Design of a Class Library for Association Relationships

Kasper Østerbye
IT University of Copenhagen
kasper@itu.dk

ABSTRACT

Association is an important relationship that is supported in both UML and entity relationship database modeling tools. However, there is no language or library support in current object-oriented languages for implementing an association relationship. Instead, a complex implementation using references or collections of references must be handcrafted and laboriously maintained each time an association relationship is needed. In this paper, we develop an approach to supporting the association relationship through the design of a reusable class library that hides most of the complexity and guarantees that the consistency of the relationship is maintained automatically. Our current library implementation in C# draws on generic types with runtime type instantiation, runtime reflection on type parameters, annotations on classes, and runtime code generation. This set of language features seems to be necessary to provide effective support for association relationships.

1 INTRODUCTION

The association relationship is an important modeling concept that is found both in entity-relationship modeling and in UML modeling. However, there is no language or library support in current object-oriented languages for implementing an association relationship. Instead, a complex implementation using references or collections of references must be handcrafted and laboriously maintained each time an association relationship is needed. This can be compared to translating while-loops into goto-statements by hand. It is error prone, blurs the intended design, lowers the abstraction level, and complicates maintenance.

As well, an important success factor for object-oriented programming is to be able to move from design to implementation within the same paradigm. The design notations of classes, specialization, fields, and objects all have a direct linguistic counterpart in object oriented programming languages. The direct support of design concepts in the programming languages minimizes the semantic gap and assists in keeping the implementation in accordance with the original design.

In this paper, we develop an approach to supporting the association relationship through the design of a reusable class library Noiai (short for "no object is an island" inspired by [Beck & Cunningham, 1989] page 2). Noiai hides most of the complexity and guarantees that the consistency of the relationship is maintained automatically. Our current library implementation in C# draw on generic types with runtime type instantiation, runtime reflection on type parameters, annotations on classes, and runtime code generation. The lack of this combination of features prior to the .Net platform might in part explain that no association library has surfaced before. At least, the implementation strategy of Noiai cannot be used in languages lacking this set of features.

LCSD'07 October 21, 2007, Montréal, Canada
 ISBN 978-1-60558-086-9 . . . $5.00

2 BACKGROUND

The semantic gap in the transition from design to implementation of association relationships has been noted by several researchers. Some have designed language extensions to incorporate associations, the most recent of these being [Bierman & Wren, 2005], but the idea of direct language support can be traced to Rumbaugh's work on DSM [Rumbaugh, 1987][Rumbaugh, 1988][Shah *et al*. 1989]. Some work has taken place in the area of object-oriented databases as well, notably [Albano *et al*. 1991]. In [March & Rho, 2000] the original ideas of Rumbaugh are translated into a Smalltalk implementation. Our own work [Østerbye, 1999] also embeds associations as a language construct in Smalltalk.

Associations seem closely related to collections, but the major collection libraries like Smalltalk [Goldberg & Robson, 1989], STL [STL, 1994], Java [JavaSDK 5.0], or C5 [Kokholm & Sestoft, 2006] have no support for associations. However, an association library for the Aspect/J language is presented by [Pearce & Noble, 2006].

Another approach to handle the implementation of associations is a pattern-based approach. A small pattern language for association is given in [Noble 1995], and [Génova *et al*. 2003] has a rather thorough discussion of how to implement associations in Java. A closer discussion of related work will be presented in Section 5 after our own work has been introduced.

Terminology

In the literature at large, there is no consensus of the terminology. 'Relationship' and 'Association' seems to be used interchangeably. In the rest of this paper, relation will be used to denote a mathematical term, whereas association will denote a library or linguistic construct to support the UML notion of an association relationship. As it is useful to distinguish between type level and instance level, *association* will be reserved for the type level, and *linkage* for the instance level, e.g., an association Employment can contain the linkage (John,Dell). An association is specified using a *name* of the association, e.g., Ownership. It associates two *participants*, e.g., Person and Company. For convenience, we refer to the first participant as From, and the second as To. The two participants play *roles* in the association, e.g., owner and company in the Ownership association.

3 DESIGN OF THE NOIAI ASSOCIATION LIBRARY

The presentation of Noiai is done in two stages; first, the design and use of Noiai is presented, followed by an in-depth discussion of implementation issues.

To illustrate the Noiai framework, examples will be drawn from the simple model in Figure 1, illustrating associations declared

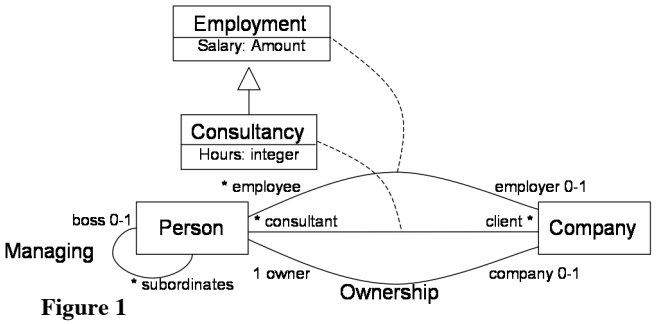

Figure 1

between class Person and class Company. The model represents a number of choices primarily intended to illustrate Noiai.

The ownership association specifies that a person can only own one company, that all companies are owned by exactly one person. The managing association specifies that all a subordinate have one boss, who might have several subordinates. The association Consultancy is a specialization of the association Employment. The employment association declares that a person can have at most one employer, while the employer can have multiple employees. The employment association declares a salary as an association field, and consultancy adds an hours field. Linkages of type consultancy will therefore have both a salary and an hours field.

The design of Noiai has been guided by the following main criteria:

1. Domain capture. It should be possible to use association names (e.g., Employment) and role names (e.g., Employer) rather than generic names.
2. Concise declaration and navigation. There should be as little boilerplate code as possible.
3. Early warning. Inconsistent declaration and usage is best caught at compile time, secondly at load-time, and thirdly at run-time.
4. Supporting the design space. The library should enable the library user to make choices of data structure, cardinality, association attributes etc.

