Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 163 (2007) 29–43
www.elsevier.com/locate/entcs

Using Aspect-orientation Techniques to Improve Reuse of Metamodels ⋆

A. M. Reina Quintero[1] J. Torres Valderrama[2]

Department of Languages and Computer Systems
University of Seville
Seville, Spain

Abstract

Metamodelling is an activity that attracts attention of the research community dealing with the Model-Driven Development (MDD). To be reusable in different MDD approaches a metamodel should be unaware of being extended by another metamodel. This property of metamodel is called obliviousness. This paper shows that current techniques implementing metamodels do not maintain obliviousness when some elements of the extended metamodel and the elements of the original model have association relations. Three different approaches to reuse of metamodels are analyzed. One of the approaches uses traditional object-oriented techniques. Two other approaches use aspect-oriented techniques. The paper shows that the third approach, which considers relationships as first-class citizens at the implementation level by using relationship aspects, guarantees obliviousness of metamodels.

Keywords: Metamodelling, Aspect-Oriented Programming, Model-Driven Architecture.

1 Introduction

Model Driven Development (MDD) is supported by two main approaches: Software Factories (SF) [11] promoted by Microsoft and Model Driven Architecture (MDA) [19] promoted by the Object Management Group (OMG) [16,5].

The Software Factories approach proposes the use of extensible and configurable tools to automate the development and maintenance of different software product families. The automation is obtained by means of the composition and configuration of different components. Thus, the Software Factories approach integrates multiple activities and techniques. One of these activities is the development of various modelling languages and domain specific tools.

⋆ This work has been partially supported by the Spanish Ministery of Science and Technology and FEDER funds: TIC-2003-369.

[1] Email:mailto:reinaqu@lsi.us.es

[2] Email:mailto:jtorres@lsi.us.es

doi:10.1016/j.entcs.2006.10.014

The Model Driven Architecture approach is based on the modelling standards proposed by the OMG: the Unified Modelling Language (UML) [22] and the Meta Object Facility (MOF) [20]. MDA proposes a framework composed of different levels of modelling: Computation Independent Model (CIM), Platform Independent Model (PIM), and Platform Specific Model (PSM) and model transformations. Thus, the models and transformations have become the first-class citizens. Figure 1 shows a MOF diagram of the main elements of the MDA approach.

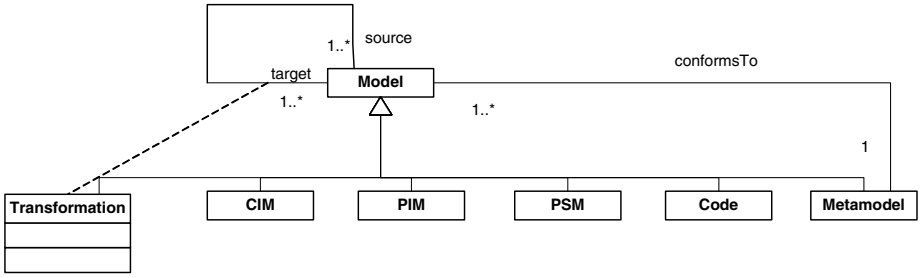

Fig. 1. Metamodel of the MDA approach

In addition to the CIM, PIM, PSM and transformation, Figure 1 shows another important element of MDA: a metamodel. A metamodel is a special kind of model which describes an abstract syntax of another model.

So, the metamodel have become the key elements for both MDD approaches. In this context it is very promising to define and implement metamodels that can be reused [2].

Catalysis method [7] introduces two mechanisms of metamodel reuse: the package extension and the package template mechanism [4]. In this paper we focus on the package extension mechanism. Metamodel `MMA` extends metamodel `MMB` if `MMA` specializes `MMB`. Thus, any of the elements of the `MMA` metamodel has a relationship with any of the elements of the `MMB` metamodel. This mechanism allows defining the `MMA` and `MMB` metamodels separately, and merging them together. In order to improve reuse of metamodels it is important to maintain the obliviousness, i.e. the unawareness of `MMB` about `MMA`.

We have analyzed three different approaches to implementation of metamodels in order to understand if they maintain the obliviousness of metamodels. The first approach uses inheritance to make the original metamodel oblivious so that the problem is solved with traditional object-oriented techniques. The second and third approaches are based on aspect-oriented techniques [9,10]. The second approach introduces inter-type declarations, while the third one treats the relationships in metamodels as aspects [23]. In this paper we show that the third approach guarantees obliviousness of metamodels which facilitates metamodel reuse.

This paper is structured as follows: section 2 introduces the main metamodelling concepts. In section 3 the Eclipse Modelling Framework for implementing models and metamodels is presented. Section 4 presents the problems that arise when metamodels are extended. Section 5 analyses three different approaches to extension of metamodels. Section 6 draws conclusions from the analysis.

