ASPECTOS GENÉRICOS Y ASOCIACIONES
Una Propuesta para Reutilizar Aspectos en AspectJ

Verónica Vanoli, Sandra Casas, Eugenia Márquez
Unidad Académica Río Gallegos, Universidad Nacional de la Patagonia Austral
Río Gallegos, Santa Cruz, Argentina
e-mail: {vvanoli, lis}@uarg.unpa.edu.ar

and

Claudia Marcos
ISISTAN Research Institute, Facultad de Ciencias Exactas, UNICEN
Tandil, Buenos Aires, Argentina
e-mail: cmarcos@exa.unicen.edu.ar

Abstract

In aspects-oriented development is necessary to reuse the aspects, since certain non-functional behaviours are common in different software applications. AspectJ provides a mechanism based on the extension of abstract aspects. This scheme presents certain limitations. In this work an alternative proposal to reuse aspects implemented in AspectJ is presented. The strategy is based on the creation of a generic aspects repository and the handling of associations. These mechanisms are used to generate the concrete aspects automatically.

Keywords: Aspect-Oriented Programming, Aspect Reuse, Generic Aspects, Associations.

Resumen

En el desarrollo orientado a aspectos es necesario reutilizar los aspectos, dado que ciertos comportamientos no funcionales son comunes en distintas aplicaciones de software. AspectJ proporciona un mecanismo basado en la extensión de aspectos abstractos que presenta ciertas limitaciones. En este trabajo se presenta una propuesta alternativa para la reutilización de aspectos codificados en AspectJ. La estrategia se basa en la generación de un repositorio de aspectos genéricos y manejo de asociaciones que permiten vincular a éstos con aspectos concretos en AspectJ creados automáticamente.

Palabras claves: Programación Orientada a Aspectos, Reutilización de Aspectos, Aspectos Genéricos, Asociaciones.

1 INTRODUCCION

Una aplicación de software esta compuesta por un conjunto de propiedades o áreas de interés (*concerns*). En el desarrollo de software un concern será un servicio o una funcionalidad del sistema que puede ser identificada y limitada. Los concerns son de diferente naturaleza y especie: temas referidos a las reglas de negocio, la interfaz de usuario, la seguridad, la persistencia, el registro de actividades, las comunicaciones, etc.

Las herramientas de desarrollo (lenguajes de programación, notaciones de diseño) están centradas en la modularización de unidades funcionales (clases, objetos, funciones, etc.), por lo tanto, los demás concerns entrecruzan y atraviesan las unidades funcionales para adicionar comportamiento que responde principalmente a requerimientos de diseño e implementación. Los mecanismos de abstracción e implementación de las herramientas de desarrollo convencionales no soportan la separación de aquellos concerns diferentes a la funcionalidad básica. De esta manera, el código resultante de cada módulo responde a varios objetivos y se torna confuso; simultáneamente el comportamiento de los concerns esta disperso y duplicado en distintos módulos de la aplicación. Las desventajas más significativas son [8]: código de mala calidad, trazabilidad pobre, baja productividad, baja reusabilidad, dificultades en el testing, mala adaptabilidad y evolución pobre.

La Programación Orientada a Aspectos [14] permite a los programadores codificar estas propiedades técnicas, que se entrecruzan en unidades llamadas aspectos, separadas de la funcionalidad básica. Los beneficios que se obtienen al aplicar una mayor descomposición es que las aplicaciones de software resultan más fáciles de diseñar, codificar, mantener y reutilizar, superando los problemas de código mezclado y código diseminado [12].

Los Lenguajes de Programación Orientados a Aspectos (LOA) son extensiones de lenguajes de programación existentes que incorporan los mecanismos necesarios para dar soporte a los aspectos. En la mayoría de los LOA, es posible definir unidades de descomposición primaria que representen aspectos. Un aspecto es una entidad muy parecida a una clase, en esencia es una unidad de código con un nombre, variables y métodos propios. 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 [17]. 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 *join points* (puntos de unión) y podrían interpretarse como eventos que al ocurrir, activan la ejecución de un aspecto. Desde la perspectiva de los aspectos, éstos incluyen *pointcuts* (puntos de corte) que se asocian a los puntos de unión del código funcional.

