Implementing UML Associations in Java

A Slim Code Pattern for a Complex Modeling Concept

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

ABSTRACT

Relationships are difficult to implement for two main reasons: Firstly, they provide a complex semantics for relating entities and secondly, relationships are not *first class* constructs in modern programming languages.

The challenge of implementing relationships in code is to resolve the semantics of abstract model elements and turn them into references or pointers of the target language.

Language extensions or libraries are often discussed as a means of facilitating the translation of relations into code. We present an approach that uses plain Java language concepts for the implementation of UML associations. We focus on symmetry of associations as well as on navigability, visibility and ownership of association ends and present a code pattern, that can easily be used for automatic code generation.

Keywords

UML, Associations, Relationships, Language Extensions, Code Generation

1. INTRODUCTION

Models are a means of describing a system on a high level of abstraction that is closely related to the problem domain. Model-driven development (MDD) is an approach in software development in which models become essential artifacts of the development process, rather than merely serving an inessential supporting purpose [15]. A core theme of MDD is raising the level of automation by using computer technology to bridge the semantic gap between the specification and the implementation [15]. We assume that this requires considering all semantic details of a model when generating code. Thus, it is ensured that characteristics of the models also are characteristics of the implementation. For each omitted detail, it must be proved that a validation of an input model is still applicable to the generated code.

The Model Driven Architecture (MDA) [8] of the Object Management Group (OMG) is an approach for MDD. It defines three levels of abstraction, the computational independent model (CIM), the platform independent model (PIM) and the platform specific model (PSM).

The PSM provides the details needed for implementation and serves as the input for the code generation process. Code generation can also be done on the basis of the PIM, but a preceding transformation to a PSM is considered to be more comprehensible and facilitates a simple one-to-one mapping of concepts, provided that the model and the code meet on the same level of abstraction. Otherwise, the transition from model to code includes a decomposition of high level modeling concepts to lower level concepts of the target language unless programming languages are being extended.

We prefer executing model transformations in the run-up to code generation. These transformations cover the separation of the generated and the programmer's code as well as a decomposition of the complex semantics of UML associations, which is detailed in Sect. 2. Problems of existing approaches are shown in Sect. 3. Afterwards, we introduce the aforementioned model transformation for association decomposition in Sect. 4.

The essence of our approach presented in Sect. 5 is how to support two-way navigable associations with a restricted visibility of association ends by a significantly enhanced mechanism of our previous work [7]. In Sect. 6 we provide more details of how our approach deals with the complex semantics of UML associations.

How generated code is separated from the programmer's code and thus reusability of classes is achieved is shown in Sect. 7. Related work is discussed in Sect. 8.

We contribute to a better implementation of UML associations by

- supporting restricted visibilities for fully navigable associations
- separating generated code from user code
- describing a set of transformations for generating an adequate input for our code generator

2. SEMANTICS OF UML ASSOCIATIONS

The UML provides "an abstract concept that specifies some kind of relationship between objects" [12, p. 132]. "The various subclasses of Relationship will add semantics appropriate to the concept they represent" [12, p. 132]. One of these subclasses is the metaclass Association.

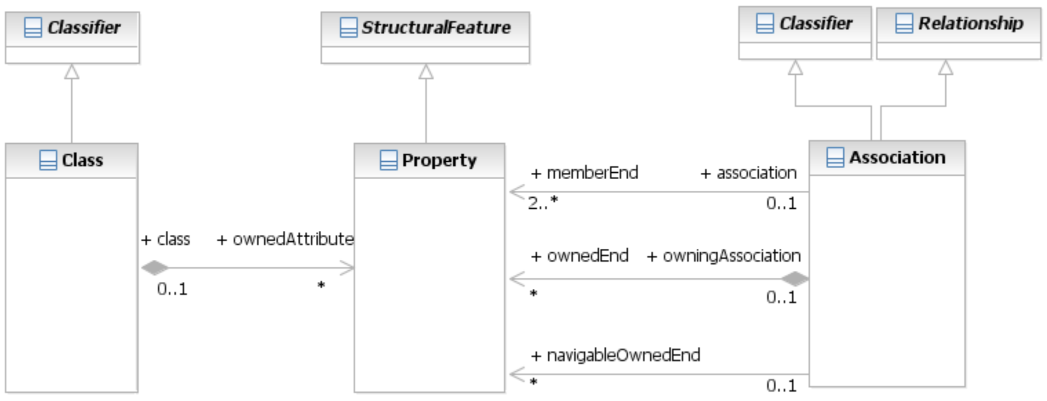

Figure 1: Excerpt of the *Classes Diagram of the Kernel Package* [12, Fig. 7.12]

