Associations as a Language Construct

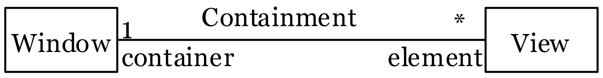
Kasper Østerbye
Norwegian Computing Centre,
Kasper.Osterbye@nr.no

Abstract

A cornerstone in the object-oriented paradigm is the abstraction mechanisms transcending analysis, design, and implementation. The notions of class, object, behaviour, and inheritance are supported in both design notations and through programming language constructs in object-oriented languages. An association is a declarative mechanism to associate objects, and it is well known from analysis and design notations. However, associations are not matched by a programming language construct. This paper eliminates the semantic gap between design and implementation by proposing a programming language construct to support associations directly. An important issue in designing such a language construct is to make it as efficient and flexible as a manual translation to pointers and containers, and that it gives added benefits compared to a manual translation. Our association compiler can generate code that is as efficient as hand produced code, and it is our experience that it is easier to make changes to program structure and implementation when using language supported associations.

1. Introduction

The history of programming languages can be seen as the story of introducing new and more powerful abstraction mechanisms and trying to devise better and more efficient implementations for those mechanisms. The analysis and design notation UML [18] depicts associations as shown below.



Here the association named Containment associates one window object to many view objects. The views play the role of elements in the containment association, whereas the window is the container.

The aim of this paper is to demonstrate that the association specified above can be expressed as a programming language construct that can be efficiently compiled. An association construct will narrow the semantic gap between design and implementation, in the same way as the inheritance mechanism of programming languages is well suited to realise the specialisation hierarchies used during analysis and design.

There are many interpretations of what an association represents at the modelling level. However, this paper will not provide a taxonomy of associations. Using the terms of [6], this paper addresses referential associations between nodes of the same meta-type. [2][3][5] discuss in the literature whether an association is uni-directional or bi-directional. [5] in particular argues strongly for the uni-directionality of associations. However, there seems to be no consensus, which has led us to design the language construct to support both kinds of directionality.

The introduction of associations as a replacement for hand coded usage of pointers and containers are expected to be beneficial for several reasons.

- The compiler can produce access methods and choose appropriate implementation strategies depending on the type of association. This will produce safer code than what is normally produced by human translation, pointer errors are well known and can be hard to debug.
- Associations can provide two-way navigation while ensuring consistency. They make sharing explicit, or prevent it altogether depending on their cardinality constraints; sharing is a common source of programming errors.
- Providing associations for programming languages will narrow the semantic gap between analysis and design where associations are now well established.
- In current object-oriented programming languages, aggregation is only supported through static references. In object-oriented analysis and design notions, aggregation has evolved to become a rich notion, which has no direct support in programming languages.
- Explicit association declarations in the source code make re-engineering to UML easier.

In general, a binary association is a rather complex data structure, which allows objects to be associated to several other objects, it might be navigated in both directions, and it can have attributes. However, often a concrete association is not so general, but is only one-to-many or unidirectional, or it carry no attributes. An important part of this paper is to examine how a compiler can choose the best implementation based on the characteristics of the association.

There are several ways to implement associations, which can be grouped into two broad categories: internal and external implementations. A representation where the data structure is kept local to an object is called an internal representation. An external representation is an implementation where the data structure representing the association is kept outside of the associated objects. Both categories cover a number of variations, from pairs of hash-tables to simple references. The compiler should be able to use both types of representations.

There are two distinct styles of navigating (or accessing) an association. Either one navigates the association directly, as in `Containment.getContainer(aView)`, or one navigates using role names on the objects, as in `aView.getContainer`. Navigation choice is independent of representation, since role based navigation can be syntactic sugar for association-based navigation, and vice versa.

Using associations as a semantic construct in object-oriented programming is not a new idea. It was first proposed by Rumbaugh [14] in the programming language DSM, and further described in [15] and [16]. This paper shows how associations can be compiled efficiently, and demonstrates that the analysis can be operational. The contribution of [14] was to lay the conceptual groundwork, showing that associations can be incorporated into object-oriented programming languages. Section 2 presents examples of using associations. Section 3 describes how associations can be efficiently realised depending on their characteristics, which is the main contribution of this work. The paper is concluded with a discussion of related work (Section 4), further work (Section 5), and summary of the results (Section 6). The association compiler is implemented for Smalltalk. The implementation will be discussed in Section 3.4.

