Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity

Omar Badreddin, Andrew Forward, and Timothy C. Lethbridge

Abstract. UML classes involve three key elements: attributes, associations, and methods. Current object-oriented languages, like Java, do not provide a distinction between attributes and associations. Tools that generate code from associations currently provide little support for the rich semantics available to modellers such as enforcing multiplicity constraints or maintaining referential integrity. In this paper, we introduce a syntax for describing associations using a model-oriented language called Umple. We show source code from existing code-generation tools and highlight how the issues above are not adequately addressed. We outline code generation patterns currently available in Umple that resolve these difficulties and address the issues of multiplicity constraints and referential integrity.

Keywords: Associations, Model Driven Development, UML, Code Generation, Umple, Reverse Engineering.

1 Introduction

A UML association is a relationship among classes within an object-oriented system. It specifies the connections, called links, which may exist among instances of classes at run time. A notation called multiplicity appears at each end of an association to describe the number of links other objects may have to the object in question. In this paper we focus on binary associations, which have just two ends. Examples of UML diagrams with associations can be found in Figures 1 and 2.

Omar Badreddin · Andrew Forward · Timothy C. Lethbridge
School of Electrical Engineering and Computer Science,
University of Ottawa, Canada K1N 6N5
e-mail: {obadr024,aforward,tcl}@eecs.uottawa.ca

R. Lee (Ed.): *SERA*, SCI 496, pp. 129–149.
DOI: 10.1007/978-3-319-00948-3_9 © Springer International Publishing Switzerland 2014

Current programming languages do not directly support associations, which are coded by hand or by using a UML code generator tool. However, current code generators have weaknesses such as not dealing with referential integrity [1].

This paper introduces a syntax called Umple for defining associations textually, as well as a series of patterns for generating high-level language code from associations. We analyse all possible combinations of multiplicity that may appear on association ends, highlighting the underlying complexity in properly implementing associations in a target executable language. We analyse existing UML code generation tools to investigate the completeness of their implementation, or lack thereof. We provide code-generation patterns covering all multiplicity combinations that ensure referential integrity as well as adherence to multiplicity constraints.

Umple enables modelling concepts to be described textually with a similar syntax to Java. We present our code generator for Java, but Umple also supports PHP and Ruby and can conceptually support any object-oriented language.

In translating UML associations into an executable language, it is more efficient (but currently uncommon) to include access methods (get, set, add, remove) to manage links of associations. These methods would maintain the multiplicity constraints of the association and preserve referential integrity – ensuring that both ends of an association are properly updated when adding or removing links.

2 Associations in Practice

In a separate paper in this conference [3] we discussed an empirical study that analyzed the use of attributes in open source systems. We used the same systems as a starting point for the research presented here.

The seven projects selected for analysis include fizzbuzz, ExcelLibrary, ndependencyinjection, Java Bug Reporting Tool, jEdit, Freemaker; and Java Financial Library. We documented all member variables and recorded the project, namespace, object type, and variable name for each, as well as related characteristics such as constructors, and set/get methods.

To find variables representing associations, we used a two-step manual process. The first step recursively eliminated attributes. An attribute is considered to have as its type either: a) a simple data type including String, Integer, Double, Date, Time and Boolean, or b) a *complex attribute* type, i.e. a class that itself only contains instance variables meeting conditions a and b, with the proviso that in this recursive search process, if a cycle is found, then the variable is deemed a candidate association end. This process resulted in 350 candidate association ends.

In the second step, we filtered the set down to 235 association ends by removing *internal* variables. Internal variables represent data that is not an intrinsic part of the permanent state of the object, but is used for some algorithm or temporary process. They are neither set in the constructor nor are available via set/get methods. Examples of the types of such variables include Readers, Streams, and Maps.

Table 1 highlights distribution statistics of the 235 association ends. The results are not mutually exclusive so the column sum will not be 100%.

Table 1 Distribution of set / get and availability in constructor

Category	Freq	%	Description
Set/Get Methods	67	29%	All variables that had both a set and get method.
Set Method	89	38%	All variables that at least had a set method.
Get Method	120	51%	All variables that at least had a get method.
No Set Method	54	23%	All immutable variables, as there is no set method
Only Get Method	39	17%	Internally managed (no set method, not in constructor).

In addition to tracking the distribution of set and get methods, we also noted that some implementations provided direct access to the list structure and others provided methods like add and remove. Of the 235 association ends, 42 (17.9%) were defined using Map, Set, Hash, or List classes and hence likely represented associations with an upper bound greater than one.

When analyzing the open-source systems, it was difficult to match association-end variables to one another because many associations linked to external resources and most likely represented one-way associations. There was little evidence of referential integrity between association ends, implying that the application developer *using* the object model would have to write code to maintain the correct multiplicities and inverse pointers. This difficulty in analyzing how associations are used in practice provides motivation for our work. It would appear to be beneficial for developers to be able to define associations in one location, and have links created, accessed and modified in a consistent manner.