> An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end. [12, p. 40]

If multiple links between the same objects may occur or links are to be ordered, links carry information in addition to their values.

Associations are not directed and therefore the relation of two instances being associated is symmetric: if an instance i_1 is linked with an instance i_2, then i_2 is also linked to i_1.

The UML metamodel defines an association as a specialization of Classifier (see Fig. 1) which is a classification of instances. Hence, instances of associations can be created. The specification claims that links are firstly instances of the association and secondly tuples with values that are instances of the associated types. This definition is problematic because it suggests the presence of association instances for every link as discrete objects. On the other hand, the metamodel defines an association as a compound of the two metaclasses Association and Property. If a property is owned by a class that is participating in an association, then the property becomes a structural feature of the class and its value is the instance associated with the opposite end. This situation is similar to a property with the type of another classifier. If both ends of a binary association are owned by the associated classes, these properties become structural features of the associated classifiers and the values are not represented by a tuple as an object of its own right but by references. A matter of particular interest is that attributes for the owned ends may be explicitly modeled within the owning classes but do exist even if not explicitly modeled [12, p.46]. Fig. 2 shows the two different notations for the same model.

But the situation is different if a property is owned by an association. In this case, such a property is not a feature of the class that is participating in the association but of the association itself. According to this, the properties are structural features of the association and instances can serve as tuples. Association ends of associations with more than two ends must be owned by the association [12, p. 40].

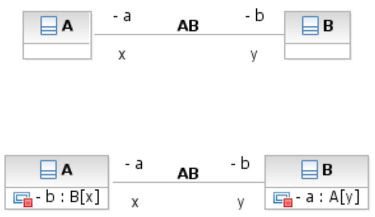

Figure 2: Two different notations for a binary association with both ends owned by the associated classes [12, Fig. 7.24]

Besides the concept of ownership that defines a property's featuring classifier, the visibility of a property can be used to show the end's visibility as an attribute of the featuring classifier. This allows the owning classifier to grant access to an association to other classifiers or not.

Another characteristic of association ends is navigability.

> Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. The precise mechanism by which such access is achieved is implementation specific. [12, p. 41]

3. EXISTING APPROACHES

Implementing relationships and UML associations is a topic often discussed. In general, there are three approaches to bridge the gap between modeling languages and programming languages: implementing relationships by means of code patterns, libraries or extending languages.

3.1 Code Patterns

Two main alternatives of code patterns are known for the implementation of associations:

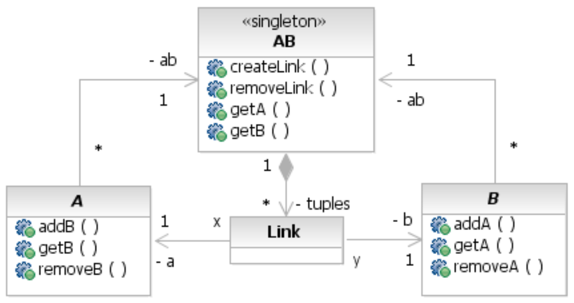

Figure 3: The relationship object pattern.

- The attribute pattern: binary associations are implemented as a pair of pointers.
- The relationship object pattern: an association is implemented as a class of its own right.

Noble [11] discusses both patterns and proposes using the attribute pattern with mutual update methods to ensure a synchronization of pointers. This requires all ends to be visible to all associated classes. However, if an end's visibility is private, the attribute (pointer) representing that end must be private, too. Thus a mutual update is not possible.