The fourth criterion assumes knowledge of the design space. In brief, there are three main axes to be aware of in the design of binary associations:

a) *Storage*. There are two major ways to store the linkages of an association, as a global data-structure, or distributed in the linked objects.
b) *Declaration*. Again, there are two major ways to declare an association. Either as a top-level declaration akin to classes, or as roles in the participating classes.
c) *Manipulation*. Again, there are two major ways to manipulate an association. Either, by method calls on a top-level object, or as method calls on participating roles.

A contribution of [Østerbye, 1999] is to show that these three axes are independent. Noiai has been designed to aim for this goal, though it is not completely possible without language support.

3.1 *Declaration of associations*

In Noiai, associations are *declared* as classes. Basing associations on classes rather than objects was chosen because associations appear on the same meta-level as classes in UML and ER diagrams. Figure 2 in section 3.8 gives an overview figure of the entire library.

The Employment association from Person to Company is declared as:

```
[Cardinality( From=Cardinality.Unique )]
class Employment : BaseAssociation<Person, Company, Employment,
                                   Contract>{}
```

A BaseAssociation can be used as a root in an association hierarchy. The two first type parameters indicate the participants in the association. To differentiate the participants, the terminology is that the association Employment is *from* Person *to* Company. The seemingly redundant third parameter ensures that each association (E.g., Ownership, Employment, and Consultancy) from Person to Company each will get a unique instantiation of BaseAssociation. This will be further elaborated in section 4.1. The fourth parameter specifies which class to use for association fields, i.e., to store the salary and hours fields. Ideally, such fields should be declared as part of the Employment class. Unfortunately, this brings us in conflict with our implementation strategy where each declared association must have its own instantiation of a generic class (see sections 4.1 and 4.2), and we run into problems with variance of generic parameters. Alas, a separate class is used to store association fields.

Cardinality is specified using C# attributes (C# attributes are written in [] before the declaration of the class). If no cardinality constraints are defined, the association is assumed many-to-many. Unique signifies the role to be unique (that is, each Person is uniquely determining a Company). Because no To-constraint is specified, the Employment association becomes Many-One.

Sub-associations are declared using SubAssociation as the base class. The fourth parameter states the super association.

```
class Consultancy : SubAssociation<Person, Company, Consultancy,
                                   Employment, ConsultancyContract>{}
```

The attribute class ConsultancyContract must be a subclass of the corresponding attribute class in the super association.

From a subtype perspective, the cardinality-constraints of a sub-association must be invariant (i.e., neither covariant nor contra variant) because associations can be both inserted into and enumerated. However, our library takes a slightly broader stance.

The Consultancy association is many-many, as no cardinality-constraints are declared. Had the From-constraint in Employment been UniqueSub, uniqueness would have propagated down into sub-associations. Unique specifies that "A person has only one employer, but nothing is known about cardinalities of sub-associations". It is *explicit ignorance*. It is possible to extend the uniqueness to sub-associations as well (specifying Cardinality.UniqueSub). In the example model, Consultancy is a sub-association of Employment, but it is decided that while a person can only be employed at one company, it is possible to do consultancy for several companies.

Most associations do not carry attributes. However, Employment and Consultancy do, and the last type parameter tells which class to use for storing attributes of the employment. In C# it is possible to declare two generic classes with the same name if they differ in the number of the generic arguments. Hence, it is possible to provide a declaration syntax in which the attribute class seems optional.

3.2 *Behavior of associations*

The basic operations on associations are to add, lookup, and remove. When an association is declared, a singleton object representing the list of linkages is inherited from the instantiated super class. This singleton (named Assoc) is used to access the behavior of the association. The following code does add, lookup, and remove using the associations declared above. Assume John and Jane are declared as Person, and Dell and HP are declared as Company:

```
Employment.Assoc.Add(John,Dell);      // John is employed at Dell
Consultancy.Assoc.Add(Jane,Dell);     // Jane consults for Dell
Employment.Assoc.GetFromSet(Dell);    // returns an IEnumerable<Person> of
                                      // John and Jane.
```

```
Consultancy.Assoc.GetFromSet(Dell);  // returns an IEnumerable<Person> of
                                     //Jane.
Consultancy.Assoc.Add(John,HP);      // John consults for Dell
Employment.Assoc.GetToSet(John);     // returns an IEnumerable<Company>
                                     // of Dell and HP.
Consultancy.Assoc.Remove(Jane,Dell);// Jane no longer consults for Dell
```

It has not been possible to find a way to (conveniently) introduce role names into the association declaration. Hence querying is done using generic names like GetFromSet rather than Employees, and GetToSet rather than Employers. However, some headway can be made in C++ using pointers to members. We will briefly take that up in the conclusion.

The GetFromSet and GetToSet methods return an IEnumerable rather than a collection to enable lazy enumeration over associations. In addition, IEnumerable is the foundation of the proposed language integrated query facilities of C# 3.0 [Linq, 2007]. Thus, one can write queries such as:

```
from p in Employment.Assoc.GetFromSet(Dell)
where Managing.Assoc.GetFromSet( p ).Count() > 10
select p.Name
```

This query selects the names of all employees (including consultants) of Dell that hava more than 10 subordinates.

The action Consultancy.Assoc.Remove(Jane, Dell) removes the Consultancy association from Jane to Dell. Two removal methods are provided. Remove removes the linkage from an association (and thereby also in all super associations), and RemoveSub removes the linkage in the association and in all sub-associations.

Consider the following statement:

```
Employment.Assoc.Add(John, HP);      // sets John's employer to be HP,
                                     // removes prior employment
```

The semantics of the Add operation is significant. It ensures cardinality consistency by first removing the association from John to Dell, and then inserting the association from John to HP. As Employment is a many-one association, the choice is to ensure consistency, as done in Noiai, or to throw an exception if John is currently employed. Integrating the test, removal and addition into a single operation, giving fewer internal method calls, fewer state tests, and less dereferencing. However, testing and explicit removal is supported by Noiai.

