen de conjuntos de objetos AtomicChange. Los objetos consequence representan información acerca de los falsos positivos/negativos o interacciones (tales como join-point, aspectos o pointcut) que será dado a los usuarios. La Figura 3 es un esquema simple del diseño.

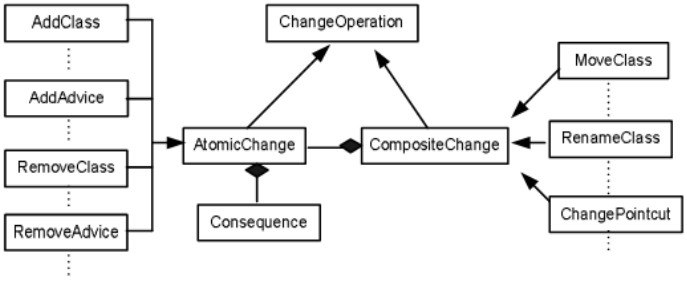


Figura 3. Diagrama de clases de las operaciones de cambio.

La Figura 4 presenta un escenario muy sencillo en el que tenemos que evaluar las consecuencias de la eliminación del método *setPoint* de la clase *Point.*

Chop = new RemoveMethod (“setPoint” , “Point”)

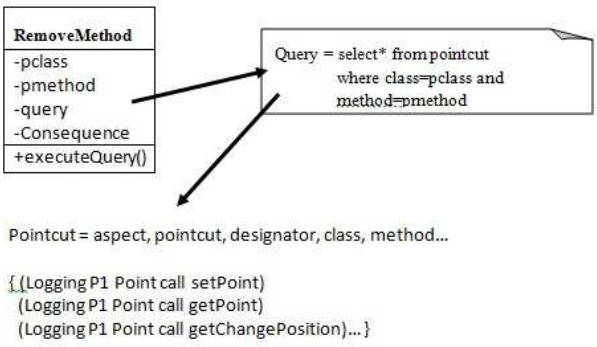
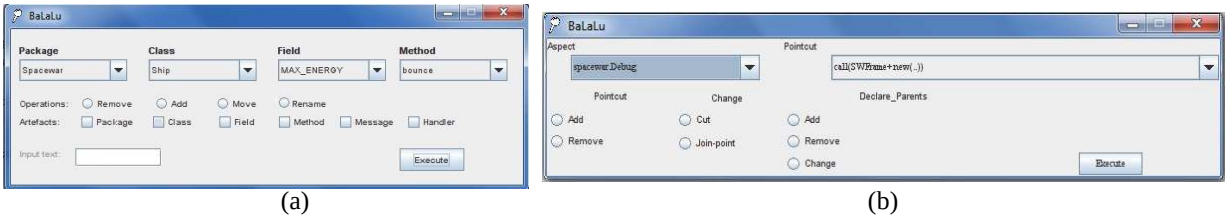


Figura 4. Instancia de operación de cambio atómica.

El aspecto Logging está capturando los métodos de la clase Point. La tabla pointcut contiene todos los join-point interceptados por Logging. Una instancia de la clase RemoveMethod se crea con los atributos "setPoint" y "Point". Luego una consulta se configura con estos valores.
El método executeQuery ejecuta la consulta y mapea los resultados en un objeto Consequence.

Para introducir las operaciones de cambio BaLaLu consta de un formulario que se puede apreciar en las Figuras 5 (a) y (b), la primera de ellas permite seleccionar los elementos del dominio requeridos como entrada de la función que se pretende aplicar. Esta se configura particularmente con los elementos de cada aplicación. Además de las operaciones de cambio que se efectúan en el dominio, BaLaLu permite analizar las operaciones de cambio que se realizan en los aspectos, como se observa en la Figura (b); la cual muestra que seleccionado el aspecto, lista todos los pointcuts relacionados a ese aspecto, para luego permitir la selección de la operación que se desea realizar.



(a) (b)

Figura 5. Vistas principales de BaLaLu para el ingreso de las operaciones de cambio.

