JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 5, September-October 2003

Mapping UML Associations into Java Code

Gonzalo Génova, Carlos Ruiz del Castillo and Juan Llorens,
Computer Science Department, Carlos III University of Madrid, Spain

Abstract

Object-oriented programming languages do not contain syntax or semantics to express associations directly. Therefore, UML associations have to be implemented by an adequate combination of classes, attributes and methods. This paper presents some principles for the implementation of UML binary associations in Java, paying special attention to multiplicity, navigability and visibility. Our analysis has encountered some paradoxes in the specification of visibility for bidirectional associations. These principles have been used to write a series of code patterns that we use in combination with a tool that generates code for associations, which are read from a model stored in XMI format.

1 INTRODUCTION

One of the key building blocks in the Unified Modeling Language [UML] is the concept of association. An "association" in UML is defined as a kind of relationship between classes[1], which represents the semantic relationship between two or more classes that involves connections (links) among their instances [UML, p. 2-20][2].

As it has been denounced long ago [Rumbaugh 87], object-oriented programming languages express classification and generalization well, but do not contain syntax or semantics to express associations directly. Therefore, associations have to be implemented by an adequate combination of classes, attributes and methods [Rumbaugh 96a, Noble 96, Noble 97, Ambler 01]. The simplest idea is to provide an attribute to store the links of the association, and accessor and mutator methods to manipulate the links. Other approaches emphasize the use of Java interfaces to implement associations with some practical advantages [Harrison 00].

[1] Actually classifiers. `Classifier` is a superclass of `Class` in the UML metamodel.
[2] The current submission of communityUML to the OMG for the development of UML 2.0 [cUML] proposes a change in terminology: "association" instead of "link" and "association type" instead of "association". We support this change, but in this paper we are going to follow the current official terminology in UML.

Cite this article as follows: Gonzalo Génova, Carlos Ruiz del Castillo and Juan Llorens: "Mapping UML Associations into Java Code", in *Journal of Object Technology*, vol. 2, no. 5, September-October 2003, pp. 135-162. http://www.jot.fm/issues/issue_2003_09/article4

CASE tools often provide some kind of code generation starting from design models[3], but limited to skeletal code involving only generalizations and classes, with attribute and method signatures, but no associations at all[4]. The programmer has to manually write the code to manage the associations in a controlled way, so that all constraints and invariants are kept for correctness of the implementation. This is usually a repetitive task that could be automated to a certain extent. Besides, the number of things that the programmer should bear in mind when writing the code for the associations is so large, that he or she continuously risks forgetting some vital detail. This is specially true when dealing with multiple (with multiplicity higher than 1) or bidirectional (two-way navigable) associations. Moreover, the final written code is frequently scattered over the code of the participating classes, making it more difficult to maintain.

The aim of this work is two fold. First, write a series of code patterns that will help programmers in mapping UML associations into a target object oriented programming language. In this work, the language has been chosen to be Java, although the principles we have followed may be applied to other close languages like C++ or the .NET framework. Second aim, construct a tool that generates code for associations using these patterns, the associations being read from a model stored in XMI format[5]. A third aim will be to enable reverse engineering, that is, obtaining the associations between classes by analyzing the code that implements them. Our tool does not presently accomplish this task, although it is a very simple and straightforward procedure if the code has been written with our patterns. A complete description of the patterns and the tool is outside the scope of this paper, but can be found elsewhere [Ruiz 02].

Associations in UML can have a great variety of features. The present work is limited to the analysis and implementation of multiplicity, navigability and visibility in binary associations. It excludes, therefore, more complex kinds of associations such as reflexive associations, whole/part associations (aggregations and compositions), qualified associations, association-classes, and n-ary associations. It excludes, too, features such as ordering, changeability, etc.

The three following sections of this article are devoted to studying the features of multiplicity, navigability and visibility of associations, with a detailed analysis of the possible problems and proposed solutions. Then, Section 5 contains the description of a uniform interface for all kinds of associations from the point of view of the participating classes, such as it is implemented by our patterns and tool. Finally, Section 6 describes briefly how our tool works.

[3] We distinguish here between analysis and design models. An analysis model is an *abstraction of the problem* (the real world as it is before the proposed system is built) whereas a design model is an *abstraction of the solution* (the proposed system's internal construction) [Kaindl 99], therefore code generation has sense only for a design model.
[4] Some tools are an exception to this rule [Fujaba, Rhapsody].
[5] XML Metadata Interchange [XMI], an XML-based format designed to store and interchange UML models between different tools.

2 THE PROBLEM OF MULTIPLICITY

The multiplicity of a binary association, placed on an association end (the target end), specifies the number of target instances that may be associated with a single source instance across the given association, in other words, how many objects of one class (the target class) may be associated with a given single object from the other class (the source class) [RM, p. 348; UML, p. 2-23][6].

The classical example in Figure 1 illustrates binary multiplicity. Each instance of `Person` may work for none or one instance of `Company` (0..1), while each company may be linked to one or more persons (1..*). For those readers less familiarized with UML notation, the symbol (*) stands for "many" (unbounded number), and the ranges (1..1) and (0..*) may be abbreviated respectively as (1) and (*).

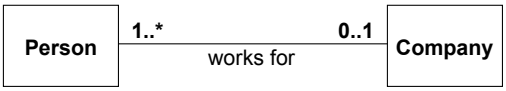


Figure 1. A classical example of binary association with the expression of multiplicities

The potential multiplicities in UML extend to any subset of nonnegative integers [RM, p. 346], not only a single interval as (2..*), or a comma-separated list of integer intervals as (1..3, 7..10, 15, 19..*): specifications of multiplicity like {prime numbers} or {squares of positive integers} are also valid, although there is no standard notation for them. Nevertheless, in UML as in other modeling techniques, the most usual multiplicities are (0..1), (1..1), (0..*) and (1..*). We are going to restrict our analysis to multiplicities that can be expressed as a single integer interval in the form of *(min..max)* notation.

The multiplicity constraint is a kind of *invariant*, that is, a condition that must be satisfied by the system. A possible practice when programming is: do not check always the invariant, but only at the request of the programmer, after completing a set of operations that are supposed to leave system in a valid state (a *transaction*). This practice is more efficient in run-time, and gives the programmer more freedom and responsibility in writing the code, with the corresponding risk that he or she forgets putting the necessary checks and carelessly leaves the system in a wrong state. On the other side, we think that checking multiplicity constraints is not very time consuming (inefficient), especially when compared with the time required to manage collections or synchronize bidirectional associations (see Section 3). Therefore, we think that it is worth doing as much as we can for the programmer, so that our first target will be to analyze the possibility of performing automatic checks for multiplicity constraints.

[6] Other notations invert the placement of multiplicity values, following the near-end convention instead of the far-end convention, which is the one used in UML. It has been well established that the semantics of both conventions are equivalent for binary associations, but differ substantially when they are applied to associations of higher degree [Song 95, McAllister 98, Castellani 00, Génova 02, Génova 03b].

Optional and mandatory associations

The value of minimum multiplicity can be any positive integer, although the most common values are 0 or 1. When the value is 0 we say the association is *optional* for the class on the opposite end (class `Person` in Figure 1), when the value is 1 or greater we say it is *mandatory* (class `Company`). Optional associations pose no special problems for the implementation, but mandatory associations do. From a conceptual point of view, an object participating in a mandatory association needs to be linked *at any moment* with one object (or more) on the other side of the association, otherwise the system is in a wrong state. In the example given in Figure 1, an instance of `Company` needs always an instance of `Person`. Therefore, in the same moment you create the instance of `Company`, you have to link it to an instance of `Person`.

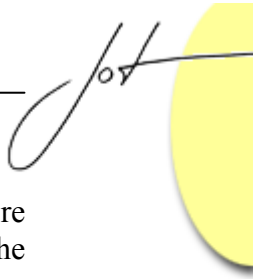
This can happen in three different ways:

- An instance of `Company` is created by an instance of `Person` and linked to its creator.
- An instance of `Company` is created with an instance of `Person` supplied as a parameter.
- An instance of `Company` is created and it issues the creation of an instance of `Person`.