Associations can be equipped with attributes. Assume Employment is declared with a Contract class with a Salary field, and Consultancy uses a subclass ConsultancyContract with a LastDay field. Add returns a new Contract object.

```
Employment.Assoc.Add(John,Dell).Salary = finaloffer;
```

To retrieve the attributes for a given employment, an indexer (C# construct) is provided:

```
Employment.Assoc[John,Dell].First().Salary  += 300;
```

The indexer returns an IEnumerable<Contract> (as there might be several linkages between the two). Notice that indexing (Jane, Dell) in Employment will return an enumerable of Contracts, of which some might be ConsultancyContracts, while indexing in Consultancy will return an enumerable of ConsultancyContracts.

3.3 *Role-based specification and navigation*

Sometimes it is convenient to access associations through role names rather than working on the singleton of the association class. This section shows how Noiai enables definition of roles in the participating classes as a supplement to the singleton based access. However, the actual storage of the linkages is still done in the singleton object; section 3.4 explains how Noiai supports storing the linkages in the roles themselves.

Noiai allow roles employer and employees in Person and Company to be declared as:

```
class Person{
  Employment.FromEntityRef Employer;
  public Person(){  Employer = new Employment.FromEntityRef( this );  }
}
class Company{
  Employment.ToEntitySet Employees;
  public Company(){  Employees = new Employment.ToEntitySet( this ); }
}
```

The role Employer is declared of type Employment.FromEntityRef. The explicit mentioning of Employment serves to bind the two roles to the same association. BaseAssociation provides four nested classes: FromEntitySet, FromEntityRef, ToEntitySet and ToEntityRef. These classes are intended for declaring roles as illustrated above. FromEntitySet and FromEntityRef declares a class that can be used within the type in the "From" position (first type parameter of the association), and similarly for the To-classes. As an association can link two objects of the same type (e.g., Managing), one cannot determine automatically which role a field plays, and it is necessary to specify this directly, here done by having From and To be part of the class name.

Declaring roles enables association behavior to be accessed as:

```
John.Employer.Set(Dell);                // Method "Set" for Entity<i>Ref</i>
Dell.Employees.Add(Jane );              // Method "Add" for Entity<i>Set</i>
foreach (Person p in Dell.Employees )   // ToEntitySet roles implements
                                        // IEnumerable<Person>
                                        // yields both John and Jane.
```

EntitySet implements the IEnumerable interface directly. Hence, the C# 3.0 query example

```
from p in Employment.Assoc.GetFromSet(Dell)
where Managing.Assoc.GetFromSet( p ).Count() > 10
select p.Name
```

can be rewritten using the role based syntax:

```
from p in Dell.Employees
where p.SubOrdinates.Count() > 10
select p.Name
```

Including role-names significantly enhances readability.

3.4 *Role-based storage*

As shown in the previous section, it can be convenient to allow roles to be declared in the participating classes, as it gives nice navigation syntax. Sometimes it is also preferable to let the storage of the linkages be handed over to the participating classes as it alleviates some of the performance penalties of a central storage, e.g., weak dictionaries, locking, and hashing. Noiai allow the application programmer to chose to let the state of an association be stored in the roles rather than in the singleton of the association declaration. This is declared as:

```
[Cardinality(From=Cardinality.Unique)]
class Managing : RoleAssociation<Person, Person, Managing >{}
```

RoleAssociation declares the same nested classes (FromEntitySet, FromEntityRef, ToEntitySet, and ToEntityRef) with the same signature as in BaseAssociation. However, the participating classes (Person) must declare roles – as the roles contain the actual storage of the state of the association.

```
class Person{
  Employment.FromEntityRef Employer =
          new Employment.FromEntitySet( this );
  Managing.FromEntityRef Boss = new Managing.FromEntityRef( this );
  Managing.ToEntitySet Subordinates = new Managing.ToEntitySet( this );
}
```

Roles are declared the same way as for BaseAssociation roles, and they implement the same behavior. Changing from one representation to the other is as simple as changing the base class of Employment.

It is possible to access the role-stored associations through association-based syntax. Managing is declared a subclass of RoleAssociation, but it is still possible to add new bindings as Managing.Assoc.Add(John, Jim). This will update the roles in John and Jim.

3.5 *Compositions*

One of the appealing aspects of relations is that one can define operators to obtain new relations. Noiai has three such operators: Composition, Inverse, and Closure. Such derived associations are read-only in Noiai, and they are not top-level declarations, but rather the result of expressions.

Based on the Ownership and Employment associations it is possible to obtain a combined association representing the relation between the owner of a company and all the employees of the company. Ownership and Manages can be combined into a Governs association:

```
IAssociation<Person,Person> Governs
        = Ownership.Assoc.CombineWith( Employment.Assoc.Inverse());
```

IAssociation is a read-only interface for associations, exposing the GetFromSet and GetToSet projection methods discussed earlier. This can then be used as in Governs.GetToSet(Bill) which returns an IEnumerator<Person> of all the employees in the company owned by Bill. Notice, it is necessary to get the inverse of the Employment, as Employment is declared from Person to Company. The inverse is from Company to Person, which is necessary to combine with Ownership. Noaia implements these operators:

- ComposeWith, discussed above.
- Inverse, discussed above
- Closure. Construct the transitive (non-reflexive) closure of an association. The association must associate classes of the same type.

All three kinds of associations (Base, Sub, and Role) can be combined, that is, if Employment was defined as a RoleAssociation rather than a BaseAssociation, the above combination and inverse operations are still valid, and have the same semantics. As the example shows, the result of an association operation can be used as argument in further operations (an Inverse is used as argument for a ComposeWith).

3.6 *Subscribing to change*

The association relationship is particularly useful when implementing support for domain models. For example, they are likely to be useful when implementing a model in the MVC design pattern, or when implementing the role of subject in the observer design pattern. In both cases, being able to subscribe to changes is important. To accommodate this, all associations (including those obtained through composition) expose two .Net events. The event Added will notify when a linkage is added to an association. The event Removed will notify when a linkage is removed from an association. Both events are based on the following delegate type:

```
public delegate void
AssociationAction<FROM,TO>(FROM from, TO
to);
```

For example, the following will print whenever a new Employment takes place:

```
Employment.Assoc.Added +=
    delegate(Person p, Company c)
        { Console.Writeline(p.Name +
            " is now employed at " + c.Name); }
```

Composed associations cannot be modified directly, but their underlying associations will ultimately be a declared association, which can be modified. Changes to the underlying associations are propagated to the composed associations. Adding a single new linkage to an association might result in a large number of linkages being added in the closure (though the closure does not actually store all these linkages).