To better understand the types of associations used in practice, we analyzed 1400 associations in UML diagrams of real systems (not association ends as discussed above). These were found in two UML specifications (v1.5 / v2.1.2) and some UML profiles (MARTE, Flow Composition, ECA, Java, Patterns, rCOS).

Table 2 Industry Usage of Association Multiplicities in UML

Industry Usage			Rank in the various sets of examples		
Multiplicity	**Frequency**	**Ratio**	**Industry**	**Example**	**Repository**
1--*	273	19.0%	1	1	1
0..1--*	270	18.8%	2	4	2
->	179	12.4%	3	9	4
--	162	11.3%	4	2	3
0..1--1	126	8.8%	5	N/A	7
*->1	86	6.0%	6	N/A	6
*->0..1	73	5.1%	7	N/A	5
0..1--0..1	58	4.0%	8	6	N/A
--n..	54	3.8%	9	N/A	N/A
Other	157	10.9%	N/A	N/A	N/A
Total	1438	100.0%			

We also analyzed UML models in a book by one of the authors [4] (*example models*), and in our repository of UML modeled systems [5] (*repository models)* built using Umple.

The top nine associations patterns by multiplicity, ordered according to the industry examples, are in Table 2. The multiplicities include the following constraints: optional (0), one (1), lower-bound (n), upper-bound (m), and many without bounds (*). For example, a one-to-many multiplicity would be 1 -- *. The rank of actual usage is based on the UML specification. For comparison the rank of the particular multiplicity within the example and repository collection is also shown.

We performed similar analysis based on example usage from [4] and present the results in Table 3.

Table 3 Example Usage [4] of Association Multiplicities

Example Usage			Rank in the various sets of examples		
Multiplicity	**Frequency**	**Ratio**	**Industry**	**Example**	**Repository**
1--*	39	39.8%	1	1	1
--	15	15.3%	4	2	3
1--1	13	13.3%	N/A	3	N/A
0..1--*	11	11.2%	2	4	2
*->1	4	4.1%	N/A	5	6
0..1--0..1	4	4.1%	8	6	N/A
1--0..n	3	3.1%	N/A	7	N/A
*--n	2	2.0%	N/A	8	N/A
->	2	2.0%	3	9	4
Other	5	5.1%	N/A	N/A	N/A
Total	98	100.0%			

Table 4 is ordered based on our UML model repository examples [5], which has over 28 systems in domains like airlines, elevators, traffic lights and the Umple metamodel itself.

Table 4 Usage of Association Multiplicities from Model Repository [5]

Model Repository Usage			Rank in the various sets of examples		
Multiplicity	**Frequency**	**Ratio**	**Industry**	**Example**	**Repository**
1--*	108	43.4%	1	1	1
0..1--*	34	13.7%	2	4	2
--	27	10.8%	4	2	3
->	22	8.8%	3	9	4
*->0..1	21	8.4%	7	N/A	5
*->1	12	4.8%	6	5	6
1--0..1	4	1.6%	5	N/A	7
1--1..*	3	1.2%	N/A	N/A	8
1..*--*	3	1.2%	N/A	N/A	9
Other	15	6.0%	N/A	N/A	N/A
Total	249	100.0%			

The industry and example UML models share five of the top nine multiplicity usage patterns; one-to-many, optional-one-to-many, many-to-many, optional-one-to-one, and optional-one-to-optional-one.

After analyzing over 1,800 different modeled associations, approaches like eUML [6] (where only a subset of the UML multiplicities can be modeled) provide sufficient coverage for most applications. The same is true of the applications analyzed in Section 5, where most code generators provide little capability for association multiplicities beyond differentiating a *one-end* from *many-end*. As shown above, about 5% of the UML specifications fall outside of the simple cases currently supported and it should be of both academic and practical relevance to explore all types of association relationships.

3 Textual Associations in Umple

Umple is a set of extensions to object-oriented languages that provides a concrete textual syntax for UML abstractions like attributes, associations and state machines. Below, we describe how associations are represented in Umple. Please see our separate paper [3] for a discussion of attributes. For more details, and for the motivation regarding why we created Umple, the reader should refer to [7] and [8]. Umple can also be edited directly within a browser [9].

Figure 1 shows two associations. To distinguish between Umple and Java, the Umple examples use dashed borders in light-grey shading, and Java examples use solid-line borders with no shading. The UML class diagram one the right has three classes and two one-to-many associations. The code on the left is the equivalent in Umple . The '--' means that the association is bi-directionally navigable (more on this later). It is also possible to use '->' or '<-' to indicate that navigation is possible in only one direction. The full set of UML multiplicity symbols may be used.

```
class Student {}
class CourseSection {}
class Registration
{
  String grade;
  * -- 1 Student;
  * -- 1 CourseSection;
}
```

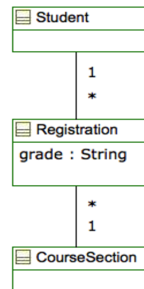

Fig. 1 Umple code/model for part of the registration system

In addition to showing an association embedded in one of the two associated classes, it is also possible to show an association 'on its own'. The association name, *Enrollment* and role names *course* and *attendee* are optional.

```
association Enrollment
{* Registration course -- 1 Student atendee;}
```

Besides providing improved abstraction, explicitly coding associations may speed development and reduce bugs, since the compiler can enforce various design constraints and less code needs to be written. The current implicit nature of associations in standard languages most likely results in code that is bug-prone since there is no general mechanism to enforce things like referential integrity.

