Model to Text Transformation in Practice: Generating Code from Rich Associations Specifications

Manoli Albert, Javier Muñoz, Vicente Pelechano, and Óscar Pastor

Department of Information Systems and Computation
Technical University of Valencia
Camino de Vera s/n
46022 Valencia (Spain)
{malbert, jmunoz, pele, opastor}@dsic.upv.es

Abstract. This work presents a model to code transformation where extended UML association specifications are transformed into C# code. In order to define this transformation the work uses a conceptual framework for specifying association relationships that extends the UML proposal. We define a set of transformation rules for generating the C# code. The generated code extends an implementation framework that defines a design structure to implement the association abstraction. The transformation takes as input models those which are specified using the conceptual framework. Transformations have been implemented in the Eclipse environment using the EMF and MOFScript tools.

1 Introduction

Model to text transformations play a key role in MDA based methods for the development of software systems. The assets that are produced by this kind of methods usually are source code files in some programming language. Therefore, model to text transformation are used in most of the projects that apply the MDA. Currently, there is a lack of specific and widely used techniques for specifying and applying this kind of transformations. The OMG "*MOF Model to Text Transformation Language RFP*" aims to achieve a standard technique for this task. Anyway, guidelines and examples of model to text transformations are needed in order to improve the way this step is performed in MDA based methods. In this work, we introduce a model to text transformation for a specific case: the generation of code to implement in OO programming languages (C# in our example) extended UML association relationships specified at the PIM level.

In order to do this, we use a conceptual framework that was introduced in [8] for precisely specifying association relationships in Platform Independent Models. This conceptual framework defines a set of properties that provide to the analyst mechanisms for characterising the association relationships. Then, we propose a software framework for implementing them. The framework, which has been implemented using the C# programming language, applies several design patterns in order to improve the quality of the final application. Using these two items, we define transformations for automatically converting the PIMs that are defined using the

J.F. Roddick et al. (Eds.): ER Workshops 2006, LNCS 4231, pp. 63 – 72, 2006.
© Springer-Verlag Berlin Heidelberg 2006

conceptual framework, into code that extends the implementation framework. We implement this model to text transformation using Eclipse plug-ins for model management. Concretely, we use EMF to persist and edit the models and MOFScript to specify and apply the model to text transformations.

In short, the main contribution of this paper is a practical application of model to text transformations for automatically generating code from PIMs. In addition, we provide knowledge (a conceptual framework, an implementation framework and a transformation mapping) for specifying and implementing association relationships. This proposal has been developed in the context of a commercial CASE tool (ONME[1]), but the knowledge can be integrated in other MDA based methods, since association relationships are widely used in OO approaches.

The paper is structured as follows: Section 2 briefly presents the conceptual framework that is used in the paper for specifying association relationships. In Section 3 we show our proposal to implement association relationships in OO languages. Section 4 describes the model to text mapping and Section 5 introduces the implementation using Eclipse and MOFScript. Finally, Section 6 contains the conclusions and our future works.

2 A Conceptual Framework for Association Relationships

The meaning of the association construct, central to and widely used in the OO paradigm, is problematic. The definitions provided in the literature for this construct are often imprecise and incomplete. Conceptual modelling languages and methods, such as Syntropy[1], UML[2], OML[3] or Catalysis[4], include partial association definitions that do not solve some relevant questions. Several works have appeared highlighting these drawbacks and answering many important questions regarding associations [5, 6, 7].

To define a precise semantics for the association abstraction, we present a Conceptual Framework [8] that identifies a set of properties that have been extracted and adapted from different OO modelling methods. These properties allow us to characterize association relationships in a conceptual model.

Fig. 1 shows the metamodel for specifying associations using our approach. The three basic elements that constitute an association (the *participating classes*, the *association ends* and the *association)* are represented by metaclasses. The attributes of the metaclasses represent the properties of the conceptual framework that is introduced in the next section.

2.1 Properties of the Conceptual Framework

In this section we briefly present the properties. We introduce the intended semantics of each property in a descriptive way, and its possible values.

Dynamicity: Specifies whether an instance of a class can be dynamically connected or disconnected (creating or destroying a link) with one or more instances of a related class (through an association relationship) throughout its life-time. The property is applied to the associated ends. The values are: *Dynamic* (the connection and disconnection is

[1] http://www.care-t.com

possible), *Static (*the connection and disconnection are no possible), *AddOnly* (only the connection is possible) and *RemoveOnly* (only the disconnection is possible).