2 Metamodelling

According to [15] a metamodel is a precise definition of the constructs and rules needed for creating semantic models. Metamodelling is a way to organize related models. The OMG defines four different levels of modelling [14]. Figure 2 shows a scheme of the relationships among the levels M0, M1, M2 and M3 defined by the OMG. The MOF is placed on the top of the hierarchy, and it is used to define itself; therefore, the level above MOF (M3) can be seen as the MOF itself. The UML is at level M2. The abstract syntax of the UML has been described using the MOF. An instance of the UML metamodel can be seen as a class diagram (level M1). An instantiation of a class diagram is an object diagram (Level M0).

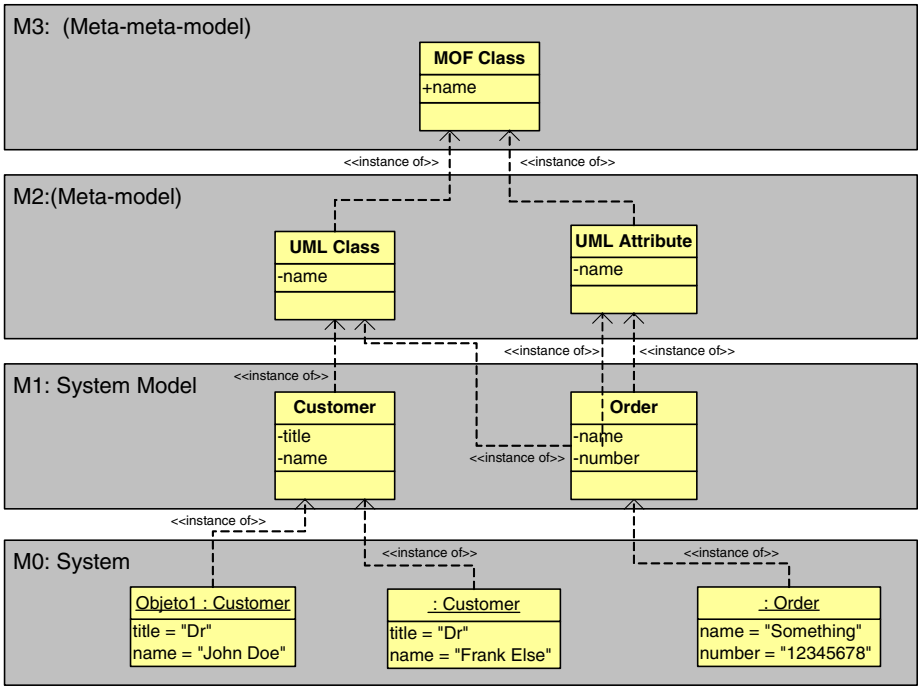

Fig. 2. The four layer metamodel architecture proposed by the OMG

The metamodel repositories can be classified in two main groups:

- *The MOF-based ones.* The MOF [20] is the metamodelling framework proposed by the Object Management Group (OMG) to define other modelling frameworks. The MOF specification is vendor and language independent. With the MOF specification, the OMG has standardized a set of mappings that specify how a specific technology represents and manages meta-data. For instance, XMI [21] is a XML representation for model interchange between tools, while JMI [25] is an abstract syntax definition for meta-data in Java applications. Some repositories that implement the JMI interface are: MDR from NetBeans [17] or NSMDF [18] from NovoSoft. The Coral Metamodelling Framework [24] has the hard coded MOF, although other different metamodels can be installed. Coral is not based on Java, but on Python.

- *The Ecore-based ones.* Ecore is the metamodel included in the Eclipse Modelling Framework (EMF) [3]. It is different from MOF. EMF is a low-cost tool to obtain the benefits of formal modelling and Java code generation and it is language-dependent. The functionality of EMF is similar to MDR.

3 Eclipse Modelling Framework (EMF)

The Eclipse Modelling Framework (EMF) [3] is a modelling framework for Eclipse. EMF is, on the one hand, a framework, and, on the other hand, a facility for defining a model in one of the following forms: Java interfaces, UML diagrams or XML Schemas. `Ecore` is the metamodel that uses EMF to represent models. Ecore is itself an Ecore model, and it is of the same OMG level (Figure 2) as MOF. Figure 3 depicts different sources of a core model.

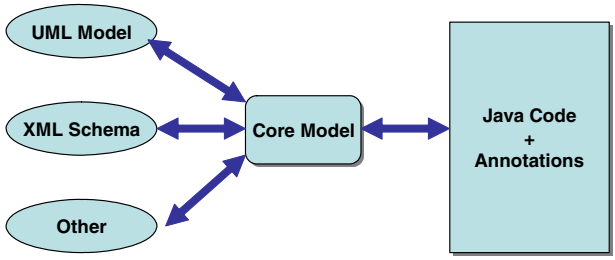

Fig. 3. Sources of a Core Model

There are four basic metaclasses to represent an Ecore model: `EClass`, `EAttribute`, `EReference` and `EDataType`.

