Clasificación y Resolución de Conflictos entre Aspectos

Sandra Casas[1] **Héctor Reinaga**[1] **Luis Sierpe**[1] **Verónica Vanoli**[1] **Claudio Saldivia**[1] **Jane Pryor**[2]

[1]Unidad Académica Río Gallegos - Universidad Nacional de la Patagonia Austral.
Río Gallegos - Santa Cruz - Argentina. E-mail: {lis, vvanoli}@uarg.unpa.edu.ar

[2]ISISTAN Research Institute - Facultad de Ciencias Exactas, UNICEN.
Tandil - Buenos Aires - Argentina. E-mail: jpryor@exa.unicen.edu.ar

Resumen: En este trabajo se introduce la problemática de conflictos entre aspectos circunscripta al paradigma de programación orientada a aspectos. En base a una taxonomía para la resolución de conflictos se analizan brevemente algunas lenguajes orientados a aspectos. El análisis nos permite establecer algunos requisitos para diseñar e implementar herramientas mas abarcativas y seguras para el desarrollo de aplicaciones bajo este enfoque, en lo que respecta al manejo de conflictos entre aspectos.

1. Introducción

La Programación Orientada a Aspectos[1] (POA) es un nuevo paradigma de programación que soporta la separación de los componentes funcionales respecto de las competencias técnicas de un sistema, tales como la sincronización de procesos concurrentes, la gestión de errores, la distribución, la persistencia, el logging, la seguridad, la traza, etc. Estas competencias técnicas son propiedades circunstanciales que afectan a la performance de los componentes funcionales; cuando se encapsulan en módulos se denominan aspectos. El objetivo de la POA es el desarrollo de aplicaciones de software más fáciles de diseñar, codificar, mantener y reusar, superando los problemas de código mezclado y código diseminado[2].

La POA permite a los programadores codificar los aspectos de una aplicación de manera separada respecto de la funcionalidad básica. Los tres componentes fundamentales para desarrollar una aplicación POA son: un lenguaje de programación de propósito general que se utiliza para desarrollar los componentes de funcionalidad básica; un lenguaje de programación orientada a aspectos (LOA), y un tejedor (weaver). Los LOA son extensiones de lenguajes de programación existentes que incorporan los mecanismos necesarios para dar soporte a los aspectos. El 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[3].

El tejedor de aspectos 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 corte (pointcuts) que se asocian a los puntos de unión del código funcional.

En la mayoría de los LOA, es posible definir entidades de primera clase que representen a los aspectos. Esta construcción es una clase o una entidad muy parecida a una clase, en esencia es una unidad de código con un nombre y con variables y métodos propios.

En general, los aspectos cortan los componentes de funcionalidad básica para comunicar que cierto código se ejecutará antes o después o durante la ejecución de un evento (método, excepción, acceso a atributos, ...); introducir atributos y métodos; modificar la estructura jerárquica; etc. Estas características se pueden encontrar en AspectJ[4], y todos aquellos LOA que se basan en un este tipo de modelo, como AspectS[5], AspectC++[6], AspectC[7], Aurelia[8], Pythius[9], AspectR[10], etc.

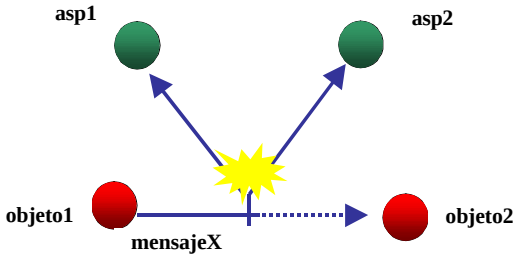
En el desarrollo de una aplicación POA es frecuente encontrar situaciones en las que un componente de funcionalidad básica esta afectado por más de un aspecto. Cada uno de éstos le adiciona diferente comportamiento. Los conflictos surgen cuando se ejecutan dos o más aspectos que están asociados a un mismo componente funcional, dado que esta situación

puede provocar un comportamiento impredecible o indeseado del sistema.

2. Conflictos

Una primera aproximación a una definición de conflicto sería:

Un conflicto ocurre cuando dos o más aspectos compiten por su activación[11].



