Language support for reusable associations and static aspects in mainstream OO languages (a proposal)

Jiri Soukup, Code Farms Inc. (codefarms.com),
Martin Soukup, Nortel (msoukup@nortel.com),
February 2, 2009

ABSTRACT

Languages such as C++, C#, and Java provide generic collections, but do not support generic associations of several participating classes such as many-to-many, general graphs, finite state machine, intrusive data structures and design patterns. This increases code maintenance. affects code reliability, and is the source of problems with mapping between the source and its UML representation. The implementation of generic associations requires insertion of members and/or methods into participating classes – a concept similar to Aspects. If we add two simple keywords (*insert, myOwn*) to the existing languages, generic associations and static Aspects become a part of the core language. The idea has been proven on a variety of complex industrial applications during the past 18 years. The code generator, which essentially does what we would have to add to the language compiler, is only 500 lines long.

1. EXISTING SITUATION

If our OO languages provided such a feature, class libraries such as Java Collections or C++ Standard Template Library could store generic bi-directional associations, more efficient graphs, and design patterns. UML code generators would be much simpler, because there would be simple one-to-one mapping between the class diagram and the code. It would also be possible to derive UML class diagrams safely from the code. And static aspects would not be needed, because the language itself would support the transparent insertion.

There are many problems in today's programming which, on the surface, appear completely unrelated. Yet they have a common root – they require the cooperation of two or more classes, and this cooperation requires a transparent insertion of members and methods into application classes. All this should be triggered by an invocation of a generic class from some new, more flexible class library just as when we use generic collections today.

1.1 Two types of data structures

In general, there are two types of data structures:

(A) Those that require the addition of references, arrays, or other members to just one class;
(B) those that require such addition to two or more classes (*intrusive* data structures, see [6]).

All STL classes, Java and C# collections are of type A, because neither of these languages has a mechanism for the coordinated insertion of members into several classes. Most examples in this article use Java, but with minor syntax modification they apply to all the three languages.

Type A data structures are usually implemented with collections, while type B are sometimes called *intrusive data structures*, because they introduce members into more classes than one.

1.2 Bi-directional associations

As reflected in UML, we often need bi-directional associations. By definition, any bi-directional association must insert references into two or more classes, and is therefore of type B. The following three examples use Java notation:

```
// bi-directinal OneToMany
class Parent {
    Hash<Child> children;
}
class Child {
    Parent parent;
}

// directed Graph
class Node {
    Edge first;
}

class Edge {
    Node to; // see[1]
    Edge next;
}

// ManyToMany association (bi-directional)
class Source {
    Hash<Link> links; // see[2]
}

class Link {
    Source from;
    Target to;
}

class Target {
    Hash<Link> links; // see the same footnote
}
```

1.3 Intrusive data structures are sometimes better than existing collections

For example, one of the most frequently used data structures, OneToMany uni-directional association, can either represent a *bag* or a *set*. In a bag, the same child can be referred to several times. In a set, there can be at most one reference to any child. A set can be implemented either as type A or as type B:

```
// implementing a Bag with Java Collection (style A)
class Parent {
    Collection<Child> children;
}

// implementing a Set, with Java or STL Hash (style A)
class Parent {
    Hash<Child> children;
}
// before adding a Child, we must search the hash table
```

[1] Edges leaving the given node form a *set*.
Regardless of how we implement this *set*, the presence of reference 'to' implies type B.

[2] Our preferred implementation would be pointer chains, but as you see here, even when using Java Hash, this still leads to type B.

```
// whether the Child is already there

// the intrusive implementation is a natural Set (style B)
class Parent {
    Child head; // to the ring of Children
    Parent(){head=null;}
}

class Child {
    Child next; // ring of children, not null ending list
    Child(){next=null;}
}
// simple test if(next==null) tells us whether a Child
// is already there
```

1.4 Storing design patterns in a library

Structural design patterns are really a combination of associations with inheritance. If we can expand Java Collections to include general associations, we could also store reusable design patterns in the same library. The concept has been proven by the Pattern Template Library (PTL, this is a C++ library, see [7]), which contains patterns *Composite*, *Flyweight, and FSM (Finite State Machine).*

1.5 UML class diagram inside the code

If all the *Association* statements are kept together, possibly in one file, we can consider them a textual form of the UML class diagram. Association bevome first class entities just like classes, and the programmer can always instantly see the relations among the classes. Converting these diagrams into the UML diagram is trivial – each Association line describes one relation in the diagram. The UML code generator also becomes trivial: All it has to do is to generate a set of Association lines. Everything else happens automatically and transparently.