A continuación (en la Figura 6) se exhiben las consecuencias de una operación de cambio, en (a) aquellas que se realizan en el dominio generan un reporte que indica la cantidad de falsos positivos y/o negativos que se han producido, y el detalle de dónde se producen (aspecto, pointcut, expresión). Por otro lado en (b) las consecuencias de una operación de cambio que se realiza en los aspectos, generan un reporte que además de indicar la cantidad de falsos negativos y positivos como en el caso anterior, también muestra la cantidad de interacciones e inalterables, y el detalle de cada uno de ellos.

(a) (b)

Figura 6. Reporte de consecuencias para operaciones de cambio.

Los casos estudiados son Telecom y Spacewar, ejemplos que son distribuidos con AspectJ para Eclipse, a continuación se mostrarán algunas de las operaciones que se realizaron con BaLaLu, y para poder exponer que la herramienta permite analizar las operaciones de cambio atómicas y compuestas, se mostrarán primero aquellas operaciones que se realizaron en el dominio de Telecom, y luego las operaciones que se realizaron en los aspectos de Spacewar.

ICT-UNPA-74-2013
ISSN: 1852 - 4516
Aprobado por Resolución Nro. 1053/13-R-UNPA

6.1 Caso de estudio Telecom

Telecom es una simple aplicación distribuida con AspectJ, que simula conexiones telefónicas. La aplicación tiene tres funcionalidades principales: Customers, Calls y Connections. En la Figura 7 se muestra un Diagrama del caso de estudio y las características de las clases se describen a continuación:

- **Call:** Representa una llamada, crea una conexión de tipo local o larga distancia
- **Connection:** Representa una conexión entre los clientes, y lo modela a través de una máquina de estados simples (las conexiones son inicialmente PENDING, luego COMPLETED y finalmente DROPPED). Los mensajes se imprimen en la consola de manera que el estado de las conexiones pueden ser observadas. La clase Connection es abstracta con dos subclases: Local y LongDistance.
- **Costumer:** Representa un cliente, tiene métodos de call(), pickup(), hangup() y merge() para administrar las llamadas.
- **Timer:** Representa un contador de tiempo, y registra los tiempos cuando se inicia y se detiene una conexión; así también, devuelve la diferencia del tiempo transcurrido.

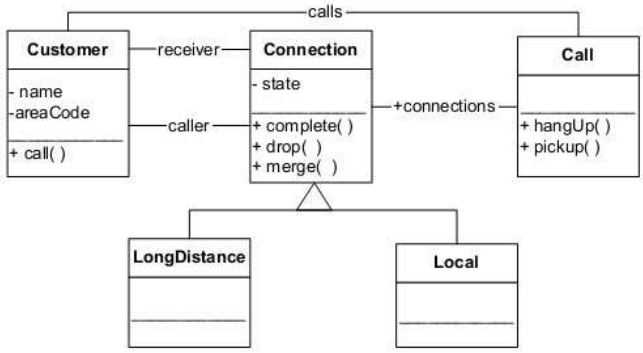


Figura 7. Diagrama de Telecom.

Telecom también incluye tres aspectos, los cuales implementan los requerimientos de tiempo y facturación de las conexiones, los que se describen a continuación:

- **Timing:** añade funcionalidad para registrar el tiempo de conexión telefónica utilizando un timer a cada conexión.
- **Billing:** calcula el coste de cada conexión y se basa en la información de tiempo obtenida del aspect Timing.
- **TimerLog:** guarda en un log el registro de cuando empieza y termina un timer, por lo tanto, este aspecto intercepta la ejecución de los métodos Timer.start() y Timer.stop().

En la Tabla 3, se detallarán las operaciones de cambio en Telecom que se analizaron con la herramienta presentada. Al realizar una operación de cambio atómica AddClass “Celular” se obtuvo un falso positivo que corresponde al pointcut aspect1 del aspecto Billing, el mismo atrapa todos los métodos new de la clase Connection y todas las que son subtipos de ésta (ChOp #1).