The third case poses additional problems. The creation of a `Person` will probably require additional data, such as name, address, etc., and it does not seem very sensible to supply them in the creation of a `Company`. This problem becomes much worse if `Person` has other mandatory associations, for example one with the `Country` where he or she lives: if this were the case, the creation of a `Company` would require supplying data for creating a `Person`, for creating a `Country`, etc.

The most obvious solution is to allow only the first and second forms of instantiation. But then suppose the association is mandatory in both ends. Which instance is to be created first? We have not a satisfactory choice, since we will put the system in a wrong state until both creations are finished. We could think of an *atomic creation* of both instances, but this is valid only for the simplest case in which only two classes are involved. Should we define atomic creators for two, three, any number of classes? Similar problems arise when dealing with object deletion.

Imagine now that we are not creating or deleting instances, but changing links between instances. If you want to change the instance of `Company` that is linked with a given instance of `Person`, simply delete the link with the old `Company` and add a new link with the new `Company`. This works as far as the old `Company` is linked to other instances of `Person`; you can even delete the link and add no new one, since the association is optional for `Person`. If you had only one `Person` linked to a given `Company`, you should supply a new `Person` to the `Company` before deleting the link with the old `Person`, but this is only the specified behavior (the association is mandatory for `Company`) and you cannot complain about it.



Nevertheless, we find new problems here. If the association with `Company` were mandatory for `Person` too (that is, 1..1 multiplicity instead of the current 0..1), the instance of `Person` could not delete the old link with a `Company` and then add the new one, nor it could do it in the reverse order, "first add then delete", because it would go through a wrong system state. An *atomic change* of links would be valid only for the simplest cases, but not for more complex ones such as the following, rather twisted case (see Figure 2): consider classes `A` and `B`, which are associated with multiplicity 1..1 on both ends, and the corresponding instances `a1`, `a2`, `b1` and `b2`. In the initial state, we have the links `a1-b1` and `a2-b2`. In the final state, we want to have the links `a1-b2` and `a2-b1`. Even if we can change atomically `a1-b1` to `a1-b2` without violating the multiplicity constraints on `a1`, this would leave `b1` without any links and `b2` with two links until the final state is reached. We should have to perform the whole change atomically by means of an *atomic switch* implemented in a single operation.

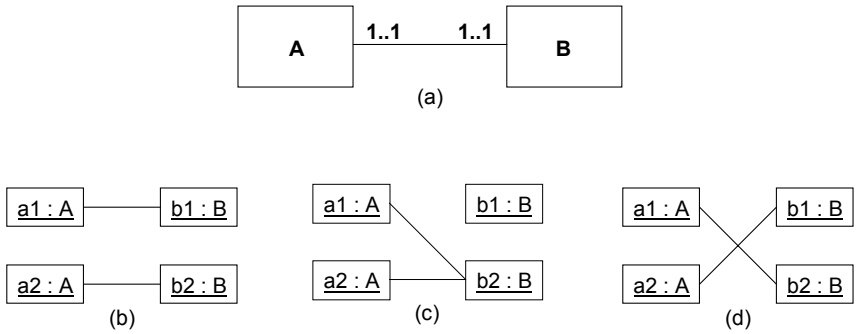


Figure 2. Multiplicity constraints can make very difficult changing links between instances without entering a wrong system state: a) class diagram; b) initial state; c) intermediate wrong state; d) final desired state

Obviously, we cannot define a new operation to avoid any conceivable wrong state involving several instances. In consequence, we think that mandatory associations pose unsolvable problems regarding the creation and deletion of instances and links: we cannot achieve with a few primitive operations that a mandatory association is obeyed *at any time*, and we cannot isolate, inside atomic operations, the times when the constraint is not obeyed. Therefore, we have to relax the implications of mandatory associations for the implementation, as other methods do [Harrison 00]. Our proposal is as follows: *do not check the minimum multiplicity constraint when modifying the links of the association* (mutator methods, or setters), *but only when accessing them* (accessor methods, or getters). The programmer will be responsible for using the primitives in a consistent way so that a valid system state is reached as soon as possible.

For example, you will be allowed to create a `Company` without linking it to any `Person`, and you will be allowed to delete all the links of a `Company` with instances of `Person`; but before accessing, for other purposes, the links of that particular instance of `Company` towards any instances of `Person`, you will have to restore them to a valid state,

otherwise you will get an *invalid multiplicity exception*, which shall be defined in the code that implements the associations according to our proposal.

Single and multiple associations

The value of maximum multiplicity in an association end can be any integer greater or equal than 1, although the most common values are 1 or *. When the value is 1 we say the association is *single* for the class on the opposite end (class `Person` in Figure 1), when the value is 2 or greater we say it is *multiple* (class `Company`). Single associations are easier to implement than multiple associations: to store the only possible instance of a single association we usually employ an attribute having the corresponding target class as type, but to store the many potential links of a multiple association we must use some kind of collection of objects, such as the Java predefined `Vector`, `HashSet`, etc. In the general case we cannot use an array of objects, because it gets a fixed size when it is instantiated. Since collections in Java can have any number of elements, the maximum multiplicity constraint cannot be stated in the declaration of the collection in the Java code, but it must be checked elsewhere during run-time.

We need two kinds of mutators, `add` and `remove`, which will accept as a parameter either single objects or entire collections. Because of the problems with minimum multiplicity explained above, the remover sometimes will leave the source instance in a wrong state; we can't avoid this situation. The adder, instead, leaves us a wider choice. If we try to add some links above the maximum multiplicity constraint, we can choose between rejecting the addition or performing it; in the latter case we violate temporarily the constraint until a call to the remover restores the source instance to a safe state; the wrong state would only be detected by accessor methods, as we settled in the case of minimum multiplicity. However, this is true only for multiple associations implemented with a collection; in single associations implemented by means of an attribute we simply cannot violate the maximum multiplicity constraint: we are forced to reject the addition.

If we choose to reject the addition, instead, besides having an asymmetric behavior between remover and adder, we can find precedence problems when invoking the adder and the remover in succession. Consider class `Game` associated with class `Player` with multiplicity 2..4 (see Figure 3), and suppose an instance `g1` of `Game` is linked to two instances `p1`, `p2` of `Player`. We want to replace these two players by four new different players `q1`, `q2`, `q3`, `q4`. If we issue "first remove then add", we get finally what we want; if we issue "first add then remove", the addition is rejected and the remotion leaves the instance of `Game` in a wrong state.

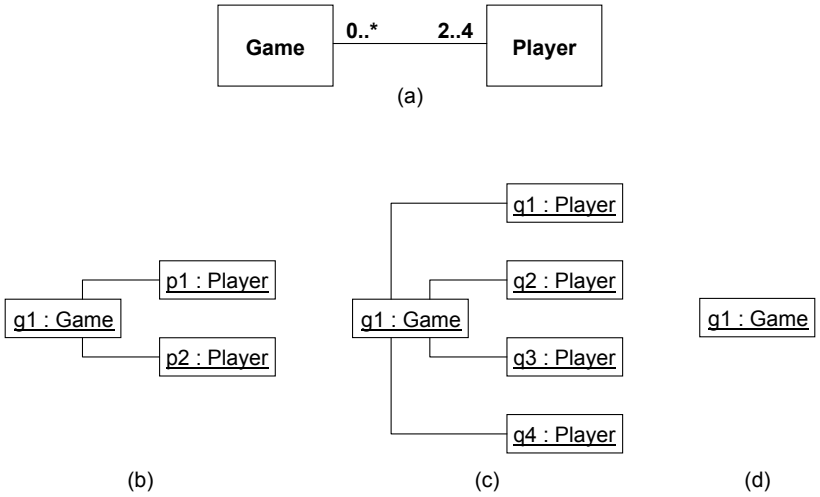


Figure 3. Precedence problems found when invoking the adder and the remover in succession: a) class diagram of `Game-Player` association; b) initial state with players `p1`, `p2`; c) final desired state after removing players `p1`, `p2` and then adding players `q1`, `q2`, `q3`, `q4`; d) final wrong state after unsuccessfully trying to add players `q1`, `q2`, `q3`, `q4` and then removing players `p1`, `p2`

In the end, we have preferred to *reject the addition if it violates the maximum allowed*, and ask the users of mutator methods to use them always in the right order, first remove then add, so that we can get an analogous behavior for single and multiple associations. Therefore, the remover does not check the minimum multiplicity constraint (possibly leaving empty a mandatory association), the adder does check the maximum multiplicity constraint, and the getter raises an exception if either constraint is not fulfilled.