Conflicto entre asp1 y asp2

En forma similar, se puede referir a los conflictos como interacciones de aspectos[12].
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. Por ejemplo, un aspecto específico debe ser ejecutado antes que otros, o la ejecución de dos aspectos producen inconsistencia de tal manera que se requiera la ejecución de sólo uno. En estos y en otros casos, el desarrollador debe poder especificar las potenciales situaciones y la ejecución deseada de acuerdo al tipo de conflicto y/o el dominio de la aplicación, determinando las prioridades y políticas de activación de los aspectos.
En un contexto más amplio, en el cual para cada elemento de programa (clase, objeto, método e instancia) se pueden asociar aspectos en forma independiente, existe una posible situación conflictiva que puede o no ser tratada. Dados dos aspectos asp1 y asp2, existe un conflicto potencial sí:

- **asp1** y **asp2** son asociados a la misma clase base.
- **asp1** y **asp2** son asociados al mismo método y clase base.
- **asp1** y **asp2** son asociados al mismo objeto base.
- **asp1** y **asp2** son asociados al mismo par objeto-método base.
- **asp1** es asociado a una clase y **asp2** es asociado a un método de la misma clase.
- **asp1** es asociado a una clase y **asp2** es asociado a un objeto de la misma clase.
- **asp1** es asociado a una clase y **asp2** es asociado a un objeto-método de una instancia de la clase.
- **asp1** es asociado a un método y **asp2** es asociado a una instancia de la clase a la cual el método pertenece.
- **asp1** es asociado a un método y **asp2** es asociado a un objeto-método del mismo método e instancia de la clase.
- **asp1** es asociado a un objeto y **asp2** es asociado a un objeto-método del mismo objeto.

En este sentido, y según las propiedades que deben cumplir los LOA definidas en[13], los conflictos entre aspectos suponen una posible violación del principio que denota que “los aspectos no deben interferir entre ellos”.
En general el manejo de conflictos es responsabilidad del programador, ya que las herramientas POA existentes brindan un escaso soporte para la evaluación o inspección anticipada de conflictos. Por ejemplo, en una aplicación en las que se codifican varios aspectos, es el programador el que debe llevar el registro y control de los posibles conflictos, y en consecuencia, aplicar los mecanismos que le ofrezca la herramienta para su resolución. La situación es más engorrosa y complicada en los casos que tratan de una aplicación existente, que ante nuevos requerimientos demanda la adición o remoción de aspectos, circunstancia que requiere del análisis de toda la arquitectura en búsqueda de los posibles nuevos conflictos.
En todos estos casos, se advierte que sería beneficioso y de gran ayuda que la tarea de detectar conflictos fuera automática. Esta función puede añadirse al tejedor de la herramienta o como un proceso anterior que se responsabilice de buscar y hallar potenciales conflictos antes de la compilación o ejecución de la aplicación.

3. Taxonomía de conflictos

La acción a realizar cuando un conflicto es detectado depende de las características de la aplicación. En algunos casos será necesario indicar un orden específico de ejecución de los aspectos involucrados. En otros casos, un conflicto puede requerir que uno o ambos aspectos no sean activados. Se han identificado seis categorías de conflictos para la activación de políticas cuando un conflicto es identificado: de Orden, de Orden Inverso, Opcional, Exclusivo, Nulo y Dependiente del Contexto, conformando una taxonomía para la resolución de conflictos[14].

A continuación cada una de estas estrategias se presenta:

- *de Orden:* en este tipo de conflicto, se establece un orden de ejecución para los aspectos involucrados en él.
- *de Orden Inverso:* en este tipo de conflicto, se establece un orden inverso de ejecución para los aspectos involucrados en él.
- *Opcional:* en algunos casos puede ser necesario que el sistema tome la decisión de qué aspecto ejecutar - en casos de haber varios compitiendo por ser activados -, dependiendo de alguna política interna o aleatoria.
- *de Exclusión:* resulta imprescindible contar con un tipo de conflicto cuando los aspectos en cuestión tienen una funcionalidad contradictoria, en este caso es necesario que sólo se ejecute uno de ellos, pero no ambos.
- *de Nulidad:* mediante este mecanismo se anula la ejecución de ambos competidores.
- *Dependiente del Contexto:* a diferencia de los mecanismos anteriores, el funcionamiento de este tipo de conflicto brinda la posibilidad de determinar los objetos que se van a ejecutar y en qué orden, dependiendo del contexto en el que se encuentran.

Según nuestra opinión, la taxonomía descripta es la manera más amplia y flexible porque ofrece la máxima cantidad de alternativas de resolución posible. En la realidad, la forma de resolución de conflictos depende en gran medida de los mecanismos que ofrezca el LOA, siendo en general muy pobres y restringidos, como se describe a continuación.

4. Resolución de Conflictos en POA

En la presente sección se analiza brevemente los dispositivos que proporcionan algunas herramientas POA con el propósito de resolver conflictos entre aspectos.