Otro ejemplo es renombrar la clase Connection por Linkage, en esta operación de cambio se producen tanto falsos positivos como negativos, debido a que se trata una operación compuesta (aplica las operaciones RemoveClass y AddClass), siendo interceptados por los aspectos Timing, endTiming, y Billing, para aquellos casos donde el designador de clase corresponda con la clase a renombrar (ChOp #2). Otro caso es renombrar el método start() de la clase Timer por init(), esta operación produce falsos negativos (ChOp #3).
Eliminando la clase Connection del paquete Telecom, se producen 5 falsos negativos (ChOp #4). Al mover el método complete() de la clase Connection a la clase Call, se produce sólo un falso negativo, ya que el pointcut que intercepta este método es aspect5 del aspecto Timing (ChOp #5).

Identificador operación	Operación de cambio	Descripción	Falsos Positivos	Falsos Negativos
ChOp #1	Add Class	Celular	1	0
ChOp #2	Rename Clase	Connection por Linkage	3	5
ChOp #3	Rename Method	start() por init()	0	1
ChOp #4	Remove Class	Connection	0	5
ChOp #5	Move Method	Complete() a la clase Call	0	1

Tabla 3. Detalle de las operaciones de cambio realizadas en el código base de Telecom.

En el gráfico de la Figura 8 se pueden observar la cantidad de falsos positivos y falsos negativos resultantes de las operaciones de cambio realizadas; contrastándolas con el total de los join-point capturados por los Pointcut originales de Telecom (barra de color verde).

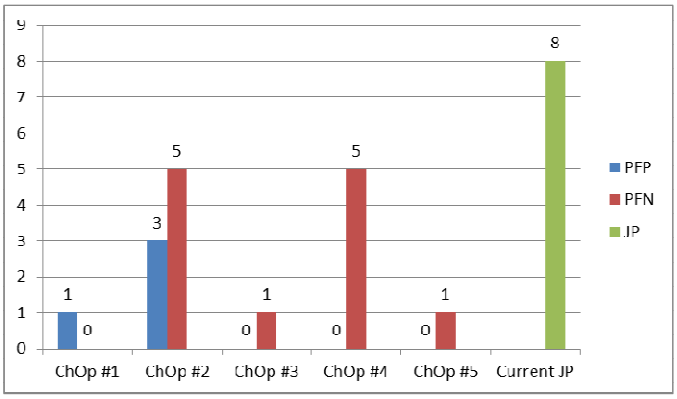


Figura 8. Consecuencias de las operaciones de cambio en Telecom.

6.2 Caso de estudio Spacewar

Spacewar es una implementación del clásico juego de batalla espacial de vídeo. La clase main o principal es Game. Cuando se ejecuta la aplicación, se observan dos pantallas diferentes, estos son los dos aspectos incorporados en la pantalla del juego. En cada uno se observa una sola nave blanca y dos naves rojas, la blanca es la que controla el usuario, y las rojas son las naves enemigas. La nave blanca es controlada con las cuatro teclas de flecha para girar, impulsar,

detener, y se dispara con la barra espaciadora. El juego se muestra en ambas ventanas. Además de una ventana que muestra el debugging.
Spacewar contiene 17 clases y 8 aspectos, además de importar el paquete Coordination que a su vez consta de 8 clases y 1 aspecto; conteniendo 3053 líneas de código.
Algunos de los ejemplos de operaciones realizadas en Spacewar con BaLaLu son las siguientes:

Al agregar el pointcut fire: call(void Ship.fire()) al aspecto Debug, se obtiene como resultado 2 falsos positivos y una interacción (ChOp #1). Otro ejemplo es agregar un pointcut al aspecto Debug, lowest : call(boolean Ship.expendEnergy (double amount)) && args (p) && if (p.getEnergy() < 0.10) para interceptar las actualizaciones de energía de las Naves (Ship), lanzando una alerta al llegar al mínimo; resultando 4 falsos positivos (ChOp #2).