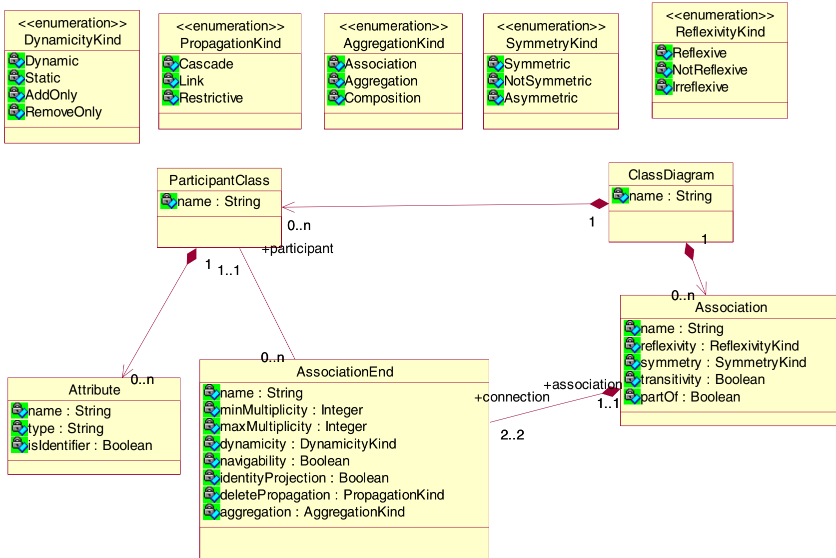

Fig. 1. Metamodel for specifying association following our conceptual framework

Multiplicity (maximum and minimum): Specifies the maximum/minimum number of objects of a class that must/can be connected to one object of its associated class. The property is applied to the associated ends.

Delete Propagation: Indicates which actions must be performed when an object is destroyed. The property is applied to the associated ends. The possible values are: *Restrictive* (the object cannot be destroyed if it has links), *Cascade* (the links and the associated objects must also be deleted) and *Link* (the links must be deleted).

Navigability: Specifies whether an object can be accessed by its associated object/s. The property is applied to the associated ends. The property value is *true* if the objects of the opposite end can access the objects of the class; otherwise the value is *false*.

Identity Projection: Specifies whether the objects of a participating class project their identity onto their associated objects. These objects are identified by their attributes and by the attributes of their associated objects. The property is applied to the associated ends. The property value is *true* if the class of the opposite end projects its identity; otherwise the value is *false*.

Reflexivity: Specifies whether an object can be connected to itself. The property is applied to the association. The possible values are: *Reflexive* (the connection is mandatory), *Irreflexive* (the connection is not possible) and *Not Reflexive* (the connection is possible but not mandatory).

Symmetry: Specifies whether a **b** object can be connected to an **a** object, when the **a** object is already connected to the **b** object. The property is applied to the association.

The possible values are: *Symmetric* (the connection is mandatory), *Antisymmetric* (the connection is not possible) and *Not Symmetric* (the connection is possible but not mandatory).

Transitivity: Specifies whether when an `a` object is connected to a `b` object, and the `b` object is connected to a `c` object, it implies that the `a` object is connected to the `c` object. The property is applied to the association. The property value is *true* if the implicit transition exists; otherwise the value is *false*.

Using this conceptual framework we can specify associations in a very expressive way. Furthermore, these properties have been used in [8] for characterizing the association, aggregation and composition concepts in the context of a commercial tool that follows the MDA proposal (the ONME Tool).

In the next section, we present the software representation of an association relationship that is characterized by the framework properties.

3 Implementing Association Relationships

Most object oriented programming languages do not provide a specific construct to deal with associations as first level citizens. Users of these languages (like C# and Java) should use reference attributes to implement associations between objects. Following this approach, an association is relegated to a second-class status. In order to solve this situation, several approaches have been proposed to implement association relationships (as it has been presented in [9]). Nevertheless, in these approaches some expressivity is missed in order to support those properties which are widely used for specifying association relationships.

3.1 Design Patterns

Our proposal for implementing associations provides a software framework that combines a set of design patterns [10]. The goal of our framework is to provide quality factors like **loose coupling** (since most of the implementation proposals introduce explicit dependencies which make difficult the maintainability of the application), **separation of concerns** (since the objects of the participating classes could have additional behaviour and structure to those specified in the domain class) and **reusability and genericity** (since most of the association behaviour and structure can be generalized for all associations.).

In order to achieve these goals, we present a solution that combines three design patterns: the **Mediator,** the **Decorator** and the **Template Method**. The next section shows how these patterns are applied to implement associations.

3.2 Framework Structure

We combine the design patterns selected to obtain a composite structure of design classes that implements an association relationship. Taking into account the association relationship and the participating classes, in this section, we present the design classes that constitute the framework.