3.7 *Error checking*

There are a number of rules to be followed in the Noiai framework. For example, the from-type of a RoleAssociation is expected to declare exactly one role-field of type FromEntitySet or FromEntityRef. Similarly, the to-type must declare one field of type ToEntitySet or ToEntityRef. It has not been possible to get the type system of C# to perform such checks at compile time. Noiai checks such constraints at class-load time, and issues a load-time error if the role-fields are not declared correctly. The following aspects are checked:

- RoleAssociations. Roles must be declared in each of the two participating classes. In case a class plays both roles, two roles must be declared in that class.
- BaseAssociations. At most one role can be declared in each participating class. In case a class plays both roles, two roles can be declared.
- SubAssociations. The from-class, to-class, and attribute class must be subclasses or same as the corresponding classes in the super association.
- SubAssociations. If the cardinality of any super-association is specified as UniqueSub, all sub-associations must be UniqueSub as well.
- No associations can be made between associations. The associated classes cannot themselves be associations.

Associations between associations imply that a linkage object exists to partake in associations. Noiai has been designed not to expose linkages in order to enable a wider range of implementation strategies, and associations therefore cannot be associated themselves.

Checking these aspects at load-time causes some performance penalty of cause. However, a library designer cannot normally control the compilation of a library. In C++ some control can be gained at compile time through template meta programming. To a Java or C# programmer load-time checks allow some of the consistency checks to be moved from actual run-time to load-time. In addition, writing the load-time checks gives a pretty good idea of

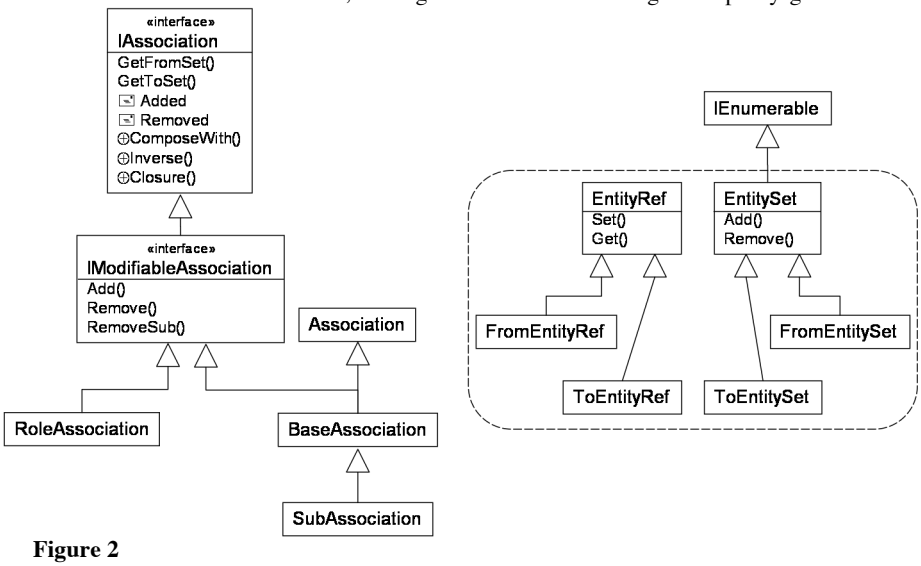

Figure 2

which aspects should be done at compile time – if one later decides to do a language extension.

3.8 *Summary*

The storage of an association is done either in the association singleton (BaseAssociation and SubAssociation) or in the roles (RoleAssociation). Both provide the same mechanisms for navigation and manipulation, and they are declared almost the same way.
Figure 2 shows the main classes of Noiai. IAssociation is an interface declaring read-only aspects of associations. The two events Added and Removed are marked with a small mail symbol (subscription). It is not possible to provide implementation of methods in interfaces. However, through the extension method construct in C# 3.0, it is possible to attach implementation of methods to interfaces. The interface IAssociation provides composition operators to all associations. The three compositions ComposeWith, Inverse, and Closure are marked with a circled plus to indicate extension method. RoleAssociation and BaseAssociation both implement the IModifiableAssociation interface. Association is an un-typed super-class used internally in Noiai to gain polymorphic behavior across different instantiations of BaseAssociation.
In addition, both RoleAssociation and BaseAssociation contain the same four nested classes FromEntityRef, FromEntitySet, ToEntityRef, and ToEntitySet. These are used for declaring roles in the participating classes. We have not seen an interface concept that states that a class should provide nested classes. In Noiai this is a merely a convention.

4 NOIAI IMPLEMENTATION

As stated earlier, run-time supported generics, load-time reflection, and easy load-time code generation all play a crucial role in the internals of Noiai. In addition, C# provides special support for writing lazy iterators. This enables a succinct implementation of many operations. This section will examine the details of the Noiai implementation to make clear where and how these language features were utilized.

4.1 *Creating a singleton*

All declarations of associations in Noiai are done by declaring a class that is a subclass of a selected Association kind (e.g., Base, Sub or Role). These association classes all take at least three types as parameters, of which the third is the class being declared, e.g:

```
[Cardinality(From = Cardinality.Unique, To = Cardinality.Unique)]
class Ownership : BaseAssociation<Person, Company, Ownership >{}
[Cardinality(From = Cardinality.Unique)]
class Employment : BaseAssociation<Person, Company, Employment,
Contract>{}
```

Each association is implemented as a singleton. The type of that singleton is supposed to be the domain type rather than BaseAssociation. In C# each instantiation of a generic class gets its own set of static fields. Passing the Employment class itself to the super class, as done above, allows a suitably typed singleton to be created for each association declared from Person to Company i.e., the singleton of Employment is different from that of Ownership. The construction of the singleton in BaseAssociation is done as follows:

```
class BaseAssociation<FROM,TO,THIS>
  where THIS : BaseAssociation<FROM,TO,THIS>, new()
  where FROM : class where TO : class
{
  private static THIS _Assoc;  // Field for storing the singleton
  public static THIS Assoc{ get{ return _Assoc;} } // C# property with getter
  static BaseAssociation(){
    _Assoc = new THIS(); // Eager construction at load-time
  }
}
```

The "static BaseAssociation" is a static constructor executed at the load-time of the class. The where-clause specifies that the generic parameter THIS must be a subclass of the present instantiation of BaseAssociation, and must have a parameter-less constructor. The run-time support for the generic parameter allows instantiation of the generic parameter THIS (e.g., Employment). The implementation of the singleton is eager – the singleton is created at load-time. Note in a language like Java, where generic types are erased, "new THIS" is not allowed. Factory objects would have to be defined by the application programmer. However, that would lead to boilerplate code in the declaration of the association.