4 Analyzing Association Multiplicity

Consider the following association between a Mentor and a Student.

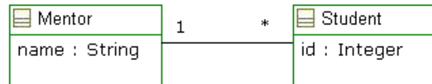

Fig. 2 An example binary association

An association has a multiplicity at each end describing how many instances of one class can be linked with the other class. In Figure 2, a Mentor can link to any number of Students, but a Student must be assigned to one and only one Mentor.

In general, there are nine distinct categories of multiplicity, shown in Table 5. These largely require different treatment in the code, except that the 1..n case can be coded as a special case of m..n and 1..* can be coded as special case of m..*.

Table 5 Multiplicity Possibilities for Associations

Multiplicity Notation	Lower Bound	Upper Bound	Description
0..1	0	1	Optional-One – Item is either present or not
0..n	0	n > 1	At Most n – At most n items, or none at all
0..* (*)	0	undef. > 1	Many – Any number of items present. or none at all
1..1 (1)	1	1	One – The item is mandatory
1..n	1	n > 1	Mandatory at Most n – At least one item is mandatory up to a maximum of n items
1..*	1	undef. > 1	Mandatory Many – One item is mandatory (no max)
n..n (n)	n > 1	n	Exactly n – Exactly m items are mandatory
m..n	m > 1	n > m	From m to n – At least m items are mandatory up to a maximum of n items
m..*	m > 1	undef. > m	At least m – At least m items mandatory (no max)

4.1 Bidirectional Associations Between Two Different Classes

We analyzed associations between two classes that are navigable in both directions. Both linked objects are conceptually *aware* of the relationship. There are 28 different patterns of binary associations, as listed in Table 6.

Note that x -- y is equivalent to y -- x, so for example, 0..1 -- n is equivalent to n -- 0..1; for this reason, the upper diagonal in Table 6 is left blank. Also note that the variables ***m*** and ***n*** are not assumed to be the same on both ends of the association. For example, 0..6 -- 3..4 association would be in the 0..n -- m..n category.

Our examples model a Mentor and Student, allowing us to vary the multiplicity in a sensible way. Figure 3 shows a Student always has one Mentor, but a Mentor can have zero or more students.

Our code written in Java assumes a Mentor has a *student* variable when the multiplicity upper bound (ub) is 1, or *students* if ub > 1. A Student, has a *mentor* (ub =1), or *mentors* (ub > 1). The Umple notation for Figure 3 is:

```
association {1 Mentor -- * Student;}
```

Table 6 All 28 Bi-Directional Non-Reflexive Association Patterns

0..1	**0..n**	*****	**1**	**n**	**m..n**	**m…***
0..1 -- 0..1						
0..1 -- 0..n	0..n – 0..n					
0..1 -- *	0..n -- *	* -- *				
0..1 -- 1	0..n -- 1	* -- 1	1 -- 1			
0..1 -- n	0..n -- n	* -- n	1 -- n	n -- n		
0..1 -- m..n	0..n -- m..n	* -- m..n	1 -- m..n	n -- m..n	m..n -- m..n	
0..1 -- m..*	0..n -- m..*	* -- m..*	1 -- m..*	n -- m..*	m..n -- m..*	m..* -- m..*

Shaded cells show cases where both sides are *mandatory*. Associations with thick borders indicate the common cases as observed in the previous section. Note that 1..* is a subset of the more generic m..* and 1..n is a subset of m..n.

4.2 Unidirectional (Directed) Associations

A directed association is navigable in one direction only. Only one object of the pair is aware of and can manage the relationship. For example, one could write in Umple:

```
association {* Mentor -> * Student;}
```

In this, a Mentor is aware of the associated Students, but a Student is unaware of any Mentor's to which he/she might be associated. The end that is unaware of the link can be unknowingly linked to multiple objects (i.e. a * relationship is generally implied); resulting in seven possible combinations for code generation.

```
* -> 0..1, * -> 1, * -> *, * -> m..n, * -> n, * -> m..*, * -> 0..n
```

Without injecting additional complex code, the system will be unable to manage the association when changes occur to the unaware side, such as a Student deleting itself. If such situations must be managed, then a bi-directional association must be used. In practice, we find that the vast majority of associations would benefit from being bidirectional. Doing so enables functionality that tends to be required anyway, and where the functionality is not immediately required, the code is better suited to meet unanticipated future needs. However, bi-directional association can increase coupling unnecessarily.

4.3 Reflexivity and Symmetry

Where both ends of an association are the same class, we must consider several special cases. A reflexive association allows an object to be linked to other objects of the same class including itself. An example of such an association might be “lives-at-same-address”.

