DIVISION OF COMPUTER SCIENCE

Implementing Associations between Objects

Audrey Mayes
Bob Dickerson
Carol Britton

Technical Report No.205

August 1994

Implementing Associations between Objects

Audrey Mayes, Bob Dickerson and Carol Britton
School of Information Sciences, University of Hertfordshire, College Lane,Hatfield, Herts AL10 9AB, UK
Tel 0707 284763 Fax 0707 284303
email comrjam@herts.ac.uk or comqcb@herts.ac.uk

1. Introduction

This paper presents an alternative design method for the implementation of *conceptual associations* identified during the analysis of a problem.

Object-oriented development methods such as OMT [1], identify associations between objects. Some of these associations represent aggregations of objects such as a wheel is part of a car. Other associations represent conceptual links between objects such as a person has an account. These types of association are shown in figure 1.

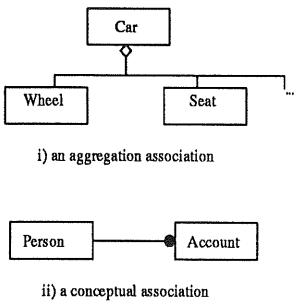

Figure 1. Associations between objects

These two types of association clearly represent different concepts. However, both are usually implemented in object-oriented programming languages by using the client-server relationship. The class representing an aggregation declares an instance of the classes which represent its parts. A class involved in a conceptual association declares an instance of the classes with which it is associated. The addition of an association usually requires the declaration of new subclasses of the classes involved. The result of using the client-server relationship to add conceptual associations is that the implemented objects become less like the objects identified during analysis and are bound together in the same way as aggregations. The effects are:

- the classes used to define the objects are application specific and therefore less reusable [2].
- the structure of the system is difficult to understand.

It is agreed by Kilian [2] and Rumbaugh [3] that the provision of associations as separate constructs in object oriented systems would increase reusability and improve the clarity of system design.

The design method presented here allows conceptual associations from the analysis model to be implemented directly by using Sociable Classes. These Sociable Classes have the ability to participate in associations without the addition of attributes.Conceptual associations are provided as instances of generic classes. New associations can be added as required by introducing a new instantiation of the required type of association to the system. It is not necessary to define new subclasses of the participating objects. The conceptual associations retain the object-oriented structure by storing the associations with the objects. Relational tables of associations are not added to the system.The design technique overcomes some of the problems mentioned above.

2. Basis of the design

In order to differentiate between conceptual associations and aggregations, a mechanism must be provided to allow different degrees of binding between objects. The different degrees of binding would result in:

- groups of objects which are tightly bound because they represent aggregations, such as a car.

- objects which are loosely coupled to other objects because they take part in conceptual associations, such as a person has an account.

In this suggested design, the client-server relationship is used to implement aggregations. The loose coupling between objects is produced by adding conceptual associations between objects. For example, an association is added between a person and an account to implement the association 'a person has a bank account'. It is not necessary for all objects of a class to be involved in all types of association.

In order for a design to provide loosely coupled objects, the following facilities should be available:

- a means of explicitly implementing associations between objects.

- the ability for objects to take part in many different associations. These associations must be added to objects without changing the definition or implementation of the classes of which they are instances. Therefore, the classes should have no knowledge of the specific associations in which any or all of its objects are involved.

- the ability to add new logical relationships without producing subclasses of the classes involved.

The next section describes the classes used in the Sociable Class design technique which attempts to meet the above criteria.

3. Sociable Classes and related constructs.

Sociable Classes define objects which have the ability to take part in a potentially unlimited number of different associations. The design technique using Sociable Classes requires the declaration of two abstract base classes, **Social** and **Assoc**. Sociable Classes are subclasses of **Social**. The associations between objects are formed by instances of subclasses of class **Assoc**.

When an association is made, instances of associations become linked to the part of the object which was inherited from **Social**. Associations therefore become part of the objects involved in the association not separate entities stored in a data structure. They remain part of the object until the association is broken. When an association is broken, the object ceases to have any knowledge of that type of association.

One instance of each type of association required in the system is declared and used to create, access and delete all other instances of that type of association.

3.1. Class Social

This class provides the ability to add, retrieve and delete associations from an object. In order to provide this ability, the class **Social** declares a collection of associations as a private attribute and provides features to access this attribute. The collection may be implemented by any appropriate data store.

The features to access the private attribute are not made publicly available. The only classes which can access the attribute are **Assoc** and its derivatives. The access is limited in this way to encapsulate knowledge of the implementation of associations. All Sociable Classes are derived from **Social** and do not need to access any of its features.

