Estudio de Revisión sobre las Métricas en la Programación Orientada a Aspectos

A. J. Orozco
j.orozco@unisimon.edu.co

Resumen—En la actualidad, la literatura se refiere a nuevos paradigmas de programación que hacen las que las líneas de código de los programas hagan procesos convencionales pero de forma más eficiente y con un menor número de líneas de codificación fuente. Este es uno de los objetivos de la Programación orientada a aspectos, que tiene sus bases en la programación orientada a Objetos, pero tiene características, ventajas, conflictos y métricas, las cuales son tratadas en este trabajo.

Palabras clave —Aspecto, Métrica, Conflicto, Encapsulamiento y Advice

I. INTRODUCCIÓN

En la programación existen variadas tendencias en estructura y aplicaciones. Dentro de estas se encuentra con más frecuencia la programación estructurada y la Orientada a objetos. Esta última, ha tenido unas variantes y una de las más conocidas es la Programación orientada a Aspectos, que es el referente del presente trabajo, que inicia con un recorrido por los conceptos preliminares de la temática, pasando por los conflictos entre aspectos y terminando con las respectivas métricas.

A. Programación orientada a aspecto

Es un término usado para describir una técnica de programación y una forma de pensamiento acerca de la construcción de aplicaciones software que complementan la forma de expresión encontrada por la programación orientada a objetos [1]. Las tecnologías de programación Orientadas a aspectos tienen como objetivo mejorar el sistema mediante la modularización funciones transversales. Las propiedades globales y cuestiones de programación y el diseño pueden conducir a conceptos transversales, por ejemplo, el manejo de errores o el código de transacción, funciones que interactúan, fiabilidad y la seguridad [2]. Según [3] es un paradigma emergente que se basa en el principio de separación de conceptos. Este ofrece la idea de una nueva unidad modular que encapsula los problemas transversales. Así mismo, un sistema orientado a aspectos consiste en aspectos y las clases base (o componentes) que se pueden ser enlazados en un ejecutable conjunto. Las clases base en un sistema orientado a aspectos también pueden ejecutarse de forma independiente.

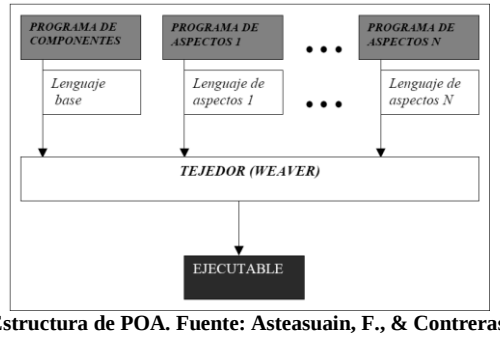


Fig. 1 Estructura de POA. Fuente: Asteasuain, F., & Contreras, B. E. (Octubre de 2002)

