Mapping the UML2 Semantics of Associations to a Java Code Generation Model

Dominik Gessenharter

Ulm University, Institute of Software Engineering
and Compiler Construction, D-89069 Ulm, Germany
Dominik.Gessenharter@uni-ulm.de

Abstract. It is state of the art to provide UML modeling by means of class diagrams and code generation from there. But whereas drawing diagrams is most often well supported, code generation is limited in scope. Association classes, multiplicities, aggregation and composition are not correctly or not at all processed by most code generators. One reason may be that the UML semantics is not formally defined in the UML specification. As a result of that, associations are usually transformed into code by using properties of the same type as the associated classes or corresponding typed sets. This approach must fail although the UML2 Superstructure Specification considers association ends owned by a class to be equal to a property of the owning class. In this paper, we describe why associations should be implemented as classes when generating code from class diagrams.

Keywords: UML, Associations, Code Generation, Java.

1 Introduction

OMG's Model Driven Architecture [18] is an approach to a Model Driven Development (MDD) process using the Unified Modeling Language (UML). MDD concentrates on models, whereas code is generated from there. Only the quality of a model should influence the code's quality, i. e. a highly detailed model covering all aspects of an application should result in automatically generated code with no necessity for adaptations. This ambition requires good code generation tools. But most code-generators produce code that does not cover all the elements of the input model.

This situation is problematic: Models are the result of the analysis and design phases and must conform to all requirements of a system and therefore, generated code should cover the entire semantics of the model. Otherwise, the conformance of the code to the model remains in the sole responsibility of the developer. This entails more lines of hand-written code and is error prone. Furthermore, the verification of a model is invalidated. Generated code should enforce all constraints to be always held, as it is the case in modern database systems, where foreign key definitions allow to define the rejection of the deletion of a dataset if it would result in a violation of the underlying ER-Model.

With regard to associations, both the UML specification [21] and code generation should be improved. We contribute to both in the following way:

- We shortly introduce the specification [21] of associations in Sect. 2 and show its shortcomings.
- An evaluation of some code generators reveals, that only the code pattern discussed in Sect. 3 is used for the implementation of associations.
- In Sect. 4 we point out, why this pattern is invalid and how it relates to deficiencies of the UML specification [21].
- In Sect. 5, we discuss how another approach of translating associations considers the characteristics of associations that are explained in Sect. 2.

The context of this work is the ActiveCharts project [23], which requires a code generator for static structure modeled in UML 2 class diagrams. It is based on the UML 2.1.2 Superstructure Specification [21]. Keywords of the UML as well as class or instance names are written in *italics*. Instances of classes are named with the class name in lowercase letters.

2 Associations at a Glance

In this section, we describe the concept of an association. We concentrate on binary associations unless otherwise stated.

2.1 Elements of Associations

> "An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. (...) Each end represents participation of instances of the classifier connected to the end in links of the association." [21, p. 39]

An association end may be adorned with a role name, a multiplicity, a property string, a navigability and a visibility modifier. It may be owned by the association itself or by the classifier connected to the opposite end of the association.

The UML provides no means of defining a uni-directional association. Relating a class A to a class B entails that both classes are related to each other and associations therefore cannot be uni-directional.

> "An association describes a set of tuples whose values refer to typed instances. An instance of an association is called a link." [21, p. 39]

However, the UML specification is general enough to allow both uni-directional and bi-directional links. We write a bi-directional link for an association L between classes A and B as $(a \leftrightarrow b)_L$, a uni-directional link from a to b as $(a \rightarrow b)_L$. For brevity, the index $_L$ will be omitted when no confusion is possible. A reference on b held by a (in Java) is denoted by $a \rightarrow_{ref} b$.

Note that uni-directional links are not contradictory to bi-directional associations. A bi-directional association may be implemented by two uni-directional links as long as they are consistent as defined in the following section.

2.2 Consistency of Associations

A tuple (a, b) in an association L can be represented by a bi-directional link $(a \leftrightarrow b)$ or two uni-directional links $(a \rightarrow b)$ and $(b \rightarrow a)$. Note that the roles of a and b may be chosen arbitrarily, but must be kept consistent for a given association.