Accessor methods of multiple associations have another peculiarity, when compared with the accessors of single associations: they return a collection of objects, not a single object, therefore the returned type is that of the collection, not that of the target class. In our implementation, the returned type is the Java interface `Collection`, which is implemented by all standard collections. Internally, we use a `HashSet` collection, which ensures that there are no duplicate links in an association, as the UML requires [UML, p. 2-19][7].

Finally, the standard collections in Java are specified to contain instances of the standard class `Object`, which is a superclass of every class in Java. You cannot specialize these collections to store objects pertaining only to a particular class[8]. This means that, if we use a `HashSet` inside `Company` to store the links to instances of `Person`, we must ensure on our own that no one puts a link to an instance of another class such as `Dog` or `Report` (this could happen if a collection of objects is passed as a parameter to the `add` method). Therefore, the mutator methods must perform a run-time

[7] In other places we have given conclusive arguments against the no-duplicates restriction in UML associations [Génova 03b], but here we have respected the current specification of UML.

[8] That is, you cannot specialize them to modify their storage structure, but you can modify their behavior so that they store in effect only the required objects, precisely by means of the run-time type checking method we describe.

type checking by means of explicit *casting*. If the type-check fails, then the link is not set to that object, and a *class cast exception*, which is predefined in Java, is raised.

3 THE PROBLEM OF NAVIGABILITY

The directionality, or navigability, of a binary association, graphically expressed as an open arrow at the end of the association line that connects two classes, specifies the ability of an instance of the source class to access the instances of the target class by means of the association instances (links) that connect them[9]. If the association can be traversed in both directions, then it is *bidirectional* (two-way), otherwise it is *unidirectional* (one-way).

A navigable association end, which is referenced by its rolename, defines a pseudoattribute of the source class, so that the source instance can use the rolename in expressions in the same way as it uses its own attributes [RM, p. 354]. An instance can communicate (by sending messages) with the connected instances of the opposite navigable end, and it can use references to them as arguments or reply values in communications [UML, p. 2-114]. Similarly, if the association end is navigable, the source instance can query and update the links that connect it to the target instances.

The examples in Figure 4 illustrate navigability. The association `Key-Door` is unidirectional, meaning that a `Key` can access the `Door` it can open, but an instance of `Door` does not know the set of instances of `Key` that can open it: the `Door` cannot traverse the connections (links) against the navigability of the association. On the other side, the association `Man-Woman` is bidirectional, meaning that connected instances of these classes know each other.

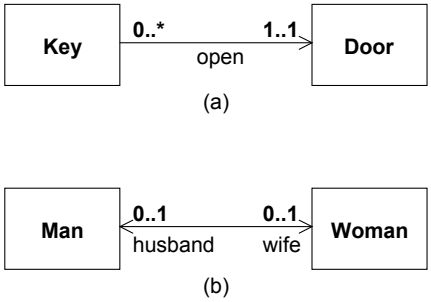


Figure 4. Examples of a) unidirectional and b) bidirectional associations

The arrowheads can be shown or omitted in a bidirectional association [UML, p. 3-73]. Unfortunately, this leads to an ambiguity in the graphical notation, because we cannot

[9] An alternate definition: the possibility for a source object to designate a target object through an association instance (link), in order to manipulate or access it in an interaction with message interchanges. The Standard does not give a clear definition of navigability, as we have shown in previous works where we have tried to clarify this topic [Génova 01, Génova 03a, Génova 03b]. In this paper, we take navigability and directionality as synonyms.

distinguish between bidirectional associations and associations with unspecified navigability. Or, worse, unspecified associations are assumed to be bidirectional without further analysis [Génova 01].

Unidirectional associations

A *single unidirectional association* is very similar to a single valued attribute in the source class, of the type of the target class: an embedded reference, pointer, or whatever you want to call it. The equivalence, however, is not complete. Whereas the *attribute value* is "owned" by the class instance and has no identity, an *external referenced object* has identity and can be shared by instances of other classes that have a reference to the same object [Rumbaugh 96b] (see Figure 5). Anyhow, the equivalence is satisfactory enough to serve as a basis for the implementation of this kind of associations. In fact, in Java there is no difference at all: except for the case of primitive values, attributes in Java are objects with identity, and if they are public you cannot avoid them to be referenced and shared by other objects.

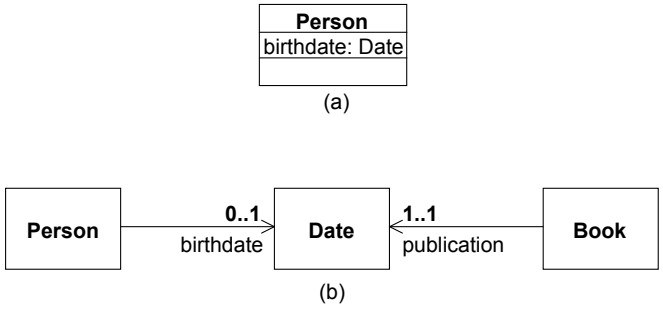


Figure 5. Partial equivalence between a) attribute and b) single unidirectional association

A *multiple unidirectional association* is a bit more complicated, although the implementation can be based on the same principles, since it can be assimilated to a multivalued attribute[10]. To manage the collection of objects on the navigable end, however, we need an additional object of a standard collection class, which is a `HashSet` in our implementation (see Figure 6).

[10] UML allows multiplicity in attributes, thus multivalued attributes [UML, p. 2-50].

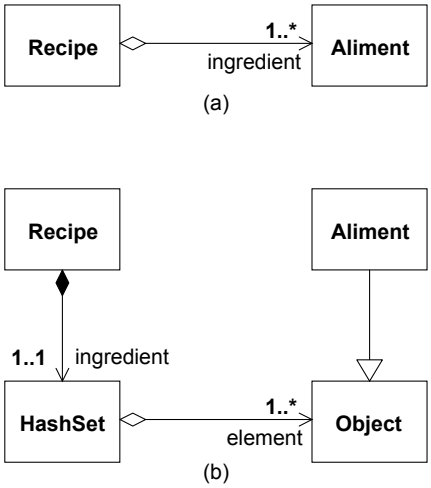
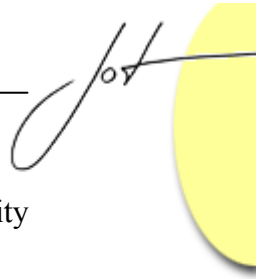


Figure 6. Multiple unidirectional association: a) analysis diagram and b) design diagram. A new object must be inserted to manage the collection of target objects. The standard collections in Java, such as `HashSet`, are defined for the standard class `Object`, which is a superclass of every class; therefore, mutator methods must ensure that the objects contained in the collection parameter are of the appropriate type before adding them to the collection attribute.

Therefore, the type of the attribute used to implement the association inside the source class is not any more the target class itself, but the `HashSet` class or another convenient collection class. The methods to manage the association will have to accomplish some additional tasks. *Mutators* can add or remove not only single objects of the class target, but also entire collections; thus, the type of the parameter will be either the target class of the association or the intermediate collection class. In this case, mutator methods must ensure that the objects contained in the collection parameter are of the appropriate type before adding them to the collection attribute. *Accessors*, as we have already explained (see Section 2), do not return a single object, but a collection of objects, even when the collection is made up of only one element. The returned collection object is not identically the same one that is stored inside the source class, but a clone (a new object with a collection of references to the same target elements), because the original collection object must remain completely encapsulated inside the source object (represented by the composition in Figure 6).

