Desarrollo de Software Orientado a Aspectos, DSOA 2006
Asociado a XV Jornadas de Ingeniería del Software y Bases de Datos
J. Araújo, J. Hernández, E. Navarro y M. Pinto (Eds)
Sitges (Barcelona), Octubre - 2006

# UN ENFOQUE PRÁCTICO PARA REUSAR ASPECTOS EN ASPECTJ

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

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

**Palabras clave:** Programación Orientadas a Aspectos, Reuso de Aspectos, AspectJ

**Resumen**. *En el Desarrollo de Software Orientado a Aspectos es frecuente encontrar situaciones en las que un mismo aspecto se repite y se utiliza en diferentes aplicaciones de software. En AspectJ el reuso de los aspectos es limitado. En este trabajo se propone un enfoque que radica en establecer un mecanismo que permita definir separadamente aquellos elementos de AspectJ que obstaculizan su reuso. Se plantea la extensión de la herramienta ASTOR para incluir un repositorio de aspectos genéricos y manejo de asociaciones, factores en los cuales se basa la estrategia.*

## 1. INTRODUCCIÓN

La Programación Orientada a Aspectos (POA) [1] es una técnica de programación que propone la implementación de los requerimientos no funcionales en unidades separadas e individuales a los requerimientos funcionales centrales.

En el Desarrollo de Software Orientado a Aspectos (DSOA) [2] es frecuente encontrar situaciones en las que un mismo requerimiento no funcional se repite y utiliza en diferentes contextos software. El reuso de estas unidades o módulos se presenta como una necesidad para el programador. En el lenguaje de aspectos AspectJ [3] el reuso de los aspectos es limitado, ya que estos quedan explícitamente ligados a determinados elementos que obstaculizan su reuso. El establecimiento expreso y no ambiguo en tiempo de compilación de los puntos de unión, designadores de cortes y tipos de avisos, imposibilitan el reuso de la implementación del requerimiento no funcional. En este trabajo se presenta un enfoque

alternativo para el reuso de aspectos codificados en AspectJ. El diseño e implementación de la propuesta se plantea como una extensión de la herramienta ASTOR [4] para incluir un repositorio de aspectos genéricos y manejo de asociaciones, factores en los cuales se basa la estrategia.

## 2. EL MODELO DE ASPECTJ

AspectJ [3] es el lenguaje de aspectos más popular y difundido. En AspectJ un requerimiento no funcional se modulariza en una unidad denominada aspecto. Los componentes de un aspecto son: miembros, introducciones y avisos [5]. 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 reuso. La inclusión de estos elementos de manera estática hace que la implementación dependa del requerimiento no funcional 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.

```
aspect PointObserving
   {
(1) private Vector Point.observers=new Vector();
(2) pointcut changes(Point p): target(p) && call(void Point.set*(int));
(3) after(Point p): changes(p)
      {
      ...
      }
   }
```

Figura 1. Ejemplo del aspecto *PointObserving* en AspectJ

En la Figura 1 se representa el aspecto *PointObserving*, cuyo objetivo es monitoriear las modificaciones de los objetos de la clase *Point*. Analizando este ejemplo desde el punto de vista de reuso se plantea 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 *observers* se ha restringido a la clase *Point*.
- Designadores de corte: El punto de corte *changes* indica que es activado únicamente por 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 el aviso *before* si se requiere antes.

Si se necesita un aspecto observador para otro tipo de objeto en la misma u otra aplicación, a pesar de 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 aspecto no puede ser utilizado en relación a otros componentes funcionales y/o de otras formas.
Los aspectos abstractos son un mecanismo para reusar la implementación 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. En la Figura 2, por ejemplo, algo usual es implementar los sistemas de log como aspectos. Así si se quiere mantener un historial de ciertas acciones, se puede definir un aspecto abstracto con un punto de corte abstracto, donde se indique cómo se quiere escribir la información, pero no cuáles van a ser los puntos de unión. Los aspectos que extiendan dicho aspecto, son los que concretan los puntos de corte, dando la posibilidad de definir clara y separadamente los elementos que se quieren controlar.

```
abstract aspect Logger
   {
   abstract pointcut logPoint();
   before(): logPoint() && !within(Logger+)
      {
      System.log(thisJoinPoint.getTarget()+
                 thisJoinPoint.getThis());
      }
   }
```

```
aspect LogDataBaseExec extends Logger
   {
    pointcut logPoint():
                call(* DataBase.*(..));
   }
```