2. Examples

This section will discuss two examples of associations. The `Containment` association between windows and views as presented in the introduction will be used to introduce the basic concepts. Then doubly linked binary search trees will be discussed, showing the more advanced aspects of associations. The prime motivation for this work has been to narrow the semantic gap between design and implementation by providing linguistic support for associations. However, we believe that associations are a useful language construct in their own right. Both represent a usage of associations that are not grounded in domain modelling, but demonstrate the power of associations in programming in general.

2.1. Window - view containment

The containment association between a window and its views is a prototypical situation where it is useful to have two-way navigation, enabling navigation from a window to its contained views, and from a view to its window. The monitoring association between subjects and observers in the observer pattern from [4] is another example where a two-way navigational association which associates one subject to a number of observers is useful.

In our Smalltalk association compiler, Containment is declared as:

```
Assoc named: #Containment
   relating: #( one container Window)
   and: #( many element View)
```

Here the association between window and view is named Containment. The objects of type Window play the role of container, and View objects play the role of element. Compiling the containment association creates several access methods in Window and View. Navigation from a view object to the window object is done as aView getContainer. This returns the unique window object that contains the view. One window object can contain several view objects, and aWindow getElement returns a set of view objects. This style of access is called *role-based* access. The association compiler implements the containment association itself as a meta-class (using meta-classes is just a way to implement the Singleton pattern [4] in Smalltalk), and it also generates access methods on this class. To navigate from a view to its window can also be done using the *association-based* access style Containment getContainer: aView. Both styles of access have their advantages, and the association compiler presently generates both.

To set the window of a view can be done as aView setContainer: aWindow. If aView were contained in another window, this association will be removed before the dependency of aWindow is established. This is because it is declared that a view can only be contained in one window (the container role has a cardinality of one). It is possible to reverse the roles: aWindow addElement: aView will have the same semantics, that is, removing aView from any previous dependencies, and associating aWindow and aView. Depending on the cardinality of the roles, the method name is either setRole (cardinality one), or addRole (cardinality many). The corresponding association-based access method is Containment addElement: aView toContainer: aWindow, which is equivalent to the other two mentioned above.

When a window needs to be re-drawn, it will ask all of its views to re-draw themselves. The getElement method generated for Window will return a collection of the views in the window. Because Smalltalk handles collections very well, we have not felt a need to provide access methods that can provide indexing, enumeration, or other similar functionality for associations. Instead, we rely on the standard collection methods to do so. In the following re-drawing method for Window, the do: method enumerates all elements in a collection, binding them to element one at a time.

```
reDraw
   "re-draw all my views"
   self getElement do:[:element | element reDraw]
```

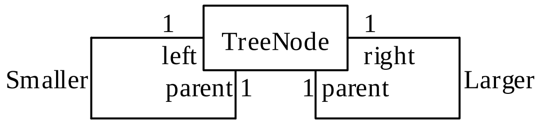
2.2. The binary search tree

Another example is a binary search tree. The idea is that the association could improve on traditional implementation techniques in two ways.

- In a traditional implementation, each node has a left and a right pointer. If there are N nodes in such a tree, there will be N+1 nil pointers. Let us assume that space is more important than time and we prefer an external representation for left and right sons.

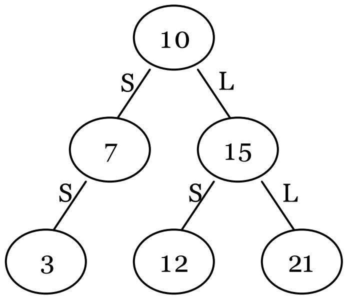
- Traditionally binary trees have only downward pointers, so one has to start from the root to remove a node. The association compiler can produce two-way associations, and in a doubly linked tree, a node can be removed by sending it a message to remove itself.

The structure of the tree is defined by the following specification, which states that the Smaller association associates one object of type TreeNode playing the role parent, to one other object of type TreeNode playing the role left. The Larger association is declared similarly.