As the diagrams in Figures 5 and 6 show, in our opinion *the multiplicity constraint in a design model can be specified only for a navigable association end*[11]. Indeed, the multiplicity is a constraint that must be evaluated within the context of the class that owns the association end; if that class knows the constraint, then it knows the association end, that is, the end is navigable. You cannot restrict the number of objects connected to a given instance unless this instance has some knowledge of the connected objects, that is, unless you make the association end navigable. Therefore, *the need for a multiplicity constraint other than 0..** (that is, unrestricted) *is an indication that the association end*

[11] This principle does not apply to analysis models, which usually do not deal with navigability [Fowler 97, Stevens 00]. Obviously, code generation only has sense when starting from design models.



must be navigable. In consequence, unidirectional associations with multiplicity constraints on the nonnavigable association end must be rejected in code generation.

Bidirectional associations

The partial equivalence between attributes and unidirectional associations is not any more found among bidirectional associations. Instead, an instance of a bidirectional association is more like a *tuple of elements* [UML, p. 2-19]. Combining the multiplicities in both association ends, we can have three cases: single-single, single-multiple, and multiple-multiple.

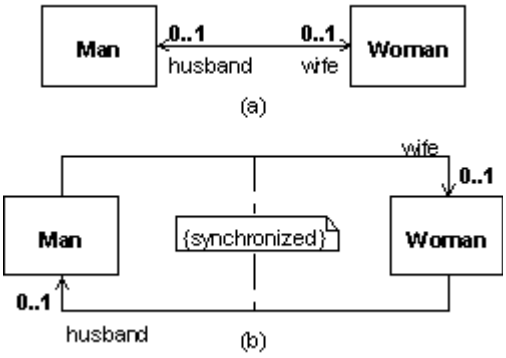


Figure 7. Single-single bidirectional association: a) analysis diagram and b) design diagram. The implementation of the association's mutators must ensure that the husband of the wife of a given man is that man himself, and vice versa

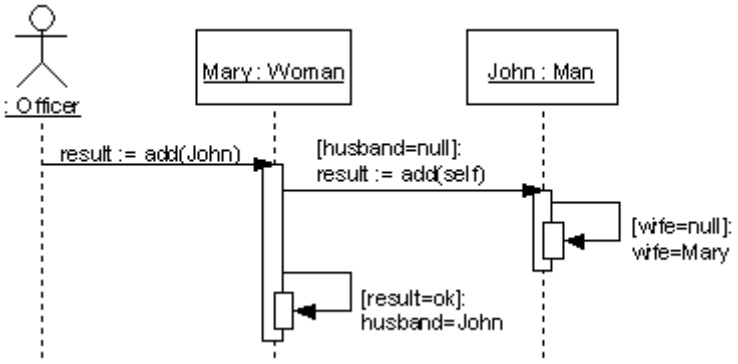


Figure 8. Sequence diagram illustrating the synchronization of a bidirectional association. The update of the attribute `Woman.husband` to "John" (last operation) takes place only after the update of the attribute `Man.wife` to "Mary" has been correctly accomplished. If the woman were already married, then she would not request the man to update the marriage association on his side; if the update on the man's side fails (because he is already married), then the woman does not update her side. To achieve this behavior, the `add` method returns a convenient result that is checked by the client object

An easy way to implement a *single-single* bidirectional association is by means of two synchronized single unidirectional associations (see Figure 7). The synchronization of the

two halves must be preserved by the mutator methods on each side: every time an update is requested on one side, the other side must be informed to perform the corresponding update; the update is accomplished only if both sides agree that they can perform it while keeping maximum multiplicity constraints[12] (see Figure 8).

A *single-multiple* bidirectional association can be implemented in a similar way, combining a single unidirectional association and a multiple unidirectional association. And, finally, a *multiple-multiple* bidirectional association is achieved by means of two multiple unidirectional associations (see Figure 9).

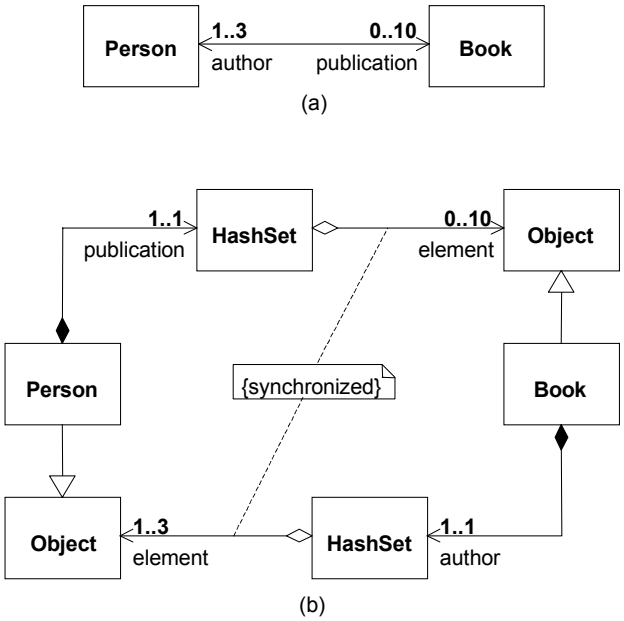
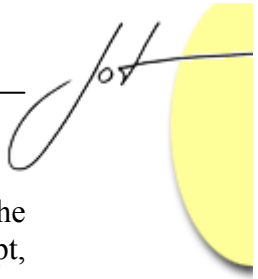


Figure 9. Multiple-multiple bidirectional association: a) analysis diagram and b) design diagram

Synchronization becomes progressively a more and more complex issue when one or both association ends are multiple. Consider the example given in Figure 9. Suppose you want to add an author to a particular `Book` instance; you do this by issuing the `add` method on the `Book` instance, and passing a `Person` instance as a parameter. If the `Book` can have more authors without violating its maximum multiplicity (which is 3), then it requests the author to add the `Book` itself to the collection of publications the `Person` has; this can fail if the maximum multiplicity constraint for the number of publications (in this case, 10) is violated. If the request to the author succeeds, then the `Book` updates its side.

Now, you can try adding a collection of authors to a `Book`, too. As one can expect, the `Book` requests each one of the authors to add the `Book` itself as a publication; if only one of the authors fails to add the `Book`, then the whole operation must be undone, since an update must be atomic: all or none.

[12] We have already justified in Section 2 that mutators must be allowed to violate the minimum multiplicity constraint.



Similar considerations apply to the `remove` mutator, bearing in mind that the `remove` method is performed even if the minimum multiplicity constraint is not kept, therefore it can leave the source instance or any of the affected target instances in an invalid state.

In UML, an association is defined as a "set of tuples" [UML, p. 2-19], meaning that you cannot have twice the same tuple in the collection of links of an association[13]. This is automatically safeguarded if we follow the implementation scheme explained above. Anyhow, it suggests also a different kind of implementation that could have some advantages. Instead of synchronizing two unidirectional associations to get a bidirectional one, we can directly store the collection of bidirectional links (see Figure 10).

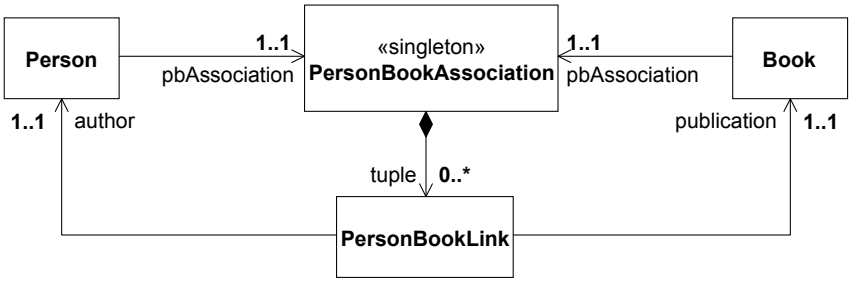


Figure 10. An alternative scheme for implementing bidirectional associations by means of a collection of "reified" tuples

Within this alternative scheme the links are "reified" and become objects on their own [Rumbaugh 87]. To manage the collection of links, or tuples, we need an object, which will be the only instance of a class (in application of the "Singleton" design pattern [Gamma 94]) representing the association itself. The main advantage of this approach is that it avoids the dispersion of the information about the association instances (links), so that updates are effected in only one place, without synchronization problems. It is easily extended to implement association-classes and associations of higher degree (ternary associations, etc.). However, these advantages have a high cost, as we can appreciate by comparing Figures 9 and 10.