El LOA más popular y difundido es AspectJ [9]. Este lenguaje es una extensión de Java [5], de propósito general, que proporciona una nueva unidad de modularización llamada aspecto. Los aspectos cortan las clases, interfaces y a otros aspectos mejorando la separación de concerns y haciendo posible localizar en forma limpia los conceptos de diseño. El tejedor de AspectJ (ajc) realiza la composición de aspectos y clases, y compila la aplicación, generando código objeto conforme a la especificación Java byte-code, ejecutable por la JVM (Java Virtual Machine). El modelo propuesto por AspectJ ha sido adoptado por otras herramientas como son: AspectS [7], AspectC++ [21] [4], AspectC [20], Aurelia [16], Pythius [23], AspectR [22], etc.

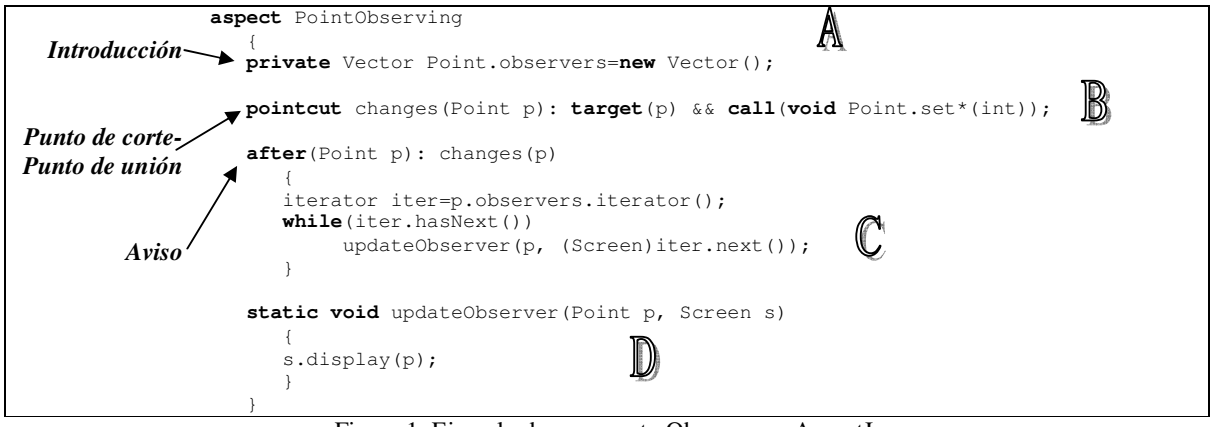
En el Desarrollo de Software Orientado a Aspectos (DSOA) [10] es frecuente encontrar situaciones

en las que un mismo aspecto puede ser utilizado en diferentes aplicaciones. En AspectJ la reutilización de aspectos es limitada, ya que éstos quedan explícitamente ligados a ciertos componentes funcionales (en la definición puntos de corte y puntos de unión) y/o de ciertas formas (definición de avisos). Algunos de estos inconvenientes pueden ser superados mediante el uso de aspectos abstractos. En este trabajo se presenta una propuesta alternativa para la reutilización de aspectos codificados en AspectJ. Se plantea la extensión de la herramienta ASTOR [3] para soportar la estrategia que se basa en la generación de un repositorio de aspectos genéricos y manejo de asociaciones, que permiten vincular a estos aspectos genéricos con los aspectos concretos o subaspectos en AspectJ creados automáticamente.

Este trabajo esta organizado de la siguiente manera: en la Sección 2 se analizan las dificultades que presenta la reutilización de aspectos en AspectJ; en la Sección 3 se describe la estrategia y mecanismos propuestos para reutilizar el código no funcional en AspectJ, mediante la extensión de la herramienta ASTOR; en la Sección 4 se analiza brevemente otros trabajos relacionados; y finalmente, en la Sección 5, se exponen las conclusiones.