AspectJ

AspectJ[4] es una extensión de Java[15], de propósito general, considerada la herramienta POA de mayor popularidad y difusión. En AspectJ un aspecto puede declarar que los avisos (advices) que define, dominen sobre los avisos de otros aspectos, lo que implica que el aviso dominante tiene mayor precedencia que los especificados en los otros aspectos. Por defecto, en AspectJ no existe ningún orden definido, por lo que, si se precisa ejecutar los avisos en determinado orden, es necesario especificarlo con la cláusula *dominates*. La semántica es que si un aspecto A domina sobre un aspecto B, entonces los avisos de A tienen más prioridad y se ejecutan antes que los de B. Por lo tanto, AspectJ sólo cubre la categoría de orden de resolución de conflictos. En la versión 1.1 de AspectJ la sintaxis de *dominates* se ha reemplazado por: *declare precedence*: A, B; (los avisos del aspecto A tienen precedencia sobre los avisos del aspecto B).

DJ Aspect

DJ Aspect[16] es un prototipo desarrollado para el soporte de POA. Fue diseñado bajo una arquitectura reflexiva, utilizando para ello JMOP[17], un framework reflexivo construido en Java. DJ Aspect trata los conflictos de una manera simple, a través de la asignación de prioridades a los aspectos utilizando la instrucción *dominates*. De esta manera, si se desea dar precedencia a un aspecto de otro, se le asigna una prioridad más alta al primero (regla de precedencia).

Este esquema lo maneja el AspectManager a través del orden de la colección de aspectos que se le envía a activateMetaObjets(). Este orden es:

1. Se ejecutan los métodos *before* enviándolos a los aspectos ordenados por mayor prioridad.
2. Se ejecuta el mensaje interceptado.
3. Se ejecutan los métodos *after* enviándolos a los aspectos ordenados por menor prioridad.

4. Opcionalmente, se ejecutan los métodos *exception* ordenados por mayor prioridad.

Alpheus

Alpheus[14] es una herramienta visual diseñada para ayudar a los usuarios en el desarrollo de aplicaciones orientadas a aspectos. Aquí es importante el concepto de abstracción de conflictos al mas alto nivel de granularidad, permitiendo declaraciones en niveles de funcionalidad y no sólo entre aspectos. Soporta un manejo flexible de conflictos con la introducción de los planos, otorgándole facilidad en la definición y mantenimiento, siendo los niveles de granularidad de conflictos soportados: de aspecto-aspecto (determina un conflicto entre dos aspectos específicos); de aspecto-plano (establece conflictos entre un aspecto y plano); de plano-plano (especifica un conflicto entre todos los aspectos de un plano con respecto a todos los aspectos de otro plano); y de aspecto-todos (permite la especificación de conflictos "uno a muchos", donde un aspecto específico tiene un conflicto con todos los otros aspectos).

Alpheus soporta un manejo amplio de conflictos que son clasificados en: de orden, opcional, exclusivo, nulo o dependiendo del contexto. Permite que el usuario defina los conflictos potenciales entre aspectos o planos, especificando la política a seguir si el conflicto es detectado; esto se realiza identificando los componentes involucrados (por ejemplo, aspecto-aspecto, aspecto-plano), el tipo de conflicto (por ejemplo, en orden) y, en el caso de conflictos dependientes del contexto, el código específico según el conflicto detectado. Esta última característica sólo existe en Alpheus.

Para facilitar el diseño y especificación de los componentes de aplicación, sus asociaciones y conflictos, Alpheus ofrece diferentes vistas del sistema, posibilitando identificar dos grupos principales de diagramas: los relacionados a los planos, conflictos y niveles de asociación; y los relacionados a diagramas UML.

AspectC/C++

AspectC++[18] es una extensión de C++[19], de propósito general, muy similar a AspectJ. La precedencia es usada para determinar el orden de ejecución en el caso de que más de un aspecto afecte al mismo punto de unión. El compilador chequea las siguientes condiciones para determinar la precedencia de aspectos:

- Orden de declaración: el programador provee un orden de declaración que define la relación de precedencia entre dos aspectos para un punto de unión. La sintaxis de orden de declaración es la siguiente: *advice* pointcut-expr : *order* (high, ..., low). La lista de argumentos debe contener al menos dos elementos. Cada aspecto en una lista de argumentos tiene una precedencia mayor que los demás, por ejemplo, '("A1" || "A2", "A3" || "A4")', significa que A1 tiene una precedencia sobre A3 y A4, como así A2 tiene una precedencia sobre A3 y A4.
- Relación de herencia: mediante una relación de herencia se define que el aspecto derivado tiene una mayor precedencia sobre el aspecto base.

La precedencia de avisos es determinada a través de un esquema simple:

- si dos declaraciones de avisos pertenecen a distintos aspectos, se asumirá la misma relación de precedencia entre aspectos.
- si dos declaraciones de avisos pertenecen al mismo aspecto, entonces el primer aviso declarado tiene la mayor precedencia.