The first point is addressed by stating that the left and right roles are *external*, since the association compiler supports external as well as internal representations. When a role is declared external, a table based representation is created, which is accessed using the same access methods as would have been available with an internal representation. The definition of the Smaller association then becomes:

```
Assoc named: #Smaller
   relating: #( one parent TreeNode)
   and: #( one left TreeNode external)
```



The second point, however, is troublesome. For a tree node to remove itself from the tree, it must know whether it is in a smaller or larger association with its parent. Node 15 plays the parent-role for both 12 and 21. If node 12 is removed, node 12 does not know if its parent is larger or smaller, hence it does not know from which association to remove itself. There are two ways to find out, either by comparing with its parent or by storing its association with its parent. The problem is intrinsic to the notion of a doubly linked binary tree, and is not specific to associations.

If we chose not to compare with the parent node, there are several ways in which to represent the role of a node. One possibility is to create specialisations of TreeNode, a LeftTreeNode and a RightTreeNode. That way a node knows its role in relation to its parent. However, when a node is removed from a tree, the tree is restructured, and a node that was previously a left tree node might change to become a right tree node. While Smalltalk can change the class of an object at runtime, such a solution can not be used in other languages, and we would like to solve the problem using techniques that will transfer to other programming languages.

Another way is to use an association variable that remembers which association (Smaller or Larger) a tree node has to its parent. If this association variable is called parentAssociation, then the code needed to remove an object from the appropriate association becomes:

```
parent := parentAssociation getParent: self.
parentAssociation  removeParent: parent from: self
```

Because associations are first class, they can be stored in variables, passed as parameters etc.

Incidentally, the access method for removing associations only use the first role name (parent), so that the same method name can be used for removing left and right offspring. In general we have strived to provide specific access method names, e.g., the remove method should have read removeParent: parent fromLeft: self. However, when associations are stored in variables, it is necessary to access these associations in more general terms, e.g., removeRoleA: obj1 fromRoleB: obj2, where roleA and roleB are generic names for the first and second role. The compiler generates these generic methods for association based access, and the last line in the example above should read: parentAssociation removeRoleA: parent fromRoleB: self, as parent is the first declared role.

3. An association compiler

The purpose of this section is to provide an overview of the different implementation approaches for associations and to show in which situation each approach should be chosen. The results in this section are implemented in the Smalltalk association compiler, which is presented in section 3.4.

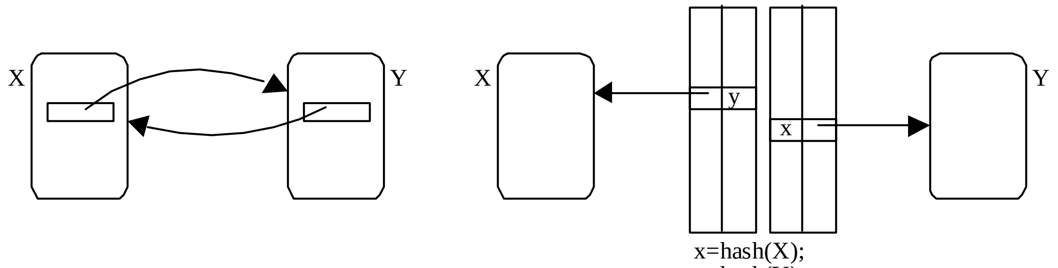
We have identified five orthogonal characteristics of associations that affect the choice of representation and navigation style. These will be discussed in detail in the next sections.

- *Cardinality*. If a role has cardinality of one, a simple reference can be used to point out the other part in an association. For a cardinality of many, the association must include a variable size data-structure, which gives a significant overhead. Small cardinalities such as 2,3, or 4, normally indicate that there are several roles, and are really an abstraction of several associations, such as the Smaller and Larger associations in the previous section.
- *Attributes*. Whether or not the associations have attributes is a major issue. If one of the roles have a maximum cardinality of one, the attributes of the association can be stored with that object in a marsupial manner. The case of many-to-many association is especially troublesome, as the attributes cannot be stored with any of the participating objects.
- *Navigation*. If the association is not navigable in both directions, it becomes much easier to maintain referential integrity, as changes only need to be done in one data-structure.
- *Density*. If a role has minimum cardinality of zero, not all objects of a given class will be associated. The density of a role is the ratio of associated objects to the total number of objects.
- *Duration*. If an object is associated through a role in only part(s) of its life span, this influences both choice of representation and navigation style. A role will often have a shorter life span than the object [9].