It is also common to have *irreflexive* associations with both ends being the same class. The added constraint is that a given object cannot be linked to itself. For example, a *mother* relationship among Person objects is irreflexive as you cannot be your own mother.

A *symmetric* association describes a mapping that reads the same as its inverse; for example, a *spouse* association. An asymmetric association is not reversible; for example a *child* relationship. Finally, an anti-symmetric association is asymmetric except that it allows a relationship to self. For example the relationship *being-present-at-birth* is anti-symmetric.

One could encode the mentor-student example using a single class, where all objects are Persons, and some persons can mentor others. This is an irreflexive asymmetric association, since one cannot mentor oneself and the meaning of the association is different in each direction. The Umple notation would be:

```
association {0..1 Person mentor -- * Person student;}
```

The Umple language natively supports symmetric and asymmetric associations. Anti-symmetric associations are currently not supported as we find them to be quite rare (they may be supported in the future). The distinction between reflexivity and irreflexivity is currently not managed, but applications can relatively easily be coded to prevent (or allow) an object to be linked to itself.

Asymmetric associations require code that is effectively identical to associations between two different classes (see Table 6) - except that the lower bound of both ends must be zero. The top seven associations from Table 6 are therefore possible as asymmetric associations from one class to itself. The reason the lower bound must be zero is to prevent infinite regress. For example, in the case of class Person in the asymmetric association previously discussed, if every mentor Person

must have a student Person, and since every student is also a mentor (by virtue of being a Person) there would be an infinite chain of persons requiring a mentor.

A symmetric association specifies links between different instances of the same class, and must have the same multiplicity on each end. The diagonal of Table 6 gives the cases to consider.

As a result of all the above analysis, a total of 42 different possible association types have been identified (28 for bidirectional associations; 7 for unidirectional associations, and 7 for symmetric associations). In the following section, we highlight certain implications that these association types will have on code generation. This overview serves as a guide when comparing existing code generation tools and also as a template for building code generator for systems programmed in the Umple modeling language.

4.4 Implications for Code Generation

The implementation of associations in a language like Java impacts the following aspects of a class. First, the class will have an additional member variable to reflect the other end of an association. *One* and *optional-one* multiplicities can be declared as a member variable of the other type, while many multiplicities are declared as lists (implemented as a collection class) of objects of the other type.

Second, the constructor may need an additional parameter to ensure *mandatory* association ends like 1 or 1..*.

Finally, the class requires methods to set, get, add and remove links between objects. To be consistent with the model of the associations, the implementation of those methods should enforce the referential integrity between pairs of objects, as well as ensuring that multiplicity constraints are upheld.

In the following section, we analyze how existing code generators deal with the various combinations of multiplicities and to what extent they behave according to the structure outlined above. We then discuss the code generation available from the Umple language.

5 Existing Code Generators

In this section we look at existing tools to see how well they translate the semantics of associations into a programming language.

The UML modeling tools considered were identified from a Gartner report [10] and an online list [11]. We selected four open-source and one closed source application to analyze, as listed in Table 7. Each tool was configured to generate Java code for a simple 1 -- * relationship shown in Figure 2.

The generated code provided in the following sections has only been modified to provide a consistent layout/format, and for space considerations comments have also been removed.

Table 7 UML code generation tools

Tool	Version	Source
ArgoUML	0.26.2	argouml.tigris.org
StarUML	5.0.2.1570	staruml.sourceforge.net
BOUML	4.11	bouml.free.fr
Green	3.1.0	green.sourceforge.net
RSA	7.5	ibm.com/software/awdtools/architect/

5.1 Code Generation Patterns

In general, all tools analyzed provided two basic code generation templates; one for 0..1 and 1 (referred to as one) multiplicities and a second for m..n multiplicities (referred to as many where $m \geq 0$, $n > m$, $n > 1$).

The template pattern for *one* would generate a member variable to refer to the other association end. The template pattern for *many* would generate a reference to a List or Set structure that could contain multiple references to the other association end. Both examples are shown below.

```
private <ClassName> <assocEndName>;  // ub = 1
private <ListStructure> <assocEndName>; // ub > 1
```

Some tools provide explicit code generation patterns for *n* relationships (where $n > 1$), as well as m..* relationships (where $m \geq 0$). Some tools provided explicit get/set methods in addition to creating the necessary member variables. A discussion of each code generation pattern will be provided based on the tools analyzed.

5.2 ArgoUML

ArgoUML is an open source modeling platform that provides code generation for Java, C++, C#, PHP4 and PHP5. Below is the generated code from Figure 2.

```
import java.util.Vector;
public class Mentor {
  public Integer id;
  public Vector myStudent; }
public class Student {
  public String name;
  public Mentor myMentor; }
```

The generated code provides a mechanism to access “each end” of the relationship. The generator provides little validation or constraint checking to ensure the relationship is maintained, and the variables are made directly available without the inclusion of accessor (get and set) methods.