- `EClass` models a class. It has an attribute called `name` to store the name of the modelled class. It has composition relationships with `EAttribute` and `EReference`. The cardinality of the composition means that an `EClass` can have zero or more attributes and zero or more references.
- `EAttribute` models an attribute. It has an attribute (`name`) and an association with `EDataType`. The association represents that an attribute must have a type.
- `EReference` models one of the two ends of an association between classes. It has two attributes: `name` and `containment`. The attribute `containment` is true if the association end represents a composition relation. Finally, `EReference` has an association with `EClass`. This relation models the target type, that is, the class which is at another end of the association.
- `EDataType` models the type of an attribute. It can be a primitive type (`int`, `float`, ...) or an object type.

We use Java interfaces to define the core model. For each class of the model an interface is defined. For each attribute and for each reference contained in the class, a standard `get()` method is declared in the interface. With this information the EMF generator will deduce the model attributes and references. The Java interfaces and the `get()` methods are annotated in order to help the EMF generator to deduce

the model properties.

With the interfaces and the annotations, EMF produces two files: a `.ecore` file and a `.genmodel`. The `.ecore` is an XML file that contains the core model. The `.genmodel` is a kind of wrapper of the core model with extra information. This information is needed for generating the implementation of the model.

Once the implementation is generated, each Ecore class (that is, each `EClass`) corresponds to two things in Java: an interface and its corresponding implementation class. For example, class `Book` in our Ecore model will be modelled as an `EClass` in EMF, and it will be mapped onto a Java interface (`public interface Book`) and an implementation class (`public class BookImpl ... implements Book`). Figure 4 depicts these relations.

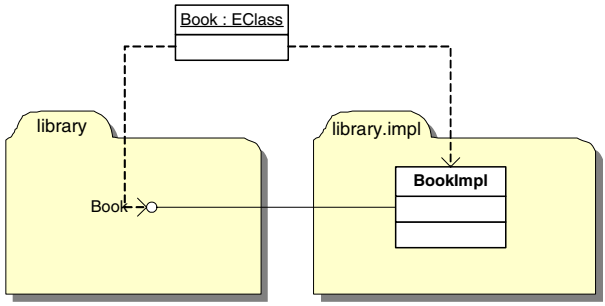

Fig. 4. Relationship between an Ecore class and the generated stuff

If we define the Ecore model using annotated Java, we will just be in charge of writing the interfaces and the `get()` methods (if needed). Afterwards, the EMF generator will complete these interfaces with more annotations and `set()` methods. Furthermore, it will generate the implementation classes and all the extra code needed.

4 Problem Statement: Metamodel Extension

If a metamodel is reused, it should be defined in such a way that it is completely unaware of being extended by another metamodel.

To introduce the problem, we have classified the possible relations between the elements of two models (the original and the extended one) into two main groups: inheritance relationships and the rest of relationships. The following subsections introduce two different examples to illustrate the impact of these two kinds of relations on the definition and implementation of metamodels.

4.1 Example 1: Extensions by means of inheritance relationships

This example has been obtained from [8]. It has been chosen because it is a very simple, introductory example. Figure 5 shows a package named `Library` which holds a MOF metamodel of a library. This package contains three metaclasses (`Library`, `Book` and `Writer`) and one enumeration (`BookCategory`). For writing

this metamodel with EMF, three interfaces (one for each metaclass) have to be defined. In each interface a number of `get()` methods should be written (one `get¬()` method for each attribute and another one for each reference). Furthermore, a new class should be defined for the enumeration.