AOP/ST

AOP/ST[20] es una extensión de VisualWorks (SmallTalk) que soporta una limitada cantidad de puntos de unión. El autor hace referencia a las relaciones entre aspectos y define tres posibles formas:

1. Pre-requisito: un aspecto puede requerir de otro aspecto para ser tejido primero.
2. Orden de composición: los aspectos necesitan ser tejidos en un orden específico.
3. Validación de composición: algunas combinaciones de aspectos pueden no ser permitidas.

AOP/ST solo maneja la relación de orden, mediante un mecanismo sencillo por el cual se establecen las prioridades de composición/ejecución asignando un valor numérico a cada aspecto, así los aspectos con alta precedencia son ejecutados primero.

5. Conclusiones y trabajo futuro

La POA resulta ser una técnica muy prometedora para desarrollar aplicaciones de software más modulares. Pero la realidad es que aún se encuentra en un ciclo temprano en cuanto a su investigación y experimentación. Se destaca una considerable y explosiva aparición de los LOA, a la vez que la principal discusión se encuentra focalizada en los tipos de tejedores[3]. Es por este motivo, que aun no se ha profundizado sobre la problemática de conflictos entre aspectos.

Se observa en primer lugar, que la mayoría de las herramientas POA ofrecen mecanismos muy restringidos y pobres para la resolución de conflictos, comparados con la taxonomía propuesta. Concretamente, la mayoría de éstos ofrece la resolución que responde a la categoría de Orden.

En segundo lugar, detectar la presencia y existencia de conflictos es una tarea prácticamente manual, por estar ausente en la mayoría de las herramientas de forma automática, principalmente en aquellas de mayor uso y difusión.

El objetivo es el estudio de estrategias tendientes a la resolución de conflictos y su implementación en herramientas. En esta primer etapa el trabajo se centra en particular en el estudio de AspectJ, con el propósito de dotarlo de mecanismos automáticos para la especificación, detección y resolución de conflictos entre aspectos.

El presente trabajo fue parcialmente financiado por la UNPA, Santa Cruz, Argentina.

6. Referencias

[1] Mens K., Lopes C., Tekinerdogan B., Kiczales G. “Aspect-Oriented Programming”. Workshop Report ECOOP. 11th. Finland. 1997.

[2] Laddad R. “I want my AOP”. Part 1,2,3 from JavaWorld. 2002.

[3] Piveta E., Zancanela L. “Aspect Weaving Strategies”. Journal of Universal Computer Sciencie. Vol.9. Num.8. 2003.

[4] Kiczales G., Hilsdale E., Hugunin J., Kersten M. , Palm J., Griswold W. “An Overview of AspectJ”. ECOOP 2001.

[5] Hirschfeld R. “AspectS - AOP with Squeak”. In Proceedings of OOPSLA. Workshop on Advanced Separation of Concerns in Object-Oriented System. USA. 2001.

[6] Sitio Web de AspectC++: http://www.aspectc.org/

[7] Sitio Web de AspectC: www.cs.ubc.ca/labs/spl/projects/aspectc.html

[8] Piveta E., Zancanela L. “Aurelia: Aspect oriented programming using reflective approach”. Workshop on Advanced Separation of Concers ECOOP. 2001.

[9] Sitio Web de Pythius: http://sourceforge.net/projects/pythius/

[10] Sitio Web de AspectR: http://aspectr.sourceforge.net/

[11] Pryor J., Diaz Pace A., Campo M. “Reflection on Separation of Concerns”. RITA. Vol.9. Num.1. 2002.

[12] Douence R., Fradet P., Südholt M. “A Framework for the Detection and Resolution of Aspect Interactions”. Proceedings of the ACM SIGPLAN SIGSOFT Conference on GPCE. 2002.

[13] Cugola G., Ghezzi C., Monga M. “Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming”. WSDAAL. 1999.

[14] Pryor J., Marcos C. “Solving Conflicts in Aspect-Oriented Applications”. Proceedings of the Fourth ASSE. 32 JAIIO. Argentina. 2003.

[15] Gosling J., Joy B., Steel G. “The Java Languaje Specification”. Addison-Wesley. 1996.

[16] Pryor J., Bastán N., Campo M. “A Reflective Approach to Support Aspect Oriented Programming in Java”. In Proceedings of the ASSE. 29 JAIIO. Argentina. 2000.

[17] Zunino A. “Brainstorm/J: un framework para agentes inteligentes”. Master's Degree Dissertation. UNICEN. ISISTAN. 2000.

[18] Gal A., Scroder-Preikschat W., Spinczyk O. “AspectC++: Language Proposal and Prototype Implementation”. ACM International Conference Proceeding Series Proceedings of the Fortieth International Confernece on Tools Pacific. Vol.10. Australia. 2002.

[19] Stroustrup B. “The C++ Programming Languaje”. Addison-Wessley. 1991.

[20] Boellert K. “On Weaving Aspects”. Proceeding of the AOP Workshop at ECOOP. 1999.