This style of coding dramatically decreases the complexity of the code, and has been one of the main reasons for the success of the Code Farms' libraries ([1],[2],[7]).

1.6 Aspects

A controlled insertion of members is the central idea of Aspects, which are becoming increasingly popular. If the core language provides a simple way of inserting members, it would automatically provide the functionality of static aspects, but without even introducing the term 'aspect'.
This would be a significant improvement of the language, and obtained with the minimum effort.

On the other hand, we would not recommend to introduce dynamic aspects into the core language. Besides a major addition to the compiler, it would increase the code complexity beyond the level we consider acceptable.

A library of generic associations has been already developed with AspectJ (see [4] and [5]). The library still does not work in some special cases, but this is just due to the existing bugs in AspectJ [2]; its internal implementation is simple and elegant.

Here is the intrusive implementation of class Aggregate (bi-directional association OneToMany), implemented with AspectJ. Note note that classes AggregateParent and AggregateChild are not visible outside of the Aggregate class.

```
public abstract aspect Aggregate<Parent,Child> {
    public static interface AggregateParent {}
    public static interface AggregateChild {}

    declare parents : Child implements AggregateChild;
    declare parents : Parent implements AggregateParent;

    private Child AggregateParent.head = null;
    private Child AggregateChild.next = null;
    private Parent AggregateChild.parent = null;

    public static void addHead(Parent p, Child c) {
        if(p.AggregateParent.head!=null){
            c.AggregateChild.next=p.AggregateParent.head;
        }
        else c.AggregateChild.next=NULL;
        c.AggregateChild.parent=p;
        p.AggregateParent.head=c;
    }
    ...
}
```

Application code using association Aggregate:

```
public class Department {...}  // same as if not using associations

// declaration of associations
aspect departments extends Aggregate<Company,Department> {};
aspect employees extends Aggregate<Department,Employee> {};
aspect boss extends OneToOne<Department,Employee> {};

// USING THE ASSOCIATIONS
Department d; Employee e;
employees.addHead(d,e);
```

In cooperation with Olaf Spinczyk, author of AspectC++, we attempted to implement reusable associations – see [3], but the experiment failed because AspectC++ cannot handle templates as parameters. I checked with Olaf recently, and it is still the case.

The C# library of associations Noiai [8] is advanced in many ways, but it uses a combination of features not available in Java: Generic types with runtime type instantiation, runtime reflection on type parameters, annotation of classes, and runtime code generation. Its weakness is that besides declaring the associations, the application must explicitely insert members (roles) into the participation classes – essentially spreading the design through the applications classes instead of having it in one place.

1.7 Experience with generic associations

Code Farms (www.codefarms.com) have been building and selling C++ libraries of reusable associations since 1989. The most recent of these libraries is also available in Java.These libraries have been used with great success on many complex industrial projects, and most of the source including the documentation is free to download [7]. For additional discussion of these and other libraries, see [1] and [2] . Most Code Farms libraries use simple code generators which do not modify the original code – they only generate classes for the objects to be inserted into the participating classes.

Data Object Library (DOL, since 1989) has been the workhorse of Code Farms. It has an option which makes all the application classes and their associations automatically persistent, and it supports rapid

design of efficient memory resident databases. Because of the persistency, the logic of the code generator is more complex, and the entire library depends heavily on the use of C macros.

Pattern Template Library (PTL, since 1996) provides both associations and design patterns, but the internal design is completely different. The required pointers and collections are inserted through multiple inheritance. The code generator which is called *Template Manager* is under 500 lines of code; it only assembles the inheritance statements.

IN_CODE modeling library (ICML, since 2005) was developed as a proof that a library of reusable associations can be designed both in C++ and in Java, applying the same interface and a similar, almost identical code generator. As in DOL, the insertion is done through members, but ICML has a clean, modern internal design.

1.8 Inside the IN_CODE modelling Library (ICML)

ICML is based on similar principals as DOL and PTL. Let's explain its internal workings on the example of the most frequently used bi-directional association, composition *Aggregate,* which is a *set.* In this example, we will assume an intrusive implementation (style B) which we discussed in Section 1.3. The result must be a generic, reusable class (or classes) which we can store in a library. The following code is in C++.