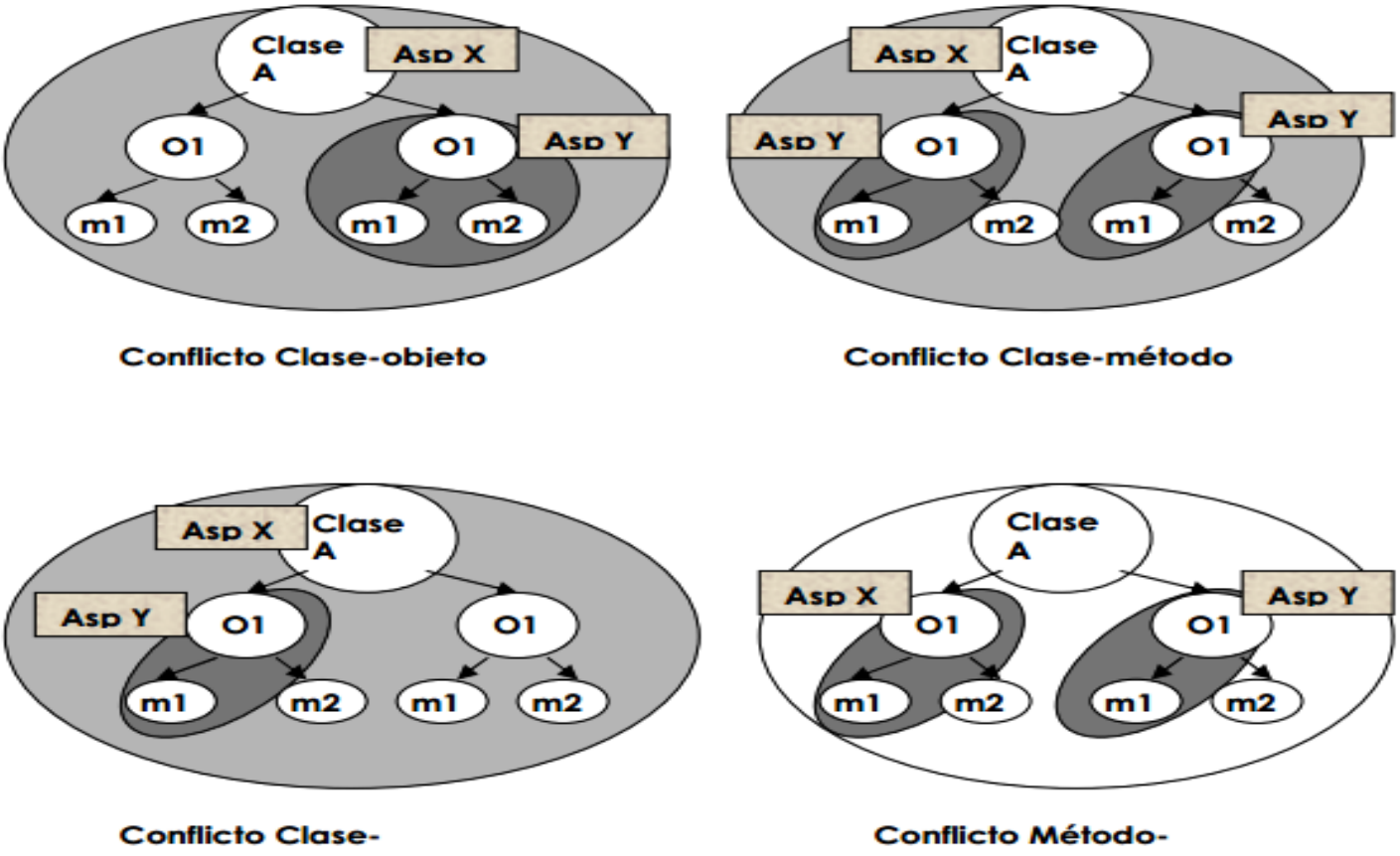
Desde la perspectiva de la arquitectura del sistema, los aspectos a menudo son atravesados por múltiples clases base. Desde la perspectiva de la clase base, sin embargo, los aspectos son esencialmente modificaciones graduales en las clases base con operaciones adicionales y limitaciones para distinguir cuestiones [4].

1) Diferencia entre orientación a objetos y orientación a aspectos

El paradigma orientado a objetos proporciona un potente mecanismo para separar intereses, sobre todo aquellos relacionados con la lógica del negocio de la aplicación, pero presenta dificultades a la hora de modelar otros intereses que no pueden ser encapsulados en una única entidad o clase, ya que afectan a distintas partes del sistema. Es decir, la diferencia radica es que mientras la POA se enfoca en los conceptos que se entrecruzan, la POO se enfoca en conceptos comunes [5].

Fig. 2. Comparación entre Lenguaje de programación generalizado (LPG) y Programación Orientada a Aspectos. Fuente: Asteasuain, F., & Contreras, B. E. (Octubre de 2002)

Fig. 3. Ejemplos de Conflictos. Fuente: Fernández, A. (Junio de 2009)



2) Aspecto

Un aspecto es una unidad modular que se disemina por la estructura de otras unidades funcionales. Los aspectos existen tanto en la etapa de diseño como en la etapa de implementación. Un aspecto de diseño es una unidad modular que se entremezcla en la estructura de otras partes del diseño. Un aspecto de programa o de código es una unidad modular del programa que aparece en otras unidades del programa [6].

Seguido de lo anterior, es posible diferenciar los aspectos de los demás integrantes del sistema: al momento de implementar una propiedad, tenemos que la misma tomará una de las dos siguientes formas:

a) Un componente

Si puede encapsularse claramente dentro de un procedimiento generalizado. Un elemento es claramente encapsulado si está bien localizado, es fácilmente accesible y resulta sencillo componerlo.

b) Un aspecto

Si no puede encapsularse claramente en un procedimiento generalizado. Los aspectos tienden a ser propiedades que afectan la performance o la semántica de los componentes en forma sistemática (Ejemplo: sincronización, manejo de memoria, distribución, etc.) [7].