4.2 *Establishing the association hierarchy*

Specialization of associations is implemented by each association singleton having a collection of its own linkages, and a list of its sub-associations. To enumerate all GetFromSet or GetToSet for a given association, first all linkages of the association itself are enumerated, and then those of the sub-associations.
Noiai establishes the run-time structures to support the association hierarchy at load-time. A global data structure is used for this. The hierarchy is represented as a Dictionary<Type, List<Association>>. It maps e.g., Employment to its list of sub-associations e.g., Consultancy.
This internal representation is illustrated in Figure 3. Association Employment has two sub-associations Consultancy and Voluntary (a new association for this example). Each association has a list of linkages E and a list of its sub-associations S. Each list of linkages E contains the association linkages of that association. The large oval illustrates all linkages in Employment, while the smaller embedded ovals represent the linkages belonging to the sub-associations Consultancy and Voluntary. To avoid further cluttering of the figure not every x has an arrow pointing to it. In the implementation of GetFromSet and GetToSet the entire extent of Employment needs to be enumerated. When enumerating Employment, the E-list from the association Employment itself is enumerated, followed by an enumeration of E-lists in all sub-associations, here E in Consultancy and Voluntary. Each association keeps an explicit list S of its sub-associations.

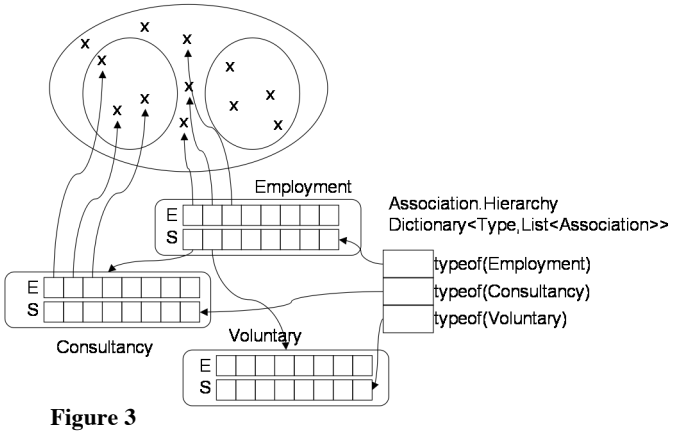

Figure 3

The S-list structure is established at load-time. When an association class is loaded, it inserts its S-list to the dictionary. This is done in the constructor of the singleton discussed in the previous section:

```
public BaseAssociation() : base(){
   Association.Hierarchy[typeof(THIS)] = this.SubAssociations;
                          // SubAssociations is the S-list
}
```

When Employment is loaded, a singleton Employment is created (see previous section). This singleton registers its S-list in the Hierarchy dictionary with the typeof(Employment) as key.
In class SubAssociation, the constructor of the singleton adds the sub-association singleton to the super-association singletons' S-list:

```
public class SubAssociation<FROM,TO, THIS, SUPER> :
                                    BaseAssociation<FROM,TO, THIS >
{
   public SubAssociation():base(){
      Association.Hierarchy[typeof(SUPER)].Add(this);
   }
}
```

When Consultancy (or Voluntary) is loaded, the Consultancy singleton is created. The constructor of the Consultancy singleton adds itself to the list of sub-associations of its super-association Employment. typeof(SUPER) is typeof(Employment). Hierarchy[typeof(Employment)] is the list of sub-associations of Employment, to which the Consultancy singleton is added. The run-time availability of the generic types (here THIS and SUPER), allows this entire structure to be built at load-time.

4.3 *Declaration checking*

Another usage of static constructors in the Noiai framework is load-time consistency checks. RoleAssociation requires the From-participant and To-participant to declare role-fields. The static constructor of RoleAssociation has access to the F, T, and THIS types given as parameter to the class. The reflected type object of F (e.g., Person) can be obtained using the run-time availability of generics through typeof(F). By examining all fields of Person using reflection, it is possible to ensure that there is exactly one field of the appropriate type. It is even possible to put together an understandable error message stating what is wrong if it is not the case.
If two associations are declared with Person as from-type e.g., Employment and Manages, the static constructor for Employment will look for fields of type Employment.FromEntitySet (or -Ref), while the static constructor for Management will look for fields of the type Management.FromEntitySet (or -Ref). In addition, it can be foreseen that a common error will be confuse To and From. Noiai can look for fields declared as Employment.ToEntitySet (or –Ref) in Person; in case any is found an error is issued.

4.4 *Nested classes*

Classes nested inside generic classes are implicitly parameterized by the same parameters as their enclosing classes, and are instantiated together with their enclosing classes. This allows Noiai to avoid role types to be explicitly parameterized. Thus, in class Person it is not necessary to state the employer role is of type Company, this is implicitly given by the type Employment.FromEntityRef.

```
class Person{
  Employment.FromEntityRef Employer;
  public Person(){
      Employer = new Employment.FromEntityRef( this );
  }
  ...
}
```

The FromEntityRef is known to refer to Company in the context of Employment. Unlike inner classes in Java, nested classes in C# do not have access to their enclosing object instance, but they do share the type parameters of the enclosing class.

