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

# HACIA LENGUAJES DE METAMODELADO ORIENTADOS A ASPECTOS

**A. M. Reina [1], J. Torres[1], M. Toro[1]**

1: Dpto. Lenguajes y Sistemas Informáticos
E.T.S. Ingeniería Informática
Universidad de Sevilla
Avda. Reina Mercedes, s/n. 41012 Sevilla
e-mail: reinaqu@lsi.us.es, web: http://www.lsi.us.es/~reinaqu

**Palabras clave:** metamodelado, desarrollo de software orientado a aspectos, desarrollo de software dirigido por modelos

**Resumen.** *En este artículo se propone la extensión de los lenguajes de metamodelado con constructores de la orientación a aspectos. Tras justificar brevemente el interés que en la actualidad está teniendo el metamodelado, y sentar las bases necesarias para entender la propuesta, se presenta, a través de ejemplos, la necesidad de extender un lenguaje de metamodelado como Kermeta con conceptos introducidos en el ámbito de la orientación a aspectos.*

## 1. INTRODUCCIÓN

Hoy en día hay un interés creciente por el Desarrollo de Software Dirigido por Modelos (DSDM), propiciado, sobre todo, por el hecho de que en los últimos años han mejorado mucho las herramientas para dar soporte a todo el proceso de desarrollo software. Así, destacan dos aproximaciones a este tipo de desarrollo: las Factorías Software (FS)[8], promovidas por Microsoft, y *Model Driven Architecture (MDA)* [10] promocionada por la Object Management Group (OMG).

En ambas propuestas los metamodelos juegan un papel fundamental. Mientras que en las factorías software los metamodelos entran en juego en la actividad de desarrollo de diferentes lenguajes de modelado y de las herramientas específicas para el dominio de los mismos, en MDA los metamodelos son el apoyo de los diferentes niveles de modelado (Modelos Independientes de Computación, Modelos Independientes de Plataforma y Modelos Específicos de Plataforma). Además, aunque MDA impulsa el uso de UML, también proporciona MOF [11] como estándar para describir nuevos lenguajes de modelado.

En este artículo se propone la incorporación de algunos elementos definidos en los lenguajes de orientación a aspectos a los lenguajes de metamodelado, con objeto de separar competencias y mejorar la reutilización de los mismos.

Cuando se definen metamodelos, se puede trabajar en diferentes espacios tecnológicos: con lenguajes gráficos; con XML mediante el estándar XMI [12]; con una implementación en Java; o con un lenguaje diseñado específicamente para definir metamodelos como Kermeta [14]. En los dos primeros casos, no hay muchos problemas a la hora de reutilizar metamodelos. Sin embargo, en los dos últimos se puede perder la propiedad conocida en el ámbito de la orientación a aspectos como *obliviousness*. Es decir, no se puede conseguir mantener al metamodelo original inconsciente de ser reutilizado por otro metamodelo.

Por otra parte, si estos lenguajes se extienden para definir características de comportamiento, tal y como ocurre en Kermeta, se tiene que no proporcionan una buena separación de conceptos, ya que se basan en los constructores definidos en la orientación a objetos.

El artículo se estructura como sigue: en la sección 2 se explican algunos conceptos necesarios para entender el resto del artículo, mientras que en las secciones 3 y 4 se muestra, mediante un ejemplo práctico cómo se podrían beneficiar los lenguajes de metamodelado, en este caso Kermeta, de la filosofía de la orientación a aspectos. Finalmente, el artículo se concluye.

## 2. METAMODELADO Y LENGUAJES DE METAMODELADO

Se puede considerar que un metamodelo es una definición precisa de los constructores y reglas necesarios para definir la semántica de los modelos. Además, el metamodelado puede verse como una actividad que está tomando auge en los últimos años y que sirve para organizar los modelos en diferentes niveles, de tal modo que un modelo se describe por otro modelo que está situado en un nivel superior (su metamodelo).

A la hora de definir metamodelos han surgido diferentes propuestas, así MOF es un lenguaje gráfico, para el cual la OMG ha estandarizado una serie de correspondencias que especifican cómo se gestionan los metadatos en una tecnología determinada: XMI [12] es el estándar para definir metamodelos MOF en XML, mientras que JMI [15] define la sintaxis abstracta para trabajar con metadatos en Java.