3.1. Choosing a representation

This section will examine how these five dimensions influence the choice of representation. In section 3.2 we will examine how the dimensions influence choice of access style.

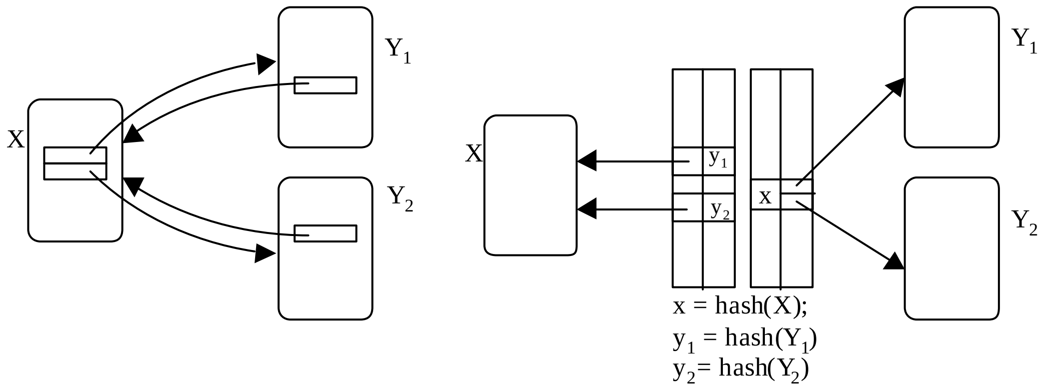
3.1.1 Cardinality: The first thing to notice is that an internal representation is always more time-efficient than an external representation. The Figure below illustrates this.



The objects X and Y are associated. To the left, the association is represented as a pair of mutual pointers. To the right the association is represented using a pair of index structures - here hash tables. To find Y given X, in the internal case is simply to de-reference the pointer. In the external representation, we need to lookup Y with X as the key. The difference in time-performance is the table lookup.

Updating the association, so X becomes associated with Y’ is a matter of changing two pointers in the internal representation. In the external representation, the values of two tables must be changed, but to do so requires two lookups as well.

The better efficiency of the internal representation is not affected by cardinality of the association. The Figure above illustrated a simple one-to-one association. The Figure below illustrates a one-to-many association.

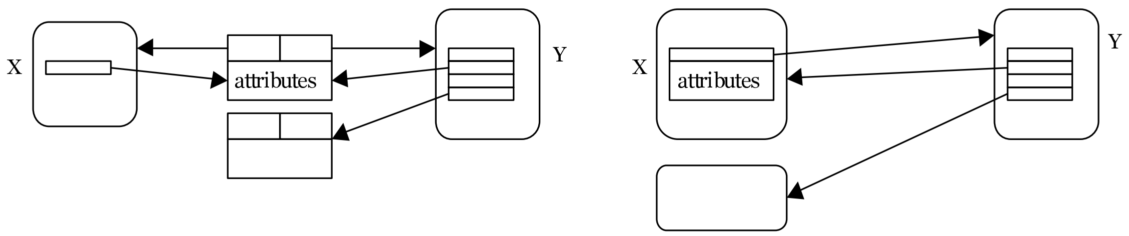


The object X is associated to Y_1 and Y_2. In the internal representation of X, it is no longer sufficient to have a simple pointer to an Y object, but a collection of some sort must be used(left part of the illustration). However, this is also true for the external representation. Looking up X must yield both Y_1 and Y_2, and hence the value to be stored can no longer be a simple pointer to a Y object, but must be a collection as well. However, the external representation still requires indexing to find the collection, where the internal representation does not.

To add or change the association between X and some Y object requires one collection operation in X, and a pointer change in the Y-object. This is also the case for the external representation; but again, two index operations must be added to the cost.