The code generation of Fujaba [5] is based on the attribute pattern and considers navigability as well as visibility. Consequently non-compilable code is generated for associations with navigable ends of private visibility.

The relationship object pattern shown in Fig. 3 supports the implementation of all combinations of navigability and visibility. Although Génova et. al. [6] use this pattern (termed *"reified" tuples* in their work) for implementing bidirectional associations, this approach is not used to resolve the conflict between navigability and visibility.

In our previous work [7] we point out why the relationship object pattern is to be used for association implementation in general. The main arguments are, that this pattern affords a straight forward implementation of n-ary associations (with $n > 2$), of association classes and association specialization. Furthermore it supports a free combination of navigability, visibility and ownership of association ends.

3.2 Libraries

Using code patterns can be facilitated by providing libraries that implement these patterns. A library of that kind is introduced by Nelson et. al [10]. It may be a powerful way of introducing relationships to OO languages. However, we focus on an automated generation of code from models. We assume that a free combination of navigability and visibility cannot be achieved by using libraries because the implementation necessarily needs code contained in the associated classes as described in Sect. 5.

A benefit of libraries is that association implementation code is hidden from the programmer. When generating code, the complex implementation of relationships can be hidden in superclasses as well. Furthermore, implementations are adjusted to the actual needs of the concrete model for which code is generated. A library must cover all possible aspects of associations regardless of whether these aspects are actually needed. This may lead to a situation where not all delivered source code is being used.

We consider the substantial advantage of our approach in contrast to libraries to be the support of navigability and visibility without limitations. Libraries are valuable as well, but lack the possibility of freely combining navigability and visibility. We consider that kind of combinability useful and applicable in system modeling.

3.3 Language Extensions

Having a look at the whole process of compiling a model to executable code, the programming language is just one level of abstraction that is passed on the way from the high level of the model down to the very low level of the machine's code. From a strict MDD's point of view, the level of abstraction of the programming language is not relevant. Although we do not reject the idea of a first class construct for associations in programming languages, we consider it not to be necessary for the implementation of UML models.

4. TRANSFORMING MODELS TO LOW LEVEL CONCEPTS

If libraries and language extensions are not used, models must be mapped to the low level concepts of a programming language. This means in particular that associations must be implemented by references (pointers) and objects. In [7] we presented a pattern where tuples represent the association instances as objects. The instances of classes that are to be related must reference a tuple that is holding a reference to the instance of the other side of the association. However, using this pattern, it is possible that an instance discontinues referencing a tuple that the associated instance is still pointing at. Fixing this problem, we ended with the pattern of Génova et. al. [6] shown in Fig. 3.

Note that all associations contained in this model are navigable in only one direction and hence can be implemented by only one attribute in the class connected to the non-navigable end. This is possible, because the non-navigable ends are either not constrained with an upper or lower bound (associations from A to AB and B to AB) or the upper bound cannot be exceeded (association from AB to Link, because only one instance of AB exists). A violation of the upper bounds of the association ends on the Link-side of the associations to A and B can be avoided by implementing the `createLink` method in AB with a check for the number of existing Link-instances. If the creation of a new link would violate the upper bound of the multiplicities x or y, an exception must be thrown as well as if the lower bound of x or y would be violated after removing a link.

Transforming the model shown in Fig. 2 to the model shown in Fig. 3 can be performed automatically by inserting the classes AB and Link. Stereotypes can be used declaring those classes being inserted for implementation purposes.