The consistency of unidirectional links for an association is assured when $\forall a : A, b : B : (a \rightarrow b) \Leftrightarrow (b \rightarrow a)$ holds.

2.3 Multiplicities at Association Ends

An essential feature of associations are multiplicities at association ends.

> "A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound." [21, p. 94]

Multiplicities are used to determine how many instances of the associated classes can or must be linked at a time.

2.4 Ownership of Association Ends

The UML specification defines that an association end may be owned by the association itself or by a member end class.

Association ends are properties. Owning an association end means that the property becomes a structural element of the owning classifier, and consequently, part of the object's state. Therefore, adding or removing a link will have an impact on the object's state in cases where the end is owned by a class. For ends owned by associations, the same rules hold, but the impact is on the association.

The relation between the objects remains unaffected, but a directive where to store links at runtime is made. Links may not appear in objects of classes, that do not own the corresponding association end. So far, a correlation between ownership of association ends and implementation of links (in particular where links are to be stored) as suggested by Diskin and Dingel [8] is even mandatory.

2.5 Navigability of Association Ends

> "Navigability means instances participating in links at runtime (instance of an association) can be accessed efficiently from instances participating in links at the other ends of the association." [21, p. 41]

An association end is navigable when it is either a *navigableOwnedEnd* of the association or an *ownedAttribute* of an end class. Otherwise, it is not navigable [21, p. 39]. The specification does not claim that a non-navigable end must not be accessed by opposite ends. It rather states that it may or may not. Accordingly it is not necessary to store links directed in a non-navigable direction of an association, however, it is not invalid to store them either. For an association between A and B navigable from A to B, a link $a : A \rightarrow b : B$ is sufficient.

An informal convention whereby non-navigable ends are assumed to be owned by the association whereas navigable ends are assumed to be owned by the classifier at the opposite end is deprecated [21, p.43], but we find a dependency when having a look at the inversion. A non-navigable end must be owned by the association. If owned by the class via *ownedAttribute*, it is navigable.

2.6 Visibility of Association Ends

The visibility "determines where the NamedElement appears within different Namespaces within the overall model, and its accessibility" [21, p.98]

The specification allows to define a non-navigable end as public. Such an end must be owned by the association (because it is non-navigable) but may be accessed. However, a non-navigable end not necessarily grants access to the opposite classifier. The effect of accessibility to a non-navigable end is unclear.

2.7 Aggregation Types of Association Ends

An association end has an *aggregationType* which may have one of the values *none*, *shared* or *composite*. The values *shared* and *composite* may only appear in binary associations and indicate a whole/part relationship. Aggregation is transitive and may not be modeled in a cyclic way. An association end with *aggregationType composite*

> "indicates that the property is aggregated compositely, i. e., the composite object has responsibility for the existence and storage of the composed objects (parts)." [21, p. 39]

This implies that the deletion of the composite entails the deletion of all of its current parts.

2.8 Piecing it Together

The specification states that "aggregation type, navigability, and end ownership are orthogonal concepts," [21, p. 43]. However, the concepts are not independent since an association end owned by a class via *ownedAttribute* implies that this association end is navigable.

Being non-navigable implies that the association end is owned by the association via *ownedEnd* and must not be included in the subset *navigableOwnedEnd*.

The specification does not define where links are to be stored. This implies that parts of links may not be stored at all if these parts are derivable from other information. Diskin and Dingel [8] give a detailed discussion about that.

Figure 1 shows an association that is navigable from A to B. An instance $b : B$ does not need any information about which instances $a_i : A$ it is related to. This information is derivable from the references stored in each a_i. As these links may be implemented in code by a static `map<A,List<B>>`, accessible through the class A, the multiplicity constraint of the A-end can be satisfied. Each instance $b : B$ must not be contained in more than 10 lists, i. e. be referenced by more than 10 $a_i : A$.