2 DIFICULTADES PARA REUTILIZAR ASPECTOS EN ASPECTJ

En AspectJ, la porción de código que representa a un requerimiento no funcional en un aspecto, es el código que corresponde a los avisos e introducciones. El modelo de estructura sintáctica y semántica de AspectJ establece que la declaración tanto de los puntos de corte como de las introducciones se conforme obligatoriamente de elementos que obstaculizan su posterior reutilización. La inclusión de estos elementos de manera estática dependizan la implementación del aspecto de manera muy fuerte a determinados componentes funcionales y/o formas de corte. Los tres elementos obligatorios que ocasionan esta dependencia son los puntos de unión, designadores de cortes y avisos. En la Figura 1 se representa al aspecto *PointObserving*, cuyo objetivo es monitorear las modificaciones de los objetos de la clase *Point* y actualizar en consecuencia el objeto *Screen*. Este aspecto básicamente captura las llamadas a los métodos de la clase *Point* que comienzan con el prefijo *set* con un argumento del tipo *int*, mediante la declaración del punto de corte *changes* (B). En consecuencia, luego que una llamada a un método es atrapada, se ejecuta el aviso *after* asociado (C), el cual actualiza un objeto *Screen*. Además, se ha introducido (A) un nuevo atributo a la clase *Point*, denominado *observers*. El método *updateObserver* (D) es una función del aspecto.




```
aspect PointObserving
   {
private Vector Point.observers=new Vector();

pointcut changes(Point p): target(p) && call(void Point.set*(int));

after(Point p): changes(p)
   {
   iterator iter=p.observers.iterator();
   while(iter.hasNext())
        updateObserver(p, (Screen)iter.next());
   }

static void updateObserver(Point p, Screen s)
   {
   s.display(p);
   }
}
```

Figura 1: Ejemplo de un aspecto Observer en AspectJ.

La reutilización del aspecto *PointObserving* esta limitada por lo siguiente:

- Puntos de Unión: El punto de corte *changes* esta vinculado a todos los métodos cuyo nombre comiencen con el prefijo *set* de la clase *Point*. No pudiendo ser aplicado a otros métodos de la misma clase o a otras clases. De manera similar, la introducción del atributo *observer* se ha restringido a la clase *Point*.
- Designadores de corte: El punto de corte *changes* indica que se activa únicamente por medio de la llamada al punto de unión, ya que se ha utilizado el designador *call*. No pudiendo ser aplicado a otros designadores válidos. Para este caso, podría escogerse el designador *execution*.
- Avisos: El punto de corte *changes* esta relacionado a un aviso *after*, no pudiendo ser aplicado *before* si se requiere antes.

Si se necesita un aspecto *observer* para otro tipo de objeto en la misma u otra aplicación, a pesar que algorítmicamente sea igual al aspecto *PointObserving*, se deberá codificar un nuevo aspecto. Esto significa que el código de los avisos e introducciones que se refiere específicamente al comportamiento del requerimiento no funcional no puede ser utilizado en relación a otros componentes funcionales y/o de otras formas. La limitación planteada tiene por origen las reglas de tejido que aplica el tejedor de AspectJ (ajc), que requiere esta información para efectuar la composición.

Para superar en parte estas limitaciones se puede optar por utilizar aspectos abstractos. Los aspectos abstractos son un mecanismo para reutilizar la implementación o lógica de los avisos, al diferir ciertos detalles de la implementación a los aspectos concretos. Un aspecto abstracto puede declarar los puntos de corte o métodos como abstractos, lo que permite al aspecto base implementar la lógica sin la necesidad de definir el punto de unión. Por ejemplo, en la Figura 2 se presenta el aspecto abstracto *AbstractLogging*. En el mismo, el punto de corte *logPoints()* se ha declarado como abstracto con el propósito de que un aspecto concreto le proporcione una definición exacta. Similarmente el método abstracto *getLogger()*. A pesar de las declaraciones abstractas, el aviso *before* asociado al punto de corte *logPoint()* utiliza ambas entidades abstractas para realizar su tarea. El objetivo es que la lógica de logging queda embebida en el aviso, mientras que cada aspecto concreto completará en detalle el punto de unión y designadores de corte específicos de dicho aspecto.

```
public abstract aspect AbstractLogging
   {
   public abstract pointcut logPoints();

   public abstract Logger getLogger();

   before(): logPoints()
      {
      getLogger().log(Level.INFO, "Before: " + thisJoinPoint);
      }
   }
```

Figura 2: Aspecto abstracto AbstractLogging en AspectJ.