Al mismo nivel que MOF, pero como una propuesta más ligera, está Ecore, el lenguaje de metamodelado asociado al Eclipse Modelling Framework (EMF) [3]. EMF también puede trabajar a partir de modelos definidos en UML, XML o interfaces Java.

Además, han surgido lenguajes de metamodelado diseñados específicamente para definir metamodelos y que normalmente están asociados a herramientas, tales como KM3 [2] que está asociado a la plataforma AMMA, o Kermeta [14].

Los ejemplos introducidos en este artículo, se basan en Kermeta. Kermeta es un lenguaje definido con el propósito de servir a todas las posibles manipulaciones de modelos, es decir, ha sido pensado para que sirva tanto para la definición de metamodelos y modelos como para la definición de acciones, consultas, vistas y transformaciones. En la figura 1 se muestra la situación del lenguaje Kermeta como el conjunto de constructores comunes a los lenguajes de metamodelado, de acciones, de transformaciones y de restricciones.

Así, Kermeta es una extensión de EMOF (Essential Meta-Object Facilities) 2.0 para

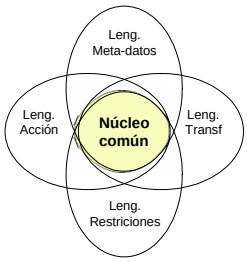



Figura 1: Situación de Kermeta

dar soporte a la definición del comportamiento. Proporciona un lenguaje de acción para especificar el cuerpo de las operaciones en los metamodelos.

## 3. APLICANDO ORIENTACIÓN A ASPECTOS EN LA ESTRUCTURA ESTÁTICA

Este apartado muestra cómo se puede utilizar la orientación a aspectos para mejorar la reutilización de metamodelos. En este caso, lo que se desea modificar es la estructura estática del metamodelo.

Para introducir la problemática se va a utilizar un ejemplo, que puede ser consultado con más detalle en [13]. Supongamos que necesitamos definir un metamodelo para Java2, y que vamos a definir ese metamodelo con Kermeta. El metamodelo de Java2 se ha obtenido de [4], y su implementación se recogerá en un paquete, denominado `<<Java2>>`.

En la figura 2 se muestra un extracto de este metamodelo definido en Kermeta. En concreto, la definición Kermeta de la metaclase `Class` representada en la parte central de la figura. En el código se puede ver que `Class` hereda de `Classifier` (línea 01), y que tiene definido dos atributos (`staticInit` y `instanceInit`, en las líneas 02 y 03, respectivamente). En esta fase, las líneas 05 y 06 no son necesarias, ya que como veremos un poco más adelante, se deberán introducir al extender este metamodelo.

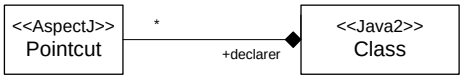
Supongamos que una vez que tenemos definido nuestro metamodelo para Java2, surge la necesidad de definir un metamodelo para AspectJ [1]. Como ya tenemos nuestro metamodelo Java2, queremos aprovechar el trabajo realizado, y definir el metamodelo de AspectJ en base al de Java2. En este caso, el metamodelo de AspectJ escogido es el que se define en [9].

Eso implica que tendremos un paquete `Java2` y otro paquete `AspectJ`, y que habrá metaclases del paquete `AspectJ` relacionadas con metaclases del paquete `Java2`. En la figura 2, se muestra un ejemplo de estas relaciones. En el gráfico, aparecen dos metaclases estereotipadas: `Class` y `Pointcut`. El estereotipo indica el paquete al que pertenecen. En este caso, la relación que existe entre ellas es una relación de composición.

Aunque no aparece en el gráfico, si nos fijamos en el trozo de código Kermeta correspondiente a `Pointcut` (línea 01), se puede comprobar que también mantiene una relación

con `Feature` (que pertenece al paquete `Java2`). En este caso, es una relación de herencia.