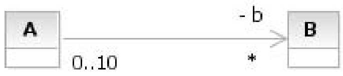

Fig. 1. An association navigable from *A* to *B*

3 Code Generation – State of the Art

We evaluated Altova umodel [3], ARTiSAN Studio [4], Borland Together [5], ChangeVIsion Jude Professional [6], Gentleware Apollo [14], IBM Rational Software Architect [16], No Magic MagicDraw [19], Sparx System Enterprise Architect [24] and Visual Paradigm for UML [27], which are linked in the UML vendor directory [26] as well as EMF [9], Fujaba [11], Gentleware Poseidon [15] and Telelogic Rhapsody [25]. Code generation from UML class diagrams is supported by all these tools, but the level of compliance to the UML specification varies considerably. Having a closer look at associations, some insufficiencies become evident.

3.1 Common Implementation of Associations

All evaluated tools generate code for associations by using attributes in the member end classes. In the case of a binary association, each class obtains an attribute that can store a reference to the opposite class.

3.2 Multiplicities at Association Ends

Except from Together [5], MagicDraw [19] and Enterprise Architect [24], upper bounds of multiplicities are rudimentarily implemented. If it is greater than 1, the attribute representing the association end is of a collection type. Thus, only two states are identified, namely an upper bound that equals to 1 or one of a higher value. None of the tools rejects a new link if the upper bound is violated. Looking at lower bounds, the situation is even worse. They are completely omitted. A compliant mapping of a lower bound to source code is more difficult than that of an upper bound. Nevertheless, it is possible (see Akehurst et al. [2]).

3.3 Ownership of Association Ends

Only umodel [3] and MagicDraw [19] allow modeling the ownership of association ends. However, all tools treat associations as if the association ends were owned by the member end classes. This implies that non-navigable ends become navigable, but the specification does not force non-navigable ends to be not accessible. Because "(. . .) an association end owned by a class is also an attribute" [21, p. 45], the above outlined policy of association implementation is no contradiction to the specification. Developers probably will not miss the feature of modeling ownership, as it is not even discussed in most UML books that address developers like UML2.0 - Das umfassende Handbuch [17] or UML 2 glasklar [22].

3.4 Navigability of Association Ends

The lack of ownership representation in code leads to associations that are navigable in both ways. It is common to omit the generation of an attribute in a member end class if the opposite end is non-navigable. It is assumed that an association where all ends are non-navigable will never appear in a model.

This assumption is problematic because non-navigable associations might be useful in combination with association classes and therefore should be allowed and translated to code.

By omitting the generation of an attribute, the ownership of a non-navigable association end is considered (see Sect 2.5) as it is no feature of the classifier.

3.5 Visibility of Association Ends

The visibility of association ends is taken over to the generated attribute representing the association end. As non-navigable ends are not implemented in code, visibility for those ends is not taken into account. Thus, all implemented association ends are supplied with the intended visibility.

Because a non-navigable end is not necessarily accessible, a visibility modifier is not needed. We consider public visibility of an inaccessible feature useless.

3.6 Consistency of Associations

Many tools just provide the static structure for associations, i. e. the attribute generation in the member end classes. When using those tools, managing the association remains in the scope of the developers' work.

However, Fujaba [11], Rhapsody [25] and EMF [9] generate code that allows to add or remove instances from associations. These tools even implement the consistency of associations:Whenever an instance a is referenced by an instance b (i. e. $b \rightarrow_{ref} a$), the instance b calls a generated method in a passing itself as a parameter. The instance a then obtains a reference to b ($a \rightarrow_{ref} b$) [10].

3.7 Aggregation and Composition

The situation regarding code generation for aggregate associations is staggering. The evaluated tools do not consider the difference between "plain" associations and aggregations or compositions. A more detailed evaluation of the tools will be published in a forthcoming paper. Akehurst et al. [2] evaluated a selection of more tools describing nearly the same situation in 2006. As far as we know, the situation has not substantially changed. In this regard, the modeling facilities of the current tools allow the violation of UML semantics like having a part included in multiple composites at a time. Probably, the reason for this is that a composite cannot be implemented by the standard association implementation. A discussion of this issue is given in Sect 5.5.

3.8 Summary of State-of-the-Art Code Generators

The coverage of code generation ranges from preparing associations by generating adequate attribute definitions to sophisticated mechanisms that manage association consistency at runtime.