First, note that *the original multiplicity constraints are not expressed in this scheme*: the multiplicity of roles `Person.pbAssociation` and `Book.pbAssociation` must be obviously 1..1, since there is only one instance of the object that manages the association considered as a collection of links; besides, a link is the connection of two instances, therefore a link has exactly one "leg" on each side [Génova 02, Génova 03b], that is, multiplicities must be 1..1 on the roles `author` and `publication`; finally, the role `tuple` has multiplicity 0..* regardless of the multiplicity of the original association, even if it was single-single, because it stores all the links that may exist between any instances on each side. In consequence, multiplicity constraints become more difficult to keep, since the control cannot consist simply in "counting links".

[13] The convenience of this constraint, inherited from Entity-Relationship modeling, is disputed by many authors [Genilloud 99, Stevens 02, Génova 03b].

Second disadvantage, *the uniqueness of each tuple, required by the "set of tuples" constraint, is not automatically safeguarded*. Suppose we implement the collection of tuples by means of a `HashSet` of objects, each object storing two references, `author` and `publication`. As each tuple object has its own identity, two different tuples referencing the same two targets would be considered as different objects, therefore the `HashSet` collection would not check the uniqueness of each tuple for us[14].

Considering all these factors, in our implementation we have discarded the "reified tuples" approach in favor of the previous "synchronized cross-references" scheme.

[14] This is not anyway an unsolvable difficulty. For two “equal” tuples to be recognized and their uniqueness to be warranted, you must redefine the `equals` and `hashCode` methods, inherited from `Object` and employed by `HashSet` with this purpose [Eckel 00].

4 THE PROBLEM OF VISIBILITY

So far we have dealt only with the Java implementation of two features of UML associations: multiplicity and navigability (directionality), but we are interested also in the implementation of visibility. According to the Standard [UML, p. 2-23], the visibility of an association end "specifies the visibility of the association end from the viewpoint of the classifier on the other end". The Standard assimilates the visibility of an association end to the visibility of an attribute, and gives the same four possibilities:

- `public` - Other classifiers may navigate the association and use the rolename in expressions, similar to the use of a public attribute.
- `protected` - Descendants of the source classifier may navigate the association and use the rolename in expressions, similar to the use of a protected attribute.
- `private` - Only the source classifier may navigate the association and use the rolename in expressions, similar to the use of a private attribute.
- `package` - Classifiers in the same package (or a nested subpackage, to any level) as the association declaration may navigate the association and use the rolename in expressions[15].

In Java we find the same four kinds of visibility for attributes and methods (not a chance, of course), known as *access control levels* [Gosling 96, Arnold 00], although their semantics is not exactly the same as in UML[16]. Package visibility is the default for unspecified access control, usually known as *friendly*. Since we have implemented UML associations by means of Java attributes and methods, it seems that we should not find special problems with the implementation of visibility[17]; on the contrary, it should be rather easy.

This is true for unidirectional associations: if we declare the Java attributes and methods with the same access control as the UML association end we want to implement, we automatically get the desired behavior. But the story runs differently for bidirectional associations. In principle, it seems sensible to declare private one or both ends of a binary association. We can think of an association with two private ends as a "secret" relationship that is not known outside the participating classes, such as a `Bank-Client` association, for example. Similarly, an association with one public and one private association ends would be only partially known from the outside. But there are problems.

[15] This last kind of visibility, appended in version 1.4 of the Standard, is ambiguously defined, since an association could be declared between classifiers from two different packages. Which package does the association declaration belong to, then? We suggest this wording instead (additions in italics): "Classifiers in the same package (or a nested subpackage, to any level) as the *source classifier* may navigate the association and use the rolename in expressions, *similar to the use of an attribute with package visibility*.

[16] The `protected` access control means in Java the union of `protected` and `package` visibilities in UML, that is, the protected element is visible for descendants as well as for other elements in the same package [Arnold 00, Eckel 00, Gosling 96].

[17] Except that `protected` will have the Java meaning, not the UML meaning. The Standard acknowledges that all forms of nonpublic visibility are language-dependent [UML, p. 3-42].

Consider the bidirectional association `Lecturer-Subject` with public and private visibilities (see Figure 11).

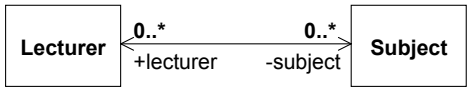


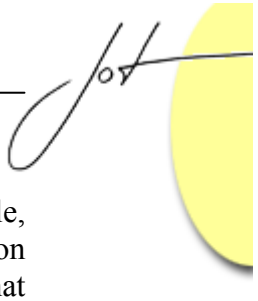
Figure 11. A bidirectional association with public and private association ends

The public association end, `lecturer`, can be used by any other class in the model with visibility to the `Subject` class, that is, the collection of lecturers that teach on a given subject can be queried and updated directly by any class in the model that sees the subject. Instead, the private association end, `subject`, meaning the collection of subjects on which a given lecturer teaches, is known only to the lecturer itself, just as a private attribute. The `Lecturer` class could declare other public methods that internally refer to the `subject` rolename, thus providing *indirect access* to the private association end, but *direct access is restricted to the owner class itself*. This is no more than the idea of declaring something private.

Now, we have got a paradox here about the bidirectionality of the association. The association end with the private rolename `subject` is known only to its owner, that is, the `Lecturer` class. We repeat: only to its owner. That means that the `Subject` class does not know the `subject` association end! The `Subject` class knows that it is associated with the `Lecturer` class, but it does not know that the `Lecturer` class is associated with it in return. *Is this really a bidirectional association?*

In our implementation, based on synchronized cross-references as explained above, this paradox manifests itself in the impossibility to reciprocally update the association ends. Remember that, when an instance of `Subject` tries to add an instance (`newLecturer`) of the `Lecturer` class to its collection of lecturers (`lecturer.add(newLecturer)`), it first has to invoke the `add` method on the reciprocal side (`lecturer.subject.add(self)`)[18]; but now this is impossible due to the private visibility of the `subject` association end. The same happens with the `remove` method. On the contrary, if an instance of `Lecturer` tries to update the association, it can issue the update method on the opposite side, because it is public, and it can update its own private side, thus the whole operation succeeds. At first sight, then, it seemed that the association could be managed via the class that owns the public association end (in this case, the `Subject` class), but this has turned to be false: in fact, only the class that owns the private association end (`Lecturer`) can manage the association, and direct access from outside the two participating classes is impossible. However, as it has been explained above, the `Lecturer` class could declare public methods to provide indirect access to the private association end from the outside.

[18] An object refers to itself in UML by means of the `self` keyword, equivalent to Java's `this`.



Even worse, if both association ends were private, as in the `Bank-Client` example, the association would become inaccessible from both sides[19]. The approach based on "reified tuples" does not solve the problem either, since it involves auxiliary classes that cannot provide "private" access to the main classes, excluding all other classes.

Summing up, a public-private bidirectional association can be managed only from the class that owns the private end, and other classes, including the class on the other end of the association, can have only indirect access if this class provides the adequate public methods. A private-private bidirectional association, on the contrary, cannot be managed at all. Similar considerations can be made for package and protected visibility, which behave in this case respectively like public and private visibility. In consequence, bidirectional associations with visibility other than public or package in both ends must be rejected in code generation. We think this result is not only a bias of our particular implementation, but a real semantic difficulty of the definition of visibility in bidirectional associations. Visibility in UML is not specified for associations but for association ends, and it is assimilated to the visibility of attributes [UML, p. 2-23]. We need in UML a definition of visibility that fits better with the concept of bidirectional association.

[19] A reflexive association (an association between instances of the same class) is an exception to this rule, since private association ends are visible inside the class, that is, for both sides of the association.

5 A UNIFORM INTERFACE

The code required to implement a UML association consists basically of an adequate combination of Java attributes and methods, but this combination depends on the multiplicity, navigability and visibility of the association. With respect to *multiplicity*, to store the only possible instance of a single association we can employ an attribute of the corresponding target class, but to store the many potential links of a multiple association we need some kind of collection of objects: we have chosen a `HashSet` collection, since it ensures that there are no duplicate objects in the collection. With respect to *navigability*, a unidirectional association is implemented only in the source class, whereas a bidirectional association is implemented in both classes, with code that ensures the synchronization of both ends. The *visibility* of the association ends maps directly onto the visibility of the required methods; the attributes, instead, will remain always private, to keep a controlled access through the interface methods.