A la hora de extender el paquete `Java2` mediante el diagrama MOF, no encontraríamos ninguna dificultad, ya que lo único que habría que hacer sería crear un paquete nuevo llamado `AspectJ` e incluir las metaclases de `Java2` estereotipadas, para indicar que son clases de otro paquete.




```
01 class Class inherits Classifier{
02     attribute staticInit:Block [0..1]
03     attribute instanceInit:Block[0..*]
04     reference thrownBy:BehavioralFeature[0..*]#thrownExceptions
05     //Añadido al extender con el paquete de AspectJ
06     attribute pointcut:Pointcut[0..*]#declarer
07 }
```

```
01 class Pointcut inherits Feature{
02    attribute paramPC:Parameter[0..*]#pointcutPar
03    reference declarer:Class#pointcut
04    attribute pce: PointcutExpression[0..1]#pc
05    reference expression: PointcutExpression[0..*]#pcOperand
06 }
```

Figura 2: Extendiendo metamodelos en Kermeta

Cuando vamos a definir el metamodelo de `AspectJ` en Kermeta, nos encontramos con alguna dificultad. En primer lugar, tendremos que crearnos un paquete `AspectJ` donde definamos las nuevas metaclases. Si la relación de estas metaclases con las metaclases del paquete `Java2` es una relación de herencia (como es el caso de `Pointcut-Feature` en la figura 2), no hay ningún problema, se incluye el paquete y la claúsula `inherits` (línea 01 en la definición de `Pointcut`).

En cambio, si la relación es una asociación o una composición (como en el caso de `Pointcut-Class`) hay que modificar la definición del metamodelo original, con lo cual deja de ser reutilizable.

Para implementar una asociación en Kermeta se coloca una cláusula `reference` en cada una de las clases involucradas en la asociación. Si la asociación tiene semántica de composición (tal como ocurre en el caso `Pointcut-Class`), en lugar de dos cláusulas `reference` hay que escribir una cláusula `reference` y otra `attribute`. En el caso de la figura 2, en la clase `Pointcut` se puede observar la cláusula `reference` que define un extremo de la composición (línea 03, resaltada en negrita). Pero además, para definir el otro extremo de la composición (el que está marcado con el rombo), hay que escribir una cláusula `attribute` en la metaclase `Class` (línea 06, en negrita).

Como se puede observar, se ha tenido que modificar el metamodelo original de `Java2`, para poder incluir la asociación con el elemento `Pointcut` del paquete `AspectJ`.

Para atacar este problema, proponemos una ampliación del lenguaje inspirada en AspectJ de forma que se defina un nuevo constructor `aspect` para introducir declaraciones

inter-tipo.

## 4. APLICANDO ORIENTACIÓN A ASPECTOS EN LA DEFINICIÓN DEL COMPORTAMIENTO

Además de la definición de metamodelos, Kermeta permite definir comportamiento. En este apartado, por tanto, se mostrará a través de un ejemplo, como se puede mejorar la separación de conceptos en la parte de comportamiento.

El ejemplo escogido está relacionado con un problema que se está abordando tanto desde la comunidad de la orientación a aspectos como desde la comunidad del desarrollo de software dirigido por modelos: la trazabilidad. Una de las maneras de abordarlo es definir un metamodelo de la traza, independiente de las transformaciones y del metamodelo utilizado. Un ejemplo de este enfoque es la propuesta presentada en [6], en la que se define un metamodelo para la trazabilidad y un *framework* para la misma.

```
01 operation transform (source: ClassHierarchy ) : DataBase is do

02     result:= DataBase.new //Inicializar el modelo destino

03     trace.initStep ( "minuml2mindb ") // Trazar el código generado

04     source.hierarchy.each {cls | //Iterar por las clases del modelo origen
05        var table: Table init Table.new // Crear una tabla
06        table.name := String.clone (cls.name) //Copiar el nombre de la clase a la tabla

07        result.table.add (table) // Añadir la tabla al modelo destino

08        // Trazar el código generado
09        trace.add link ("minuml2mindb" , "class2table" , cls , table )

10        cls.ownedAttribute.each { prop | // Iterar por los atributos de la clase
11            var col : Column init Column.new // Crear una columna nueva

12            col.name:= String.clone ( prop.name )
13            table.column.add ( col ) // Añadir la columna a la correspondiente tabla

14            // Trazar el código generado
15            trace.addlink ( "minuml2mindb", "property2column" , prop, col)
16        } // Fin de la iteración por los atributos
17     } //Fin de la iteración por las clases
18 end
```