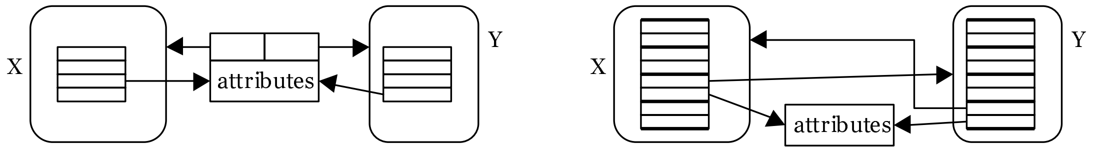
Notice that the choice of external or internal representation can be made at the role level rather than the association level. Thus it is possible, and our compiler supports this, to choose an external table for the representation of one role, and an internal pointer for the other. In the example of the binary tree in Section 2.2 a table was used to represent offspring, whereas a pointer was used to represent parents.

3.1.2 Attributes: Adding attributes to an association (also known as UML association classes) does not necessarily affect efficiency. In the left side of the Figure below, which depicts a one-to-many association between X and Y, attributes are stored in a separate link-object. To access the Y object from X, this introduces one more pointer to follow, as navigation goes through the link-object. However, if the association is one-to-one or one-to-many the attributes of the association can be kept as an attribute in the object that only associates to one other object.



This is shown in the right part of the above Figure, and will be called marsupial attribute representation (named after a somewhat similar concept in [7]). Using the marsupial attribute representation, navigation is not affected by attributes. However, access to the attributes becomes asymmetric, as it becomes necessary to obtain the X object to get to the attribute values from Y. This asymmetry can be hidden in access methods. The left and right sides of the Figure provide the same efficiency for Y, while the right hand representation is most efficient for X.

With many-to-many associations, the marsupial approach cannot be used. When an object is associated to many others, pointers to these must be kept in a collection - both in the internal representation and in the external representation. The left-hand side of the following Figure represents the direct implementation of attributes in a many-to-many situation.



A link-object is created for each pair of associated X and Y object. As before, this solution affects navigation efficiency. The right hand side of the Figure represents a different approach; instead of storing only the reference to the link-object in the collection, we can store references to both the associated object and an object keeping the attributes. Navigation efficiency is not affected using this approach.

The important point is, however, that introducing attributes is equally hard for both the internal and external representations. Again, the internal representation is more efficient as it does not require lookup.

It should be noted that the only efficiency difference between internal and external representation is that the external representation requires lookup, whereas the internal representation does not. With the overhead of cardinality (introduction of collections) and attributes, the relative cost of lookup is reduced.

3.1.3 Navigation: Uni-directional navigation further stresses the efficiency of internal representation, since a uni-directional navigation makes it possible to realise an X-to-one association as a simple pointer, without the overhead of consistency maintenance. The overhead of a lookup is proportionally larger than in the case where consistency has to be maintained as well.

3.1.4 Density and Duration: There are also problems with the internal representation. If the association has low density, i.e. few objects are associated, the internal representation becomes space expensive, because a reference (or a table) will be allocated for each object although only some objects use it. However, it is easy to design an external representation, which will use memory in proportion to the number of, associated objects rather than the internal representation, this will use memory in proportion to the number of objects.

A similar problem arises if the associations have a short duration compared to the life span of its associated objects. The internal representation requires that the association representation is present at all times. With a small ratio between the duration of an association and the life span of an object, this leads to a waste of space for the internal representations. The external representations can be designed to adjust memory needs in proportion to the number of associated objects at any point in time.

A rough estimate is that an external representation uses at least twice as much memory for each associated object than the internal representation (as it stores both the key and value of a mapping). Thus, an external representation should only be chosen when less than half the objects are associated, or when they are only associated in less than half of their life span. Neither density nor duration aspects can be inferred from the cardinality or from other aspects usually present in a UML specification of an association. Since these are the crucial deciding factors for choosing an external representation, the choice of representation should be part of the specification of the association. Nevertheless, we feel it is necessary with a default representation to make the specification closer to the UML specification. In the Smalltalk compiler, we have (somewhat arbitrarily) chosen to make the internal representation the default because it provides better time efficiency.

An association can be dense in one direction and sparse in the other direction. An example of this is an owner association between private jets and persons. All private jets have an owner, but few persons own a jet. The choice of representation should therefore be done at the role level, not at the level of an entire association. The Smalltalk compiler supports role level specification.