In general, all 0..1 and 1 multiplicities generate similar structures as seen in the Student class above, and all m..n multiplicities (where $m \geq 0$ and $n > m$ and $n > 1$) generate similar structures to the Mentor class.

5.3 StarUML

StarUML is an open source modeling tool. StarUML's generated code does not account for the *many* multiplicity, resulting in unusable generated code. Below is the generated code for the Mentor and Student example:

```
public class Mentor {
  public String name;
  public Student student; }
public class Student {
  public Integer id;
  public Mentor mentor; }
```

5.4 Bouml

Bouml is a free tool based on UML 2 that provides source code generation for C++, Java, Idl, Php and Python.

The source code generated below is very similar to that of ArgoUML. This code does not provide any mechanism to test or ensure the constraints outlined in the model; this code must be written by hand after code generation. In addition, the source code is incomplete as no reference the *java.util.List* class is provided, which means that the generated code must be maintained by hand to ensure proper compilation into byte code.

```
class Mentor {
  private List<Student> student;
  private String name; }
class Student {
  private Mentor mentor;
  private int id; }
```

5.5 Green Code Generator

Green UML is another UML editor that can generate source code from a class diagram. Below is the generated code for the Mentor and Student example.

```
import java.util.List;
public class Mentor {
  private List<Student> student;
  java.lang.String name;
  public Mentor(List<Student> student2)
   { student = student2; }
}
public class Student {
  private Mentor mentor;
  int id;
  public Student(Mentor mentor2)
   { mentor = mentor2; }
}
```

Green does provide additional code generation support by creating custom constructors based on the association. Green supports the following types of multiplicities: 1, n, m..*, and * (where n > 1 and m >= 0).

Green provides some enforcement of constraints; although the implementation is awkward and not scalable. For example, the implementation of the constraint of a *mandatory* relationship where a Mentor must have *n* Students (e.g. n = 3) is shown below.

```
public class Mentor {
  private Student student3;
  private Student student2;
  private Student student;
  java.lang.String name;
  public Mentor(Student student4,
              Student student5,
              Student student6) {
    student3 = student4;
    student2 = student5;
    student = student6;   } }
```

This implementation provides little opportunity to access or manage the collection of students, and instead each must be accessed explicitly by name. It also does a poor job of maintaining the constraint; as the variables could be set to *null*, violating the model's intention.

Green also provides an enforcement of m..* relationships. Below is an example implementation of a 2..* relationship.

```
import java.util.List;
public class Mentor {
  private List<Student> student;
  java.lang.String name;
  public Mentor(List<Student> student2) {
    student = student2;
    student.add(new Student());
    student.add(new Student());  }  }
```

The implementation above presents two issues. First, the potentially unwanted side effect of creating and inserting additional entities into the list argument (i.e. students). Second, the code generator assumes that a default (and empty) constructor exists for the Student object; an assumption that might not always be valid and could result in a generated system that does not compile.

Although Green UML does attempt to provide some additional source code generation to manage the various types of association multiplicities available; the results provide little, if any, added benefit in representing the model's intentions.

5.6 Rational Software Architect (RSA)

IBM's Rational Software Architect (RSA and RSA Real-Time) are full-fledged development environments that support model-driven development including source code generation from UML diagrams.

```
import java.util.Set;
public class Mentor {
  public Set<Student> students;
  public Set<Student> getStudents()
   { return students; }
  public void setStudents(Set<Student>students)
   { this.students = students; }
}
public class Student {
  public Mentor mentor;
  public Mentor getMentor()
   { return mentor; }
  public void setMentor(Mentor mentor)
   { this.mentor = mentor; }
}
```

RSA's model transformation into Java code provides some flexibility regarding the template patterns including (a) which Java collection to use, and (b) whether or not to include get/set methods for the attributes and association ends. As with all other source code generators, no distinction between the various possible one or many relationships are present in the generated code; leaving the implementation of the modeling constraints up to manually-written code. In addition to providing simple set and get methods, RSA's member variables representing the association ends was also public; presenting an encapsulation issue (especially considering the code already provides set and get methods).

6 Association Code Generation in Umple

The existing UML code generation tools analyzed in the previous section fall short of providing robust code to implement associations. The generated code provided little implementation support either to manage referential integrity or to ensure multiplicity constraints (beyond the 'one' vs. 'many' distinction).

In this section, we present our approach to code generation and identify implementation patterns that go beyond the capabilities of current tools. This approach is instantly accessible from Umple online [5].

6.1 Defining Association Variables

The first pattern to emerge is the distinction between having one object in the association and having many (i.e. upper bound equal to one versus greater than one). For convenience, we will use UB for upper bound and LB for lower bound.

Table 8 Member Variable Patterns

Mult. Constraint	Pattern	Example
UB = 1	ObjectType associationEnd;	Student student;
UB > 1	List<ObjectType> associationEnd;	List<Student> students;

6.2 Constructor Parameters for Associations

The next patterns relate to a class' constructor signature. The constructor defines how objects should be created and indirectly affects the order in which objects can be instantiated. Three signatures emerge from the various multiplicities:

- The association end is not required (LB=0) and not be part of the constructor
- Exactly one, the upper and lower bounds are exactly one
- Mandatory Many, (LB > 0 and UB > LB)

The patterns in Table 9 work well when the multiplicity of at least one end of the association is zero; allowing the creation of one object before the other. Below are example implementations of the constructors above.