Al cambiar el join-point call (Ship Game.newShip (Pilot)) && args(p) perteneciente al aspecto SpaceObjectPainting por call(Game.*(..)), el resultado son 32 falsos positivos, dos interacciones y 13 inalterables (ChOp #3). Otro ejemplo analizado con BaLaLu es eliminar el pointcut call(Game+.new(String)) && args(s) del aspecto DisplayAspect, resultando un solo falso negativo (ChOp #4). Cambiando el corte del pointcut execution(void Ship.bounce(Ship, Ship)) && args(s, s1)) del aspecto Debug, se obtienen 1 falso positivo y 1 falso negativo (ChOp #5).
Agregado un nuevo pointcut al aspecto Debug de Spacewar cuya expresión es call(Robot.*(..)) se obtiene como resultado 15 falsos positivos (ChOp #6).
Cuando se cambia el corte del pointcut que se encuentra en el aspecto Debug, target(r) && (call(void register(..)) || call(void unregister(..)); se obtienen 1 falso negativo y una interacción (ChOp #7).
Agregando un pointcut al aspecto Debug, preinitialization((spacewar.*&& !(Debug+ || InfoWin+)).new(..)), se obtiene como resultado 19 falsos positivos (ChOp #8). Al eliminar el pointcut execution(* (spacewar.* && !(Debug+ || InfoWin+)).*(..)), se obtienen 108 falsos positivos (ChOp #9).

Tabla 4. Detalle de las operaciones de cambio realizadas en el código de los aspectos en Spacewar.

Indicador de operación	Operación de cambio	Descripción	Falsos Positivos	Falsos Negativos	Interacciones	Inalterables
ChOp #1	Add pointcut	call(void Ship.fire())	2	0	1	0
ChOp #2	Add pointcut	call(boolean Ship.expendEnergy(double amount)) &&args(p)&&if(p.getEnergy()<0.10).	4	0	0	0
ChOp #3	Change join point	call(Game.*(..))	32	0	2	13
ChOp #4	Remove pointcut	call(Game+.new (String)) &&args(s)	0	1	0	0
ChOp #5	Change pointcut (designator)	execution(void Ship.bounce (Ship, Ship)) && args (s, s1))	1	1	0	0
ChOp #6	Add pointcut	call(Robot.*(..))	15	0	0	0
ChOp #7	Change pointcut (designator)	target(r) && (call(void register(..)) || call(void unregister(..))	0	1	1	0
ChOp #8	Add pointcut	preinitialization((spacewar.* && !(Debug+ || InfoWin+)).new(..))	19	0	0	0
ChOp #9	Remove pointcut	execution(* (spacewar.* && !(Debug+ || InfoWin+)).*(..))	0	108	0	0

En el gráfico de la Figura 9, se pueden observar las consecuencias de las operaciones indicadas en la Tabla 4 que reflejan falsos positivos, falsos negativos interacciones e inalterables, en relación a la totalidad de join-point capturados por los aspectos de Spacewar.

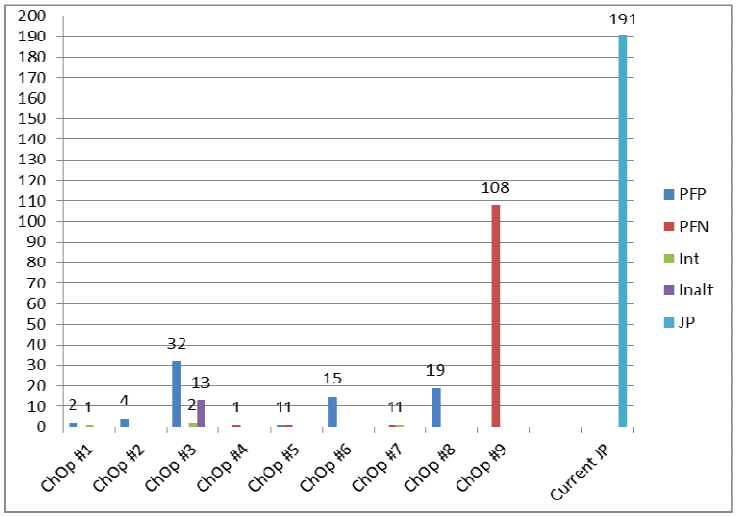


Figura 9. Consecuencias de las operaciones de cambio en spacewar.