Figura 2. Aspecto abstracto Logger y aspecto concreto LogDataBaseExec en AspectJ

Del punto de corte se excluyen el propio aspecto y sus subclases para evitar la recursividad. El aviso recoge el objeto sobre el que se llaman las funciones y el objeto actual y los escribe en el log del sistema. Este aspecto proporciona una definición del punto de corte *logPoint*, que en este caso son todas las llamadas a funciones de un paquete de base de datos *DataBase*.
El mecanismo descrito es una forma de reuso, 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. 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 [6].

## 3. REUSO DE ASPECTOS EN LA HERRAMIENTA ASTOR

ASTOR es una herramienta que soporta una serie de mecanismos y estrategias [7] para mejorar el tratamiento de conflictos entre aspectos en AspectJ [3]. 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 [4] y la resolución se efectúa siguiendo las directrices de una taxonomía que proporciona seis categorías de resolución [8]. La implementación de la herramienta esta basada en el pre-procesamiento de código AspectJ, siendo además éste el único requisito para su uso.
El diseño y arquitectura original de la herramienta ASTOR son lo suficientemente flexibles y adaptables para aceptar fácilmente extensiones que permiten ensayar y experimentar nuevos conceptos y propiedades. A continuación se describen los componentes, que constituyen el eje principal de la estrategia, incorporados a la herramienta ASTOR y un ejemplo.

Administrador de Repositorio de Aspectos Genéricos: El Administrador del Repositorio de Aspectos Genéricos (*AGenericAspectManager*) 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. Un aspecto genérico es una entidad codificada en lenguaje Java, compuesta de un conjunto de métodos y atributos independientes, es decir, no están vinculados a ningún componente funcional, ni proyecto en particular. La definición y declaración de los mismos sigue el patrón de los métodos y atributos de una clase. Los métodos y atributos de los aspectos genéricos corresponderían al código que se incluye en los avisos o las introducciones en AspectJ. Así, por ejemplo, un aspecto genérico denominado *Logging* contendría todos los posibles métodos y atributos para acceder a un sistema. El uso y utilidad del repositorio de aspectos genéricos es similar al de una API o librería de programas, pero a diferencia de éstas el repositorio es dinámico.

Administrador de Proyectos: El Administrador de Proyectos (*AProjectManager*) gestiona los Proyectos (*AProject*) de software. Los proyectos se forman de componentes (*AComponent*). Un componente puede ser una clase (*AClass*), una interface (*AInterface*) o una asociación de aspecto (*AAspectAssociation*).
Las asociaciones son las unidades que permitirán vincular un elemento de un aspecto genérico (método) a un componente funcional determinado (punto de unión). Las asociaciones son particulares a cada proyecto y existen dos tipos de asociaciones: las asociaciones de corte y las asociaciones de introducción. Las asociaciones tienen la siguiente estructura:

*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)

<u>Generador de Aspectos</u>: El generador de aspectos (*AAspectGenerator*) es el componente de software responsable de crear en forma automática los aspectos en AspectJ, según lo que las asociaciones indican. Este componente obtiene información de las asociaciones y el repositorio de aspectos genéricos para crear los aspectos. Este proceso se puede ver con más detalle en el diagrama de secuencia mostrado en la Figura 3, que describe el flujo de control que se genera una vez ejecutado el generador de aspectos.