We have designed a uniform interface for all kinds of associations, that is, an interface that is as independent as possible of the multiplicity, navigability and visibility of the association ends. The interface comprises accessor and mutator methods, as well as other auxiliary methods to learn the state and definition of the association. Our intention is that the client code can use the interface of the association without knowing *a priori*, when possible, what kind of association it is; this will make the client code much more stable with regard to changes in the design (for example, a unidirectional association that becomes bidirectional).

In fact, the implementation of unidirectional and bidirectional associations is different, because only bidirectional associations have to be synchronized, but both kinds present exactly the same interface on each end. On the contrary, single and multiple associations have not only different implementations, but a slightly different interface, because single associations do not manage collections of objects as parameters or return values. We could treat single associations as a particular case of multiple associations and provide no special implementation or interface for them, but we consider that they are used so frequently that the benefits in efficiency are proportionate to the losses in interface uniformity.

In the following paragraphs, "`%Target`" means the name of the target class in the association, which will have to be substituted by its real name when the code is generated for each concrete association in the design model. The source class of a unidirectional association presents an interface to query and update the opposite association end; the target class does not present an interface, because it is not aware of the association. In contrast, the two sides of a bidirectional association present an interface; in this case the terms "source" and "target" become relative to the class that is seeing the opposite association end.

Accessor methods

We have two accessor methods, `test` and `get`, with the following signatures:

- `boolean test(%Target query_link);`
- `boolean test(Collection query_links);`
- `%Target get();`
- `Collection get();`

The `test` method checks whether a given instance of the target class (the `query_link` parameter) is linked with the instance of the source class that receives the method invocation. The second version of this method is defined only when the association end is multiple; in this case the method checks whether *all the instances* contained in the collection parameter are linked to the source instance[20].

The `get` method returns the target instances that are linked with the source instance. The first version is for a single association end and it returns a unique value, the type of which is the `%Target` class, whereas the second version is for a multiple association end, so that it returns a value of type `Collection`. According to the justification given above when dealing with the problems of multiplicity, mutator methods warrant that maximum multiplicity is not violated, but regarding minimum multiplicity, it can happen that the number of linked instances is smaller than the minimum required by the design, in which case the *invalid multiplicity exception* is raised[21].

Mutator methods

We have also two mutator methods, `remove` and `add`, with the following signatures:

- `int remove();`
- `int remove(%Target old_link);`
- `int remove(Collection old_links);`
- `boolean add(%Target new_link);`
- `boolean add(Collection new_links);`

The `remove` method deletes target instances from the opposite association end, and returns a convenient error code. It can remove all instances (first parameterless version), one instance (second version), or a collection of instances (third version, available only when the opposite association end is multiple). In the third version, if any instance in the collection parameter is not of type `%Target`, then no link is removed, following the "none or all" semantics, and a *class cast exception* is raised. On the contrary, if any instance in the collection (or the single instance, in the second version of the method) is simply not linked to the source instance, then the operation proceeds without considering it an error. In a bidirectional association, the method invokes a reciprocal `remove` on each one of the instances to be deleted. The `remove` method can leave the source

[20] The parameter type is defined as `Collection` to get more generality. `Collection` is an abstraction (technically, an interface) realized by library classes such as `ArrayList`, `HashSet` and `TreeSet`.
[21] In fact, the check is performed by the `isValid` method, which can be more elaborated than simply verifying that the number of linked instances is not smaller than the minimum required; the programmer can modify manually the code of `isValid` to implement a more complex constraint.

instance or some of the target instances in an invalid state regarding the minimum multiplicity constraints, in which case an error code is returned, but no exception is raised. If a subsequent `get` method were invoked, the *invalid multiplicity exception* would be raised.

The `add` method appends a new target instance or a collection of target instances to the opposite association end, and returns a `boolean` value to indicate whether the operation was performed or not. The second version of this method is defined only when the association end is multiple; if any instance in the collection parameter is not of type `%Target`, then no link is added, following the "none or all" semantics, and a *class cast exception* is raised. On the contrary, if any instance in the collection (or the single instance, in the first version of the method) is already linked to the source instance, then the operation proceeds without considering it an error (and, of course, without adding a duplicate). In a bidirectional association, the method invokes a reciprocal `add` on each one of the instances to be appended. The `add` method checks whether the source instance, or any of the target instances, would be left in an invalid state regarding the maximum multiplicity constraints, in which case the operation is cancelled, no link is added, and a `False` value is returned[22].

If you want to substitute some target instances by other target instances, you must invoke first the `remove` method and then the `add` method; otherwise the result could be different from expected (see Section 2). Beware that this is valid *even for single associations*: there is no implicit `remove` of the old instance when you add a new instance (this is done this way in order to get the most similar behavior between single and multiple associations).

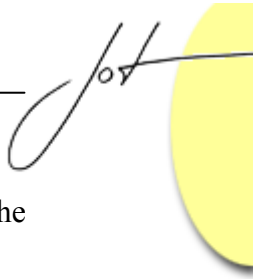
Auxiliary methods to learn the state of the association

We have two auxiliary methods to know the state of the association from the point of view of a particular source instance:

- `boolean isValid();`
- `long numberOfLinks();`

The `isValid` method determines whether the source instance sees the right number of target instances on the opposite side of the association, according to the multiplicity constraints specified in the design model. The tool generates code only for the simplest case, where the multiplicity constraint consists of a single `MIN..MAX` interval. Anyway, the programmer can modify manually the code to implement a more complex constraint, and the changes will affect the execution of accessor and mutator methods, since they check the multiplicity constraints by means of this method. This method is useful too when the automatic check of multiplicity constraints is disabled and the programmer assumes the responsibility for checking them manually at specific points in the source code.

[22] As in the previous case, the check is performed by the `isValid` method, which can achieve a more general behavior.



The `numberOfLinks` method returns the number of target instances linked to the source instance.

Auxiliary methods to learn the definition of the association

We have five auxiliary methods to know the definition of the association from one side of the association, that can be useful for the client code:

- `boolean isBidirectional();`
- `boolean isMandatory();`
- `boolean isMultiple();`
- `long getMIN();`
- `long getMAX();`

The `isBidirectional` method determines whether the reciprocal association end is navigable too. The `isMandatory` method determines whether the minimum multiplicity is greater than zero. The `isMultiple` method determines whether the maximum multiplicity is greater than one. The `getMIN` method returns the value of the minimum multiplicity constraint. The `getMAX` method returns the value of the maximum multiplicity constraint. A special value is returned when this is *many* ('*'). The interface also defines the constant value `int MANY` (actually –1).

6 THE CODE GENERATION TOOL

In this section we are going to present briefly the tool we have developed: *JUMLA* (Java code generator for Unified Modeling Language Associations). This tool reads a UML model, stored in XMI format, and generates Java code to manage the UML associations contained in the input model, according to the technique described in this paper. The tool generates code for associations only: it ignores every other UML artifact that is not directly related to associations, such as generalization between classes, class attributes and methods, etc. The tool presents the classes and associations found in the model, and the user can select which associations he or she wants to generated code for.

The tool creates output Java files for the involved classes and inserts into them the code for the associations, with convenient labels to mark the start and the end of the generated code. If the class file already exists, the code is inserted at the end of the class file, respecting any other class code that the programmer may have written manually (on the contrary, if the programmer changes the association code and then re-generates it, the manual changes are lost).

Figures 12 shows a sample model and Figure 13 shows how it is presented in the main window of the JUMLA tool. The left pane of the tool shows the classes contained in the model, in a tree structure corresponding to the package structure of the model. The right pane shows the associations contained in the model. For each association, the following information is presented: source and target classes; rolename (optional), multiplicity, navigability and visibility of source and target association ends; association

name (optional). The user can select with check boxes the associations he or she wants to generated code for.