A problem not yet addressed is the restriction of visibility of association ends to private. Génova et. al. [6] state that the pattern shown in Fig. 3 does not solve the problem, since the class AB cannot provide access to methods for classes A and B excluding all others. In general, this statement is right, however, we developed a mechanism to solve that problem. We term this mechanism *Adornment with a Handle* or briefly a *Handle*.

```
public class AB {

  /* singleton pattern */
  private static final AB fInstance = new AB();
  private AB(){}

  /* providing handles */
  public static void getHandle(Class<?> End){
    if (End == A.class)
      A.takeHandle(fInstance);
    if (End == B.class)
      B.takeHandle(fInstance);
  }

  /* methods for managing the association */
  private List<Link> tuples =
    new LinkedList<Link>();
  public void createLink(A a, B b){
    if (a != null && b != null)
      tuples.add(new Link(a, b));
  }
  public void destroyLink(A a, B b){
    for(Link l : tuples){
      if(l.a == a)
        if (l.b == b){
          tuples.remove(l);
          break;
        }
    }
  }
  public List<B> getLinks(A a){
    List<B> result = new LinkedList<B>();
    for(Link l : tuples){
      if (l.a == a)
        result.add(l.b);
    }
    return result;
  }

  /* implementation of links */
  private class Link{
    private A a;
    private B b;

    public Link(A a, B b){
      this.a = a;
      this.b = b;
    }
  }
}
```

Figure 4: Implementation of class AB of Fig. 3 with a global handle

```
public class A {
  /* association implementation code */
  static { AB.getHandle(A.class); }
  private static AB fAB;
  public static void takeHandle(AB assoc){
    fAB = assoc;
  }

  /* methods for owned end */
  public void addB(B b){
    fAB.createLink(this, b);
  }
  public void removeB(B b){
    fAB.destroyLink(this, b);
  }
  public List<B> getB(){
    return fAB.getLinks(this);
  }
}
```

Figure 5: Implementation of class A of Fig. 3 with a global handle. Class B is implemented analogously.

```
public class AB{
  public static final AB fInstance = new AB();
  private AB(){};
  private Object handle = new Object();

  public void getHandle(){
    A.takeHandle(handle);
  }
  public List<B> getLinks(A a, Object handle){
        if(this.handle != handle) return null;
    // original implementation here
  }
  ...
}
public class A {
  static { AB.fInstance.getHandle(); }
    private static Object handle;
  public static void takeHandle(Object h){
    handle = h;
  }
  public List<B> getB(){
    fAB.getLinks(this, handle);
  }
  ...
}
```

Figure 6: Implementation of classes AB and A of Fig. 3 with a method handle.

5. HANDLES

The idea of *Handles* is that an instance having a valid handle can use it for the invocation of methods that are hidden for classes not having a valid handle. We term methods that require a handle a *partial private* method.

We consider three kinds of handles:

- method handles
- global handles
- distinguishable handles

A *method handle* allows for methods requiring a handle as well as methods not requiring a handle to be contained in the same class.

A *global handle* may be used, when all methods of a class are partial private.

Distinguishable handles enable partial private methods to have different effects for callers with different handles. Instances can be grouped by providing the same handle to them.

5.1 Implementing Global Handles

If all methods of a class are private or partial private, it is possible to use the class itself as the handle. Not the handle is to be checked when invoking a method but the methods are invoked on the handle. Thus, only instances having a handle can invoke partial private methods.

The implementation of a global handle is similar to the visitor pattern. Fig. 4 shows the implementation of the visitor pattern by the method `getHandle` that passes the singleton instance of the association to the callers object class. The method `getHandle` is called when a class participating the association is loaded as shown in Fig. 5. Thus getting a reference is only done once per associated class when creating the first instance of it. Afterwards all instances share the same handle.

5.2 Implementing Method Handles

A method handle is implemented as a parameter that is to be passed when invoking a partial private method. The implementation of the invoked method checks the handle for validity and executes if the handle is valid.

Its implementation for getting the handle is that of a global handle. However, a dedicated handle object is passed to the caller and checked to be equal to the local handle held by the association implementation when partial private methods are invoked. The parts of the code that need to be adapted are listed in Fig. 6.

5.3 Implementing Distinguishable Handles

As in our work distinguishable handles are not needed their implementation is only roughly described. Distinguishable handles are implemented basically like method handles. The difference to method handles is that the class containing partial private methods holds more than one single handle. When a partial private method is invoked the instance of the handle can be compared to the handles held by the class. Depending on which handle instance is passed, not only authorization of method invocation can be checked but the identity of the handle may have effects on the execution of the invoked method.