However, as we will see in the following section, many deficiencies remain in the common patterns of generated code for associations.

4 Why Do Common Code Generation Methods Fail?

There are two reasons why code generation as described in the previous section fails. The first is that a mutual update of references requires association ends to be visible to the opposite end, the second is that if for non-navigable ends no attribute is generated in a member class, information of multiplicities may be left out.

4.1 The Informally Defined Set of Links

The specification gives an informal definition of the set of tuples as follows:

> "For an association with N ends, choose any N-1 ends and associate specific instances with those ends. Then the collection of links of the association that refer to these specific instances will identify a collection of instances at the other end. The multiplicity of the association end constrains the size of this collection."[21, p. 40]

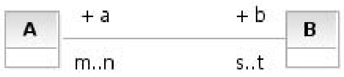

Fig. 2. A binary association

A formalization of the cited definition is given by Diskin and Dingel [8]. The association shown in Fig. 2 defines tuples (a_i, b_j) and for each arbitrarily chosen $a : A$, there must exist at least m and at most n tuples, for each arbitrarily chosen $b : B$, at least s and at most t tuples must exist.

4.2 Implementation of Links by the Use of References

In general, a single reference cannot implement a link. However, two references do as long as the following condition holds:

$\forall\ a_i : A \leftrightarrow b_j : B\ \exists\ a_i \rightarrow_{ref} b_j\ \wedge\ \exists\ b_j \rightarrow_{ref} a_i$

This condition requires a mutual update of both references if a link is changed.

4.3 Dependency between Navigability and Visibility

Navigability requires efficient access to the opposite side of an association. In case of a binary association with both ends navigable and owned by the member end classes, both classifiers can access each other. The use of references as described above suffices.

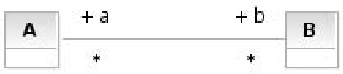

Fig. 3. An association with two navigable ends, each of which has the *VisibilityKind* public

A mechanism for satisfying the condition formulated in Sect. 4.2 is described in [10]. It is based on the idea introduced in Sect 3.6. The code generated for class A and the association in Fig. 3 looks like Listing 1, class B may be implemented analogously. EMF [9], Fujaba [11] and Rhapsody [25] implement this pattern.

```
public class A{
   private B b;
   public boolean setB (B value){
      if (this.b == value) return false;
      B oldValue = this.b;
      this.b = value;
      if (oldValue != null)
         oldValue.setA (null);
      if (value != null)
         value.setA (this);
      return true;
   }
   public B getB () { return this.b; }
}
```

Listing 1. Implementation of mutual updates for links of an association

A problem is that if one (or both) association end(s) is (are) of private visibility, one (or both) side(s) must not be allowed to update the links at the opposite end [13]. The implementation of Fujaba [11] generates non-compilable code when modeling an association with all ends defined private. EMF [9] and Rhapsody [25] violate the visibility and define the attributes as public. It must be considered that the Ecore-model of EMF [9] relates to the MOF [20] (see [1]) and not to the UML [21]. In the Essential MOF (EMOF) that only slightly differs from the Ecore, associations are not defined. It is only possible to link a property to another property called the *opposite*. EMF [9] thus implements the Ecore-model correctly. However, tools claiming to be UML compliant must implement associations, but all evaluated tools except from Fujaba [11] and Rhapsody [25] do not generate code for associations as specified by the UML (see [21]) but in the style of the MOF (see [20]).

In general, the problem can be formulated as follows:

Note 1. If an association is navigable in more than one direction, the *visibility* of the navigable ends must allow the connected classifier to access this end.

4.4 Attributes, Associations and Links

As described in Sect 3.4, navigability is implemented by the suppression of attribute generation for non-navigable ends. This technique follows a transformation of notations (which we term $\mathbf{T_1}$) defined in the specification as follows:

"The attribute notation can be used for an association end owned by a class, because an association end owned by a class is also an attribute. This notation may be used in conjunction with the line-arrow notation to make it perfectly clear that the attribute is also an association end." [21, p. 45]

Instead of *"may be used"*, it should rather state *"can only be used* in conjunction with (...)". A proper application of this transformation is discussed by Crane et al. [7], where classes are enriched by attributes instead of replacing associations.