Table 9 Constructor Signature Patterns

Multiplicity Constraint	**Pattern**	**Example**
LB = 0	Empty	N/A
LB=UB=1	ObjectType anAssociationEnd	Student aStudent;
LB > 0 && UB > 1	List<Student> someAssociationEnds	List<Student> allStudnets;

By using the *setStudent* method (which we discuss in the interface patterns section), we are able to encapsulate *how* students are set; including the verification that the set operation is indeed valid (i.e. association multiplicity constraints are not violated). If we are unable to assign the student, then an exception is thrown. The exact error message is not shown for simplicity.

```
public Mentor(Student aStudent) {
  boolean didAddStudent = setStudent(aStudent);
  if (!didAddStudent) {
    throw new RuntimeException("***"); } }
```

When the upper bound is greater than one (and the lower bound is not zero), we must initialize a list of associated members. We can delegate the action and verification using the *setStudents* (instead of setStudent like above) method.

```
public Mentor(List<Student> allStudents) {
  students = new ArrayList<Student>();
  boolean didAddStudents =
              setStudents(allStudents);
  if (!didAddStudents) {
    throw new RuntimeException("***");  } }
```

A chicken-and-egg issue manifests itself when neither end has a lower bound of zero; meaning that each end requires the other, resulting in deadlock as neither constructor can be called before the other. This issue has been divided into three domains: One to One, One to Mandatory Many and Mandatory Many to

Mandatory Many. To highlight the implementation of each situation above, we added a *name* attribute to the Mentor, and a *number* attribute to the Student class.

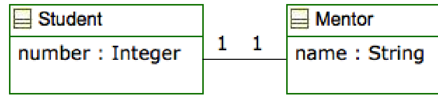

Fig. 3 One-to-one Student and Mentor association

In 3, a Mentor must have exactly one Student, and vice versa.

```
public Mentor(String aName, Student aStudent) {
  name = aName;
  if (aStudent == null || aStudent.getMentor() != null)
    { throw new RuntimeException("***"); }
  student = aStudent; }
public Mentor(String aName, int aNumberForStudent) {
  name = aName;
  student = new Student(aNumberForStudent, this); }
```

The second constructor *Mentor(String aName, int aNumberForStudent)* takes all the required parameters for both objects; allowing both objects to be created *effectively* instantaneously. Please note that in this case, there is no *setStudent* interface and the logic to verify that the student is valid is provided directly in the constructor. We do not include a *setStudent* or *setMentor* interface because the one-to-one constraint means you cannot re-assign a mentor or a student, as the *replaced* object would then be an orphan; violating the one-to-one constraint.

In the *One* to *Mandatory Many* case a Mentor must have more than one Student (m..n, n, or m..*) and a Student must have exactly one Mentor.

Table 10 Constructor Signature Patterns When Both Association Ends are Mandatory

Multiplicity	Constructor Implementation Multiplicity
1 -- m..*	1 -- *
1 -- m..n	1 -- 0..n
1 -- n	1 -- 0..n

The Mentor's constructor is implemented as though the lower bound was zero; allowing the objects to exist in an invalid state immediately following its construction. To verify the status of an object in such a case, we add an additional method `isNumberOfStudentsValid`, which checks if the number of students is valid.

The *mandatory-many* to *mandatory-many* constructor is similar, is that *both* constructors are initiated without the constraint being satisfied, and an additional method isNumberOfMentorsValid() is provided. The implementation as proposed

for these latter two cases allows for handling of cases that are in fact rare. Our implementation provides the developer with the necessary tools to query the association to verify the satisfaction of the constraint.

6.3 Get Method Code Generation Patterns

Table 11 outlines an the interface to access an association end available to a Mentor based on the multiplicity end of the Student. The method pattern is based on a generic association end name (*name*), and the association end's type (*type*).

Table 11 Method Signature Patterns for Get Methods

Mult. Constraint	Pattern	Example
UB = 1	getName() : Type	getStudent() : Student
UB > 1	getName(int index) : Type getNames() : Iterator<Type> indexOfName(Type aName) : int numberOfNames() : int hasNames() : boolean	getStudent(int index) : Student getStudents() : Iterator<Student> indexOfStudent(Student aStudent) : int numberOfStudents() : int hasStudents() : boolean

The *getStudent* implementation is shown below.

```
public Student getStudent() { return student; }
```

The difference between *mandatory one* (1) and *optional one* (0..1) is that the student member might be null in the optional case; whereas the 1 multiplicity end will never be null.