5.4 Hiding Details from the Programmer

Note that all classes of Fig. 3 are generated automatically and are overwritten when code is generated from the model. The code of a programmer is located in subclasses of A and B. The handle is defined as a `private static` field, so subclasses cannot access it. The implementation of handles, how to get a handle and how to use it is hidden from the programmer.

5.5 Using Handles for Accessing Associations

If both ends of a binary association are owned by the associated classes, methods for managing the association are not to be accessed by other classes than the owning ones. As the association itself has no features accessible to other classes, a global handle can be used as shown in Fig. 4.

If one end is owned by the association and the other one is owned by a class, methods for the end owned by the association are accessible in the same way to all classes according to the end's visibility whereas methods for accessing the other end must be restricted to the owner of that end. In this case, `fInstance` of class AB is public so that other classes may access it and its visible features. The methods for managing the other end are protected by a method handle (see Fig. 6).

For associations of a higher arity, all ends must be owned by the association and a handle is not required because classes participating the association are not privileged.

Note that links are always consistent because link objects are only created if all association ends are supplied with an instance (see Fig. 4). Note that before adding new links, upper bounds must be checked. Before removing links, lower bounds must be checked. Both checks can be implemented by the class AB but are omitted in Fig. 4 for brevity.

5.6 Summarizing the Handle Pattern

The handle pattern facilitates giving access to an association to only those classes participating in it excluding all other classes. Using the visitor pattern, it is guaranteed that only classes participating in the association get a handle for invoking partial private methods. Holding the handle in a `private static final` field guarantees that all instances of the participating classes have access to these methods. Costs for obtaining the handle are minimized by sharing the same handle.

While visibilities are a static concept defining the accessibility of members of a class to other classes, handles are a runtime concept. Errors like accessing members that are not visible in the context where they are tried to access are found at compile-time. The validity of a handle can only be checked at runtime. However, by using handles that do not expire we eliminate one source of errors. By generating all code needed for obtaining handles we guarantee that handles are valid at any time. Provided that the generated code contains no errors, runtime errors will not occur.

6. FACING THE COMPLEX SEMANTICS OF UML ASSOCIATIONS

In Sect. 2, we focus on 4 characteristics of associations: symmetry, ownership, navigability and visibility.

In Sect. 5, we present a pattern for partial private visibility that is designed to grant access to methods managing an association to some classes excluding others. We can address the UML semantics of associations as follows.

6.1 Symmetry of Associations

The symmetry of an association is guaranteed because of the structure of link objects. A link provides a value for each participating object. By that, values are grouped with each group representing one instance of the association. As links are created or removed but never changed, it is assured that a link always consists of a valid group of values.

6.2 Ownership of Association Ends

The ownership of an association end determines the featuring classifier of which this end becomes an attribute. For ends owned by the associated classes, this leads to a separation of the values of a tuple which are independently stored within the instances of the classes.

Following the idea of encapsulation, we provide methods for managing the associations within the featuring classifier of the association ends. For ends owned by a participating class, method calls are delegated to the implementation class of the association. Apart from being the implementation of an association such a class serves as a part of the internal structure of the associated classifier. The implementation class may store the values of tuples itself or use a separate link class instead.

6.3 Navigability of an Association End

Navigability is achieved by allowing classifiers to access the values of a tuple representing a navigable end. A class participating in an association has efficient access to all classifiers connected to navigable ends of the association. Non-navigable ends are not guaranteed to be navigable, but might be so. In the UML it is not possible to prohibit access to instances participating in associations. We prefer to reject access to tuple values for non-navigable ends but allow associated classifiers to create or destroy links. These operations do not need to navigate to opposite ends. The definition of navigability however does not clearly state whether a classifier may access an association in order to create or destroy a link or not if the opposite end is not navigable.

According to this, for non-navigable ends, only add and remove methods are generated whereas get methods are omitted.

6.4 Visibility of Association Ends

With regard to the visibility, we consider three situations:

- all ends are owned by the associated classifiers
- all ends are owned by the association
- ownerships of association ends are mixed

In the first case, the visibility of methods managing the association corresponds to the visibility of the ends.