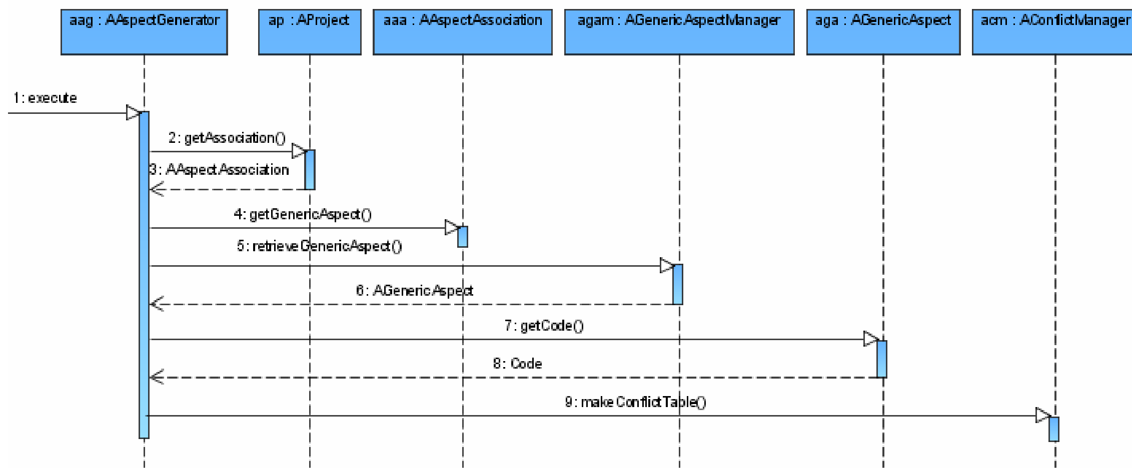


Figura 3. Flujo de Control de Generador de Aspectos

Cuando el método *execute* recibe un mensaje, solicita al proyecto actual (*AProject*) la colección de asociaciones (*AAspectAssociation*). De aquí en adelante comienza un ciclo que finaliza cuando se hayan creado todos los aspectos. Cada asociación dispone de toda la información necesaria que se requiere para crear un aspecto en AspectJ. Se obtiene el identificador del aspecto genérico (método *getGenericAspect()*) para recuperar al mismo del repositorio, operación que facilita el *AGenericAspectManager* (método *retrieveGenericAspect()*). El aspecto genérico (*AGenericAspect*) proporciona el código que representa la implementación del requerimiento no funcional (método *getCode()*). Finalizada esta operación el generador de aspectos dispone de todos los elementos para definir el punto de corte y el aviso, si se trata de una asociación de corte; o simplemente del método o atributo si se trata de una asociación de introducción.

A medida que el generador de aspectos crea cada aspecto, se actualiza la tabla dinámica de puntos de corte que utiliza el Administrador de Conflictos (*AConflictManager*). Esta última operación permitirá detectar y resolver posteriormente conflictos entre aspectos, de acuerdo a las estrategias previamente establecidas [4]. Luego que todos los aspectos se han generado, se procede con la detección e identificación de conflictos, de la manera original en la herramienta ASTOR.

En la Figura 4 se ilustra mediante un ejemplo sencillo un proyecto en desarrollo utilizando la herramienta ASTOR. El proyecto se ha denominado *Library Application* y se refiere a la administración de préstamo de libros a los socios de una biblioteca. Está

compuesto de las clases *Book*, *Borrow*, *BorrowManager* y *Member*, en estas unidades se ha codificado la funcionalidad central de la aplicación, como se pueden distinguir jerárquicamente en el panel superior izquierdo. En el panel inferior izquierdo, se visualizan los aspectos genéricos disponibles en el repositorio, como ser *ActivityRecord*, *Authentication*, *Connection*, etc. Algunos de estos aspectos genéricos ya han sido vinculados al proyecto mediante las asociaciones *BookPersistence*, *BorrowManagerActivityRecord*, *Borrow-ManagerPersistence*, *MemberPersistence*, etc. (adicionadas a la jerarquía en el panel superior izquierdo). El formulario de edición activo, indica que se esta creando una asociación de corte (denominada *BorrowManagerLogging*) entre el aspecto genérico *Logging* (con el método *checkIn()*) y el método *addBorrow(Borrow)* de la clase *BorrowManager*, para lo cual se han especificado todos los elementos necesarios, tanto para el punto de unión como para el tipo de aviso. Esta forma de establecer asociaciones garantiza que la definición será correcta y válida, ya que asegura: (a) la existencia del aspecto genérico y el método correspondiente; (b) la existencia del punto de unión completo (clase, método/atributo); (c) coherencia de puntos de corte; (d) campos vacíos y necesarios; (e) parametrización.