En la Figura 3 se muestra como un aspecto concreto proporciona una definición exacta para los puntos de corte y métodos abstractos. En principio el aspecto concreto *BankLogging* extiende del aspecto abstracto *AbstractLogging*. Mas adelante el punto de corte *logPoints()* se vincula a un componente funcional especifico (la llamada a todos los métodos de la clase *Banking*). Por último,

proporciona una implementación acorde al método *getLogger()*.

```
public aspect BankLogging extends AbstractLogging
   {
   public pointcut logPoints(): call(* Banking.*(..));

   public Logger getLogger()
      {
      return Logger.getLogger("Banking");
      }
   }
```

Figura 3: Aspecto Concreto BankLogging en AspectJ.

El uso de aspectos abstractos es una forma de reutilización, sin embargo debe notarse que el tipo de aviso ha sido definido en el aspecto abstracto imposibilitando su redefinición; un aspecto concreto puede extender sólo de un aspecto abstracto imposibilitando que el mismo reutilice avisos de más un aspecto abstracto. Las introducciones de atributos y métodos definidas en los aspectos abstractos deben ser redefinidas en los aspectos concretos si es necesario introducirlos en otras clases. Estos factores hacen que este mecanismo no siempre sea útil y aplicable. Otros inconvenientes del uso de aspectos abstractos se reporta en [13], los cuales son indicados por los autores como: la carencia de soporte para la multi-abstracción de aspectos, problemas en el mapeo de aspectos abstractos y la declaración *declare parents*, ligaduras (binding) de aspectos acoplado a la implementación y pérdida del polimorfismo aspectual.

3 ESTRATEGIAS Y MECANISMOS DE REUTILIZACION DE ASPECTOS CON LA HERRAMIENTA ASTOR

Con la extensión de la herramienta ASTOR [2] se pretende sumar beneficios y facilidades en el desarrollo orientado a aspectos en AspectJ. La estrategia para lograr la reutilización de aspectos se basa en incorporar a la herramienta nuevas características y funcionalidades.

3.1 Herramienta ASTOR

ASTOR es una herramienta que soporta una serie de mecanismos y estrategias [2] para mejorar el tratamiento de conflictos entre aspectos en AspectJ [9]. Los mismos se basan en 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 [3] y la resolución se efectúa siguiendo las directrices de una taxonomía que proporciona seis categorías de resolución [18]. La implementación de la herramienta esta basada en el pre-procesamiento de código AspectJ, siendo además éste el único requisito para su uso.

3.2 Propuesta de Extensión

La propuesta se basa en extender la herramienta ASTOR con nuevos componentes para soportar los mecanismos necesarios para reutilizar aspectos. El conjunto de características incorporado a la herramienta está compuesto por: Aspectos Genéricos, Administrador de Repositorio de Aspectos Genéricos, Asociaciones y Generador de Aspectos.

Aspectos Genéricos: Un aspecto genérico es una entidad compuesta de un conjunto de elementos (métodos y/o atributos) independientes, es decir, no están relacionados o vinculados a ningún componente funcional, ni proyecto en particular. Los elementos de los aspectos genéricos corresponderían al código que se incluye en los avisos e introducciones de los aspectos en AspectJ. Así por ejemplo un aspecto genérico denominado *Logging* contendría todos los posibles elementos para de acceder a un sistema.

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.

Asociaciones: Las asociaciones son las unidades que permitirán vincular un elemento de un aspecto genérico (método/atributo) a un componente funcional determinado (punto de unión). Las asociaciones son particulares a cada proyecto. Se distinguen dos tipos de asociaciones: asociaciones de corte y asociaciones de introducción.

