Reuso de Aspectos en AspectJ

Verónica Vanoli[1], Sandra Casas[1], Claudia Marcos[2] y Eugenia Marquez[1]

[1]*Unidad Académica Río Gallegos. Universidad Nacional de la Patagonia Austral*
Lisandro de la Torre 1070. CP 9400. Río Gallegos. Santa Cruz. Argentina
Tel/Fax: +54-2966-442313/17.
E-mail:{vvanoli, lis}@uarg.unpa.edu.ar

[2]*ISISTAN Research Institute. Facultad de Ciencias Exactas. UNICEN*
Paraje Arroyo Seco. CP 7000. Tandil. Buenos Aires. Argentina
Tel/Fax: + 54–2293–440362/3.
E-mail: cmarcos@exa.unicen.edu.ar

ABSTRACT

La Programación Orientada a Aspectos (POA) permite implementar los requerimientos no funcionales de los sistemas software en unidades separadas denominadas aspectos. Ciertos aspectos son comunes a distintas aplicaciones de software, lo que hace necesario y beneficioso su reusabilidad. Pero no todas las herramientas y/o lenguajes de programación orientados a aspectos están preparados para dar soporte a esta importante característica, como es el caso de AspectJ. En este trabajo se presenta una estrategia sencilla que apunta a posibilitar el reuso del código de los aspectos en AspectJ.

1. INTRODUCCIÓN

Los mecanismos de abstracción de la mayoría de los lenguajes de programación (subrutinas, procedimientos, funciones, objetos, clases, módulos e interfaces) pueden adaptarse al modelo de procedimiento generalizado. El estilo de diseño que soporta este modelo divide un sistema en componentes parametrizados que pueden ser llamados en la ejecución de una función.

Los sistemas actuales tienen propiedades que no necesariamente se alinean con los componentes funcionales del sistema. Manejo de fallas, persistencia, comunicación, seguridad, replicación, coordinación, manejo de memoria, restricciones de tiempo real, etc., son aspectos del comportamiento de un sistema que tienden a cortar transversalmente a algunos grupos de componentes funcionales. Mientras que estos aspectos pueden ser analizados por separado de la funcionalidad básica, su programación, usando los actuales lenguajes orientados a componentes, tiene como resultado que el código de estos aspectos se esparce en muchos componentes. En consecuencia, el código fuente se convierte en una "maraña o enredo" de instrucciones de diferentes propósitos [1].

Este fenómeno ("tangling phenomenon") es la causa de la mayoría de la complejidad en los sistemas de software existentes. Incrementa las dependencias entre los componentes funcionales, distrae sobre lo que estos componentes deben hacer, introduce numerosas posibilidades de que ocurran errores de programación, y hace que los componentes funcionales sean menos reusables. En resumen, el "tangling phenomenon" hace que el código sea difícil de desarrollar, entender y evolucionar.

La Programación Orientada a Aspectos (POA) [2] es un nuevo paradigma que pretende reducir el problema del "tangling code" y por lo tanto, mejorar la calidad del código. La POA permite a los programadores expresar los aspectos de un sistema de manera separada respecto de la funcionalidad primaria. Un proceso denominado tejido (realizado por un nuevo tipo de compilador o intérprete) combina el programa de la funcionalidad primaria con los programas de los aspectos, para formar el programa ejecutable [3].

Los aspectos en el diseño, o los concerns, son entidades un tanto difusas. Tienden a ser unidades que cortan transversalmente (crosscut) a varios componentes en el diseño de acuerdo a alguna división natural de actividades, e interactúan con los componentes de acuerdo a interfaces bien definidas. En cambio, los aspectos de implementación son entidades más concretas, pero su definición precisa depende del lenguaje de aspectos. Pero en todos los lenguajes de aspectos, un aspecto es una unidad que encapsula estado, comportamiento y mejoras de comportamiento.

En el Desarrollo de Software Orientado a Aspectos (AOSD) [4] es frecuente encontrar situaciones en las que un mismo aspecto es requerido en diferentes aplicaciones. En el lenguaje de aspectos AspectJ [5] el reuso de los aspectos es complejo, ya que éstos hacen una referencia explícita a la parte funcional de la aplicación, desde el código. Esta imposición liga al aspecto a determinados componentes funcionales.
En este trabajo se presenta una propuesta para que los aspectos codificados en AspectJ puedan ser reusados. Se plantea la extensión a ASTOR para incluir un repositorio de aspectos genéricos y manejo de asociaciones, elementos en los cuales se basa la estrategia.