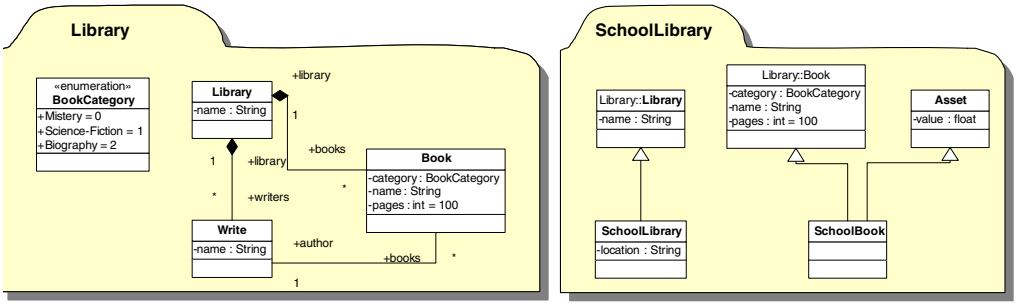

Fig. 5. Exteding a metamodel by means of inheritance relationship

Figure 6 shows a part of the implementation of the Library metamodel with EMF: the `Library` metaclass and the `BookCategory` enumeration.

```
package library;
import java.util.List;
/**
 * @model
 */
public interface Library
{
  /**
   * @model
   */
  String getName();

  /**
   * @model type="Writer" containment="true"
   */
  List getWriters();

  /**
   * @model type="Book" containment="true"
   */
  List getBooks();
}
```

```
package library;
/**
 * @model
 */
public class BookCategory
{
  /**
   * @model name="Mystery"
   */
  public static final int MYSTERY = 0;

  /**
   * @model name="ScienceFiction"
   */
  public static final int SCIENCE_FICTION = 1;

  /**
   * @model name="Biography"
   */
  public static final int BIOGRAPHY = 2;
}
```

Fig. 6. Part of the Library EMF metamodel with Java notation

Figure 5 shows the `SchoolLibrary` package which extends the `Library` package. From the MOF model point of view this extension implies that the `SchoolLibrary` package includes two classes (`Book` and `Library`) with the stereotype `<<from Library>>`. This stereotype means that those metaclasses belong to the `Library` package. In Figure 6, this is depicted with the name of the package placed just before the name of the class (`Library::Book`). Figure 7 shows the `extends` clause in the `SchoolLibrary` and `SchoolBook` interfaces. These sentences express the inheritance relationship between the pair of classes `SchoolLibrary-Library` and `SchoolBook-Book`. The classes `Book` and `Library` should not be modified. Thus, as

a conclusion, if we extend a metamodel by means of inheritance the obliviousness of the original metamodel is maintained.

```
package schoollibrary;
import library.Library;
/**
 * @model
 */
public interface SchoolLibrary extends Library
{
  /**
   * @model
   */
  String getLocation();
}
```

```
package schoollibrary;
import library.Book;

/**
 * @model
 */
public interface SchoolBook extends Book, Asset
{
}
```

Fig. 7. Part of the extended SchoolLibrary package

4.2 Example 2: Extensions by means of associations

Figure 8 shows a relation between the two packages. The original package is called `com.metamodels.java2`, it has been obtained from [6], and it is a version of the UML metamodel which tailors the UML metaclasses to the Java2 Specification. The `com.metamodels.aspectj` package contains an AspectJ metamodel which extends the Java metamodel according to the AspectJ specification [1]. AspectJ is an extension for Java to develop aspect-oriented applications. The AspectJ metamodel has been obtained from [12].

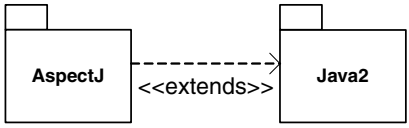

Fig. 8. Package relationship

Figure 9 shows the MOF Java 2 metamodel. Most of the metaclasses use the same name as the corresponding UML metaclasses. We focus on the metaclasses that are relevant to our example: `Element`, `Generalization`, `Feature` and `Parameter`.

The `Generalization` class models the `extend` and `implements` relationships between classifiers (classes and/or interfaces). An instance of this class represents an inheritance relationship between a subtype and a supertype. It can also represent the relationship between a class and an interface.

A `Feature` represents something that can be declared in a class or an interface (a field, a constructor or an ordinary method).

The `Parameter` class abstracts the parameters in a method or constructor. This relationship is represented by a composition relation between `Parameter` and `BehavioralFeature`. A parameter is also related to a `Type`.

That specification of this metamodel in annotated Java contains one interface for each metaclass and one final static class for each enumeration. Figure 10 shows