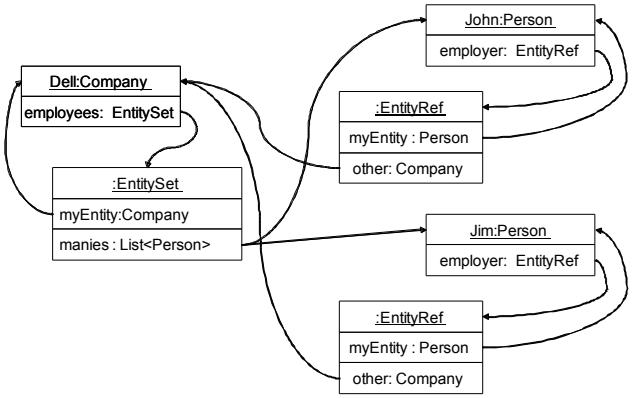

Figure 4

4.5 *Run-time code generation*

The implementation of the role-based associations is somewhat elaborate inside the abstraction, and is illustrated in Figure 4. A statement like John.employer.Set(HP) must first disconnect John from his employment at Dell. The Set operation is called on the topmost EntityRef object. First, it can be noticed that the EntityRef object needs a reference to John to be able to remove John from Dell's list of employees. This is possible through the field myEntity. This field is initialized by passing "this" as argument to the constructor (see example code in section 4.4). The company (Dell) from which John is to be removed is found in the other field in EntityRef. The list to remove John from is the manies list in the EntitySet stored in the employees field of Dell. After having removed John from this list, it is time to add John to the employees EntitySet in HP.
The problem is that there is no generic way to get to the right EntitySet or EntityRef of an object participating in an association. The field name is given by the domain programmer, and is needed in Noiai to keep the data structure consistent. Fortunately, load-time reflection establishes which field represents the right role (load-time reflection has already ensured that each of Person and Company has exactly one role-field of the appropriate type). However, accessing fields by means of reflection is expensive with respect to performance.
To eliminate the performance penalty of reflection, a delegate (C# for lambda expression) is created at load-time. Two delegates are created for each association, one that can map a from-type object to its corresponding EntityRef or EntitySet, and one that handles the to-type. Expression trees of C# 3.0 standardize the construction of abstract syntax trees for delegates, and make it convenient to compile abstract syntax into CIL byte code at run-time. The actual code to do this is five lines long.

4.6 *Composed associations*

C# 3.0 has a mechanism called extension methods. It allows a non-virtual method to be implemented outside a class, and it permits implementing methods on interfaces. All three kinds of associations (Base, Sub, and Role) implement the IAssociation interface. Extension methods enable the implementation each of the three compositions ComposeWith, Inverse, and Closure to be declared and implemented on the IAssociation interface. The definition for Closure is:

```
public static IAssociation<F,F> Closure<F>(this IAssociation<F,F> assoc)
where F: class {
      return new AssociationClosure<F> (assoc);
}
```

The keyword this in the signature indicates that the method Closure will be available on any IAssociation with the same type in the from

and to positions. The actual work is done in an auxiliary class named AssociationClosure:

```
internal class AssociationClosure<F> : IAssociation<F,F> where F: class {
    private IAssociation<F,F> original;
    public AssociationClosure(IAssociation<F,F> original){
      this.original = original;
    }
    public IEnumerable<F> GetToSet(F from){
      foreach(var t in  original.GetToSet( from )){
        yield return t;
        foreach( var tt in this.GetToSet(t))
          yield return tt;
      }
    }
    public IEnumerable<F> GetFromSet(F to){
      // similar
    }
     ...
  }
```

The class implements the IAssociation interface. The yield return construct of C# is quite important to keep the implementation of these two methods concise. The construct is especially designed to return a lazy implementation of an IEnumerable, and is essentially a simplified co-routine construct. When yield return is encountered, the method returns the result of the return expression. When asked for the next element, the method resumes immediately after the yield return, and runs until it next encounters a yield return. The GetToSet method above returns all the GetToSet of the original association, and then calls itself recursively to return the closure of the original association (assuming the association to be non-cyclic). The yield return construct has been very helpful in the construction of Noiai at many locations. Explicitly hand coding the corresponding construct without the yield return construct would be hard and error prone. Especially lazy enumerations over recursive structures are cumbersome to translate into plain C#.

4.7 *Garbage collection*

The association-based storage could potentially keep objects alive if nothing was done to address this issue. Internally, the linkage objects are not stored in a list, but in two weak dictionaries to allow fast lookup in the GetFromSet and GetToSet methods. Notice, both objects of a linkage should be dead before the linkage itself should die. Weak dictionaries build on top of weak references, which allow us to check if the referenced object is dead. It is thus possible to check this situation explicitly, and remove the linkage from the dictionary when both objects are dead. Thus, the infrastructure allows the right testing to be done. The problem is then when to examine the linkages for dead references. Noiai checks at all operations if a garbage collection has taken place. If so, at every 10th garbage collection, Noiai will traverse its structures and release dead objects (a weak dictionary need a policy for releasing its internal key-value pairs).

The role-based storage strategy does not have any issues relating to garbage collection as the linkages are stored and garbage collected together with the objects. Weak references are necessary if one wants to use association-based storage. Weak references as implemented in .Net platform are sufficient to do the job, and we expect the same construct on other platforms will do the job as well.

5 RELATED WORK

Linguistic support for associations has been proposed a number of times, but it has never made it into any mainstream object-oriented programming language. The first comprehensive work on language support was done by Rumbaugh and colleagues in late 1980's [Rumbaugh, 1987], [Rumbaugh, 1988], [Shah *et al*. 1989] describing a language named DSM. DSM supports not only binary but general N-ary relations with cardinality constraints for each element in the relation. It is a bit unclear to what extent cardinality constraints were implemented, but the syntax examples show a wide range of possibilities. In [Rumbaugh, 1988] propagation of operations along associations was discussed. The propagations patterns described in [Lieberherr, 1996] are to some extent an elaboration of this idea, though it is much further developed. In [Rumbaugh, 1987] a special case of tertiary relationships called Qualified Associations was introduced. These are not followed up in later work. The DMS language was implemented on top of C.

The semantic model of DMS was later implemented in Smalltalk, first described in [March & Rho, 1996] (later published as [March & Rho, 2000]). The malleability of Smalltalk enabled an implementation without any change to syntax of the language.

The ODMG standard [Cattell *et al*. 2000] has a role-based syntax for specifying associations. The simplicity of defining an association only in the roles seems at first very attractive. However, it has a number of drawbacks. There is no easy way to define attributes on the association. It does not allow for sub-associations. Associations are not first class objects, which means they cannot be passed as parameters, and operations such as compositions cannot be defined.