La estructura básica de los dos tipos de asociaciones sería:

```
Asociación de Corte = (nombre de la asociación, aspecto genérico (nombre y elemento),
          nombre del punto de corte, {designador de corte, punto de unión,
                [operador algebraico]}, tipo de aviso)

          Asociación de Introducción = (nombre de la asociación,
               aspecto genérico (nombre y elemento), clase)
```

Básicamente las asociaciones permiten definir puntos de corte e introducciones pero en unidades separadas a los aspectos. Un punto de unión 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 punto de unión, etc.; (ii) coherencia de puntos de corte; (iii) campos vacíos y necesarios; (iv) parametrización. El proceso de creación de asociación requiere una interactividad, donde el desarrollador participa en la selección de los elementos que componen dicha asociación y así luego, un proceso automático genera la asociación determinada, creando el aspecto respectivo propio del proyecto.

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.

3.3 Diseño y Ejemplo

El diseño resultante de la herramienta ASTOR se indica en la Figura 4. Los nuevos componentes (AGenericAspectManager, AGenericAspect, AGenericAspectElement, AaspectAssociation, AAspectGenerator) se adicionan sin alterar el diseño general de la herramienta. 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). El Administrador de Aspectos Genéricos (AGenericAspectManager) como los aspectos genéricos (AGenericAspect) que gestiona, son independientes de los proyectos de software particulares. El generador de aspectos (AAspectGenerator) obtiene información de las asociaciones para crear los aspectos en AspectJ.

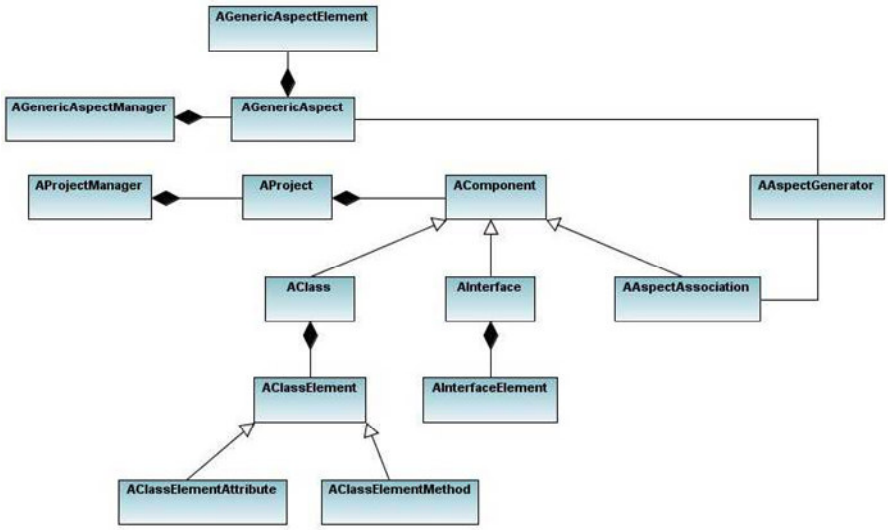


Figura 4: Diagrama de clases extendido de la herramienta ASTOR.

Mediante este enfoque, el desarrollador de una aplicación orientada a aspectos procede de la siguiente manera:

1) codificar las unidades funcionales (clases e interfaces);
2) establecer 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 [3]. 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 resulten necesarios.

En la Figura 5 se proporciona un ejemplo básico de la propuesta. El aspecto genérico *Logging* cuenta con un único método *checkIn*, el cual no esta ligado o relacionado a ninguna clase, método, atributo o interface, es simplemente código Java. Se establecen dos asociaciones *BankLogg* y *LibLogg* para proyectos diferentes, que relacionan al aspecto genérico con las clases *Account* y *BookManager*, y los métodos *extract* y *borrow* respectivamente. 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 *Logging* cuenta con un único método, pero la idea es que éste contenga todos los métodos posibles relacionados al comportamiento de logging.