7. 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. SpyWare [13] [14, 15, 16] es un IDE que incorpora el enfoque de evolución de software basado en cambios (CBSE). Este enfoque trata a los cambios como entidades de primera clase y el propósito es definir la historia de un programa como la secuencia de cambios que éste atravesó. Al disponer de la historia de cambios se puede reconstruir cada estado sucesivo del código fuente de los programas. El éxito de este modelo requiere que pueda ser implementado en los IDES o herramienta de desarrollo.

Varias herramientas como PCDiff [22], Celadon [23], y Souyoul [24] y [25] analizan los impactos de los cambios en los programas OA. En general, estas herramientas analizan y comparan dos o más versiones de los códigos fuente de los programas. Las diferencias observadas se utilizan para derivar un grupo de operaciones de cambio de atómicas. Estas herramientas emplean alguna representación abstracta de programas tales como árboles de sintaxis, grafos de llamadas, grafos de dependencia, e incluyen también casos de prueba. Una diferencia importante entre estos trabajos y nuestra propuesta son: a) Estas herramientas suscriben a métodos basados exclusivamente en la comparación de versiones de programas, el análisis del impacto de los cambios se hace "después" que se producen los cambios. Nuestro objetivo es proponer un método que pueda identificar las consecuencias de los cambios “antes”

de que ocurran. b) Debido a que estas herramientas y enfoques funcionan con versiones del programa, sólo pueden encontrar diferencias en términos de operaciones de cambio "atómicas". Nuestro enfoque propone analizar además operaciones compuestas (move-rename). Aquí es clave, mientras que los trabajos citados analizan los cambios en forma aislada, independiente y disociada y no como parte de una estructura de operaciones más “complejas” que las contienen, los análisis y resultados inevitablemente pierden la semántica y la integridad que han originado dichos cambios.

ViDock [26] es una herramienta para analizar el impacto del tejido de los aspectos sobre los casos de prueba. El enfoque se basa en realizar análisis estático para identificar el subconjunto de casos de prueba que se ven afectadas por el tejido de los aspectos. Esta herramienta funciona después de realizar los cambios en el código fuente.

Un método para analizar los efectos de los cambios de aspectos tejidos/compuestos se propone en [27], aunque el método no está soportado por una herramienta. Este trabajo analiza cómo los aspectos pueden cambiar el flujo de control, entrada / salida de parámetros, los valores de los miembros de datos, y las dependencias de herencia del código base. También se describen las influencias y los posibles efectos de las declaraciones pointcut sobre sucesiones y dependencias primordiales y cómo el efecto dominó se pueden calcular.

Herramientas automatizadas como AJDT [20] y PointcutDoctor [21] ante una expresión de pointcut muestran los join-points efectivamente interceptados y también aquellos “casi” interceptados, lo cual resulta útil ante una operación de cambio en los aspectos, pero resulta insuficiente para las operaciones de cambio que tienen lugar en el dominio.

8. CONCLUSIONES

Se han presentado diversas propuestas para lograr identificar los impactos que tiene realizar un cambio durante la evolución y mantenimiento del software OA, como se mencionó en la Sección anterior. Sin embargo la mayoría de ellas permiten analizar las consecuencias una vez realizado el cambio, o bien no contemplan las modificaciones realizadas en el dominio de la aplicación.

Nuestra propuesta permite anticiparnos a las consecuencias que un cambio podrá tener en una aplicación que usa aspectos, permitiendo analizar no solo los cambios que se quieran realizar en el código base, sino también aquellos que se llevarán a cabo en el código de los aspectos.
Este modelo de anticipación tiene como objetivo evitar o al menos disminuir la cantidad de veces que sea necesario deshacer cambios o volver a versiones anteriores de los programas. Por lo tanto pretende hacer las tareas de mantenimiento y evolución del software que usan aspectos, más predecibles y menos costosas.