An analogous transformation from the attribute to the association notation (which we term $\mathbf{T_2}$) is stated as follows:

"An attribute may also be shown using association notation, with no adornments at the tail of the arrow (...)" [21, p. 56].

The item *"with no adornments"* inhibits the explicit definition of multiplicities and navigability which leads to a multiplicity of [1..1] and an undefined navigability. $\mathbf{T_2}$ is supposed to be the inverse to $\mathbf{T_1}$ (for which we write $\mathbf{T_1^{-1}}$ and $\mathbf{T_2} = \mathbf{T_1^{-1}}$ must hold).

Note 2. In general, an association end owned by an end class is not equal to a property. It is strongly required that the information about the association of which the property is an end is not lost even if the opposite end is not navigable.

Note 3. The limitations forced for transformation of an attribute to an association must be forced analogously for a transformation vice versa.

4.5 Links Defined by Attributes

Figure 4 shows two class diagrams. $\mathbf{T_1}$ allows the transformation from the left to the right diagram, $\mathbf{T_2}$ allows the inversion. The multiplicity of the A-end of the association is equal to the default of a multiplicity element 1, because a multiplicity for this end cannot be modeled using the attribute notation. This entails that every instance $b_i : B$ must be linked to an instance $a_j : A$ and $\forall i_1, i_2 : i_1 \neq i_2 \leftrightarrow j_1 \neq j_2 \ \wedge i_1 = i_2 \leftrightarrow j_1 = j_2$. Furthermore, we can see that an instance $b : B$ must not be referenced by more than one instance $a : A$ at a time. This is much more restrictive than the common view of attributes and the way all code generators we evaluated generate code or implementors would write it.

Having a look at the metamodel, we consider the underlying models of the two diagrams not to be identical. An association has at least two ends, each of

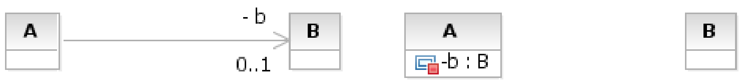

Fig. 4. Attribute notation vs. association notation

which is a *Property*, a subtype of *MultiplicityElement*. An attribute is a *Property*, too, but there is no second *MultiplicityElement* contained in the model.

Note 4. Association notation and attribute notation are defined to be exchangeable. This analogy does not reflect the difference between the concepts on the meta-level.

Modeling a reference from A to B with no restrictions about how many references to B may exist at a time cannot be achieved by an attribute but only by means of an association with appropriate multiplicities. On that issue, the semantics of the specification might be seen as unnatural and we assume that this semantics is not intended by the specification.

5 A Mapping of the UML Semantics to Java Code

This section discusses concepts that can be used to overcome the defiencies outlined in the previous sections. It addresses the issue of extending currently used code generation patterns and presents a promising alternative pattern.

5.1 Extending State-of-the-Art Code Generation

Code generation of EMF [9], Fujaba [11] and Rhapsody [25] is missing consideration of multiplicities, ownership and orthogonality of navigability and visibility. With little effort, checking bounds of multiplicities can be implemented as discussed by Akehurst et al. [2]. As ownership and the conflict between mutual updates and visibility is not addressed there, we suggest the use of separate code classes for associations as it is described in the Complete MOF (CMOF) part of the MOF specification [20] and give a Java implementation for it.

5.2 Using a Separate Code Class for an Association

Note 1 of Sect. 4.3 is a limitation to the use of associations. To overcome this limitation, associations could be translated to classes of their own right. Two alternatives are possible: storing all links in a separate object or creating an object for each link (see Fig. 5). We prefer the second idea because a link is an instance of an association which is a classifier and hence has its own identity.

Note that figure 5 is intended to illustrate the code pattern used to translate the model shown in figure 2, but not to be seen as a transformation of that model. Generating a class for an association is a straight-forward implementation of a link that is defined as a tuple of values by the UML specification. Accordingly, we term instances $ab : AB$ links and the references to instances $a : A$ and $b : B$ the values of a link.