Este trabajo esta organizado de la siguiente manera: en la Sección 2 se proporciona una breve descripción de AspectJ, en la Sección 3 se describe la estrategia y mecanismos propuestos para reusar los aspectos en AspectJ, en la Sección 4 se analizan brevemente otros trabajos relacionados y finalmente se exponen las conclusiones.

2. ASPECTJ: REUSO DE ASPECTOS

AspectJ [5] es el lenguaje de programación orientado a aspectos más popular y difundido, inicialmente desarrollado en los laboratorios de Xerox PARC Research Center. AspectJ es una extensión del lenguaje Java [6], de propósito general. El modelo propuesto por AspectJ ha sido adoptado por otras herramientas como ser: AspectS [7], AspectC++ [8] [9], AspectC [10], Aurelia [11], Pythius [12], AspectR [13], etc.

Básicamente AspectJ agrega a la semántica de Java cinco entidades principales: puntos de unión (join point), puntos de corte (puntos de corte), avisos (advices), introducciones y aspectos [14]. Un aspecto constituye en AspectJ, la unidad modular o constructor que representa un tipo entrecruzado que corta las clases, interfaces y a otros aspectos mejorando la separación de concerns y haciendo posible localizar en forma limpia los conceptos de diseño. Un aspecto es un conjunto de puntos de cortes, avisos e introducciones. Con estos dispositivos, en AspectJ la implementación de una aplicación consiste en un conjunto de clases y aspectos. Esta descomposición de concerns, es compuesta por un proceso denominado tejido. En el caso particular de AspectJ, el tejido se realiza en tiempo de compilación, por lo cual genera código objeto conforme a la especificación Java byte-code, ejecutable por la JVM.

```
aspect Trace
  {
  static void print(String message)
     {                                          introduction
     …
     }

  pointcut creation(): call(Point.new());       pointcut -
  pointcut method(Point p): execution(* Point.*(*));  join point

  after(): creation()
    {
    Trace.print(thisJoinPoint.className+"new(…)");
    }

  before(Point p): method(p)                    advice
    {
    Trace.print(thisJoinPoint.className+"."+
          thisJoinPoint.methodName+" "+p.toString());
    }
  }
```

Fig. 1: Ejemplo de un aspecto Trace en AspectJ.

La Figura 1 muestra la implementación del aspecto Trace. Este aspecto modifica el comportamiento de la clase Point mediante los pointcut creation y method. Cada vez que se crea un objeto Point, el aspecto imprime la leyenda especificada en el advice after (asociada al pointcut creation) y cada vez que se ejecuta un método de la clase Point, se imprime la leyenda especificada en el advice before (asociado al pointcut method).

De esta manera, no ha sido necesario modificar el código de la clase Point para adicionar la traza del mismo.

De acuerdo a la sintaxis de AspectJ, los aspectos quedan ligados explícitamente a determinados puntos funcionales (join points) en la declaración de los pointcuts. Por ejemplo, en el aspecto Trace de la Figura 1, los advices solo son aplicables a los constructores y métodos de la clase Point. De esta forma, el código de los advices que se refiere específicamente al comportamiento no funcional no puede ser utilizado en otros componentes funcionales, por ejemplo, en la clase Line.

Ciertos requerimientos no funcionales, como el de persistencia y logging, suelen estar presentes en la mayoría de las aplicaciones software. En estos casos seria deseable reusar los aspectos que implementan estos comportamientos en la nueva aplicación. Dicho reuso como se puede desprender del ejemplo anterior no es posible realizar en AspectJ debido a que el aspecto hace una referencia explícita a los componentes funcionales que afecta. En la próxima sección se presenta una propuesta y estrategia para posibilitar el reuso del código de los aspectos en AspectJ.

3. EXTENSIÓN DE ASTOR

Astor [15] es una herramienta que propone una serie de mecanismos y estrategias para mejorar el tratamiento de conflictos entre aspectos en AspectJ [5]. Los mismos se soportan mediante la adición de un componente Administrador de Conflictos que cumple principalmente con las funciones de detectar automáticamente conflictos y aplicar estrategias de resolución más amplias que las que AspectJ tiene por defecto, en forma semiautomática. La detección de conflictos actúa por una clasificación de los mismos por niveles de semejanza [16] y la resolución se efectúa siguiendo las directrices de una taxonomía que proporciona seis categorías de resolución [17]. La implementación de la herramienta esta basada en el pre-procesamiento de código AspectJ, siendo además éste el único requisito para su uso.
Con la extensión de la herramienta Astor se pretende sumar beneficios y facilidades en el desarrollo orientado a aspectos en AspectJ.