In the second case, all methods for managing the association are defined in the scope of the implementation class for the association with no need for privileging participating classes to access these methods. Due to this, handles are inherently dispensable for n-ary associations as those must own all ends [12].

The last case is the most complex one. For the end owned by the association, methods for managing the association are defined within the scope of the implementation class of the association. For the other end, those methods are defined in the scope of the class owning this end. Calls of that methods are delegated to partial private methods of the implementation class of the association (see Fig. 6). Note that we must not drop the restriction of partial private visibility even if this end's visibility is public since the attribute representing that end is a feature of the class rather than the association. This situation enforces the use of a method handle for methods managing the end not owned by the association.

6.5 N-ary Associations

The implementation of n-ary associations requires the representation of links by n-tuples. This requires that the class Link as shown in Fig. 3 is associated to $n-2$ more classifiers participating the association. Methods for adding and removing links must be supplied with parameters for each end. The `get` methods must be adapted to return an $m-z$ tuple with $m = |navigable\ ends|$ and $z = 1$ if the instance for which associated objects are queried is connected to a navigable end, $z = 0$ else.

7. CODE SEPARATION

A problem of code generation is how to ensure that modifications of the code and implementations for generated method stubs are not affected when the code is re-generated. For the purpose of separating generated code and programmer's implementations, we suggest that generated code is located in abstract superclasses as shown in Fig. 7.

A problem within this approach is, that methods generated for private association ends must be declared private but must also be accessible to a direct subclass where the programmer's code is located. To achieve this, methods for private ends are generated with protected visibility. This makes the method available in the class where the programmer adds his code. As long as the model does not contain a subclass, the generated subclass is the only specialization. Thus, the methods are not visible within a wider scope than intended. However, a class owning an end that is of private visibility may have subclasses in the model as shown in Fig. 8.

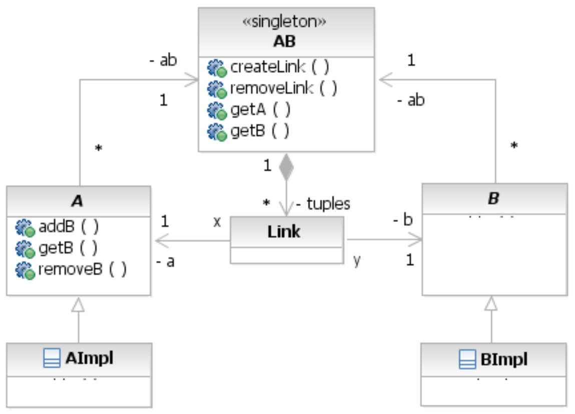

Figure 7: The relationship object pattern with separate classes for user code.

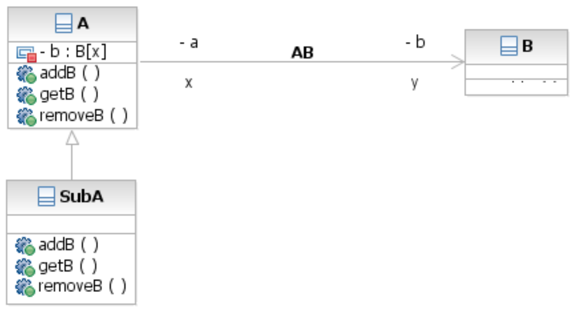

Figure 8: A specialization of a class participating in an association with private ends.

The class SubA of Fig. 8 must not have access to the methods implementing the private association end b even though the class AImpl must have. The visibility of methods cannot be narrowed in subclasses in most OO languages. A way of breaking the chain of accessibility to those methods in subclasses is to generate the methods with the modifier `final` and an empty body in the generated class SubA. The methods can be invoked in subclasses but cannot have any effect.

Fig. 9 shows the model that is derived from the model of Fig. 8. Besides the classes for implementing the association, implementation classes (AImpl and BImpl) that contain user code are generated and the subclass SubA of class A contains all private methods of A as final methods with an empty body. A drawback of this approach is, that all names associated with private features of the class A must not be used in subclasses of A. However, we consider this to be a less important aspect as equality of names suggest equality of meaning that is not held for a member that is named equal to a feature of a more general classifier without being coupled to it in any way.