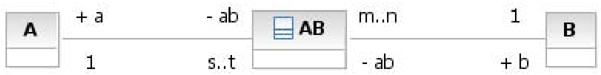

Fig. 5. Association representation by one object per link

Implementing Ownership. An association end that is owned by a member end class is a structural feature of that class. We assume the b-end of Fig. 2 to be owned by A, the a-end to be a *navigableOwnedEnd* of the association AB. Listing 2 is an implementation of A and uses a list of links (line 4) as an implementation of the b-end.

Listing 3 is an implementation of the class B. Note that no links are stored within that class. The implementation of the association uses a static map that contains a list of links for each instance of B that is associated with one or more instances of A (see listing 4, line 5).

```
1  public class A implements ABEnd{
2    private int upper = 2, lower = 0;
3    private B b = null;
4    private List<AB> links = new Vector<AB>();
5    public void addB(B value) throws Exception{
6      b = value;
7      AB.getLink(this);     }
8    public void take(AB link) throws Exception{ link.setB(b); }
9    public void notifyRelation(AB link){ links.add(link); }
10   public boolean canAcceptLink(AB link){ return (links.size()<upper); }
11 }
```

Listing 2. Implementation of a class A of Fig. 2

```
1  public class B implements ABnavigableOwnedEnd{
2    private A a = null;
3    public void take(AB link) throws Exception{ link.setA(a); }
4    public void addA(A value) throws Exception{
5      a = value;
6      AB.getLink(this);     }
7  }
```

Listing 3. Implementation of a class B of Fig. 2

Implementing Navigability. Listings 2, 3 and 4 implement navigable ends. If the a-end is not navigable, the implementation of class B may be reduced to an empty class. The implementation of the association may be shortened as the method `getLink` (lines 31 to 35) is no longer needed. Even links for associations with both ends being non-navigable are possible. However, the question arises which element should be responsible for creating, accessing and removing links.

Implementing Visibility. For an association end that is private, access methods to the corresponding set of links must be defined private (see listing 2 line 5 and listing 3 line 4).

Furthermore, it must be assured that links are only created by instances of the classes participating in the association. This can easily be achieved by passing references to only those types. The required code for this is an application of the visitor pattern [12] and implemented in listing 4 by the `getLink` methods and the `take` methods in the associated classes. The interfaces of listing 5 are needed to guarantee that the associated classifiers allow the passing of the link.

```
1  public class AB{
2    private A a;
3    private B b;
4    int lowerB = 0, upperB = 3;
5    private static HashMap<B, Vector<A>> as = new HashMap<B,Vector<A>>();
6    private AB() {}
7    public A getA() { return a; }
8    public B getB() { return b; }
9
10   public void setA(A a) throws Exception{
11     if(a.canAcceptLink(this)) this.a = a;
12     else throw new Exception(
13       "Violation of upper bound for association end a.");  }
14
15   public void setB(B b) throws Exception{
16     if (!as.containsKey(b)) as.put(b, new Vector<A>());
17     if (as.get(b).size() == upperB) throw new Exception(
18       "Violation of upper bound for association end b.");
19     else this.b = b;  }
20
21   public static void getLink(ABEnd r) throws Exception{
22     if(r instanceof A){
23       AB link = new AB();
24       r.take(link);
25       link.setA((A)r);
26       link.getA().notifyRelation(link);
27       as.get(link.getB()).add(link.getA());     }
28   }
29   public static void getLink(ABnavigableOwnedEnd r) throws Exception{
30     if(r instanceof B){
31       AB link = new AB();
32       r.take(link);
33       link.setB((B)r);
34       link.getA().notifyRelation(link);
35       as.get(link.getB()).add(link.getA()); }
36   }
37 }
```

Listing 4. Implementation of an association owning one of both ends

```
public interface ABnavigableOwnedEnd {
  public void take(AB link) throws Exception;
}
public interface ABEnd extends ABnavigableOwnedEnd{
  public void notifyRelation(AB link);
  public boolean canAcceptLink(AB Link);
}
```

Listing 5. Interfaces required for classes participating in associations

Implementing Consistency of Associations. By passing an instance when invoking the `getLink` method, it is assured that only this instance gets a reference to the link and that it is referenced by the link. The second value of a link is set by the invoking instance. After both values of a link are set, the referenced instances are notified and can store a reference to the link. For removing links, a similar pattern can be used. For lack of space, we give no listing for that.