3.2. Choosing access style

Binary associations can be used in two ways at the linguistic level. One is to use the association name explicitly as in Works_for.get_employee(John) which will return the employer of John. The other alternative is to use role names as access-methods, as in John.get_employer, which will also return the employer of John. If a person can have multiple employers, both access methods should return a set of employers (most object-oriented programming languages have good support for collections, and there is therefore no reason to replicate collection-functionality in the association compiler).

The most important fact of access style is that it is not dependent on representation. The choice of access style is therefore largely a matter of taste, and we feel that the role based access blends better with object-orientation. However, there is one exception, as the role based access style does not allow for generic access as discussed in the end of binary tree example. Generic access is necessary to treat associations as first class objects.

3.3. Summary

The following observations summarise the main points from the previous two sections, and represent the conclusion regarding choice of representation and access style.

- Both kinds of representation are always possible. External representation is never the most time efficient choice, but is sometimes better than, or as good as, an internal representation with respect to memory consumption.
- Adding attributes to an association makes the implementation more complex, but does not shift the choice of representation in favour of either representation style.
- The two access styles are simple translations of each other, and the choice is largely a matter of taste. The exception is that association access style can only be used with association variables.

The choice of access style becomes embedded into the source code; choice of representation does not. Since it is always possible to use the association access style, it seems to be the better choice. However, if the limitations of role based access are acceptable (not being able to use generic access) the role based access style seem to blend better with object-oriented programming.

The dimensions and issues regarding the two choices, representation and access, are summarised in the table to the right.

☺ Good solution 😐 Hard solution ☹ Bad solution		representation		access	
		external	internal	association	role
attributes	Yes	😐	😐	😐	😐
	No	☺	☺	☺	☺
cardinality	1-1	☺	☺	☺	☺
	1-M	😐	😐	☺	☺
	M-M	😐	😐	☺	☺
navigation	one way	☺	☺	☺	☺
	two way	😐	😐	😐	☺
density	Low	☺	☹	☺	☹
	High	☹	☺	☺	☺
duration	Short	☺	☹	☺	☹
	Long	☹	☺	☺	☺

3.4. Implementation in Smalltalk

The association compiler has been implemented for the Smalltalk programming language. This language was chosen because it has an easy to use meta-programming model, making it possible to experiment without changing the compiler of an existing language. However, we believe that the experiences from our experiments in Section 2 are independent of the actual implementation language, and the implementation discussions from the previous sections are language independent.

The main idea in the Smalltalk implementation is to make a class Assoc (Association is a predefined class name in Smalltalk), which can be sent a message that defines a new association:

```
Assoc named: #WorksFor
  relating: #( one employee Person)
  and: #( many employer Department)
  category: ‘Company organisation’
```

This expression creates a new association type, WorksFor, which associates Person and Department objects. The role names employee and employer are used to provide access methods for both the WorksFor association and the Person and Department class. As an example, four methods are created to associate a person object and a department object:

```
john setEmployer: salesDepartment
salesDepartment addEmployee: john
WorksFor setEmployer: salesDepartment for: john
WorksFor addEmployee: john to: salesDepartment
```

It is obviously not necessary to have four methods to do the same task. We chose to implement all four to gain experience with the different ways of manipulating associations. Notice that some methods have the prefix “set”, while others have the prefix “add”. We found it unnatural to “add” an association to a role that has maximal cardinality of one; hence, we named it set. Since the employee role has a maximal cardinality of one, all methods will remove any existing association between john and another department before associating him to the sales department. For a cardinality of “many”, no such removal is necessary; e.g., no one is fired when John is associated to the sales department.

In order to provide access methods that use the role names, we make WorksFor a class. Smalltalk makes it easy to insert methods into a class, a technique we use to insert access methods into both the Person, Department and the WorksFor classes. The current implementation does not support attributed associations, though an earlier version did.

In addition to support for the external and embedded representations, the association compiler also supports a programmer-defined role. The programmer must implement simple access methods to add and remove a one-way association, the association compiler then utilises these methods to ensure consistency in connection with bi-directional associations.