When the upper bound multiplicity is greater than 1, there are four common accessor methods as shown below.

```
public Student getStudent(int index)
 { return students.get(index);}
public Iterator<Student> getStudents()
 { return students.iterator(); }
public int numberOfStudents()
 { return students.size(); }
public boolean hasStudents()
 { return students.size() > 0; }
public int indexOfStudent(Student aStudent)
 { return students.indexOf(aStudent); }
```

Although one has access to all associated students, one is not able to alter the association by manipulating a list retrieved using the *get* methods shown above. To change the number of elements one must use the available *add* methods as shown below. The reason for this is to prevent the caller of API methods from

being able to violate the multiplicity constraints or corrupt the referential integrity. Other implementers of 'many' associations simply pass the collection of objects around, however, we explicitly ensure this never happens.

6.4 Set Method Code Generation Patterns

Next, we consider an interface to add, remove and set links of an association end. Again, we will be adding Student instances to a Mentor object based on various multiplicity constraints. Table 12 describes the generated interface.

Table 12 Method Signature Patterns for Set Methods

Multiplicity Constraint	**Pattern**	**Example**
UB = 1	setName(Type aName)	setStudent(Student aStudent)
UB > 1	addName(Type aName)	addStudent(Student aStudent)
	removeName(type aName)	removeStudent(Student aStudent)

The implementation of set methods is considerably more complex than get methods. First, set methods must undo any existing links between objects and establish the new links. Second, the methods must ensure referential integrity: when creating one end of a binary association they must create the other end as well.

Let us begin with the case where the upper bound is one. When the relationship is optional, the following scenarios must be considered.

If adding a new link, there must be code to set the inverse link as well. Conversely, if the inverse link has already been set, then it must not be set again. For example, if adding a Student to a Mentor, the code must be sure to add a Mentor to the Student (but only once).

If replacing or removing an existing link, both directions of the link must be removed. For example, if a Mentor can only have one Student, then when assigning a Mentor to a new Student, the implementation must unassign that Mentor from the existing Student.

When creating a new link, the multiplicity constraints on both ends must be satisfied. If a Mentor can only have four Students, then a Student is not allowed to add a Mentor such that the Mentor would now be linked to five Students.

Finally, whenever removing an existing link, the multiplicity constraints on the existing objects must be satisfied. If a Mentor must have at least two Students, then the implementation must not allow a Student to set itself to a new Mentor if the existing mentor is at its two-Student minimum.

Two examples are outlined below; one where UB = 1, and the other where UB > 1. First the implementation of *setMentor* from in Student class as part of the 0..1 Mentor -- 0..1 Student association.

```
public void setMentor(Mentor newMentor) {
  if (newMentor == null) {
    Mentor existingMentor = mentor;
    mentor = null;
    if (existingMentor != null &&
        existingMentor.getStudent() != null) {
      existingMentor.setStudent(null); }
    return; }
  Mentor currentMentor = getMentor();
  if (currentMentor != null &&!currentMentor.equals(newMentor)) {
    currentMentor.setStudent(null); }
  mentor = newMentor;
  Student existingStudent = newMentor.getStudent();
  if (!equals(existingStudent)) { newMentor.setStudent(this); } }
```

Next are the implementations of addStudent, and removeStudent for the Mentor class as part of the 0..1 Mentor -- * Student association.

```
public boolean addStudent(Student aStudent) {
  if (students.contains(aStudent)) { return false; }
  Mentor existingMentor = aStudent.getMentor();
  if (existingMentor == null) {
    students.add(aStudent);
    aStudent.setMentor(this);
  } else if (!existingMentor.equals(this)) {
    existingMentor.removeStudent(aStudent);
    addStudent(aStudent);
  } else { students.add(aStudent); }
  return true; }

public boolean removeStudent(Student aStudent) {
  if (!students.contains(aStudent)) { return false; }
  else {
    students.remove(aStudent);
    aStudent.setMentor(null);
    return true; } }
```

There are 42 different association combinations (28 different classes, 7 for directed and an additional 7 for symmetric). Each implementation follows the general guidelines shown above, and each combination can explored online at [9].

6.5 Patterns for Generated Support Methods

In addition to establishing relationships between objects, we include methods to query the minimum and maximum bounds of a relationship. Due to space constraints, we omit the full details of the support methods, but Table 13 highlights the interface.

Table 13 Interface for support methods

Multiplicity	Interface
m..n, m..*	minimumNumberOfStudents() : int
m..n, 0..n	maximumNumberOfStudents() : int
n	requiredNumberOfStudents() : int

7 Related Work

Several studies [12-16] propose approaches to formalizing the semantics of associations. They generally agree on the interpretation of the associations, but do not address uniqueness and ordering of associations.

Other studies refer to two types of associations; static and dynamic [17, 18]. Static associations, a view we adopt, represent structural relationships between objects, where the association is enforced throughout the lifetime of both objects. *Dynamic* (or *contextual* associations) are enforced only during the interactions of the two objects. Miliev [19] proposes yet another view of associations: *intentional* associations that encapsulate the intention of association of each participating object. Milicev highlights deficiencies in the traditional semantics of associations and multiplicities that can be overcome by the introduction of an intentional perspective on associations.