Implementing Upper and Lower Bound Checks. Checking upper and lower bounds must be done by the owning elements of the association ends. In listing 2, the `canAcceptLink` method in line 10 checks the upper bound, in listing 4 line 18, this is done for a non navigable end when setting the value of a link. Lower bounds must be checked when links are removed in a similar way as it is done when adding links. If all ends of an association have a lower bound greater than 0, a factory pattern [12] can be used for the creation of instances. A factory makes complex object structures unavailable before all links are set and thus ensures that no inconsistent state is revealed to other instances.

Updating References. Consider a 1 to 1 association between classes X and Y and the situation that $x_1 : X$ is associated to $y_1 : Y$ and $x_2 : X$ to $y_2 : Y$. To update the references so that x_1 references y_2 and x_2 references y_1, it must be possible to temporarily violate multiplicities. This issue relates to ACID-transactions of database systems with a special focus on isolation. A mechanism for deferring multiplicity checks and hiding temporarily inconsistent states is most likely possible but not yet considered.

5.3 Implementing Association Classes

For the implementation of an association class, its attributes and operations have to be added to the implementation of the association. A constructor with parameters for supplying association ends with instances could be added as well.

5.4 Making Orthogonal Concepts Independent

By using implementation classes for associations, orthogonal concepts of the model refer to independent parts of the generated code. Visibility of an association end corresponds to the visibility of access methods in the member end class, navigability is integrated in the consideration of ownership which is itself implemented by storing links within that element that owns the association end.

5.5 Implementing Aggregation and Composition

The semantic impact of aggregation is little. It just indicates, that an instance is part of another instance, representing the whole whereas a composition entails that the whole is responsible for the lifetime of its parts.

Java uses a garbage collector that removes unreachable objects. As long as a part is contained in the composite, both the part and the whole are referenced by a link and either both or none of them are removed. This effect runs contrary to lifetime control in the way that the parts may prevent the composite from being destroyed. If an element representing the whole of a composite is no longer referenced by an instance not participating in the composition, it should be removed by the garbage collector. Akehurst et al. [2] consider the use of Java weak references but discard it because this mechanism does not assuredly prevent access to deleted parts. An alternative implementation is given but a drawback of this is that objects are marked inaccessible while links persist.

For the proposed code pattern, a simple idea is applicable: Links are always accessible, either directly by the referenced instances or indirectly by the association. If the whole of a composition is to be deleted, the values of links referencing its parts must be set to `null` if the links are not an instance of the composition itself. If neither the whole nor its parts are referenced by a property of another class, both stay linked but become unreachable from anywhere else and may be removed by the garbage collector. An unsolved problem is that destroying links may violate multiplicities of associated classes.

6 Discussion and Related Work

This paper shows shortcomings of the UML specification concerning relations between associations and attributes as well as dependencies between concepts that are actually designed to be orthogonal.

A straight forward implementation for associations is to use references to the opposite class in each class participating the association. We agree to this implementation if all ends are defined as public, but in general, this code pattern is not sufficient [13]. Akehurst et al. [2] and Génova et al. [13] discuss this technique of association implementation. Unfortunately, visibility is not considered by Akehurst et al. [2] and therefore, an alternative code pattern is not required. Furthermore, it is stated that navigable ends must be owned by the classifier whereas non-navigable ends must be owned by the association. Our work differs from Akehurst et al. [2] by considering visibility and by considering that a navigable association end can be owned by the association. However, Akehurst et al. [2] discuss other characteristics like e. g. qualified associations.

Génova et al. [13] consider visibility and state that associations with both ends defined private cannot be managed because synchronizing references is not possible. Using an implementation class for an association is discussed and discarded, because it is assumed that it does not solve the problem either, since it involves auxiliary classes that cannot provide access to the methods for managing the association to some classes excluding all other classes [13]. Unlike Génova et al. [13], we show how to provide access to links to only some classes by combining the visitor pattern [12] and implementation classes for associations.