Diseño Preliminar: definición de componentes

El diseño y arquitectura original de la herramienta Astor son lo suficientemente simples y adaptables para aceptar fácilmente extensiones que permiten ensayar y experimentar nuevos conceptos y propiedades. La propuesta se plantea en un diseño preliminar que se representa en el diagrama de clases de la Figura 2.

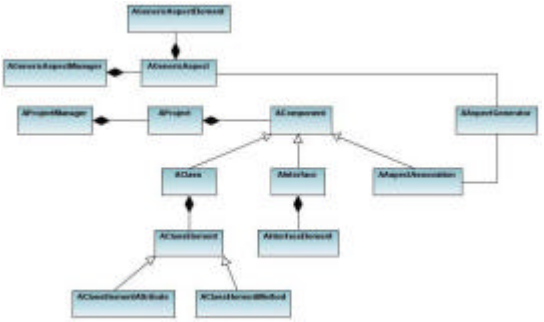

Fig. 2: Diagrama de clases extendido de Astor.

La definición de los siguientes componentes constituye el eje principal de la estrategia:

- Administrador de Repositorio de Aspectos Genéricos
- Administrador de Proyectos
- Generador de Aspectos

Administrador de Repositorio de Aspectos Genéricos: El Administrador del Repositorio de Aspectos Genéricos es el componente de software responsable de la gestión y manejo de los aspectos genéricos y el repositorio. Las funciones esenciales del mismo son las de agregar, remover, buscar y recuperar aspectos genéricos. El uso y utilidad del repositorio de aspectos genéricos será similar al de una API o librería de programas, pero a diferencia de éstas el repositorio será dinámico. El Administrador de Aspectos Genéricos (AGenericAspectManager) como los aspectos genéricos (AGenericAspect) que gestiona, son independientes de los proyectos de software particulares. Un aspecto genérico es una entidad compuesta de un conjunto de métodos independientes, es decir, no están relacionados o vinculados a ningún componente funcional, ni proyecto en particular. Los métodos de los aspectos genéricos corresponderían al código que se incluye en los advices de los aspectos en AspectJ. Así por ejemplo un aspecto genérico denominado Logging contendría todos los posibles métodos de acceso a un sistema.

Administrador de Proyectos: El Administrador de Proyectos (AProjectManager) gestiona los Proyectos (AProject) de software. Los proyectos se componen de componentes (AComponent). Un componente puede ser una clase (AClass), una interface (AInterface) o una asociación (AAspectAssociation).
Las asociaciones son las unidades que permitirán vincular un elemento de un aspecto genérico (método) a un componente funcional determinado (join point). Las asociaciones son particulares a cada proyecto. Una asociación tiene la siguiente estructura de información:

Asociación = (aspecto genérico (nombre y método), pointcut, tipo de cut[1], join point, tipo de advice[2])

Básicamente las asociaciones permiten definir pointcuts pero en unidades separadas a los aspectos. Un join point puede tener múltiples asociaciones, no existen restricciones en este sentido. Al momento de establecer asociaciones resulta necesario efectuar una serie de validaciones que garanticen la definición de asociaciones correctas, por ejemplo: (i) existencia del aspecto genérico, del join point, de la clase, etc.; (ii) coherencia de pointcuts; (iii) campos vacíos y necesarios; (iv) parametrización.

Generador de Aspectos: El generador de aspectos es el componente de software responsable de crear en forma automática los aspectos de acuerdo a la sintaxis de AspectJ, según lo que las asociaciones indican. El generador de aspectos (AAspectGenerator) obtiene información de las asociaciones y el repositorio de aspectos genéricos para crear los aspectos en AspectJ.

Desarrollo de Aplicaciones en Astor

Mediante este enfoque, el desarrollo de una aplicación orientada a aspectos procede de la siguiente manera:
1) se codifican las unidades funcionales (clases e interfaces);
2) se establecen las asociaciones necesarias entre unidades funcionales y aspectos genéricos y
3) finalmente el proceso denominado "Aspect Generator" genera los aspectos en AspectJ, en forma automática.
De aquí en adelante se prosigue con la detección y resolución de conflictos y la compilación de la aplicación. De esta forma, el desarrollador se centra en el establecimiento de las asociaciones y la actualización del repositorio de aspectos genéricos en los casos que resulte necesario.

En la Figura 3 se proporciona un ejemplo sencillo de la propuesta. El aspecto genérico Persistence cuenta con los métodos save y load, los cuales no se encuentran ligados o relacionados a ninguna clase, método, atributo o interface, es simplemente código Java. Se establecen las asociaciones ClientSave y ClientRetrieve, que relacionan al aspecto genérico con los métodos get y set de la clase Client. Por último, el generador de aspectos crea en forma automática los aspectos en AspectJ, que se denominan de la misma manera que las asociaciones. En este ejemplo, el aspecto genérico Persistence cuenta sólo con dos métodos, pero la idea es que contenga todos los métodos posibles relacionados al comportamiento de persistencia.