Acknowledging deficiencies in automated code generation of UML associations and multiplicities, Wang and Shen [20] propose a run-time verification approach for UML association constraints. Østerbye [21] proposes supporting association referential integrity with a reusable class library that ensures the consistency of the relationship is maintained.

Executable UML (eUML) [6] aims is to provide a (yet to be approved) specification of an unambiguous subset of executable UML (using model compilers). The Umple language also behaves as a model compiler and provides a concrete implementation of a subset of UML. However unlike Executable UML, Umple integrates with standard object oriented languages, and supports a wider range of multiplicity, as well as a variety of other features not present in executable UML.

Umple has been under continuous development since 2007. Experimentation with users has revealed that the comprehensibility levels of model oriented code is superior to the equivalent object oriented code [22-24]. Umple has also been used and evaluated in open source projects [25].

8 Conclusion

This paper discussed problems with generating code for UML associations, and proposed Umple as a solution. We identified the 42 combinations of multiplicity for association ends and analyzed their impact on code generation. We reviewed the code generated by five modeling tools, and found that none dealt with

multiplicity constraints or referential integrity. This may be one reason why code generation is not as widely used in practice as might be expected.. As a result developers must modify generated code by hand, which is awkward and error-prone. We provided an overview of the Umple language for associations and its model compiler that addresses the above issues.

References

1. Costal, D., Gómez, C.: On the use of association redefinition in UML class diagrams. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 513–527. Springer, Heidelberg (2006)
2. Object-Oriented Software Engineering: Practical Software Development using UML and Java. McGraw-Hill (2005)
3. Badreddin, O., Forward, A., Lethbridge, T.C.: Exploring a Model-Oriented and Executable Syntax for UML Attributes. Accepted in SERA 2013 (2013)
4. Object-Oriented Software Engineering: Practical Software Development using UML and Java. McGraw Hill (2001)
5. UmpleOnline, `http://www.try.umple.org` (accessed 2013)
6. Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley, Boston (2002)
7. Umple Language, `http://cruise.site.uottawa.ca/umple/` (accessed 2013)
8. Forward, A., Lethbridge, T.C., Brestovansky, D.: Improving program comprehension by enhancing program constructs: An analysis of the umple language, pp. 311–312 (2009)
9. Umple language online, `http://cruise.site.uottawa.ca/umpleonline/` (accessed 2013)
10. Norton, D.: Open-Source Modeling Tools Maturing, but Need Time to Reach Full Potential, Gartner, Inc., Tech. Rep. G00146580 (April 20, 2007)
11. Wikipedia Listing of UML modeling tools, `http://en.wikipedia.org/wiki/List_of_UML_tools` (accessed 2013)
12. Bourdeau, R.H., Cheng, B.H.C.: A formal semantics for object model diagrams. IEEE Trans. Software Eng. 21, 799–821 (1995)
13. Diskin, Z., Dingel, J.: Mappings, maps and tables: Towards formal semantics for associations in UML2. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 230–244. Springer, Heidelberg (2006)
14. France, R.: A problem-oriented analysis of basic UML static requirements modeling concepts. ACM SIGPLAN Notices 34, 57–69 (1999)
15. Overgaard, G.: A formal approach to relationships in the unified modeling language. In: Proceedings PSMT (1998)
16. Overgaard, G.: Formal specification of object-oriented ModellingConcepts. PhD Thesis, Dept. of Teleinformatics, Royal Inst. of Technology, Stockholm, Sweden (November 2000)
17. Stevens, P.: On the interpretation of binary associations in the Unified Modelling Language. Software and Systems Modeling 1, 68–79 (2002)
18. Genova, G., Llorens, J., Fuentes, J.M.: UML associations: A structural and contextual view. Journal of Object Technology 3, 83–100 (2004)

19. Miliev, D.: On the semantics of associations and association ends in UML. IEEE Trans. Software Eng., 231–258 (2007)
20. Wang, K., Shen, W.: Runtime checking of UML association-related constraints. In: Proceedings of the 5th International Workshop on Dynamic Analysis (2007)
21. Osterbye, K.: Design of a class library for association relationships. In: Proceedings of the 2007 Symposium on Library-Centric Software Design, pp. 67–75 (2007)
22. Badreddin, O.: Empirical Evaluation of Research Prototypes at Variable Stages of Maturity. In: ICSE Workshop on User Evaluation for Software Engineering Researchers, USER (to appear, 2013)
23. Badreddin, O., Lethbridge, T.C.: Combining experiments and grounded theory to evaluate a research prototype: Lessons from the umple model-oriented programming technology. In: User Evaluation for Software Engineering Researchers (USER). IEEE (2012)
24. Badreddin, O., Forward, A., Lethbridge, T.C.: Model oriented programming: an empirical study of comprehension. In: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp. (2012)
25. Badreddin, O., Lethbridge, T.C., Elassar, M.: Modeling Practices in Open Source Software. In: OSS 2013, 9th International Conference on Open Source Systems (to appear, 2013)