A detailed examination of the proposed code pattern with regard to n-ary associations, association classes and specialization of associations as well as a systematic evaluation of tools and code generators will be covered by forthcoming publications. An Eclipse-based implementation of the proposed code generation pattern for ActiveCharts [23] is currently under development.

References

1. The Eclipse Modeling Framework (EMF) Overview (June 16, 2005), `http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html`
2. Akehurst, D., Howells, G., McDonald-Maier, K.: Implementing associations: UML 2.0 to Java 5. In: Software and Systems Modeling (SoSyM), vol. 6(1), pp. 3-35(33). Springer, Heidelberg (2007)
3. Altova Inc. ALTOVA umodel 2008, Enterprise Edition, Version 2008, rel. 2 (2008), `http://www.altova.com/products/umodel/uml_tool.html`

4. ARTiSAN Studio (Version 6.1.21) (2006), http://www.artisansw.com/
5. Borland Software Corporation. Borland Together 2007, Service Pack 1, Version 1.0.0 (2008), http://www.borland.com/us/products/together/index.html
6. ChangeVIsion Inc. Jude Professional 5.2.1,Model Version:27 (2008), http://jude.change-vision.com/jude-web/index.html
7. Crane, M.L., Dingel, J., Diskin, Z.: Class Diagramms: Abstract Syntax and Mapping to System Model (Draft - Version 1.7), School of Computing, Queen's University, Kingston, Ontario, Canada (2006)
8. Diskin, Z., Dingel, J.: Mappings, maps and tables: Towards formal semantics for associations in uml2. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 230–244. Springer, Heidelberg (2006)
9. Eclipse Foundation. Eclipse Modeling Framework (2008), http://www.eclipse.org/emf/
10. Fujaba Associations Specification (2005),http://www.se.eecs.uni-kassel.de/fujabawiki/index.php/Fujaba_Associations_Specification
11. Fujaba Development Group. Fujaba Tool Suite 4.3.2 (2007), http://www.fujaba.de/
12. Gamma, E., Helm, R., Johnson, P., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (2003)
13. Génova, G., Llorens, J., del Castillo, C.R.: Mapping UML Associations into Java Code. Journal of Object Technology 2(5), 135–162 (2003)
14. Gentleware AG. Apollo for Eclipse, Version 3.0 (2008), http://www.gentleware.de/
15. Gentleware AG. Poseidon for UML Professional Edition 6.0.2-0 (2008), http://www.gentleware.com/apollo.html
16. IBM Corp. IBM Rational Software Architect, Version 7.0.0 (2008), http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html
17. Kecher, C.: UML 2.0 - Das umfassende Handbuch. Galileo Computing, Bonn (2006)
18. Object Management Group, MDA Guide 1.0.1, Document 03-06-01 (2003)
19. No Magic Inc. MagicDraw UML 15.5 EAP beta 1 (2008), http://www.magicdraw.com
20. Object Management Group. Meta Object Facility (MOF) Core Specification, Document formal/06-01-01 (2006)
21. Object Management Group. UML 2.1.1 Superstructure Specification, Document formal/2007-02-05 (2007)
22. Rupp, C., Hahn, J., Zengler, B., Queins, S.: UML 2 glasklar. Hanser, München Wien (2007)
23. Sarstedt, S., Gessenharter, D., Kohlmeyer, J., Raschke, A., Schneiderhan, M.: ActiveChartsIDE – An integrated Software Development Environment comprising a component for Simulating UML 2 Activity Charts. In: The 2005 European Simulation and Modelling Conference (ESM 2005) (2005)
24. Sparx Systems. Enterprise Architect,Version 7.1.829 (Build: 829)(2008), http://www.sparxsystems.com/products/ea.html
25. Telelogic. Rhapsody 7.2 (2008), http://www.telelogic.com/products/rhapsody/
26. UML Vendor Directory (2008), http://uml-directory.omg.org/vendor/list.html/
27. Visual Paradigm for UML, Enterprise Edition, Version 6.1 (2007), http://www.visual-paradigm.com/product/vpuml/