De una forma más particular, se puede definir a un aspecto como un concepto que no es posible encapsularlo claramente, y que resulta diseminado por todo el código [7, p. 11].

3) Punto de unión

Un punto de unión es cualquier punto de ejecución identificable en un sistema (por ejemplo, la llamada al método, la ejecución del método, la construcción de objetos, asignación de variables, etc.) Estos puntos son los lugares donde una acción transversal puede ser insertada [8].

4) Advice

Los avisos, éstos pueden definirse como: acciones que se ejecutan en cada punto de enlace dentro de un mismo punto de corte, estas acciones se traducen en rutinas o fragmentos de código [9].

5) Punto de corte

Un Punto de corte hace referencia a un conjunto de puntos de enlace que cumplen cierta condición, es decir, permiten exponer el contexto de ejecución de dichos puntos [9, p. 15].

6) Weaving

Es un proceso de transformación del programa que se utiliza para cambiar una programación de entorno de ejecución. Este convierte código orientado a aspectos en código orientado a objetos con los aspectos integrados en el código [10].

II. CONFLICTO ENTRE ASPECTOS

A. Definición

Un conflicto entre aspectos es, según [11], "Un conflicto ocurre cuando dos o más aspectos compiten por su

activación". Así mismo, "se puede referir a los conflictos como interacciones de aspectos. Como se dijo, un objeto de funcionalidad básica puede ser asociado a más de un aspecto, donde cada uno de estos tiene su propio objetivo de comportamiento. Si las tareas a realizar por cada aspecto son totalmente independientes del resto o entre sí, el sistema se ejecutará sin problemas. Pero el comportamiento del sistema se torna impredecible si los aspectos que compiten no son independientes."

B. Causas

Las causas de los conflictos son:

- El uso de patrones o comodines para englobar grupos de pointcuts.
- El uso de diferentes dominios.
- El uso de información dinámica en un pointcut también puede limitar el control sobre el resultado del pointcut. [12]

C. Clasificación de Conflictos

1) Según Elementos involucrados

- "Conflictos intra-aspects: si el conflicto se produce entre advices dentro del mismo aspect.
- Conflictos inter-aspects: si el conflicto es entre dos advices de diferentes aspects
- Conflictos aspect-clase: si el conflicto se produce entre un aspect y una clase" [12, p. 49]

2) Según Composición

- **"**Composición de clase: el código del aspect se activa cuando todos los objetos de una determinada clase reciben un mensaje.
- Composición de objeto: el código del aspect se activa cuando un determinado objeto recibe un mensaje.
- Composición de método: el código del aspect se activa cuando todos los objetos de una clase reciben un mensaje de un método en concreto.
- Composición de objeto-método: se trata de una mezcla entre la composición de objeto y la de método, en la cual el código del aspect se activa cuando un determinado objeto recibe un mensaje de un método en concreto." [12, p. 50]
- "Conflicto clase-objeto: si el dominio de activación de un aspect asociado a un objeto está incluido en el dominio de activación de otro aspect asociado a la clase a la que pertenece el objeto del primer aspect.
- Conflicto clase-método: si el dominio de activación de un aspect asociado a un método se encuentra incluido en el dominio de activación asociado a la clase del objeto que invoca el método. Solo se producirá conflicto cuando se invoque a ese método en concreto.
- Conflicto clase- método/objeto: si el dominio de activación de un aspect asociado al método de un objeto en concreto está incluido en el dominio de activación de la clase a la que pertenece el objeto. Este tipo tiene menos peligro que el anterior puesto que solo se producirá conflicto cuando se invoque al método asociado de un objeto determinado.
- Conflicto método-método: si dos aspects están asociados al mismo método."