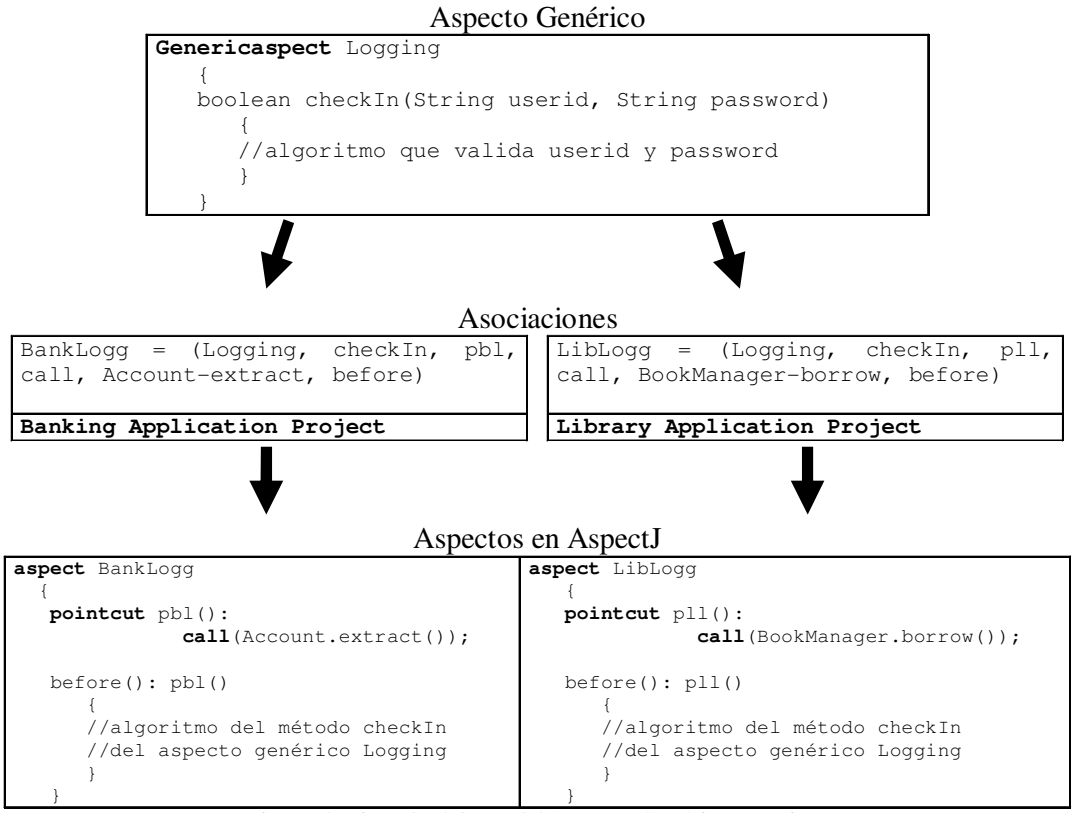


Figura 5: Ejemplo del uso del Aspecto Genérico Logging.

El código del método *checkIn* se ha utilizado en proyectos diferentes con un mínimo de esfuerzo por parte del desarrollador. De esta manera el desarrollador puede contar con el repositorio de aspectos genéricos, los cuales serán utilizados en las aplicaciones que los requieran. La implementación de los aspectos se realiza de la misma manera que se codifica una clase en Java. Con el objeto de ilustrar la esencia de la propuesta se ha simplificado la definición de las asociaciones, pero para una completa adecuación de las mismas, se podrá incluir combinaciones de cortes primitivos mediante el uso de conectores y comodines.

4 TRABAJOS RELACIONADOS

El trabajo [1] propone una arquitectura general para construir un tejedor de aspectos reutilizables. A partir de la separación de aspectos en dos partes: la semántica de los aspectos y los puntos de unión, 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 herramienta ASTOR se basa en un enfoque de pre-procesamiento, sin tomar intervención alguna en el tejedor de AspectJ (ajc). Similarmente, éste enfoque se puede aplicar a otras herramientas como AspectC++, ya que requerirá principalmente la modificación del componente “Aspect Generator”.