An excellent elucidation of the virtues of keeping an explicit representation of associations in the implementation is given in [Noble and Grundy, 1995]. The paper advocates the idea of representing associations as explicit objects at the implementation, and argues that this leads to code which is less cluttered, easier to maintain, and that explicit associations encapsulate behavior in a manner which actually reduces coupling among objects. The paper does not discuss the design of a general association library, but the rationale and insights presented in the paper are compatible with our motivations.

An association can very well be considered a cross-cutting concern, and it is hence natural to use an aspect oriented approach. In [Pearce & Noble, 2006] this is investigated in detail, and an Aspect/J implementation of their library RAL is described. To our knowledge, RAL is the only other comprehensive association library. Like Noiai, RAL can capture all three cardinality types (one-one, one-many/many-one, many-many); Noiai further strengthens this to support cardinality constraints for sub-associations differentiating between Unique and UniqueSub. Their specification style is akin to our association specification, that is, a single specification rather than one distributed in the roles. RAL, like Noiai, is able to provide role based as well as association based navigation. In RAL however, it is not possible to define domain specific role names like employer or employee. This is possible using Noiai's role specification. As with Noiai, the RAL library provides implementations that are based on either a central storage, or storage in the roles. Their notion of relationship polymorphism is the same as the one provided by Noiai through the IAssociation or IModifyableAssociation interfaces.

A crucial commonality is the singleton object behind their aspects and Noiai's association classes. In RAL, it is shown how this can be utilized to create methods that take an association as parameter. We use the same strategy for implementing the compositions of associations; composition is not discussed in their paper. In addition, Noiai supports events for addition and removal.

While the lack of explicit role names in RAL might seem a minor point, it does make it cumbersome if two classes are associated through different associations, e.g., Employment (Person to Company) and Owner (Person to Company). In RAL this is addressed by providing at set of generic relationship names, e.g., Relationship1, Rela-

tionship2, … etc. In our case, the issue does not arise. The trick of using the association class itself as the third parameter, and the run-time supported generic instantiation in C# ensure that each instantiation gets a unique singleton. Role names can be used to differentiate the two associations as well.

There is no generally accepted understanding of association specialization, e.g., the UML 2.0 standard is at best fuzzy on the issue. [Bierman & Wren, 2005] present a semantic model that allows association specialization. Their model is centered on two invariants:

"Invariant 1. *Consider a relationship r_2 that* ***extends*** *r_1. For every instance of relationship r_2 between objects o_1 and o_2, there is an instance of r_1, also between o_1 and o_2, to which it delegates requests for r_1's fields.*

Invariant 2. *For every relationship r and pair of objects o_1 and o_2, there is at most one instance of r between o_1 and o_2."*

This represents their view of association specialization, but not the one taken in Noiai. Invariant 2 implies that a person can be employed only once in the same company. At our university, some students are hired under several different contracts, and have as such multiple employments with our university, each carrying different salaries. Our example of Employment and Consultancy has led us to the conclusion that one should not expect to get the same salary independently of being permanently hired or hired through consultancy, though we do expect a salary to be involved in both cases. This is contrary to their invariant 1.

Choosing between the two approaches is better done by the library user than the library designer. However, Noiai does not currently support the semantics proposed by [Bierman & Wren, 2005] (though we see no major obstacle in implementing an association class which implements invariants 1 and 2, and implements the necessary methods of IModifyableAssociation). Indeed, we see a library approach as a more flexible way to provide semantics to associations; it is easier to provide a new kind of association to a library than to change a language.

In [Balzer *et al*. 2007] the notion of relationships are also presented in the form a language proposal. In addition to the constructs proposed by [Bierman & Wren, 2005], they focus a construct they call member interposition, and specification of invariants. Member interpositions are interesting and very useful. However, we believe that the solution lies not in the limited form proposed in their paper, but in the more general notion of roles, which we proposed in [Kristensen & Østerbye, 1996]. In Noiai, member interposition can be simulated by making association specific subclasses of the endpoint classes. E.g.,

```
public class Employment : RoleAssociation<Person, Company, Employment> {
      public class EmployeeRole : FromEntityRef {
         public string UnionName = "Fist";
         public EmployeeRole(Person p) : base(p) { }
      }
}
public class Person
{
      public Employment.EmployeeRole Employer;
      public Person(){
         Employer = new Employment.EmployeeRole(this);
      }
}
```

As part of the declaration of the Employment association, we specify that we want to store the name of the union in which the person is employed. Unfortunately, this strategy only works in connection with the explicitly stored roles, as the endpoints are there instantiated by the application, an issue also identified by Balzer *et al*. Further, making specialized endpoints makes the consistency checks harder, as we now must check for the existence of FromEntitySet or subclass thereof. Nevertheless, Balzer *et al*. has identified an important issue in member introspection.

6 CONCLUSION

It should have been nice to compare Noiai to a set of existing object oriented association libraries, but except for RAL, none seem to exist, neither in the scientific literature, nor as part of the distribution in the major languages, nor in the public domain. Perhaps the implementation details discussed in section 4 give some indication as to why. To realize Noiai, several key aspects of C# and the .Net run-time were utilized.

First, generic types must exist at run-time. The Noiai implementation crucially depends on them not being erased as in Java. Several implementation details utilize the fact that one can convert a type parameter into the corresponding Type object. Type objects serve as key in tables, and as root for reflection. Second, nested classes of generic types must be instantiated together with their enclosing type. Third, run-time code generation is used to avoid a performance penalty. Fourth, to get compositions to work for all three kinds of associations, it was very useful to utilize the C# 3.0 notion of extension methods.

The design of the Noiai library contributes in two areas. As mentioned above, it is the first published example of an object-oriented library that has extensive support for associations. Second, the implementation strategy of doing library specific checking at load-time, using reflection based on generic type parameters, is useful in other situations as well.