3) Según tipo

- **"**Conflicto estático: si se detecta al establecerse la relación en tiempo de compilación.
- Conflicto dinámico: si se detecta el conflicto en tiempo de ejecución, por ejemplo durante la fase de pruebas de la aplicación. [12, pp. 51-21]"

D. Niveles de Conflicto

En el orden de clasificación de conflictos, introducimos dos diferentes niveles. El primer nivel representa los conflictos directos. Este tipo de conflictos, se ocupan de aspectos básicos con conexiones directas que generan conflictos. [13]

E. Ejemplo de Comportamiento entre aspectos

Supongamos que hay un sistema base que utiliza un protocolo para interactuar con otros sistemas. El protocolo tiene dos métodos: uno para transmitir, sendData (String) y para recibir, ReceiveData (String). Ahora, imaginemos que nos gustaría asegurar este protocolo. Para lograr esto, encriptados todos los mensajes salientes y desciframos todos los mensajes entrantes. Podemos implementar eso, como un aviso de encriptación en el método de ejecución sendData. Sin embargo, podemos sobreponer un aviso de desencriptamiento en el método receiveData. Ahora imagine un segundo aspecto que rastrea todos los métodos y argumentos posibles. La implementación del aspecto de rastreamiento utiliza una condición para determinar dinámicamente si el método actual, deben detectarse, como el seguimiento de todos los métodos no es muy eficiente. El aspecto rastreo puede, por ejemplo, ser usado para crear un rastreamiento rutinario de la ejecución dentro de un paquete en concreto.

Estos dos avisos son superpuestos en el mismo punto de unión, este caso *Protocol.sendData*. Como los avisos tienen que ser secuencialmente ejecutados, hay dos posibles ordénenos de ejecución aquí. Ahora supongamos que queremos asegurarnos de que no hay un acceso a los datos antes de que sean encriptados. Esta restricción es violada, si dos avisos son ordenados de tal forma que el aviso de rastreo es ejecutado antes del aviso de encriptación. Podemos terminar con un archivo de registro que contiene información "sensible". La situación resultante es lo que llamamos un conflicto de comportamiento. [14]

F. Soluciones

"Luego que el Administrador de Conflictos informe los potenciales conflictos detectados automáticamente, se pretende que sea factible resolverlos aplicando la taxonomía propuesta en la sección 3. Las estrategias adoptadas para

implementar cada categoría de resolución están soportadas, en algunos casos, sobre mecanismos propios de AspectJ.
En otras situaciones se incorporan nuevos mecanismos con el objeto de que el
Administrador de Conflictos tome cursos de acción específicos.
Estrategias para la Resolución

- Orden: para la implementación de esta categoría el Administrador de Conflictos añadirá una declaración de precedencia que involucre a los aspectos que plantean el conflicto. Esta categoría sólo es válida para los conflictos de semejanza total.
- Orden Inverso: para la implementación de esta categoría el Administrador de Conflictos invertirá una declaración de precedencia que involucre a los aspectos que plantean el conflicto. Esta categoría sólo es válida para los conflictos de semejanza total.
- Opcional: la estrategia propuesta es que el Administrador de Conflictos añada una condición de ejecución de mutua exclusión a cada uno de los pointcuts de los aspectos que ocasionan el conflicto. La implementación se realizará utilizando el corte if y la elección de una condición. Esta estrategia puede ser empleada principalmente para la resolución de conflictos de semejanza total." [15]

III. METRICAS

A. Beneficio de Introducir aspectos

1) Objetivo

Medir cuál es el beneficio de introducir aspectos definiendo una relación de esfuerzo de desarrollo entre una implementación sin considerar aspectos, y otra donde se aplica el paradigma. Está definida en función de dos factores claves de un proyecto: el tamaño del producto, y el esfuerzo de desarrollo. Considera el tamaño de una implementación sin y con aspectos, y pondera estos valores con un factor de desarrollo.

2) Fórmula del Cálculo

Se calcula teniendo en cuenta el factor de desarrollo, que establece que para un sistema pequeño el tiempo de desarrollo promedio en Java es de 10 horas/programador, mientras que en AspectJ es de 2 horas/programador. A fin de considerar el tamaño de ambas implementaciones, se evalúa el tamaño físico de los archivos de las clases y del aspecto, medidos en Kb.

$$\boldsymbol{Beneficio\ POA} = \frac{(\boldsymbol{Tamaño\_sin\_Aspectos})(\boldsymbol{Factor\ de\ Desarrollo\ Java})}{(\boldsymbol{Tamaño\ con\ Aspectos})(\boldsymbol{Factor\ de\ Desarrollo\ AspectJ})}$$

Donde:

$$\boldsymbol{Tamaño\ con\ Aspectos} = \boldsymbol{Tamaño\ Funcionalidad\ Básica} + \boldsymbol{Tamaño\ Aspecto}$$