El trabajo futuro está dirigido a lograr dos objetivos: a) abarcar el análisis y evaluación de operaciones complejas y la anticipación de consecuencias, específicamente refactorings, para lo cual planteamos como estrategia realizar la composición de operaciones atómicas; b) Implementar BaLaLu plugin de Eclipse, lo cual dará más oportunidades y beneficios al desarrollador.

ICT-UNPA-74-2013
ISSN: 1852 - 4516
Aprobado por Resolución Nro. 1053/13-R-UNPA

AGRADECIMIENTOS

A la Universidad Nacional de la Patagonia Austral que ha financiado parcialmente este trabajo.

REFERENCIAS

1. Piveta E., Zancanela L.: *Aspect Weaving Strategies.* Journal of Universal Computer Sciencie. Vol.9. Num.8 (2.003).
2. The AspectJ Programming Guide, http://eclipse.org/aspectj.
3. Li W. and Henry S. An Empirical Study of Maintenance Activities in Two Object-oriented Systems, Journal of Software Maintenance, Research and Practice, Volume 7, No. 2 March-April 1995, pp.131-147.
4. Schneidewind N. The State of Software Maintenance, IEEE Transactions on Software Engineering, SE-13, No. 3, March 1987, pp.303-310.
5. Coleman, D.M. Ash, D., Lowther, B., And Oman, P. Using metrics to evaluate software system maintainability. IEEE Computer 27, 8 (1994), 44–49.
6. Erlikh, L. Leveraging legacy system dollars for e-business. (IEEE) IT Pro (May-June 2000), 17–23.
7. Coelho R., Rashid A., Garcia A., Ferrari F., Cacho N., Kulesza U., von Staa A., and Lucena C. Assessing the impact of aspects on exception ows: An exploratory study. In European Conference on Object-Oriented Programming (ECOOP), pages 207-234, 2008.
8. Soares S., Borba P., and 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., and 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, pages 261-270, New York, NY, USA, 2008. ACM.
10. Kellens A., Mens K., Brichau J., and 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, pages 501-525, 2006.
11. Koppen C. and Stoerzer M. Pcdiff: Attacking the fragile pointcut problem. In European Interactive Workshop on Aspects in Software (EIWAS), 2004.
12. Kiczales, G.: Tutorial on Aspect-Oriented Programming with AspectJ, FSE (2000).
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. AJDT: AspectJ Development Tools, http://www.eclipse.org/ajdt/
21. Ye L., De Volder, K.: Tool support for understanding and diagnosing pointcut expressions. In: International Conference Aspect-Oriented Software Development (2008).
22. C. Koppen and M. Stoerzer, "Pcdiff: Attacking the Fragile Pointcut Problem", European Interactive Workshop on Aspects in Software, Berlin, Germany, 2004.
23. S. Zhang and J. Zhao, "Change Impact Analysis for Aspect-Oriented Programs". Technical Report SJTU-CSE-TR-07-01, Center for Software Engineering, Shanghai Jiao Tong University, 2007.
24. I. Bouteraa and N. Bounour, "Towards The Use of Program Slicing In the Change Impact Analysis of Aspect Oriented Programs", ACIT'2011 Proceedings International Arab Conference on Information Technology – Arabia Saudita, 2011.
25. L. Cavallero & M. Monga, "Unweaving the Impact of Aspect Changes in AspectJ". FOAL 09 – USA.
26. R. Delamare, F. Muñoz, B. Baudry and Y. Le Traon "Vidock: a Tool for Impact Analysis of Asect Weaving on Test Cases". ICTSS'10 Proceedings of the 22nd IFIP WG 6.1 - Springer-Verlag Berlin, Heidelberg ©2010. Pp 250-265.
27. Ch. Liu, S. Chen and W. Jhu, "Change Impact Analisys for Objetc-oriented Programs Evolved to aspect-Oriented programs", SAC 2011. Taiwan.