4. Related work

March and Rho [11] describe an object-relational system, which adds E-R semantics to object-oriented modelling. Their system is very similar to the Smalltalk system we have described in this paper and in [19]. It is a direct implementation of Rumbaugh’s model [14][15][16], with some extensions for querying and navigation. One difference is that we implement associations as classes whereas they realise them as objects. We optimise the implementation according to cardinality and representation type. On the other hand, they have gone further by supporting queries for associations as well as for classes, which is natural in a system which supports E-R semantics.

The notion of associations have had a strong influence on the analysis and design notations, but has not had any impact in programming languages. This leaves us with a serious question of why that is the case. Three possible answers present themselves.

1. Supporting associations is a bad idea.
2. There is a gap in the design of object-oriented programming languages.
3. The notion of associations is not well matched to object-oriented modelling.

Our standpoint has been that 2 is the case. [17] argues that 3 is the case. They present three arguments to support the claim.

1. Associations break down class encapsulation because associations are external to the objects, but nevertheless states something about the objects.
2. Associations break down abstraction because associations become a kind of entity without real world counterpart. Their argument is that associations introduce a semantic gap between the real world and the created models.
3. Associations break extensibility because an inheritance mechanism will be needed for associations, and association inheritance will most likely differ from that of classes.

We find that all three claims can be refuted. The first two arguments are really a fundamental issue on the nature of objects. They take for granted that the fundamental abstraction is objects *and only objects*. In the Scandinavian school of object-oriented programming (e.g., Chapter 18 of [12] or [10]), there is a deeper foundation, which is that the fundamental issue is modelling the real world using natural abstractions. If associations turn out to be natural to use, modelling should support this, and so should programming to avoid a semantic gap between model and program. We are convinced that associations are a natural abstraction, and [8] gives further evidence of this.

Their last argument is correct, but under other circumstances, such an argument will be put on the agenda for further investigation, as indeed it is done in Rumbaugh's original paper [14]. In [20] we introduce the notion of covariant specialised associations - associations where the type of the associated objects are specialised simultaneously to further characterise what is associated. This is a special case of association specialisation, but it has proven to be useful.

5. Future work

The ODMG object model [1] includes a notation for specifying relationships between objects. A interesting aspect here is that their approach uses a decentralised specification of an association, where the two roles and their inverse are specified as part of the participating classes. In [5] it is argued that this role oriented specification provides a more object-oriented way of specifying an association. This is in accordance with our own experiences; i.e. the role based access blends better with the object-oriented paradigm [19]. However, it is important to notice that the techniques for efficient implementation described in this paper does not depend on the specification style. Just as the choice of access style is primarily syntactic, so is the style of specification. ODMG also specifies a language specific mapping.

The association compiler provides techniques to efficiently implement the access methods "formX" and "dropX" discussed on page 209 in [1].

The paper [8] describes complex associations. It presents a model of associations that allow multiple abstraction levels of associations, and it presents how to handle associations between objects with a rich internal structure. The paper addresses these issues at the modelling level, and does not address implementation of such associations. However, we believe that the implementation framework presented in this paper will serve as a good foundation for implementing such a rich model. In [9] we discuss the notion of roles for object-oriented programming in detail. We believe that both complex association and roles can be combined into a single linguistic framework, but that remains to be verified.

The present implementation framework chooses the same representation for all objects associated through the same association role. Choosing the representation at runtime on a per object basis will remove the present trade of between time and space.

6. Experiences

The two experiments and our experiences with using association in a larger administrative system [20] have verified some of the claims from the introduction, and pointed to some unexpected aspects of using associations.

We believed that the role names would be the primary accessing method for binary associations, as they seemed to blend well into the object-oriented paradigm as methods. This assumption has been true in almost all cases. However, there is one exception, as discussed in section 2.2. When an association is stored in a variable it is not possible to know the exact association contained in the variable. Hence, the role names of the association cannot be known, this necessitates access methods that are independent of role name.

It was also assumed that the higher level of associations would make the code robust towards changes in the implementation techniques. This has proved to be true in all cases. The association compiler makes it completely transparent if an internal or external representation is chosen. However, it is not transparent to change the cardinality of an association. If the upper cardinality for a role changes from one-to-many, the corresponding access method will return a set of associated objects rather than a single object; such a change is not transparent. One could consider having a method that returns a selected object from the set. The implementation described by [11] has two access methods, one that returns a single object, and one that returns a set. The two methods are available independent of the cardinalities. We felt that a change from one to many (or vice versa) does warrant changes to the code, but more experiments are needed to settle this pragmatic issue.