En el panel derecho se visualiza la hoja de edición de unidades, que permite la codificación de clases, interfaces y aspectos genéricos. En la vista presentada se halla editado el aspecto genérico *Logging*, ya incorporado a la lista de aspectos genéricos, como puede apreciarse.

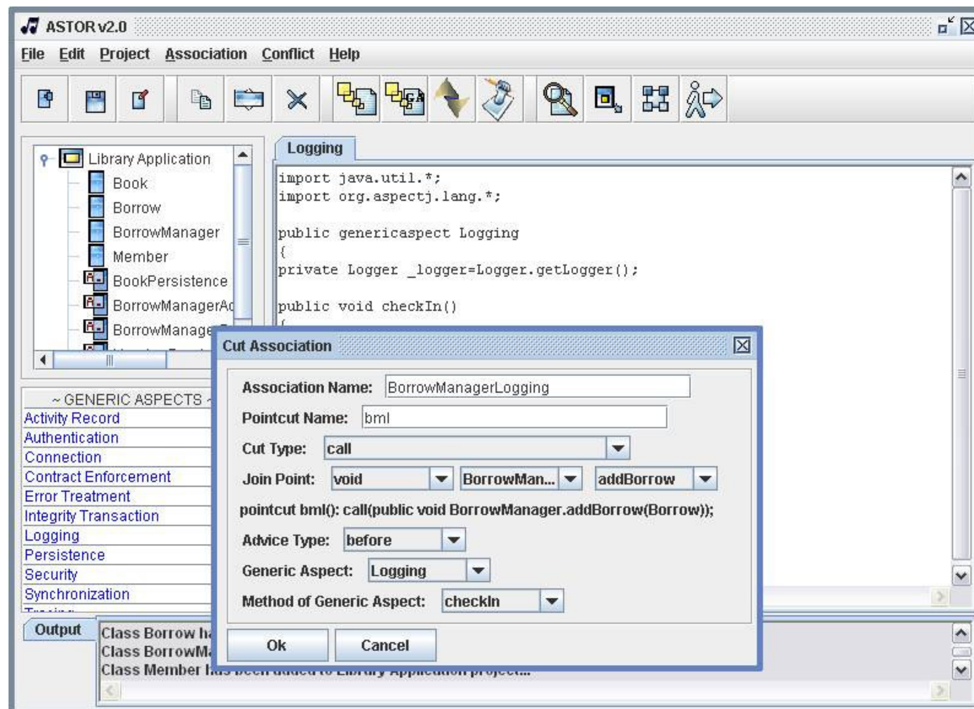


Figura 4. Vista del proyecto Library Application en desarrollo

En la Figura 5 se visualiza la ejecución del Generador de Aspectos. La ventana despliega los mensajes que indican el inicio y finalización del proceso y los resultados obtenidos. En este caso, se han generado cinco aspectos: *BookPersistence*, *BorrowManagerActivityRecord*, *BorrowManagerLogging*, *BorrowManagerPersistence* y *MemberPersistence*. Asimismo se indica (entre paréntesis) los elementos de la asociación utilizados, por ejemplo el aspecto *BookPersistence*, se ha generado a partir de una asociación de corte entre el método *save()* del aspecto genérico *Persistence*, y el punto de unión *Boook.new()*, al cual se aplica un designador de corte *execution* y un aviso *after*.

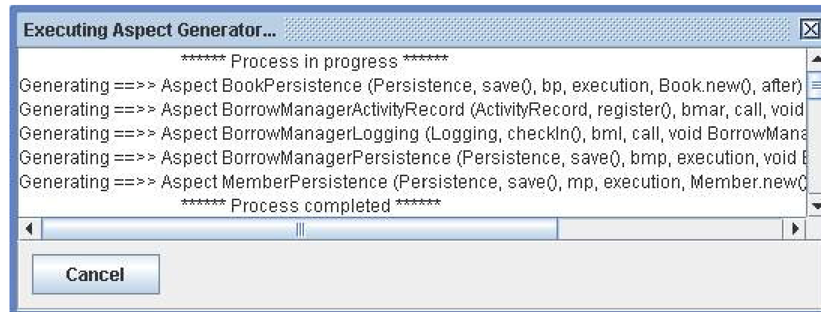


Figura 5. Ejecución del Generador de Aspectos

## 4. TRABAJOS RELACIONADOS

[9] propone una arquitectura general para construir un tejedor de aspectos reusables, separado 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, nuestra propuesta de trabajo se basa en un enfoque de pre-procesamiento de AspectJ, sin tomar intervención alguna en el tejedor (ajc).