Fig. 2 shows the generic structure of design classes that represents an archetype association (two participant classes connected through an association). Next, we describe the elements of the figure.

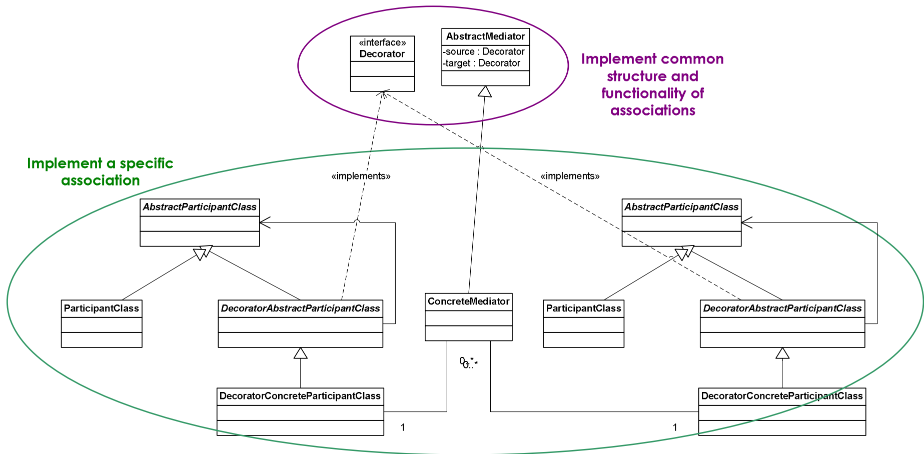

Fig. 2. Design Classes of the Implementation Framework

Participant classes (`ParticipantClass` in the figure), implement the participant classes of the conceptual model according to their specifications. The application of the **Decorator Pattern** implies the definition of decorated classes (`DecoratorConcreteParticipantClass`) that wrap the participant classes and represent the association end in which they participate. These classes implement the structure and behaviour that is added to the participant classes as a consequence of their participation in associations. Moreover, the application of this pattern results in the definition of an abstract decorator class (`DecoratorAbstractParticipantClass`) that generalizes every decorators (association ends) of a participant class. Finally, the pattern entails the definition of an abstract participant class (`Abstract ParticipantClass`), from which the participant and the decorator abstract classes inherit. The decorator abstract classes keep a reference to an object of the abstract participant class, representing the decorated object. This class structure allows to use in the same way a decorated object (an object of a participant class) and a decorator object (an object of a decorator class).

The **Mediator Pattern** is applied in order to encapsulate the interaction of the participant objects. The application of this pattern results in the definition of a mediator class (`ConcreteMediator`) that implements the structure and behaviour of an association (independently of the participant classes). This class connects the concrete decorator classes that represent the association ends (since the participant classes are implemented in an isolated way from the association in which they participate).

The **Template Pattern** is applied in the context of the mediator class. We define an abstract class (`AbstractMediator`) for the concrete mediator class to implement the common structure and behaviour of the associations, describing common execution strategies. This class defines the template methods for the link creation and destruction and includes a reference to each participant object.

Finally, we define an interface for the decorator classes in order to specify those methods that must be implemented in the decorator classes. The abstract decorator classes implement this interface.

3.3 Functionality Implementation

In this section we present the part of the framework that regards to the functionality. The definition of an association between two classes implies the implementation of new functionality. This functionality is the following:

- *Link Creation:* allows the creation of links between objects of the participating classes. The implementation of this functionality requires checking the **reflexivity**, **symmetry, transitivity** and **maximum multiplicity** properties.
- *Link Destruction:* allows the destruction of links between objects of the participating classes. The implementation of this functionality requires checking the **reflexivity**, **symmetry, transitivity, minimum multiplicity** and **delete propagation** properties.
- *Participant Object Creation:* allows the creation of decorator objects independently of the creation of their decorated objects. The implementation of this functionality requires checking the **minimum multiplicity** and **reflexivity** properties.
- *Participant Object Destruction*: allows the destruction of decorator objects independently of the creation of their decorated objects. The implementation of this functionality requires the checking the **delete propagation** property.

The next section introduces the mappings between the association specification and its implementation. We also present how the properties affect the implementation of the methods that have been introduced in this section.

4 Mapping Association Specifications into Code

This section describes the transformation from models that are specified using our conceptual framework into C# source code files. First of all, we are going to intuitively describe the mapping. Then, we show the implementation of this model to text transformation.

4.1 Metaclasses Mapping

In order to describe the mapping, we introduce the implementation classes that are generated from the metaclasses of the PIM metamodel.