The experience with the binary search tree, where association variables where used, indicates that the association-based access is more powerful than role-based access. There are situations in which the internal representation is too memory expensive (low density or short duration). These arguments suggest that one could do with only association based access and external representation. However, we are afraid that the losses in time efficiency when using an external representation might be a problem that will prevent associations to be used. Moreover, our experience has been that the role based access blends better with object-oriented programming.

The goal of our association compiler is to produce code that matches the quality of manually constructed implementations; we believe we do better. A manually constructed implementation must be efficient (concerning both time and space), and it must be maintainable. It is our claim, that our compiler produces efficient code. However, we believe that it produces code that is more efficient than what would normally be produced by a programmer. The compiler need not make the code maintainable, since it will just regenerate when a change occurs in the specification of the association. Hence, the generated implementation goes through fewer levels of abstraction than would normally be introduced to ensure maintainability.

Finally, we have experienced that it can be difficult to name associations, but it has never been a problem to name roles. This can be taken, as further indication that a role-based *specification* (as in ODMG) is fundamentally better suited to object-oriented modelling than explicit associations.

Acknowledgements. This project has been carried out with support from the Danish Centre for Information Technology, CIT/COT 74.1.

7. References

[1] R. G. G. Cattell and others. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers; ISBN: 1558604634, 1997.

[2] B. K. Ehlmann and G. A. Riccardi. *An Integrated and Enhanced Methodology for Modelling and Implementing Object Relationships*. Journal of Object-Oriented Programming, May 1997. Page 47─55.

[3] D. G. Firesmith and B. Henderson-Sellers. *Clarifying Specialised Forms of Association in UML and OML*. Journal of Object-Oriented Programming, May 1998. Page 47─54.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] I. Graham, J. Bischof, and B. Henderson-Sellers. *Associations Considered a Bad Thing. Journal of Object-Oriented Programming*, February 1997. Page 41─48.

[6] B. Henderson-Sellers, D. G. Firesmith, I. M. Graham. *OML metamodel: Relationships and state modeling*. Journal of Object Oriented Programming, March-April 1997, page 47 ─ 51.

[7] Michael Jackson. *System Development*. Prentice-Hall International, Inc. 1983

[8] B. B. Kristensen. *Complex Associations: Abstractions in Object-Oriented Modeling*. Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'94), Portland, Oregon, 1994.

[9] B. B. Kristensen, K. Østerbye. *Roles: Conceptual Abstraction Theory & Practical Language Issues*. Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-Oriented Systems, 1996

[10] Bent Bruun Kristensen and Kasper Østerbye. *Conceptual Modelling and Programming Languages*. SIGPLAN Notices, 29(9):81-90, September 1994.

[11] S. T. March and S. Rho. *A Semantic Object-Oriented Data Access System*. http://www.misrc.umn.edu/-wpaper/wp96-05.htm.

[12] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard: *The Beta Programming Language*. Addison Wesley, 1993.

[13] J. Olsson, K. H. Nielsen, K. Østerbye, A. R. Lassen. *Objektorienteret Analyse og Design af udvalgte dele af STADS*. Technical report COT/4-03, 1998. Center for IT-forskning, Forskerparken, Gustav Wieds Vej 10, 8000 Århus C, Denmark (in Danish).

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

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

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

[17] A. V. Velho and R. Carapuça. *Attribute: A Semantic and Seamless Construct*. http://albertina.inesc.pt/ESW/documents/som-tol/. 1993.

[18] UML Modeling Language, Standard Software Notation: Resource Center. http://www.rational.com/uml/-index.shtml, 1998.

[19] K. Østerbye, A. R. Lassen, J. Olsson. *Object Relational Modelling*. Technical report, COT/4-04, 1998. Center for IT-forskning, Forskerparken, Gustav Wieds Vej 10, 8000 Århus C, Denmark.

[20] K. Østerbye, J. Olsson. *Scattered Associations in Object-Oriented Modeling*. Proceedings of Nordic Workshop on Programming Environment Research, 1998, Bergen, Norway, June 14-16.