Con respecto a la propuesta [10] existe una leve similitud en el hecho de establecer las declaraciones separadas de puntos de corte, tanto la declaración como la definición de los puntos de corte del resto de los aspectos. Pero difieren en: (a) no se debe usar un punto de corte para más de un aviso; (b) los aspectos concretos están siempre vacíos y (c) el reuso de aspectos se encuentra particularmente aplicado al enfoque de la herencia de aspectos. Este trabajo presenta una forma de mejorar el reuso de los aspectos en AspectJ definiendo ciertas reglas sintácticas en AspectJ a tener en cuenta para desacoplar los aspectos de los componentes en los que serán aplicados.

Un enfoque basado en la implementación de interfaces de colaboración de aspectos (ACI) es CAESAR [6]. 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. Una diferencia importante con nuestro trabajo, es el modelo de soporte POA subyacente al que ambos enfoques se encuentran dirigidos.

ASTOR es una herramienta específica para AspectJ y podría ser adaptada a otros LOA que respondan al modelo de puntos de unión. CAESAR propone un modelo POA muy diferente, basado en otro tipo de mecanismos [11].
Otros trabajos que proponen ideas relacionadas como [12] cuyo enfoque apunta directamente a las relaciones de dependencia entre aspectos: ortogonales, unidireccionales y circulares; [13] se basa en la construcción de aspectos de aspectos y [14] 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

- Este trabajo se centra en la posibilidad de reusar el código de los aspectos en AspectJ. Para lo cual, este código se almacena en un repositorio en entidades denominadas aspectos genéricos. Mediante el establecimiento por separado de distintos tipos de asociaciones entre los aspectos genéricos y las unidades funcionales de un determinado proyecto, en forma automática se generan los aspectos en AspectJ. Así, el desarrollador de aplicaciones software escribe el código una sola vez y lo reusa tantas veces como sea necesario.
- La propuesta se proyecta como extensión de la herramienta ASTOR, para obtener mayores beneficios y flexibilidad en el DSOA. El diseño y arquitectura original de la herramienta ASTOR son lo suficientemente flexibles y adaptables para aceptar fácilmente extensiones que permiten ensayar y experimentar nuevos conceptos y propiedades.
- El presente trabajo fue parcialmente financiado por la UNPA, Santa Cruz, Argentina.

## REFERENCIAS

[1] Mens K., Lopes C., Tekinerdogan B., Kiczales G. “Aspect-Oriented Programming”. Workshop Report ECOOP. 11th. Finland. 1997.
[2] Kiczales G. AOSD 2002. 1st. International Conference on Aspect-Oriented Software Development. (Ed.). ACM Press. The Netherlands. 2002.
[3] 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. Hungary. 2001.
[4] 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 ECC. JCC–UACh. Chile. Noviembre 2005.
[5] Homepage de AspectJ™, Xerox Palo Alto Research Center (Xerox Parc), Palo Alto, California: http://aspectj.org
[6] Mezini M., Ostermann K. “Conquering Aspects with Caesar”. AOSD. USA. Pp.90-99. 2003.

[7] 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.
[8] Pryor J., Marcos C. “Solving Conflicts in Aspect-Oriented Applications”. Proceedings of the Fourth ASSE. 32 JAIIO. Argentina. 2003.
[9] Beugnard A. “How to make aspects re-usable, a proposition”. ECOOP. Workshop on Aspect-Oriented Programming. Portugal. 1999.
[10] Hanenberg S, Unland R. “Using and Reusing Aspects en AspectJ”. Workshop on ASCOOS at OOPSLA. USA. 2001.
[11] Mezini M., Ostermann K. “Integrating independent components with on-demand remodularization”. In Proceeding of OOSPLA 02, 2002.
[12] 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. 2003.
[13] 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. 2002.
[14] 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. 2003.