- **ParticipantClass**: Every *ParticipantClass* element in the model generates three classes:
  1. `AbstractDomainClass`: this class defines the attributes and operations specified in the *ParticipantClass*. Note that all the information in this class is independent of the class associations.
  2. `DomainClass`: this class implements the operations specified in the `AbstractDomainClass`.
  3. `AbstractDecorator`: this class implements the methods which are used for the management of the links (create and delete a link).
- **AssociationEnd**: Every *AssociationEnd* element in the model generates one class:
  1. `ConcreteDecorator`: this class extends the `AbstractDecorator` class, which has been generated from the *ParticipantClass* element.
- **Association**: Every *Association* element in the model generates one class:
  1. `Mediator`: this class extends the `AbstractMediator` class from the implementation framework.

The contents of these implementation classes and their methods depend on the values of properties specified in the PIM level. Next, we present briefly (due to space limitations) the representation of the properties in the framework.

4.2 Properties Mapping

Identity Projection. The value of this property determines how are implemented the identifier attributes in the `AbstractDomain` C# classes.

Dynamicity. Depending on the value of this property, methods for adding and deleting links are included in the opposite concrete decorator C# class.

Navigability. The value of this property determines if it is necessary to limit the access to the objects of the end by their associated objects.

Reflexivity, Symmetry, Transitivity, Multiplicity and Delete Propagation. The constraints that are imposed by these properties are checked by specific methods. The implementation of these methods depends on the values assigned to the properties. In section 3.3 we have described when these properties must be checked.

5 Transforming Models to Code. The Tools

We have implemented the transformation that has been introduced in this paper using the Eclipse environment. Eclipse is a flexible and extensible platform with many plug-ins which add functionality for specific purposes. In this work we have used the Eclipse Modelling Framework (EMF)[2] for the automatic implementation of the metamodel shown in Fig. 1. This metamodel provides the primitives for specifying association relationships using the properties that are defined in our conceptual framework. The EMF plug-in automatically generates the Java classes which

[2] http://www.eclipse.org/emf

implement functionality for creating, deleting and modifying the metamodel elements, and for the models serialization.

In order to implement the model to text transformation, we have used the MOFScript tool that is included in the Generative Model Transformer (GMT)[3] Eclipse project. The MOFScript tool is an implementation of the MOFScript model to text transformation language. This language was submitted to the OMG as response to the "*MOF Model to Text Transformation Language RFP"*.

In this work, **we have selected the MOFScript language/tool for several reasons**: (1) MOFScript is a *language specifically designed for the transformation of models into text files,* (2) MOFScript *deals directly with metamodel descriptions* (Ecore files) as input, (3) MOFScript transformations *can be directly executed from the Eclipse environment* and (4) MOFScript *provides a "file" constructor* for the creation of the target text files.

MOFScript provides the "*texttransformation*" constructor as the main language primitive for organizing the transformation process. A transformation takes as input a metamodel, and it is composed of one or several rules. Every rule is defined over a context type (a metamodel element). Rules can have arguments and/or return a value. The special rule called "main" is the entry point to the transformation.

5.1 Implementing the Transformation Using the MOFScript Tool

We have structured our transformation in several modules:

- We define a specific transformation for each kind of class (file) (`ConcreteDecorator`, `Mediator`, etc.).
- The root transformation is in charge of navigating the model and invoking the specific transformations.

Next we show the root transformation, which takes as input a model that is specified using our metamodel. The main rule iterates (using the *forEach* MOFScript constructor) over *ParticipantClass* and *Association* elements. Moreover, the rule iterates over the *AssociationEnd* elements of every *ParticipantClass.* The files are generated following the mapping described in Section 4.

```
import "ParticipantAbstract.m2t"
import "Participant.m2t"
import "DecoratorAbstract.m2t"
import "DecoratorConcrete.m2t"
import "MediatorConcrete.m2t"
import "Mediator.m2t"

texttransformation Association2CSharp (in asso:"http:///associationmodel.ecore"
) {

asso.ClassDiagram::main(){

   self.ParticipantClass->forEach(c:asso.ParticipantClass) {
       file (c.name+"Abstract.cs")
       c.generateParticipantAbstractClass()
       file (c.name+".cs")
       c.generateParticipantClass()
       file ("Decorator"+c.name+"Abstract.cs")
       c.generateDecoratorAbstractClass()
        c.AssociationEnd->forEach(end:asso.AssociationEnd){
           file ("Decorator" + c.name + end.association.name +".cs")
           end.generateDecoratorConcreteClass()
```

3 http://www.eclipse.org/gmt/

```
        }
      }
      self.Association->forEach(a:asso.Association){
         file("Mediator"+ a.name +".cs" )
         a.generateMediatorConcreteClass()
      }
      file("Mediator.cs")
      self.generateMediatorClass()
      }
   }
```