Inserting abstract classes for code separation purposes can be automated like the transformation described in Sect. 4. The resulting model is the input for our code generator.

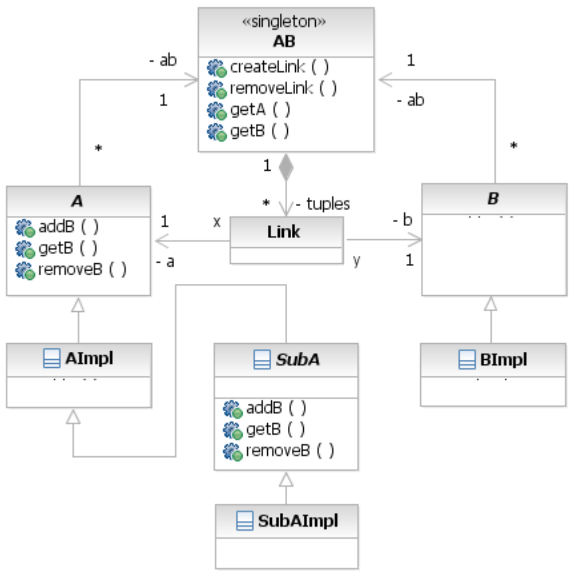

Figure 9: The relationship object pattern with a specialization of a participating class.

8. DISCUSSION & RELATED WORK

The proposed pattern is complex but in contrast to all other patterns we are aware of, it is the only one supporting navigability and visibility without restrictions. Nvertheless, its application is not always necessary and its implementation could be improved by using generic types.

However, related approaches have restrictions that can be overcome by applying the proposed pattern. We will discuss some of these related approaches in the following.

For the understanding of the semantics of UML associations, Diskin et al. [4] [3] and Milićev [9] beside other contributions are valuable. Unfortunately, visibilities of association ends and resulting problems are not discussed there.

A two-way relationship in general cannot be implemented by references in each class pointing at the opposite side as proposed by Noble [11], Diskin et al. [4], Akehurst et al. [1] and others. The required mechanism of mutual updates of the references is impossible if the references are to be private.

Génova et al. take visibilities of association ends into account but state that

> the approach based on ”reified tuples” [1] does not solve the problem either, since it involves auxiliary classes that cannot provide ”private” access to the main classes, excluding all other classes. [6]

We picked up the idea of “reified tuples” and augmented it by the proposed handle pattern. Thus, we showed that there exists a solution to the above quoted problem. However, solving it certainly requires some extra lines of code.

The libraries presented by Nelson et al. [10] and Østerbye [13] also do not support visibility of association ends as defined by the UML.

[1]According to Noble [11], we termed the same approach Relationship Object Pattern (see Fig. 3)

We assume that the examples

enrol.add(alice, programming); [10]

and

enrol.Assoc.add(alice, programming); [13]

show, that managing an association is always done by the association itself. We would like to restrict managing the association to the owners of association ends.

The language extension of Bierman and Wren [2] is based on RelJ. The given grammar of the language lacks a visibility modifier. Consequently, this approach cannot cover the visibility of association ends.

The Eiffel programming language supports a fine grained definition of visibilities. The accessibility of each members of a class can be defined separately providing access to any desired set of classes. An extension of OO languages making this feature available would afford much easier implementations of associations. The problem of mutual update methods with private visibility could be solved by making these methods visible to only those classes participating in the concerned association. In combination with code generators, we consider an Eiffel-like exporting mechanism for class members a very promising concept.

The aspect-oriented approach of Pearce and Noble [14] achieves a very clear distinction of code needed for the class implementation and the code of association implementation. However, aspect-oriented languages usually weave code of aspects into normal classes. It would require a detailed study of the produced byte code or benchmark testings to determine whether the produced code is better or not.

The contributions referred to in this section are related to our approach, however, except for Génova et al. [6], visibility is not discussed in favour of other aspects of relations not necessarily related to the UML. A more detailed discussion is hardly feasible because of too distinct ambitions of our and related approaches.