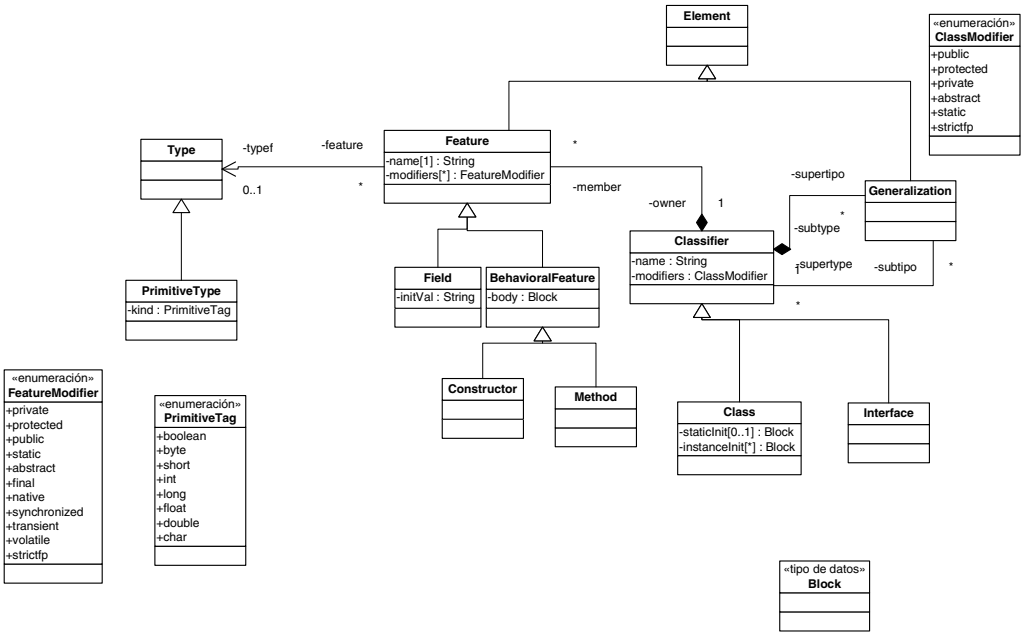

Fig. 9. AspectJ Metamodel

the code needed for generating the metaclasses `Feature` and `Class`. `Feature` has two attributes `name` and `modifiers` which are transformed in the implementation into two `get()` methods, `getName` and `getModifiers`.

`Feature` is also related to `Type` and `Classifier`. The first relation is an association while the second one is a composition relationship. From the point of view of `Feature` there is no difference between these two relationships: there are two `get()` methods (`getOwner` and `getTypefeature`) with an opposite annotation. This annotation refer to the name at the other end of the association. The difference between these two kinds of relationships is specified in the classes `Type` and `Classifier`, respectively. The `Type` interface will have a `get()` method named `getFeatures` while the `Classifier` interface will have a `get()` method (`getMember`) with an extra annotation (`containment=true`). This containment annotation has the composition semantics.

The Java2 metamodel is extended by the metamodel depicted in Figure 11. As well as in the example introduced in the previous section, at the modelling level, there is no problem to reuse some metaclasses of the `com.metamodel.java2` package. We only need to label the classes with the stereotype `<<Java2>>`. Figure 11 shows the stereotyped classes `Class`, `Generalization`, `Feature`, `Element` and `Parameter`.

The `Element` and `com.metamodel.aspectj` package have inheritance relationship. There are composition relationship between `Class` and `Pointcut`; between `Parameter` and `Pointcut`; and between `Parameter` and `Advice`. `Generalization` has association relationship with `Aspect` and `Feature` has an association with `Aspect`.

If we want to implement these relationships we need to declare `get()` methods

```
package com.metamodel.java2;

import org.eclipse.emf.common.util.EList;

/**
 * @model
 */
public interface Feature extends Element{
    /**
     * @model
     */
    String getName();

    /**
     * @model
     * type="com.metamodel.java2.FeatureModifier"
     */
    EList getModifiers();

    /**
     * @model opposite="member"
     */
    Classifier getOwner();

    /**
     * @model opposite="features"
     */
    Type getTypefeature();

}
```

```
package com.metamodel.java2;

import org.eclipse.emf.common.util.EList;

/**
 * @model
 */
public interface Class extends Classifier{
    /**
     * @model
     * dataType="com.metamodel.java2.Block"
     */
    Block getStaticInit();

    /**
     * @model
     * dataType="com.metamodel.java2.Block"
     */
    Block getInstanceInit();

    /**
     * @model
     * type="com.metamodel.java2.BehavioralFeature"
     * opposite="thrownExceptions"
     */
    EList getBehavioralFeatures();

}
```

Fig. 10. Java 2 Implementation

in the interfaces. Figure 12 shows two of these interfaces, `Pointcut` and `Aspect`. The `Pointcut` interface contains an annotated method called `getDeclarer` that models one edge of the relationship between `Pointcut` and `Class`. But in order to model the other end of the association, a new annotated `get()` method with an annotation `containment=true` should be included in the `Class` interface. Therefore, at this point we need to modify the `Class` interface and, as a consequence, the `com.metamodel.java2` package becomes aware of being extended with the `com.¬metamodel.aspectj` package.

The same problem arises when we try to implement the association between `Aspect` and `Feature`. The `getIntroducedFeatures` method implements one of the ends of the relationship, but in order to implement the other end we have to declare a new get method in the `Feature` interface.

So, if we extend a metamodel using relationships different from inheritance, the obliviousness of the original metamodel is lost.

5 Using aspect technology to improve reuse of metamodels

This section describes three different approaches for making the `com.metamodel¬.java2` unaware of being extended by another metamodel. The first one is based on the traditional object-oriented inheritance mechanism, while the second and the third ones are based on aspect-oriented solutions [13].