[1] Tipos de cut (corte) en AspectJ: call, execution, set, etc.
[2] Tipos de advice en AspectJ: before, after, around.

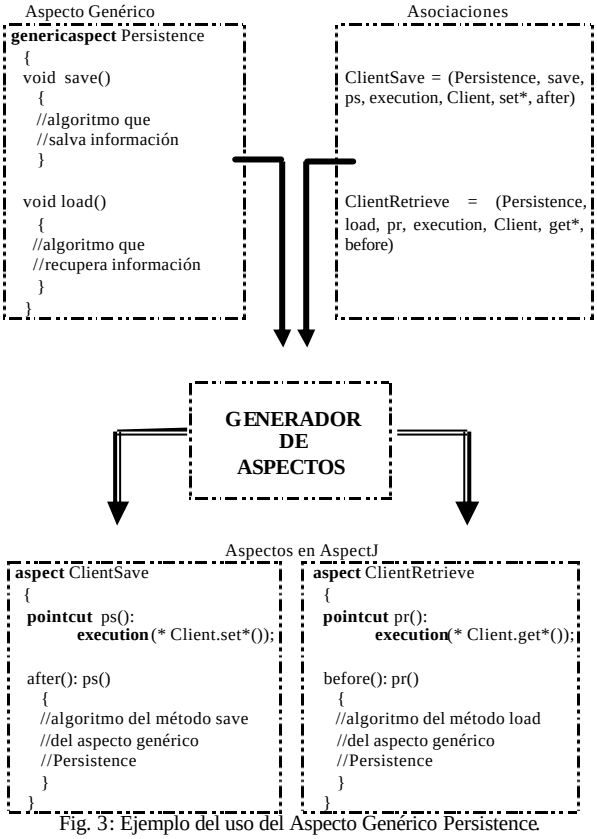

Fig. 3: Ejemplo del uso del Aspecto Genérico Persistence.

El código de los métodos save y load del aspecto genérico Persistence, pueden ser utilizados en otros proyectos simplemente definiendo nuevas asociaciones a otros puntos de unión (métodos, atributos y clases). De esta manera el desarrollador puede contar con el repositorio de aspectos genéricos, los cuales serán utilizados en las aplicaciones que los requieran. El código de los aspectos genéricos será transparente para el programador quien además no tendrá la necesidad de aprender nuevos conceptos introducidos en AspectJ. La implementación de los aspectos se realiza de la misma manera que se codifica una clase en Java.

4. TRABAJOS RELACIONADOS

El trabajo [18] propone una arquitectura general para construir un tejedor de aspectos reusables. A partir de la separación de aspectos en dos partes: la semántica de los aspectos y los join points, en la que utilizan palabras claves específicas. Esta separación es particular al tejedor, por lo tanto, ambos tienen que estar siempre ligados. Por el contrario, la propuesta de este trabajo se basa en un enfoque de pre-procesamiento de la herramienta, sin tomar intervención alguna en el tejedor de AspectJ (ajc). Esta propiedad permite que fácilmente se pueda aplicar a otras herramientas como AspectC++, ya que solo requiere la modificación del componente "Aspect Generator".

Con respecto a la propuesta de [19] existe una leve similitud en el hecho de establecer las declaraciones de los pointcuts separadas, tanto la declaración como la definición de los pointcuts del resto de los aspectos, pero se difiere en el resto de las características: (i) no se debe usar un pointcut para más de un advice; (ii) los aspectos concretos (aquellos que no contienen alguna definición de pointcut) están siempre vacíos y (iii) El reuso de aspectos se encuentra particularmente aplicado al enfoque de la herencia de aspectos.

Otros trabajos que proponen ideas relacionadas como [20] cuyo enfoque apunta directamente a las relaciones de dependencia entre aspectos: ortogonales, unidireccionales y circulares; [21] se basa en la construcción de aspectos de aspectos y [22] que aplica aspectos genéricos dentro de la programación generativa. Algunos de estos trabajos no profundizan demasiado las estrategias y mecanismos más específicos, dificultando su evaluación y comparación.

5. CONCLUSIONES

El objetivo de la propuesta es lograr un mayor reuso del código de los aspectos en AspectJ. La estrategia adoptada se basa en la definición de las entidades denominadas aspectos genéricos y asociaciones y un proceso que automáticamente genera el código de los aspectos en AspectJ. De esta forma, el desarrollador de aplicaciones software escribe el código una sola vez y lo reutiliza tantas veces como sea necesario.