The description of the transformations that generate every file can not be included in this paper due to space constraints. An Eclipse project with the transformation can be downloaded from http://www.dsic.upv.es/~jmunoz/software/. Next, we (partially) show the transformation that is in charge of generating the decorator concrete classes.

```
01 texttransformation DecoratorConcrete (in asso:"http:///associationmodel.ecore")
02 {
03 asso.AssociationEnd::generateDecoratorConcreteClass(){
04     <%using System;
05 using System.Collections;
06 using System.Text;
07 namespace org.oomethod.publications {
08    public class Decorator%> self.participant.name + self.association.name
09 <%:Decorator%>self.participant.name<%Abstract {
10
11 %>
12 //Definition of the collection reference
13 self.association.conexion->forEach(con:asso.AssociationEnd | con <> self ) {
14 <%             ArrayList %> con.participant.name <%sCollection;
15
16 %>
17 }
18 //Constructor
19 //...
20 //Insert Link Method
21 self.association.conexion->forEach(con:asso.AssociationEnd | con<>self ){
22    if ( con.dynamicity=="Dynamic" or con.dynamicity=="AddOnly" ){
23    <%          public Mediador insert%>
24    print(con.participant.name)<%(Decorador%> con.participant.name +
25    self.association.name
26     <% object) {
27        return (new Mediadora%> self.association.name <%(object, this));
28      }
29
30%>
31   }}
32 //...
33 }
```

This transformation creates a decorator concrete class for each association end associated to a class. Next, we describe the most relevant issues of the transformation:

- Lines 08-09: The class inherits from its corresponding abstract decorator class.
- Lines 12-17: A collection is defined to maintain a reference to the links. The name of this collection is based on the name of the class at the opposite end.
- Lines 20-31: An insert link method is created depending on the value of the dynamicity property of the opposite end. If the value is *Dynamic* or *AddOnly* the method is defined. Otherwise the method is not defined. The name of the method is based on the name of the class at the opposite end.

6 Conclusions

In this work we have introduced a practical case study of model to text transformation. This transformation takes as input models that are specified using the primitives of a conceptual framework for precisely specifying association relationships. The results of the transformation are C# classes which implement the association relationships using design patterns.

In the context of MDA, the transformation introduced in this paper is a PIM to Code transformation. We do not explicitly use intermediate PSMs for representing the C# classes. Currently, we are working on the development of the PIM-PSM-Code implementation of the transformation that has been introduced in this paper. This work will provide a precise scenario for the comparison of both approaches.

Another line of research includes the implementation of the association-to-C# transformation for the persistence and presentation layers. These layers are currently implemented by hand, but we have defined the correspondence mappings. Our goal is to automate these mappings using a similar approach to the one that has been introduced in this paper.

References

1. S. Cook and J. Daniels. *Designing Objects* Systems. Object-Oriented Modelling with Syntropy. Prentice Hall, 1994.
2. Object Management Group. *Unified Modeling Language Superstructure*, Version 2.0. 2005
3. Firesmith, D.G., Henderson-Sellers, B. and Graham, I. *OPEN Modeling Language (OML) Reference Manual*, SIGS Books, 1997, New York, USA.
4. D.F. D'Souza and A.C. Wills. Objects, C*omponents and Frameworks with UML*. Addison-Wesley, 1998.
5. Gonzalo Genova. "*Entrelazamiento de los aspectos estático y dinámico en las asociaciones UM*L" PhD thesis, Dept. Informática. Universidad Carlos III de Madrid. 2003.
6. Monika Saksena, Robert B. France, María M. Larrondo-Petrie. "*A characterization of aggregation.*", In Proceedings of OOIS'98, Springer editor, pp 11-19. C. Rolland, G. Grosz, 1998
7. Brian Henderson-Sellers and Frank Barbier. "*Black and White Diamonds*". In Proceedings of UML'99. The Unified Modeling Language Beyond the Standandard, 1999, Springer-Verlag, R.France and B.Rumpe editors, pp 550-565.
8. Manoli Albert, Vicente Pelechano, Joan Fons, Marta Ruiz, Oscar Pastor. "*Implementing UML Association, Aggregation and Composition. A Particular Interpretation Based on a Multidimensional Framework*". In Proceedings of CAISE 2003, LNCS 2681 pp 143-158.
9. M. Dahchour. "*Integrating Generic Relationships into Object Models Using Metaclasses*", PhD thesis, Dept. Computing Science and Eng., Université Catholique de Louvain, Belgium, Mar. 2001.
10. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.