Let's assume that we want to use this relation between application classes Department and Employee. In ICML, this is what you do:

```
class Department {
    ZZ_Department ZZds;
    ... // everything else as usual
};

class Employee {
    ZZ_Employee ZZds;
    ... // everything else as usual
};

Association Aggregate<Department,Employee> employees;

// working with this relation
Department* d=new Department;
Employee*   e=new Employee;
employees.add(d,e);
employees.remove(d,e);
```

The library also provides an interator, and other useful functions that control this relation.

The ZZ… statement (shown in blue) must be mechanically added to each class which participates in any relation. Only one statement is needed, regardless in how many relations the class participates.

The line starting with keyword *Association* declares the relation and the roles the application classes are playing. It does not instantiate any object.

Lets look at what happens under the hood. The code generator replaces the Association line by

```
typedef class employees_Aggregate<Department,Employee> employees;
```

where Aggregate is a class from a special library which, for each data organization, also keeps one participating class for each role. Beside being regular templates, these classes are also parametrized by symbols $$ and $0:

```
template<class Parent,class Child> $$_Aggregate {
    void add(Parent *p,Child *c){
        if(c->$0.next)... // error, already used
        Child* h=p->$0.head;
        if(h){c->$0.next=h->$0.head; h->$0.next=c;}
        else c->$0.next=c;
        p->$0.head=c;
    }
    ...
};

template<class Parent,class Child> $$_AggregateParent {
    Child *head; // head of the list formed as a ring
};

template<class Parent,class Child> $$_AggregateChild {
    Child *next; // ring, not 0-ending list
    Parent *parent;
};
```

The code generator modifies these classes by replacing $$ by *employees,* and $0 by *ZZds.ZZemployees*. It also generates the following classes:

```
class ZZ_Department {
    employees_AggregateParent<Department,Employee> ZZemployees;
};
class ZZ_Employee {
    employees_AggregateChild<Department,Employee> ZZemployees;
};
```

If Company had multiple Departments, and the Department kept two sets of Employees: all *employees*, and separately those ready for *promotion*, then we would have

```
class ZZ_Department {
    employees_AggregateParent<Department,Employee> ZZemployees;
    promotion_AggregateParent<Department,Employee> ZZpromotion;
    departments_AggregateChild<Company,Department> ZZdepartments;
};
```

As you can see, the logic of what the code generator does is quite simple. It consist of 500 lines of code, and it uses itself to manage its own data structures.

More complex library classes can be derived from simple ones through inheritance, just as it is done in the existing collection libraries.

This is a simple, pragmatic implementation; the following proposal replaces some steps by more appropriate object-oriented features. Note that the sole purpose of the $$ substitution is to prevent conflict between data organizations with the same parameter classes, such as *employees* and *promotions* in the

example above. The $0 substitution binds the participation classes with the class which controls the relation, e.g. class *Aggregate* above.

2. PROPOSAL:

Here we propose an addition of two keywords (and of two simple concepts) to the existing object-oriented languages. The result will be ability to build generic libraries of bi-directional associations, intrusive data structures, and design patterns. Also, static aspects will become a part of the language. For over a decade, libraries based on this idea (see Part 1) and Aspects in general have been successfully used in a variety of application.

2.1 New keywords

Within the context of any class or interface (R)[3], keyword *insert* will transparently add the specified members to a given class, S, and will also remember how to access them:

```
class R {
    insert S memberType1 memberName1;
    insert S memberType2 memberName2;
    ...
}
```

Only class R will have access to this member, which will be through keyword *myown*.
In Java

```
class R {
    S s;
    s.myown.memberName1.foo1();
    ...
}
```

The C++ syntax would simply use pointer arrows instead of dot in such expressions.

Note: A smart compiler may automatically insert *myown* before any invocation of members declared in the *insert* statements, thus completely eliminating the need for the second keyword, *myown*.

2.2 Example of using the new features for generic associations

Let's code a reusable Aggregate class in a style similar to the AspectJ implementation (Section 1.6), except that only normal Java generics are used – no Aspects are involved.

```
abstract interface Aggregate<Parent,Child,REL> { // for REL, see[4]
```

[3] Whether this class implements association, aspect or design pattern is irrelevant.