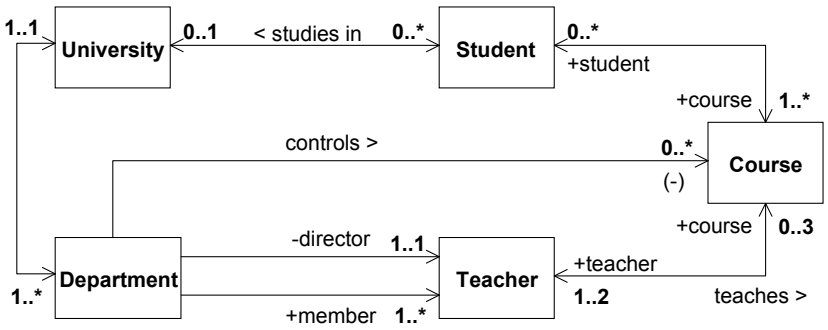


Figure 12. A sample model with some classes and associations between the classes

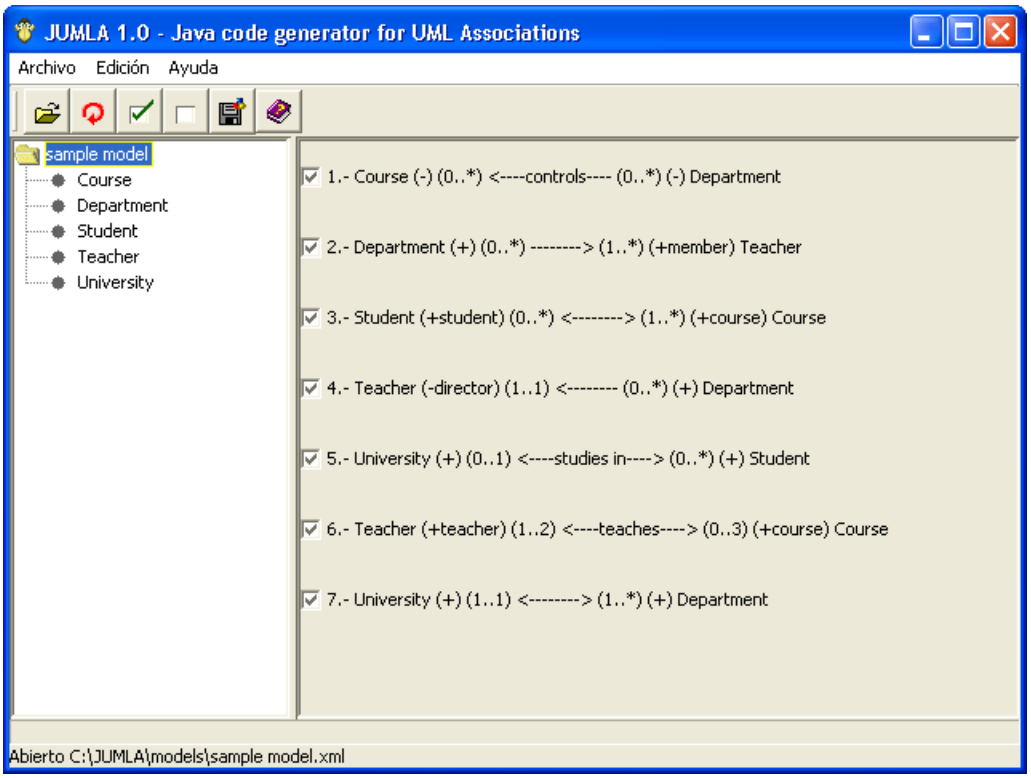


Figure 13. A snapshot of the JUMLA tool. The interface of the current version of the tool is in Spanish (Archivo = File, Edición = Edit, Ayuda = Help).

The tool behaves according to *five predefined options* which can be disabled by the user to get more flexibility in the generation of code or in dealing with the input model. Table 1 summarizes them.

Tool Option	Default
Check minimum and maximum multiplicity constraints in `get` method	Yes
Check maximum multiplicity constraint in `add` method	Yes
Check type of objects in `Collection` parameters in `add` and `remove` methods	Yes
Reject unidirectional associations with multiplicity constraint on source end	Yes
Reject bidirectional associations with one private or protected end	Yes

Table 1. Summary of tool options

The first two options refer to the *automatic checking of multiplicity constraints* in mutator and accessor methods by means of the `isValid` auxiliary method. According to the justification given in Section 2, the predefined behavior is: `get` methods raise an *invalid multiplicity exception*, defined in the code that implements the association, if multiplicity constraints are not satisfied; `add` methods reject the addition of new links if these constraints are not satisfied, but they raise no exception; and `remove` methods don't do any checking. Changing the default value of these two options allows the generation of a simplified code that omits these checks, so that the user assumes the responsibility of controlling multiplicity.

The third tool option refers to the *automatic type checking* in mutator methods (`add` and `remove`) for multiple associations, which deal with `Collection` parameters, by means of run-time explicit casting. According to the justification given in Section 2, if the type-check fails, then the links are not updated, and the Java predefined *class cast exception* is raised. Changing the default value of this third option allows the generation of a simplified code that does not check the type of objects received in a `Collection` parameter, and does not raise this exception.

The last two options refer to the *checking of the input model's correctness*. In the predefined behavior, unidirectional associations with multiplicity constraints on the nonnavigable association end are rejected (see Section 3), and bidirectional associations with visibility other than `public` or `package` in both ends are also rejected (see Section 4). Changing the default value of the fourth option allows the generation of code without checking the multiplicity on the nonnavigable end, instead of rejecting the association. Changing the default value of the fifth option allows the generation of code, instead of rejecting the association, when one of the ends is `protected` or `private` and the other end is `public` or `package`, warning the user that he or she must provide an indirect access via other methods. When both ends are `protected` or `private`, the association is allways rejected, since the generated code could not work properly.

7 CONCLUSIONS

In this work we have developed a concrete way of mapping UML associations into Java code: we have written specific code patterns, and we have constructed a tool that reads a UML design model stored in XMI format and generates the necessary Java files. We have paid special attention to three main features of associations: multiplicity, navigability and

visibility. Our analysis has encountered difficulties that may reveal some weaknesses of the UML Specification [UML].

Regarding multiplicity, we have shown that it is impossible in practice with a few primitive operations to keep the minimum multiplicity constraint at any moment on a mandatory association end; our proposal is to check this constraint only when accessing the links, but not when modifying them. The programmer will be responsible for using the primitives in a consistent way so that a valid system state is reached as soon as possible. On the contrary, it is possible to ensure the fulfillment of the maximum multiplicity constraint during run-time, and so we enforce it in our implementation. Single association ends are easily stored in attributes having the related target class as type, but multiple association ends require the use of collections to store the corresponding set of links; as collections in Java are based on the standard `Object` class, it is necessary to perform run-time type-checking by means of explicit casting when using collections as parameters in the mutator methods.

Regarding navigability, unidirectional associations are easier to implement by means of attributes than bidirectional associations, because of the difficulties in synchronizing both associations ends. An update to a bidirectional association must be performed atomically on both ends to keep them consistent; this is achieved in the source object by issuing a reciprocal update on the target object. We have considered the pros and cons of an alternative implementation, based on the storage of "reified tuples", and finally we have discarded it in favor of our "synchronized cross-references" scheme. A side consequence of our analysis is that the multiplicity constraint in a design model can be specified only for a navigable association end.

Regarding visibility, in the case of unidirectional associations it can be implemented rather easily by simply mapping the visibility of the association end onto the visibility of the corresponding accessor and mutator methods, because UML and Java visibility levels have the same semantics. However, bidirectional associations with one or two private (or protected) ends behave paradoxically, because the reciprocal update becomes impossible. Besides, we consider that package visibility is ill-defined for associations in the UML Specification, and we have suggested a new definition.

The generated code for each association is easily localized inside the involved Java classes. Each association end presents a uniform programmer's interface. The interface is exactly the same for unidirectional and bidirectional association ends, but there are slight differences for single and multiple association ends.

Our approach is rather check-exhaustive with regard to invariants. We think that it is worth doing for the programmer as much as we can, so that our tool will insert code to perform run-time multiplicity and type checking and, of course, to issue reciprocal updates on bidirectional associations. However, different tool options will allow the user to override the automatic multiplicity and type checks when generating code, in favor of efficiency. Besides, we have argued that unidirectional associations should not have a multiplicity constraint on the source end in a design model, and bidirectional associations should not have both ends with private (or protected) visibility; therefore, the tool will

reject the generation of code for these associations. Again, the user will be able to disable this model-correctness checking and issue the code generation at his/her own risk.