La extensión de la herramienta ASTOR guarda cierta similitud con respecto a la propuesta de [6]. En dicho trabajo también se trata de establecer las declaraciones de los puntos de corte separadas, tanto la declaración como la definición de los puntos de corte del resto de los aspectos, pero se difiere en el resto de las características: (i) no se debe usar un punto de corte para más de un aviso;

(ii) los aspectos concretos (aquellos que no contienen alguna definición de punto de corte) están siempre vacíos y (iii) la reutilización de aspectos se encuentra particularmente aplicado al enfoque de la herencia de aspectos.

Otros trabajos que proponen ideas relacionadas como [11] cuyo enfoque apunta directamente a las relaciones de dependencia entre aspectos: ortogonales, unidireccionales y circulares; [15] se basa en la construcción de aspectos de aspectos y [19] 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.

Un enfoque basado en la implementación de interfaces de colaboración de aspectos (ACI) es CAESAR [13]. El propósito es desacoplar la implementación de aspectos y enlazar (binding) los aspectos, los cuales son definidos en módulos independientes, indirectamente conectados. La idea es que mientras son independientes unos de otros, estos módulos implementan partes comunes de la ACI, la cual indirectamente los relaciona como partes de un todo.

5 CONCLUSIONES

La propuesta se centra en la posibilidad de reutilizar el código de los aspectos en AspectJ como alternativa al mecanismo basado en la extensión de aspectos abstractos. Se plantea que la lógica o comportamiento correspondiente a requerimientos no funcionales se organice en un repositorio en entidades denominadas aspectos genéricos. Mediante el establecimiento por separado de asociaciones entre los aspectos genéricos y las unidades funcionales de un determinado proyecto, en forma automática se generan 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.

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 diseño y arquitectura original de la herramienta ASTOR son lo suficientemente simples y adaptables para aceptar fácilmente extensiones que permiten ensayar y experimentar estos conceptos y propiedades.

AspectJ es el LOA más popular y difundido, no sólo porque es el más utilizado sino además porque una gran cantidad de lenguajes siguen su modelo de estructura, y sólo se diferencian en el lenguaje base que extienden, como ser AspectC++. Esta propuesta es válida y aplicable también en estos casos, ya que requerirá modificar principalmente el componente Aspect Generator y la estructura de las asociaciones para que genere los aspectos correspondientes a cada LOA según corresponda.

El trabajo actual y futuro esta referido a la implementación de la estrategia planteada en la herramienta ASTOR y codificación de una API de aspectos genéricos iniciales que represente requerimientos funcionales típicos tales como: Autenticación, Conexión, Tratamientos de errores, Transacciones de integridad, Logging, Persistencia, Seguridad, Sincronización, entre otros.

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

REFERENCIAS

[1] Beugnard A. “How to make aspects re-usable, a proposition”. Position paper at the ECOOP. Workshop on Aspect-Oriented Programming. Lisboa, Portugal. Junio 1999.

[2] 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.

[3] 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.

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

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

[6] Hanenberg S, Unland R. “Using and Reusing Aspects in AspectJ”. Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA. Tampa. October 2001.

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

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

[9] 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.

[10] Gregor Kiczales. AOSD 2002. 1st. International Conference on Aspect-Oriented Software Development. (Ed.). ACM Press. The Netherlands. 2002.

[11] 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.

[12] Laddad R. “AspectJ in Action: Practical Aspect-Oriented Programming”. ISBN: 1930110936. Manning Publications Co. 2003

[13] Mezini M., Ostermann K.: “Conquering Aspects with Caesar”. AOSD’03. USA. Pp.90-99. 2003

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

[15] 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.

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

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

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

[19] 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.

[20] Homepage de AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html

[21] Homepage de AspectC++: http://www.aspectc.org/

[22] Homepage de AspectR: http://aspectr.sourceforge.net/

[23] Homepage de Pythius: http://sourceforge.net/projects/pythius/