Figura 3: Trazando una transformación en Kermeta

En la figura 3 se puede ver el código Kermeta que define una operación `transform` donde se definen las transformaciones necesarias para generar un esquema de base de datos reducido a partir de un diagrama reducido de clases UML. Como se puede observar, para trazar las transformaciones, se han de insertar trozos de código para llamar a operaciones definidas en el *framework* de trazabilidad (líneas 03, 09 y 15, resaltadas en gris más claro). Por lo tanto, el código para invocar al *framework* de trazabilidad se mezcla con el código necesario para realizar la transformación.

Si volvemos a fijarnos en AspectJ, en este caso proponemos la amplicación del lenguaje

con constructores para especificar los puntos en los que se va a inyectar el código necesario para trazar las transformaciones, así como para recoger este código que se inyecta.

## 5. CONCLUSIONES

Las ideas propuestas en este artículo aún se encuentran en una fase preliminar, pero ahora que se está extendiendo con fuerza el desarrollo de software dirigido por modelos, y que el metamodelado se ha convertido en parte central del mismo, es necesario encontrar mecanismos para poder reutilizar la implementación de estos metamodelos. Aunque a nivel gráfico y trabajando con XMI no parezca necesario, al final, las herramientas acaban o bien haciendo una implementación en algún lenguaje de propósito general, o bien definiendo su propio lenguaje para describir los metamodelos.

En este artículo, se aboga por la aplicación de los conceptos desarrollados en la orientación a aspectos para poder mejorar la reutilización de los metamodelos.

## REFERENCIAS

[1] The AspectJ Team. *AspectJ Programming Guide (v.1.2)*. Avalaible at: `http://www.eclipse.org/aspectj`. 2003.

[2] Atlas Group. *KM3: Kernel MetaMetaModel Manual*

[3] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. J. Grose. *Eclipse Modelling Framework: A Developer's Guide*. Addison-Wesley, 2003.

[4] S. Dedic and M. Matula *Metamodel for the Java Language* Avalaible at: `http://java.netbeans.org/models/java/java-model.html`.

[5] T. Elrad, R. E. Filman, A. Bader. *Aspect Oriented Programming*. Communications of the ACM. Vol. 44, n. 10., Oct. 2001.

[6] J.R. Fallery. M. Huchard, C. Nebut. *Towards a traceability framework for model transformations in Kermeta*. Proceedings of the 2nd ECMDA Traceability Workshop. Held with the ECMDA Conference. July, 2006. Bilbao, Spain.

[7] R. E.Filman, D. P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000. Oct, 2000.

[8] J. Greenfield, K. Short, S. Cook, S. Kent. *Software Factories. Assembling Applications with Patterns, Models, Frameworks and Tools.*Wiley Publishing, Inc., 2004.

[9] Y. Han, G. Kniesel, A. Cremers. *Towards Visual AspectJ by a MetaModel and Modeling Notation*. Proceedings of the 6th International Workshop on Aspect-Oriented Modeling held in conjunction with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05). Chicago, Illinois, USA. Mar, 2005.

[10] OMG, *MDA Guide Version 1.0*, Eds. J. Miller and J. Mukerji. May, 2003.

[11] OMG, *Meta Object Facility Specification Version 2.0*,January, 2006.

[12] OMG. *MOF 2.0 / XMI Mapping Specification, v2.1*. Jan, 2006. Avalaible at: `http://www.omg.org/technology/documents/formal/xmi.htm`.

[13] A. M. Reina, J. Torres. *Using aspect-orientation techniques to improve the reuse of metamodels*. Proceedings of the Second Workshop on Aspect-Based and Model-Based Separation of Concerns in Software Systems (ABMB 2006). Held with the ECMDA Conference. July, 2006. Bilbao, Spain.

[14] Triskel Team. Kermeta web site: `http://www.kermeta.org`.

[15] Sun Corporation: *The JavaTM Metadata Interface (JMI) Specification*. Jun, 2002. Avalaible at: `http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html`.