AspectJ es el lenguaje de aspectos más popular y difundido, no solo porque es el más utilizado sino además porque una basta cantidad lenguajes siguen su modelo de estructura, y solo se diferencian en el lenguaje base que extienden, como ser AspectC++. Esta propuesta es válida y aplicable también en estos casos, ya que solo requiere modificar el componente Aspect Generator.

La propuesta se proyecta como extensión de la herramienta Astor, para obtener mayores beneficios y flexibilidad en el desarrollo de sistemas orientados a aspectos.

El trabajo actual y futuro esta referido a la implementación de la estrategia planteada en la herramienta Astor y prueba de casos de distinta complejidad.

El presente trabajo fue parcialmente financiado por la Universidad Nacional de la Patagonia Austral, Santa Cruz, Argentina.

6. REFERENCIAS

[1] Hürsch W., Lopes C. "Separation of Concerns". Northeastern University Technical Report NU-CCS-95-03, Boston, February 1995.

[2] Kiczales G., Lamping L., Mendhekar A., Maeda C., Lopes C., Loingtier J., Irwin J., "Aspect-Oriented Programming". In Proceedings ECOOP'97

– Object-Oriented Programming, 11th European Conference, Jyväskylä Finland, Springer-Verlang, 1.997.
[3] Piveta E., Zancanela L. “Aspect Weaving Strategies”. Journal of Universal Computer Sciencie. Vol.9. Num.8 (2003).
[4] Gregor Kiczales. AOSD 2002. 1st. International Conference on Aspect-Oriented Software Development. (Ed.). ACM Press. The Netherlands (2002).
[5] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G. “An Overview of AspectJ”. In Proc. of the 15th ECOOP. Pp. 327-357. Budapest, Hungary. June 2001.
[6] Gosling J., Joy B., Steel G. “The Java Languaje Specification”. Addison-Wesley (1996).
[7] Hirschfeld R. “AspectS - AOP with Squeak”. In Proceedings of OOPSLA. Workshop on Advanced Separation of Concerns in Object-Oriented System. USA (2001).
[8] Homepage de AspectC++: http://www.aspectc.org/
[9] 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).
[10] Homepage de AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html
[11] Piveta E., Zancanela L. “Aurelia: Aspect oriented programming using reflective approach”. Workshop on Advanced Separation of Concers ECOOP (2001).
[12] Homepage de Pythius: http://sourceforge.net/projects/pythius/
[13] Homepage de AspectR: http://aspectr.sourceforge.net/
[14] Homepage de AspectJ™, Xerox Palo Alto Research Center (Xerox Parc), Palo Alto, California: http://aspectj.org
[15] Casas S., Marcos C., Vanoli V., Reinaga H., Sierpe L, Pryor J., Saldivia C. “Administración de Conflictos entre Aspectos en AspectJ”. 34ª Jornadas Argentinas de Informática e Investigación Operativa (JAIIO 2005). VI Argentine Symposium on Software Engineering (ASSE 2005). Rosario, Santa Fe. Agosto/Septiembre 2005.
[16] Casas S., Marcos C., Vanoli V., Reinaga H., Saldivia C., Pryor J., Sierpe L. “ASTOR: Un Prototipo para la Administración de Conflictos en AspectJ”. XIII Encuentro Chileno de Computación (ECC). Jornadas Chilenas de Computación - UACh (JCC). Valdivia (X Región), Chile. Noviembre de 2005.
[17] Pryor J., Marcos C. “Solving Conflicts in Aspect-Oriented Applications”. Proceedings of the Fourth ASSE. 32 JAIIO. Argentina (2003).
[18] Beugnard A. “How to make aspects re-usable, a proposition”. Position paper at the ECOOP. Workshop on Aspect-Oriented Programming. Lisboa, Portugal. Junio 1999.
[19] Hanenberg S, Unland R. “Using and Reusing Aspects en AspectJ”. Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA. Tampa. October 2001.
[20] Kienzle J., Yu Y., Xiong J. “On Composition and Reuse of Aspects”. In Proc. of 2nd fundations of Aspect-Oriented Languages Workshop at AOSD. Pp. 17-24. Boston, MA. March 2003.
[21] Panas T., Andersson J., Assmann U. “The editing Aspect of Aspects”. In I. Hussain, editor, Software Engineering and Applications (SEA). Cambridge, MA, USA. Acta Press. November 2002.
[22] Silaghi R., Strohmeier A. “Better Generative Programming with Generic Aspects”. Second International Workshop on Generative Techniques in the Context of MDA, held at the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA. Anaheim, CA, USA. October 2003.