[4] Parameter REL allows the same association to be used several times. Instead of using the $$ substitution, we use several dummy (empty) classes REL0,REL1,REL2,…, for example

For example:

```
Aggregate<Department,Employee,REL2> research;
Aggregate<Department,Employee,REL1> admin;
Aggregate<Department,Employee,REL2> manufact;
```

In C++, we can use an integer parameter, i

```
template<class Parent,class Child,int i=0> class Aggregate {...}
```

which some of us may consider less elegant

```
Aggregate<Department,Employee> research;
Aggregate<Department,Employee,1> admin;
Aggregate<Department,Employee,2> manufact;
```

```
    insert Parent Child head;
    insert Child Child next;
    insert Child Parent parent;

    // add c as the head of the ring under p
    public static void addHead(Parent p,Child c) {
         if(p.myown.head){
             c.myown.next=p.Par.head;
         }
         else c.myown.next=NULL;
         c.myown.parent=p;
         p.myown.head=c;
    }
    ...
}
```

Using Aggregate in an application:

```
public class Department {...}  // same as if not using any associations

// declaration of associations
interface departments implements Aggregate<Company,Department,REL0>;
interface employees implements Aggregate<Department,Employee, REL0>;
interface boss implements OneToOne<Department,Employee,REL0>;

// using the association
Department d; Employee e;
employees.addHead(d,e);
```

2.3 Associations and existing class libraries

Reusable associations will be stored in the existing libraries such as C++ Standard Template Library, Java or .NET Collections, without any modifications to the existing classes. Existing classes (uni-directional associations) will be a special case of the new, bi-directional implementation. Iterators for the new associations can be coded in the style used currently for collections. Reusable design patterns will also be stored in these libraries, just as PTL does it already (see Section 1.4).

It will be easy and straightforward to derive new complex associations (or other data structures) from the existing simple ones. For example, the bi-directional many-to-many association will be derived from two Aggregates: [5]

```
abstract interface ManyToMany<Source,Link,Target,REL>
    implements Aggregate<Source,Link,Rel>, Aggregate<Target,Link,REL> {

    // add a link between the given source and target
    public static void add(Source src,Link lnk,Target tar) {
         Aggregate<Source,Link,Rel>.addHead(src,lnk);
         Aggregate<Target,Link,Rel>.addHead(tar,lnk);
    }
    ...
}
```

Using ManyToMany in an application:

[5] This is all Java code; in C++, classes will replace interfaces, and we will have multile inheritance.

```
public class Student {...}  // same as if not using any associations
public class Course {...}   // same as if not using any associations
public class InCourse (
    int mark;
}
class Rel {} // dummy class

// declaration of associations
interface isTaking implements ManyToMany<Student,InCourse,Course,Rel>;
>;

// using the association
Student s; InCourse ic; Course c;
isTaking.add(s,ic,c);
```

3. Legal Issues

The publication of this proposal is essential. Key players such as Microsoft refuse even to look at it unless it is published and thus in the public domain.

4. References

[1] Soukup M., Soukup J.: *Reusable Associations*, Dr. Dobb's Journal, Nov.2007, pp.51-56.

[2] Report from the OPPSLA 2007 workshop: *Implementing Reusable Associations/Relationships*, Montreal, Oct.22, 2007

[3] Soukup J., Pearce D.J., Soukup M., Noble J., Nelson S.: *Reusable Associations with Aspects*, article submitted to Dr. Dobb's Journal

[4] Nelson S., Pearce D.J., Noble J.: *First-Class Relationships in Object Oriented Programs*, University of Auckland Software Engineering Workshop (SIENZ) 2007.

[5] Pearce D.J., Noble J.: *Relationship Aspects*, AOSD 06 conference, March 20-24, 2006, Bonn, Germany

[6] Soukup J.: *Intrusive Data Structures*, C++ Report Vol.10 (1998), in three parts: No.5 (May) pp.22-27, No.7 (July/August) pp.22-28, No.9 (October) pp.28-32.

[7] Pattern Template Library (PTL), Data Object Library (DOL) and In-Code Modeling Library (ICML), for User Guides and free downloads see www.codefarms.com/products.htm

[8] Osterbye K.: *Design of a Class Library for Association Relationships,* ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07), at OOPSLA'07, Montréal Oct. 21-25, 2007.