B. Limpieza

1) Objetivo

Medir el porcentaje de código que se depura de la funcionalidad básica al introducir aspectos. Utiliza el tamaño de las clases, medido en Kb. [16, pp. 1167-1168]

2) Fórmula de Cálculo

$$\boldsymbol{Limpieza} = \frac{(\boldsymbol{Tamaño\ sin\ Aspectos} - \boldsymbol{Tamaño\ Funcionalidad\ Básica})}{\boldsymbol{Tamaño\ Sin\ Aspectos}} \times \mathbf{100}$$

C. Grado de Dispersión

1) Objetivo

Medir la Variación de la concentración de un aspecto sobre todos los componentes con respecto al peor de los casos. [17, pp. 2-3]

2) Formula

La concentración ($CONC$), mide cuantas de las líneas de código fuentes relacionadas en un aspecto s están contenidas dentro de un componente específico t (Por ejemplo: archivo, clase, método):

$$\boldsymbol{CONC(s,t)} = \frac{\boldsymbol{SLOCs\ in\ componente\ t\ relacionado\ en\ un\ aspecto\ s}}{\boldsymbol{SLOCs\ relacionado\ en\ un\ aspecto\ s}}$$

Donde: SLOC, son las Líneas de código fuente.
Las líneas de código fuente excluyen comentarios, líneas en blanco, y anotaciones usadas para asignaciones de aspectos. El inconveniente de CONC, es que no se tiene una idea de cómo se dispersa un aspecto, y no permite que sean comparables. Para resolver esto, fue creado el grado de Dispersión (DOS), así:

$$\boldsymbol{DOS(s)} = |\boldsymbol{T}| \left[\sum_{t}^{\boldsymbol{T}} \left(\boldsymbol{CONC(s,t)} - \frac{\mathbf{1}}{|\boldsymbol{T}|} \right)^{\mathbf{2}} \right] \left(\frac{\mathbf{1}}{|\boldsymbol{T}| - \mathbf{1}} \right)$$

Donde, $\boldsymbol{T}$ es el conjunto de componentes y $\boldsymbol{|T| > 1.}$

D. Grado de Enfoque

1) Objetivo

Medir de la varianza de la dedicación de un componente a cada aspecto con respecto al peor de los casos

2) Formula

La dedicación ($DEDI$), cuantas de las líneas de código contenidas en el componente t, están relacionadas con el aspecto s.

$$\boldsymbol{DEDI(t,s)} = \frac{\boldsymbol{SLOCs\ in\ componente\ t\ relacionado\ en\ un\ aspecto\ s}}{\boldsymbol{SLOCs\ relacionado\ en\ componente\ t}}$$

Otra vez, el inconveniente de que es difícil tener una idea de cómo los aspectos será separado de componentes. Para esto, fue creado el grado de enfoque (DOF), así:

$$\boldsymbol{DOF(t)} = |\boldsymbol{S}| \left[\sum_{s}^{\boldsymbol{S}} \left(\boldsymbol{DEDI(t,s)} - \frac{\mathbf{1}}{|\boldsymbol{S}|} \right)^{\mathbf{2}} \right] \left(\frac{\mathbf{1}}{|\boldsymbol{S}| - \mathbf{1}} \right)$$

Donde, S es el conjunto de componentes y $\boldsymbol{|S| > 1.}$ [17, pp. 2-3]

E. Cantidad de Aspectos

Al ser el aspecto la unidad básica de la POA es deseable conocer la Cantidad de aspectos que afectan al sistema total o a parte del mismo. Esto nos puede ayudar a justificar el uso de esta metodología ya que si la cantidad de aspectos es muy alta; esta metodología nos beneficia; en cambio si la cantidad es baja nos puede complicar el diseño en vez de mejorarlo.

F. Cantidad de relaciones existentes entre aspectos y una clase

Una clase puede ser atravesada por más de un aspecto, aumentando así la complejidad de la clase.

G. Cantidad de clases relacionadas con un mismo aspecto

Dado que varias clases se pueden relacionar con un mismo aspecto, es importante conocer esta cantidad para conocer así la complejidad del sistema. Ya que el diseño del comportamiento de aspectos puede verse afectado por esto.

H. Cantidad de puntos de enlace en una clase

Al ser los puntos de enlace los lugares en los que se agrega el comportamiento de los aspectos, esta cantidad varía de la cantidad de aspectos que atraviesan una clase ya que dicha clase puede ser atravesada por un mismo aspecto en diferentes oportunidades.