However, this paper is deeply related to our own prior work [7] where the necessity of representing links as objects is motivated by a far more detailed study of the UML semantics of associations. The implementation has been enhanced significantly by introducing handles in order to omit call back mechanisms. This reduces the overhead dramatically. In our current implementation, adding a link by invoking an add method triggers the following steps:

1 the method invocation must be resolved to the supertype where the generated method is located

2 a method of the object representing the association is called

3 multiplicity bounds are checked

4 the link is stored by either

   $a)_1$ storing a reference for one direction of the association

   $a)_n$ storing a reference for the other directions of the association
   or by

   $b)_1$ creating a link object with a reference on every object that is to be linked

   $b)_2$ storing the link object in a list

The overhead by using call back mechanisms is about twice as much.

Using subclassing for separating generated code from the programmer's code entails extra costs for resolving superclass methods (see step 1.). If these costs are intolerable, a weaving mechanism for combining code fragments from the generator and the programmer is imaginable. However, it affects the reusability of the compiled classes which contains parts of the association implementation. Using subclassing supports reusability because the generated (general) class containing the association implementation can be replaced if a class is to be reused in another context.

9. CONCLUSION AND FUTURE WORK

We presented a pattern, that can serve as an implementation for associations keeping the semantics of navigability, visibility, ownership and symmetry of UML associations. The presented pattern decouples navigability and visibility and facilitates code generation for associations, that are not navigable in both ways. The sense of this kind of associations is often neglected, however we consider it as useful in particular in combination with association classes. Non-navigable ends must be owned by the association and become a feature of it. The non-navigable ends are accessible through the association or the association class itself. Being not navigable does not mean being not accessible at all.

As a part of our future work a semantic foundation of class diagrams and a formal mapping to Java will be developed. This will show in detail, which concepts of programming languages are to be used for implementing models preserving the input model's semantics.

Another question that should be discussed is to what extent the current specification of associations in the UML could be revised.

References

[1] David Akehurst, Gareth Howells, and Klaus M. Maier. Implementing associations: Uml 2.0 to java 5. *Software and Systems Modeling*, 6(1):3–35, March 2007.

[2] Gavin Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *ECOOP 2005 - Object-Oriented Programming*, pages 262–286, 2005.

[3] Zinovy Diskin and Jürgen Dingel. Mappings, maps and tables: Towards formal semantics for associations in uml2. In *Model Driven Engineering Languages and Systems, MODELS06*, volume 4199, pages 230–244. Springer, 2006.

[4] Zinovy Diskin, Steve Easterbrook, and Juergen Dingel. Engineering associations: From models to code and back through semantics. In *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008*, pages 336–355, 2008.

[5] Fujaba-AS. Fujaba Associations Specification, 2005. http://www.se.eecs.uni-kassel.de /∼fujabawiki/index.php/ Fujaba_Associations_Specification.

[6] Gonzalo Génova, Juan Llorens, and Carlos Ruiz del Castillo. Mapping UML Associations into Java Code. In *Journal of Object Technology, vol.2, no. 5, pp. 135-162*, 2003.

[7] Dominik Gessenharter. Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, pages 813–827, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] MDA. Object Management Group, MDA Guide 1.0.1, Document 03-06-01, 2003.

[9] Dragan Milićev. On the semantics of associations and association ends in uml. *IEEE Trans. Softw. Eng.*, 33 (4):238–251, 2007.

[10] Stephen Nelson, James Noble, and David J. Pearce. Implementing first-class relationships in java. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL)*, 2008.

[11] James Noble. Basic relationship patterns. In *In EuroPLOP Proceedings*. Addison-Wesley, 1997.

[12] Object Management Group. UML 2.1.2 Superstructure Specification, November 2007. Document formal/2007-11-02.

[13] Kasper Østerbye. Design of a class library for association relationships. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 67–75, New York, NY, USA, 2007. ACM.

[14] David J. Pearce and James Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2006. ACM.

[15] Bran Selic. Model-driven development: its essence and opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 313–319, 2006.