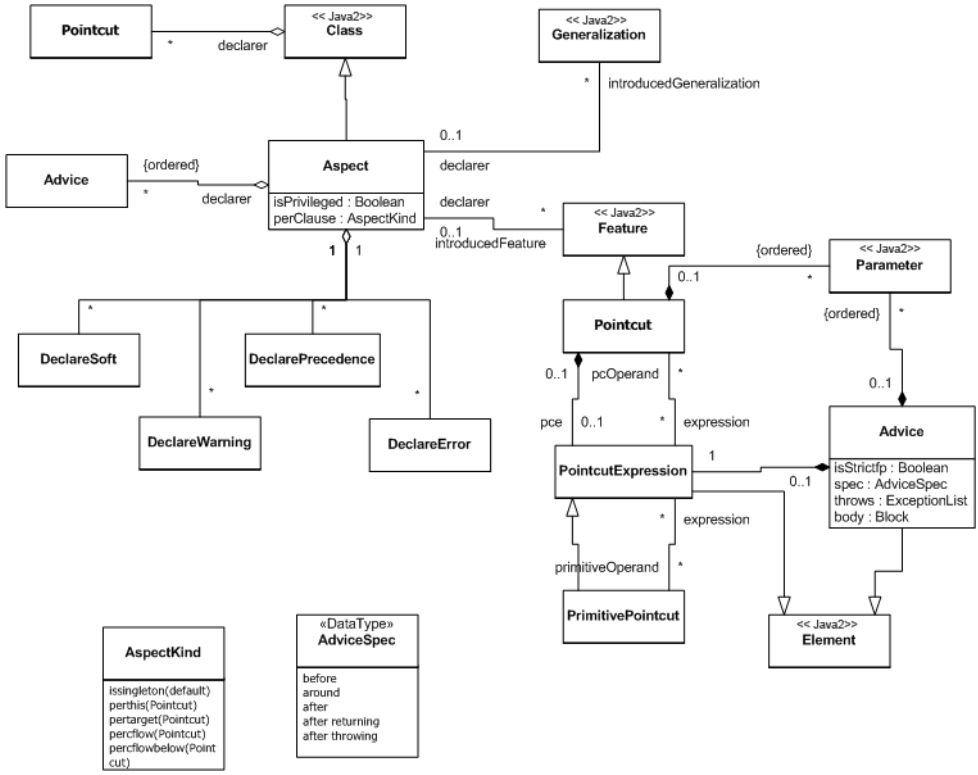

Fig. 11. Java2 Metamodel

5.1 First Approach: Inheritance

This first approach is based on a traditional object-oriented solution. The idea behind this approach is to use the inheritance as the only possible metamodel extension mechanism. This assumption implies that we have to create a new virtual metaclass in the new metamodel which will extend the class in the original metamodel. This idea is illustrate by Figure 13. This figure shows an excerpt of the `com.metamodel.aspectj` after applying this approach.

If we compare the excerpt shown in Figure 13 and the original AspecJ metamodel depicted in Figure 11, we see a new metaclass named `Class`. This metaclass inherits from the metaclass `Class` declared in the package `com.metamodel.java2`. The composition relationship between `Java2::Class` and `Pointcut` in the original metamodel (Figure 11) has been replaced in Figure 13 by the composition between `Pointcut` and `Class`.

Therefore, if we apply the same approach to all the relations between the classes included in both packages, we have four new classes in the `com.metamodel.java2` package. Although this approach is simple, it has an inconvenience: our `aspectj` metamodel implementation is tangled with a set of implementation classes.

```
package com.metamodel.aspectj;

import com.metamodel.java2.Feature;
import org.eclipse.emf.common.util.EList;
import com.metamodel.java2.Class;
import com.metamodel.java2.Parameter;

/**
 * @model
 */
public interface Pointcut extends Feature {

   /**
    * @model
    * type="com.metamodel.java2.Class"
    * lowerBound="1" upperBound="1"
    */
    Class getDeclarer();

   /**
    * @model
    * type="com.metamodel.java2.Parameter"
    * containment="true"
    */
    EList getParameters();

   /**
    * @model type="PointcutExpression"
    * containment="true" lowerBound="0"
    * upperBound="1"
    */
    PointcutExpression getPce();

   /**
    * @model type="PointcutExpression"
    * opposite="pcOperand"
    */
    EList getExpression();
}
```

```
package com.metamodel.aspectj;

import org.eclipse.emf.common.util.EList;
import com.metamodel.java2.Class;
import com.metamodel.java2.Feature;

/**
 * @model
 */
public interface Aspect extends Class{
   /**
    * @model
    */
    boolean isIsPrivileged();

   /**
    * @model
    */
    AspectKind getPerClause();

   /**
    * @model type="Advice" containment="true"
    */
    EList getAdvices();

   /**
    * @model
    * type="com.metamodel.java2.Generalization"
    */
    EList getGeneralizations();

   /**
    * @model type="com.metamodel.java2.Feature"
    */
    EList getIntroducedFeatures();

   /**
    * @model type="DeclareSoft"
    * containment="true"
    */
    EList getDeclareSoft();

   /**
    * @model type="DeclareWarning"
    * containment="true"
    */
    EList getDeclareWarning();

   /**
    * @model type="DeclarePrecedence"
    * containment="true"
    */
    EList getDeclarePrecedence();

   /**
    * @model type="DeclareError"
    * containment="true"
    */
    EList getDeclareError();
}
```