I. Cantidad de Clases Tejidas

La generación de la clase tejida depende del lenguaje OA que se utilice. En el caso que se generen clases tejidas el número de éstas me indica la complejidad y reutilización de la clase que intervino para generar la misma.

J. Reutilización de una clase

Se sabe que una clase se puede relacionar con más de un aspecto, cuanto más se relacione una clase con éstos mas disminuye la reutilización de la misma. [18, pp. 3-4]

K. Difusión de aspectos sobre componentes

Es una métrica de diseño que cuenta el número de componentes primarias cuyo objetivo principal es contribuir a la aplicación de un Aspecto. Además, se cuenta el número de componentes que acceden a los componentes primarios de su utilización en las declaraciones de atributos, parámetros formales, los tipos de devolución, declaraciones y las variables locales, o llamar a sus métodos.

L. Difusión de aspectos sobre operaciones

Cuenta el número de operaciones primarias cuyo objetivo principal es contribuir a la aplicación de un Aspecto. Además cuenta el número de métodos y avisos que tienen acceso a cualquiera de los componentes con sólo llamar a sus métodos o su utilización en los parámetros formales, los tipos de devolución, declaraciones y variables locales.

M. Difusión de aspectos sobre Líneas de Código fuente.

Cuenta el número de puntos de transición para cada aspecto a través de las líneas de código. El uso de este indicador requiere un proceso de sombreado que divide el código en zonas de sombra y las zonas no sombreadas.

N. Métricas de Unión

Es un indicador de la fuerza de las interconexiones entre los componentes en un sistema.

O. Uniones entre Componentes

Se define para un componente (clase o aspecto) como un recuento del número de otros componentes a los que está acoplado. Se cuenta el número de clases que se utilizan en las declaraciones de atributos

P. Profundidad del árbol de herencia.

Se define como la longitud máxima de un nodo al origen del árbol. Cuenta que tan abajo de la jerarquía de herencia una clase o aspecto es declarado

Q. Falta de Cohesión en operaciones.

Esta métrica mide la falta de cohesión de un componente. Si el componente C1 tiene $\boldsymbol{n}$ operaciones (métodos y avisos) $\boldsymbol{O_1, \ldots, O_n}$ entonces $\{I_j\}$ es el conjunto de las variables de ejemplo usadas por la operación O_j. Sea $|P|$ el número de intersecciones nulas entre los conjuntos de variables de ejemplo. Sea $|Q|$ el número de intersecciones no nulas entre variables de ejemplo.

R. Tamaño del Vocabulario

Cuenta el número de componentes en el sistema, clases y aspectos dentro del sistema. Esta métrica mide, el tamaño del vocabulario del sistema. Cada nombre de componente es contado como una parte del vocabulario del sistema.

S. Líneas de Código

Es el número de líneas de código.

T. Número de Atributos

Esta métrica cuenta el vocabulario interno de cada componente.

U. Peso de las Operaciones por Componente

Esta métrica, mide la complejidad de los componentes en términos de operaciones. Se obtiene, contando el número de parámetros de operación, asumiendo que una operación con más parámetros que otra va a ser más compleja. [19, pp. 5-7]

IV. CONCLUSIONES

- La POA es una metodología q mejora a la POO a la hora de desarrollar software, esta permite un seguimiento, mantenimiento y fácil manejo del software desarrollado.
- A la hora calificar el uso de esta metodología utilizamos una serie de herramientas llamadas métricas.
- En este trabajo se muestra como diferentes autores coinciden en el uso de las mismas métricas a lo largo de sus investigaciones.

- Observamos que los lenguajes orientados a aspectos actuales, no cuentan con mecanismos lingüísticos poderosos. Estos mecanismos son necesarios para respetar por completo todos los principios de diseño.
- Las métricas proporcionan una vista más objetiva del desarrollo que se realiza, esto es crucial en el momento de estimar recursos ya que no solo cumplir con los requisitos del cliente es importante, sino también limitar y estimar los recursos necesarios. Por esto, el análisis de las métricas aplicadas a distintas etapas del desarrollo de software es un tema que debe ser profundizado, en particular para la Orientación a Aspectos ya que su uso aumenta crecientemente.
- Para finalizar vemos como NO existe una métrica para evaluar conflictos entre aspectos.