This work can be continued on several lines. First, implementation of other association end properties, such as ordering, changeability, interface specifier, xored associations, and so on. Second, specific implementation of particular kinds of binary associations, such as reflexive associations, aggregations and compositions. Third, implementation of more complex associations: qualified associations, associations classes, and n-ary associations. Fourth, expand the tool to perform reverse engineering, that is, obtaining the associations between classes by analyzing the code that implements them. Our tool does not presently accomplish this task, although it is a very simple and straightforward procedure if the code has been written with our patterns. Finally, adapt the tool and the patterns so that they follow the new Java Metadata Interface (JMI) Specification [JMI].

8 ACKNOWLEDGEMENTS

The authors wish to give thanks to Perdita Stevens, since this paper was written mainly while the first author was visiting the Laboratory for Foundations of Computer Science (LFCS), part of the Division of Informatics of the University of Edinburgh, invited by her in February-April 2002; a preliminar version of this work was presented at the LFCS Lab Lunch on April 23rd 2002. This research stay was accomplished with funding by the Fundación Universidad Carlos III, Madrid, Spain. Thanks also to José Miguel Fuentes, Víctor Quintana, David Fernández and Vicente Palacios for their valuable suggestions to improve the paper.

REFERENCES

[Amble01] Scott W. Ambler. “An Overview of Object Relationships”, “Unidirectional Object Relationships”, “Implementing One-to-Many Object Relationships”, “Implementing Many-to-Many Object Relationships”. A series of tips to be found at IBM Developer Works, http://www-106.ibm.com/developerworks/.

[Arnol00] Ken Arnold, James Gosling, David Holmes. *The Java Programming Language*. Addison-Wesley, 3rd ed., 1998.

[Caste00] Xabier Castellani, Henri Habrias, Philippe Perrin. "A Synthesis on the Definitions and Notations of Cardinalities of Relationships", *Journal of Object Oriented Programming*, 13(6):32-35 (2000)

[cUML] Financial Systems Architects (New York, U.S.A.). *3C-Clear Clean Concise*. Submission to OMG. Available at http://www.community-ml.org/.

[Eckel00] Bruce Eckel. *Thinking in Java*, 2nd ed. Prentice-Hall, 2000.

[Fowle97] Martin Fowler, Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[Fujab] *The Fujaba CASE Tool*, University of Paderborn, http://www.fujaba.de/.

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Genil99] Guy Genilloud. "Informal UML 1.3 - Remarks, Questions, and some Answers". *UML Semantics FAQ Workshop* (held at ECOOP'99), Lisbon, Portugal, June 12th 1999.

[Génov01] Gonzalo Génova. "Semantics of Navigability in UML Associations". *Technical Report UC3M-TR-CS-2001-06*, Computer Science Department, Carlos III University of Madrid, November 2001, pp. 233-251.

[Génov02] Gonzalo Génova, Juan Llorens, Paloma Martínez. "The Meaning of Multiplicity of N-ary Associations in UML", *Software and Systems Modeling*, 1(2): 86-97, 2002. A preliminary version in: Gonzalo Génova, Juan Llorens, Paloma Martínez. "Semantics of the Minimum Multiplicity in Ternary Associations in UML". *The 4th International Conference on the Unified Modeling Language-UML'2001*, October 1-5 2001, Toronto, Ontario, Canada. Published in *Lecture Notes in Computer Science 2185*, Springer 2001, pp. 329-341.

[Génov03a] Gonzalo Génova, Juan Llorens, Vicente Palacios. "Sending Messages in UML", *Journal of Object Technology*, vol.2, no.1, Jan-Feb 2003, pp. 99-115, http://www.jot.fm/issues/issue_2003_01/article3.

[Génov03b] Gonzalo Génova. *Entrelazamiento de los aspectos estático y dinámico en las asociaciones UML*. PhD Thesis, Carlos III University of Madrid, 2003.

[Gosli96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Harri00] William Harrison, Charles Barton, Mukund Raghavachari. "Mapping UML Designs to Java". *The 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications-OOPSLA'2000*, October 15-19 2000, Minneapolis, Minnesota, United States. *ACM SIGPLAN Notices*, 35(10): 178-187. ACM Press, New York, NY, USA.

[JMI] Java Community Process. *Java Metadata Interface (JMI) Specification*, Version 1.0, June 2002. Available at http://www.jcp.org/.

[Kaind99] Hermann Kaindl. "Difficulties in the Transition from OO Analysis to Design". *IEEE Software*, 16(5):94-102 (1999).

[McAll98] Andrew McAllister. "Modeling N-Ary Data Relationships in CASE Environments", *Proceedings of the 7th International Workshop on Computer Aided Software Engineering*, pp. 132-140, Toronto, Canada (1995). A more recent version in: "Complete Rules for N-Ary Relationship Cardinality Constraints", *Data & Knowledge Engineering*, 27(3):255-288 (1998).

[Noble96] James Noble. “Some Patterns for Relationships”. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*, Melbourne, 1996. Prentice-Hall.

[Noble97] James Noble. “Basic Relationship Patterns”. In *Proceedings of the European Conference on Pattern Languages of Program Design (EuroPLOP'97)*. Irsee, Germany, 1997.

[Rhaps] *The Rhapsody CASE Tool*, ILogix, http://www.ilogix.com/.

[RM] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[Ruiz02] Carlos Ruiz del Castillo. *Implementación en Java de asociaciones binarias UML*. Universidad Carlos III de Madrid, Proyecto Fin de Carrera, Ingeniería Informática (Segundo Ciclo), julio 2002. Tutor: Gonzalo Génova.

[Rumba87] James Rumbaugh. "Relations as Semantic Constructs in an Object-Oriented Language", In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pp. 466-481, Orlando, Florida, 1987.

[Rumba96a] James Rumbaugh. "Models for Design: Generating Code for Associations". *Journal of Object Oriented Programming*, 8(9):13-17, February 1996.

[Rumba96b] James Rumbaugh. "A Search for Values: Attributes and Associations". *Journal of Object Oriented Programming*, 9(3):6-8, June 1996.

[Song95] Il-Yeol Song, Mary Evans, E.K. Park. "A Comparative Analysis of Entity-Relationship Diagrams", *Journal of Computer and Software Engineering*, 3(4):427-459 (1995).

[Steve00] Perdita Stevens, Rob Pooley. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 2000.

[Steve02] Perdita Stevens. “On the Interpretation of Binary Associations in the Unified Modelling Language”, *Journal on Software and Systems Modeling*, 1(1):68-79 (2002). A preliminar version in: Perdita Stevens. "On Associations in the Unified Modeling Language". *The Fourth International Conference on the Unified Modeling Language*, UML'2001,

October 1-5, 2001, Toronto, Ontario, Canada. Published in *Lecture Notes in Computer Science 2185*, Springer 2001, pp. 361-375.

[UML] Object Management Group. *Unified Modeling Language (UML) Specification*, Version 1.4, September 2001 (Version 1.3, June 1999). Available at http://www.omg.org/.

[XMI] Object Management Group. *XML Metadata Interchange (XMI) Specification*, Version 1.2, January 2002. Available at http://www.omg.org/.

About the authors

Gonzalo Génova received in 2003 his PhD in Computer Science at the Carlos III University of Madrid, Spain, where he is currently a Teaching Assistant of Software Engineering and Advanced Software Design. His main research subject is modeling and modeling languages in software engineering. He can be reached at ggenova@inf.uc3m.es.

Carlos Ruiz del Castillo received in 2002 his MS degree in Computer Science at the Carlos III University of Madrid. He is currently working for a news agency in the development of .NET applications.

Juan Llorens is Associate Professor of the Computer Science Department at the Carlos III University of Madrid, Spain, where he is the leader of the IE (Information Engineering) research group. He is also a Visiting Professor at Aland's Institute of Technology - ATL, Mariehamn, Finland. His current research involves the integration of Knowledge technologies and Software Engineering techniques towards Software and Information Reuse. He can be reached at llorens@inf.uc3m.es.