Incorrect usage of a library should be noted as soon as possible, and at the abstraction level of the library. Compile-time checking requires the consistency checks of the library to be embedded into the type system of the language, or to change the compiler. In our experience, the type system cannot capture all incorrect usage and reports errors at too low a level of abstraction. Load-time analysis can report errors at the abstraction level of the library, and captures a broader range of errors than normal type systems. In addition, load-time code is written in the same programming language as the library itself, a language assumed well known to the library developer. Load-time analysis has the obvious drawback that it gives the program slower startup time. However, in the final product the load-time checks can often be disabled. Note that load-time checks are easily tested, as errors are concentrated in the beginning of the code. Hence, the library user can be sure that once the program starts to execute, no more inconsistencies will arise. After the program passes these tests, the checks can be turned of.

Preliminary experiments show that a similar library can be built for Scala [Scala, 06], as Scala has most of the necessary language constructs found in C#. Scala has no simple model for run-time code generation, but it is possible to achieve a reasonable usage syntax with lambda expressions only. Our preliminary experiments with C++ are less clear. There is no load-time execution time, but the role-based storage does not require that. The template instantiation semantics for C++ also allow the data structures discussed in section 4.2 to be established. The C++ "pointer-to-member" construct is useful for specifying the inverse roles as template parameters, placing the declarations of the roles in the association declaration, removing the need for load-time code generation. It is less clear if the template meta programming techniques can be used to check all the constraints discussed in section 3.7, and if it will be possible to produce legible error messages at the level of associations. In addition, C++ is a single-pass language, and associations are notoriously interdependent. This leads to massive amounts for forward declarations, both internally in the libraries, and more importantly in the applications of the libraries.

Some libraries change the style of programming. It is too early to say if that is the case with Noiai. While associations are different from doubly linked lists, role-stored one-one associations make the implementation next to trivial. Two roles predecessor and successor in a class Linkage : RoleAssociation<Node,Node,Linkage> association give the basic structure, and provides strong support for consistency. The error prone removal operation for linked lists can be coded as:

```
public void Remove(Node n){
    Linkage.Assoc.Add( n.predecessor.Get(), n.successor.Get() );
}
```

The consistency preserving Add will remove the links to the present node n and splice the two ends together.

Acknowledgements

Thanks, to all who have given feedback and encouragement, both on the work and on the preliminary versions of this paper, in particular Sibylle Schupp and Liam Peyton.

7 REFERENCES:

[Albano *et al.* 1991] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. *A relationship mechanism for a strongly typed object-oriented database programming language*. In Proceedings of the Seventeenth International Conference on Very Large Data Bases (Barcelona, Catalonia, Spain, 3rd--6th September 1991)

[Balzer *et al.* 2007] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster: *A Relational Model of Object Collaborations and its Use in Reasoning about Relationships*. In ECOOP 2007, pp. 323-346. Springer, 2007.

[Beck & Cunningham, 1989] Kent Beck and Ward Cunningham. *A Laboratory For Teaching Object-Oriented Thinking*. In proceedings of OOPSLA, 1989

[Bierman & Wren, 2005] Gavin Bierman and Alisdair Wren. *First-Class Relationships in an Object-Oriented Language*. Proceedings of ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005.

[Cattell *et al.* 2000] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann Publishers Inc,US, ISBN: 1558606475.

[Génova *et al.* 2003] Gonzalo Génova, Carlos Ruiz del Castillo and Juan Llorens. *Mapping UML Associations into Java Code*. Journal of Object Technology, Vol. 2, No. 5, September-October 2003. http://www.jot.fm.

[Goldberg & Robson, 1989] Adele Goldberg and David Robson. *Smalltalk-80, the Language.* Addison-Wesley Publishing Company, 1989.

[JavaSDK 5.0] JavaTM 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5.0/docs/api/, Accessed December 2006. [Kokholm & Sestoft, 2006] Niels Kokholm and Peter Sestoft. The C5 Generic Collection Library for C# and CLI. Technical report of IT University of Copenhagen, TR-2006-76. http://www.itu.dk/research/c5/Release1.0/ITU-TR-2006-76.pdf.

[Kristensen & Østerbye, 1996] Bent Bruun Kristensen and Kasper Østerbye. *Roles: Conceptual Foundation and Practical Usage in Analysis, Design and Programming*. Theory and Practice of Object Systems (TAPOS), Vol. 2, No 3 1996 Pages 143-160

[Lieberherr, 1996] Karl J. Lieberherr. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996.

[Linq, 2007] The Linq project. http://msdn.microsoft.com/data/ref/linq/.

[March & Rho, 1996] S. T. March and S. Rho. *A Semantic Object-Oriented Data Access System*. http://www.misrc.umn.edu/wpaper/wp96-05.htm. (accessed at June 12th 2006).

[March & Rho, 2000] S. T. March and S. Rho. *A Semantic Object-Oriented Data Access System*. Information Systems, vol. 25, No 1, pp 23-41, 2000.

[Noble and Grundy, 1995] James Noble and John Grundy. *Explicit relationships in object-oriented development*. In proceedings of TOOLS 18, Melbourne, 1995.

[Noble 1995] James Noble. *Basic relationship patterns*. In EuroPLOP Proceedings, 1997.

[Pearce & Noble, 2006] David J. Pearce and James Noble. *Relationship Aspects*. In Proceedings of the ACM conference on Aspect-Oriented Software Development (AOSD'06), pages 75-86, March 2006.

[Rumbaugh, 1987] J. Rumbaugh. *Relations as Semantic Constructs in an Object Oriented Language*, Proceedings of OOPSLA'87. Pages 466-481.

[Rumbaugh, 1988] J. Rumbaugh. *Controlling Propagation of Operations using Attributes on Relations*. Proceedings of OOPSLA'88. Pages 285-296.

[Scala, 06] Scala programming language. http://scala.epfl.ch/.

[Shah *et al.* 1989] A. V. Shah, J. Rumbaugh, J. H. Hamel, and R. A. Borsari. *DSM: An Object-Relationship Modelling Language*. Proceedings of OOPSLA'89. Pages 191-202.

[STL, 1994] Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/index.html. Accessed December 2006.

[Østerbye, 1999] Kasper Østerbye. *Associations as a Language Construct*. In Proceedings of TOOLS 29, Ed. Richard Michel *et. al*., Nancy, June 7-10 1999. Pages 224-235.