V. REFERENCIAS

[1] A. Colyer y A. Clement, «Aspect-oriented programming with AspectJ,» *IBM Systems Journal*, vol. 44, nº 2, pp. 301-308, 2005.

[2] G. Murphy y C. Schwanninger, «Aspect-Oriented Programming,» *IEEE Software*, vol. I, nº 23, pp. 20-23, Enero 2006.

[3] M. Bartsch y R. Harrison, «An exploratory study of the effect of aspect-oriented programming on maintainability,» *Software Quality Journal*, vol. I, nº 16, pp. 23-44, Marzo 2008.

[4] D.-x. Xu, O. El-ariss, W.-f. Xu y L.-z. Wang, «Aspect-Oriented Modeling and Verification with Finite State Machines,» *Journal of Computer Science and Technology*, vol. 24, nº 5, pp. 949-961, September 2009.

[5] S. Manzanares Guillén y J. J. García Molina, «Programación Orientada a Aspectos: Una experiencia práctica con AspectJ,» Junio 2005. [En línea]. Available: http://dis.um.es/~jmolina/Proyecto%20POA-AspectJ.pdf.

[6] G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, L. C. Videira, J.-M. Loingtier y J. Irwin, «Aspect-Oriented Programming,» de *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, 1997.

[7] F. Asteasuain y B. E. Contreras, «Programación Orientada a Aspectos,» Octubre 2002. [En línea]. Available: http://lafhis.dc.uba.ar/~ferto/docs/tesis.pdf.

[8] A. Villazón, W. Binder, P. Moret y D. Ansaloni, «Comprehensive aspect weaving for Java,» *Science of Computer Programming*, vol. 76, nº 11, p. 1015–1036, 2011.

[9] R. Almonacid Arismendi. y S. Hernández Venegas, «Programación Orientada a Aspectos,» 28 Febrero 2008. [En línea]. Available: http://cybertesis.ubiobio.cl/tesis/2008/almonacid_r/doc/almonacid_r.pdf. [Último acceso: 8 Febrero 2013].

[10] M. S. Ali, M. Ali Babar, L. Chen y K.-J. Stol, «A systematic review of comparative evidence of aspect-oriented programming,» *Information and Software Technology*, vol. 52, nº 9, pp. 871-887, September 2010.

[11] S. Casas, H. Reinaga, L. Sierpe, V. Vanoli, C. Saldivia y J. Pryor, «Clasificación y Resolución de Conflictos entre Aspectos,» 2012. [En línea]. Available: http://sedici.unlp.edu.ar/bitstream/handle/10915/20383/Documento_completo.pdf?sequence=1.

[12] A. Fernandez, *Conflictos entre Aspectos en la Programación Orientada a Apectos*, Madrid, 2009.

[13] F. Tessier, L. Badri y M. Badri, «Towards a Formal Detection of Semantic Conflicts Between Aspects: A Model-Based Approach,» Lisboa, 2004.

[14] P. Durr, L. Bergmans y M. Aksit, «Static and Dynamic Detection of Behavioral Conflicts between Aspects.,» Centre for Telematics and Information Technology University of Twente, Enschede, 2007.

[15] S. Casas, *Administración de Conflictos entre Aspectos en AspectJ*, 2012.

[16] F. Asteasuain, B. Contreras, E. Estévez y P. Fillottrani, «Programación Orientada a Aspectos: Metodología y Evaluación,» de *IX Congreso Argentino de Ciencias de la Computación*, La Plata, 2003.

[17] M. Eaddy, A. Aho y G. Murphy, «Identifying, Assigning, and Quantifying Crosscutting Concerns,» de *First International Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07)*, Minneapolis, 2007.

[18] L. Baigorria, G. Montejano y D. Riesco, «MÉTRICAS C&K APLICADAS AL DISEÑO ORIENTADO A ASPECTOS,» de *XII Congreso Argentino de Ciencias de la Computación*, Argentina, 2006.

[19] C. Nogueira, A. García, C. Lucena, A. Von Staa y C. García, «On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework,» de *Proceedings XVII Brazilian Symposium on Software Engineering*, Rio de Janeiro, 2003.

Angelo Orozco (1983-) nació en Barranquilla, Colombia, el 7 de Octubre de 1983. Se graduó de Bachiller Académico en el Colegio Colón y es estudiante del Programa de Ingeniería Industrial de la Universidad Simón Bolívar.