Fig. 12. AspectJ Implementation

5.2 Second Approach: Inter-type declarations

The second approach has been inspired by AspectJ. The idea is to introduce a new constructor that can be understood by the EMF generator in order to deal with aspects. The constructor will be an `aspect` and, in AspectJ, the only thing we have to do is to introduce a inter-type declaration. Thus, to implement the relationship between `Class` and `Pointcut` we will write something similar to the code that appears in Figure 14.

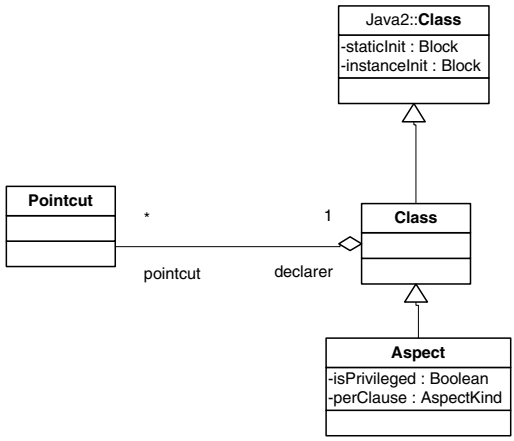

Fig. 13. Excerpt of the AspectJ implementation package applying the inheritance approach

```
import com.metamodel.java2.Class;
import org.eclipse.emf.common.util.EList;

/**
  * @model
  */
public aspect ClassAsp {

   /**
    * @model type="com.metamodel.aspecj.Pointcut"
    * containment=true
    **/
    public EList Class.getPointcuts();

}
```

Fig. 14. Definition of an aspect with an inter-type declaration

If the AspectJ syntax and compiler are used, the solution shown in Figure 14 presents a problem: Only classes may have the inter-type declarations. But, our `Class` is an interface. To solve the problem the EMF generator should ignore this error and generate an aspect for the class `ClassImpl`. The EMF generator should read and use the annotations that have been included in the aspect.

5.3 Third Approach: Relationship Aspect

The third approach is related to the treatment of relatioships as first-class citizens. Although relationships are treated as first-class citizens at the modelling level, the same treatment is not maintained at the implementation level. At this level, the implementation of relationships is hand-crafted and spread across the objects which participate in those relationships. It has been proposed in [23] to model these relationships as separable, crosscutting concerns.

In order to clarify this approach, we will introduce an example of the relationship aspect obtained from [23], and afterwards, we will apply this relationship aspect to the reuse of metamodels. Figure 15 shows an UML representation of a relationship between the students that attend some courses.

If we want to implement the UML diagram of Figure 15, we have to hard code

Fig. 15. Simple UML Diagram that shows a relationship between the students that attend some courses

the relationship in the classes `Student` and `Course`. Figure 16 shows a possible implementation of this diagram. There is an attribute `attends` in the `Student` class to implement one of the ends of the relationship, and attribute `attendees` in the `Course` class to implement the other end of the relationship. The syntax of this approach is totally compatible with the AspectJ syntax.

```
import java.util.HashSet;

public class Student {

        String name;
        Integer number;

        HashSet<Course> attends;
}
```

```
import java.util.HashSet;

public class Course {
        String code;
        String title;
        Integer workload;

        HashSet<Student> attendees;
}
```

Fig. 16. An implementation of the UML Diagram of Fig. 15

Using the *Relationship Aspect Library (RAL)* proposed in [23] the relationship `Attend` is implemented as an aspect which extends `SimpleStaticRelationship`, a generic AspectJ aspect from the RAL library. Figure 17 shows this implementation.

```
import java.util.HashSet;

public class Student {

        String name;
        Integer number;

}
```

```
import java.util.HashSet;

public class Course {
        String code;
        String title;
        Integer workload;

}
```

```
public aspect Attends extends StaticRel<Student, Course>{
}
```