3.2. Class Assoc

This base class is declared to allow all associations to be assigned to the same data structure in instances of the class **Social**. The *associate* and *disassociate* features are defined by class **Assoc**. These two features represent the minimal functionality that must be provided by all subclasses. The variable number of objects involved in asso-

ciations means that different numbers of features are required to access the objects participating in different forms of association. One access feature will be required for each object involved. These features cannot therefore be defined in the base class.

3.3. Sociable Classes

Sociable classes are implemented by declaring the required classes as subclasses of **Social**. The features defined by the analysis model are then added. The following extract of code gives two examples of the definition of Sociable Classes using Eiffel notation [4].

```
class PERSON
export        first_name,...

inherit SOCIAL

feature
       first_name : STRING;

...

end --person

class ACCOUNT

export
        accountNumber,...
inherit SOCIAL

feature
        accountNumber: INTEGER

...

end --account
```

3.4. Associations

The many different types of associations required in systems are provided by subclasses of **Assoc**. Each subclass defines a general type of association, such as a one-to-one bidirectional association. Figure 2 shows part of the association hierarchy. The subclasses of **Assoc**, in the lower level of figure 2, are generic classes which supply

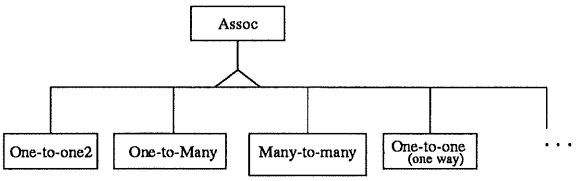

Figure 2. Association class hierarchy

the implementation of features provided by associations of that type.

For example, class **One_to_one2** implements one-to-one bidirectional associations. That is, it implements associations between two objects and allows the association to be traversed via either object. This generic class requires two formal generic parameters, one for each of the objects involved in the association. It provides implementations for the *associate* and *disassociate* features inherited from class **Assoc**. It also defines the features required to access the objects involved in a specific association. The classes used to replace the formal generic parameters must be Sociable Classes. Using Eiffel notation, the interface of this class is **One_to_one2 [A→SOCIAL, B→SOCIAL]**. An association between a person and an account is declared as `has-account : One_to_one2(Person,Account)`.

4. Using Sociable classes

This section gives an example of the use of the Sociable class design technique. The association being implemented is shown in Figure 3. The classes **Person** and **Account** are declared as indicated in Section 3.3. The root or main class is declared as follows:

```
class BANK

feature
   a,b : PERSON;
   x,y : ACCOUNT;
   has-account : ONE_TO_ONE2[PERSON,ACCOUNT];
```

```
Create is
do
--create and assign values to
--person and account variables
  a.Create;
  ...
  x.Create;
  ...

--create the association variable
  has-account.Create(hasaccount);

--(the parameter is required for
-- reasons beyond the scope
-- of this paper)

-- associate the required objects
  has-account.associate(a,x);

-- find the account belonging
-- to person a
  y := has-account.find_object2(a);

--the account can the be
--accessed via object y

-- find the person owning
-- account x
  b := has-account.find_object1(x);

--the owner of account x
--can then be accessed via object b
...

end;--Create
end --BANK
```

5. Conclusion

A system designed and implemented by using the above technique would consist of classes which are recognisable as definitions of the objects identified in the analysis model. The client-server relationship is used to implement the structure and attributes of each class. The classes would not be modified by the addition of extra attributes to provide associations with other classes of objects. New subclasses of objects would be introduced only when extra attributes or structure need to be added to existing classes. The implemented system would be simpler and therefore easier to understand, maintain and enhance. The classes would be readily available for use in other systems because application specific features would not have been added.

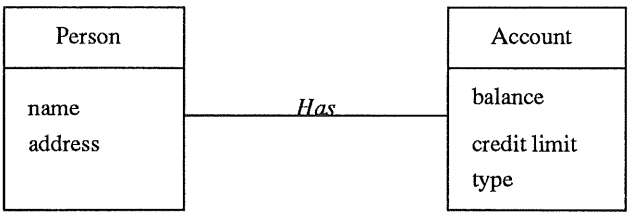

Figure 3. Simple banking application object model

Further research is being carried out into the use of other languages for implementation and into the possibility of including the constructs defined in this paper in a language definition. This work forms part of a PhD thesis to be submitted Sept 94.

REFERENCES

1. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modelling and Design.* Prentice-Hall International Editions, Englewood Cliffs, New Jersey, 1991.
2. Michael Kilian. A Note on Type Composition and Reusability. *OOPS Messenger*, 2(3), 7 1991.
3. J. Rumbaugh. Relations and semantic constructs in an object-oriented language. *OOPSLA'87*, 1987.
4. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.