Fig. 17. An implementation of the UML Diagram of Fig. 15

In Figure 18 we apply this approach to implementation of the relationships between Poincut and Class of the AspectJ metamodel. Instead of hand code `get()` methods in the interfaces representing the ends of the relationships, we define an aspect for each relationship in the metamodel. This aspect extends an abstract aspect that should be defined in an Aspect Relationship Library. This library can be the one defined in [23] or a new one of our creation. There are some limitations of the *RAL* library that should be taken into account. The current implementation of the

RAL library does not support two different relationships between the same classes due to name problems. For example, in the `com.metamodel.aspectj` package we will have problems to define the association and the composition relationships between `Pointcut` and `PointcutExpression`.

```
import com.metamodel.java2.Class;

/**
 * @model
 **/
public aspect PointcutDeclaration extends CompositionRel <Class, Pointcut>{

}
```

Fig. 18. An aspect for implementing the relationship between Pointcut and Class

6 Conclusions and further work

This paper has presented a problem that arises when trying to extend a metamodel with relationships different from inheritance. In this case, the original metamodel is not oblivious because it should be modified in order to implement the extension.

Three different approaches to solve the problem have been investigated. The first approach is based on the inheritance mechanism of traditional object-oriented languages. Although it is a simple approach, it allows for extension of the new metamodel with different virtual classes which are only needed for implementation.

The second approach introduces the inter-type declarations. This approach has an inconvenience: the AspectJ compiler does not allow the use of inter-type declarations for interfaces.

The third approach treats relationships as first-class citizens at the implementation level. The relationships are defined as aspects. The Relationship Aspect Library (RAL) [23] has been taken as an example, but this library should be extended to deal with more kinds of relationships. Moreover, the RAL library allows the definition of only one relationship between to classes, which is not sufficient for metamodelling.

As a result of the analysis of the three approaches, we consider the third one as the better option to improve the reuse of metamodels because this approach allows for

- localizing and reusing relationships;
- reducing the coupling of the metamodel implementations.

Moreover, using the third approach we can take advantage of the AspectJ compiler. The code needed in this approach can be validated by the AspectJ compiler.

In a future work we intend to adapt the RAL library to our needs. This implies the solution of some problems that the RAL library has, such as the naming problem arising when two different relationships are defined between two classes. It also implies the necessity of the definition of various kinds of relationships within the RAL library. We also plan to add these new aspect oriented features to current modelling and metamodelling frameworks.

References

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

[2] J. Bezivin, F. Jouault, P. Valduriez. *First Experiments with a ModelWeaver*. Proceedings of the Workshop on Best Practices for Model Driven Software Development held in conjunction with the OOSPLA conference. 2004.

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

[4] T. Clark, A. Evans, S. Kent. *A metamodel for Package Extension with Renaming*. Proceedings of the UML Conference (UML'02). LCNCS 2460, pp. 305-320, 2002.

[5] S. Cook, *Domain-Specific Modeling and Model Driven Architecture*, MDA Journal. Jan, 2004.

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

[7] D. D'Souza, A. Willis. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[8] Eclipse, *Tutorial: Generating Extended EMF 2.1 Model*.Avalaible at: http://www.eclipse.org/emf. July, 2005.

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

[10] 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.

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

[12] 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.

[13] S. Hanenberg, R. Unland. *Concerning AOP and Inheritance*In: Mehner, K., Mezini, M., Pulvermller, E., Speck, A. (Eds.): Aspect-Orientation - Workshop. Paderborn, Mai 2001, University of Paderborn, Technical Report, tr-ri-01-223, 2001.

[14] A. Kleppe, J. Warner, W. Best. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2004.

[15] metamodel.com "What is metamodelling?". Avalaible at: http://www.metamodel.com/staticpages/index.php?page=20021010231056977.

[16] J. Muñoz, V. Pelechano. *MDA vs. Factorías Software*. Proceedings of the Second Workshop on Model-Driven Development, MDA and Applications (DSDM'05).pp. 1-10.Granada, Spain. Sept, 2005.

[17] NetBeans. *Metadata Repository (MDR)*. Avalaible at: http://mdr.netbeans.org/.

[18] Novosoft. *NovoSoft Metadata Framework*. Avalaible at: http://nsuml.sourceforge.net/.

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

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

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

[22] OMG, *UML 2.0 Superstructure Specification*, 2004.

[23] D. J. Pearce, J. Noble. *Relatioships Aspects*. Proceedings of the 5th Aspect Oriented Sofware Development Conference (AOSD'06). Bonn, Germany. Mar, 2006.

[24] I. Porres. *A Toolkit for Model Manipulation* Springer International Journal on Software and Systems Modeling, vol: 2, num: 4, 2003.

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

[26] N. Ubayashi, T. Tamai, S. Sano, Y. Maeno, S. Murakami. *Model Evolution with Aspect-Oriented Mechanisms*. Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05). IEEE Publishing.