Implementing Associations with Ada

Catherine Jean-Pousin, Stéphane Barbey
Swiss Federal Institute of Technology
Software Engineering Laboratory
EPFL-DI-LGL
CH-1015 Lausanne
tel: +41 (21) 693.25.80
fax: +41 (21) 693.50.79
catherine.jean@di.epfl.ch
stephane.barbey@di.epfl.ch

Abstract

Object-oriented programming is a mechanism used to construct modular systems made of reusable components. Objects that compose a system are necessarily related. This is the reason why the relationships between the objects are one of the major factors in the development of software. Unfortunately, of all the relationships between the objects, the associations are often left aside. Therefore, we investigated a way to implement associations, using a language that does not provide a built-in construct for that purpose.

We start by giving a definition of associations and then an implementation-oriented taxonomy of the different kinds of associations. Then, according to this taxonomy, we propose a way to implement the associations using Ada 83. We also show, for a particular case, how Ada 9X can be used. The solutions proposed allow to translate all the semantics attached to associations while preserving the integrity of the system to develop.

Résumé

La programmation par objets est un mécanisme utilisé pour construire des systèmes modulaires dont les composants sont réutilisables. Les objets qui entrent dans la composition de ces systèmes sont inévitablement reliés entre eux. C'est la raison pour laquelle les relations entre les objets font partie des facteurs prédominants dans le développement de logiciels. Malheureusement, parmi les relations entre les objets, les associations sont souvent négligées. C'est pourquoi, nous nous sommes intéressé à la façon de coder les associations dans un langage de programmation ne fournissant pas de mécanisme direct pour les traduire.

Nous donnons en premier lieu une définition des associations, puis nous proposons une taxonomie des différentes sortes d'associations, et en fonction de cette taxonomie, nous indiquons les mécanismes utilisés pour coder les différents types d'associations en Ada 83. Le codage en Ada 9X d'un cas particulier est également donné. Les solutions proposées pour le codage des associations respectent la sémantique des associations tout en préservant l'intégrité du système englobant.

Keywords

Objects, relationships, associations, modularity, Ada 83, Ada 9X.

Mots-clés

Objets, relations, associations, modularité, Ada 83, Ada 9X.

1. Introduction

Although object modelling brings up a lot of interesting problems [Meyer 88], research in object-oriented technology does not simply deal with describing the structure and behaviour of objects; it also deals with the relationships that link those objects together. It would be senseless to create systems in which objects are all unrelated.

Of the many kinds of modelling relationships between objects and classes that inhabit a system, the most commonly used are:

- Aggregation

  Aggregation, also called a *part-whole* link, is the most obvious relationship. Just like the objects found in the real world, most of the abstract objects are not atomic, but aggregation of other objects. For example, a plane is composed of a cabin, a motor, two wings,... Each component exists by itself. However, assembled together, they form a plane. Thus, there is an aggregation relationship between a plane, and the cabin, the motor and the wings.

- Inheritance

  Inheritance, expressed by an *is-a* link, allows the creation of one abstraction based upon another, while retaining the ability to express differences with the parent abstraction. It mainly serves as a way to design specialized objects, often in order to create a taxonomy. For example, a plane is a specialization of a vehicle in the sense that it shares the property of being a means of transport; a distinctive feature of the plane is to fly.

- Association

  Unlike the two aforementioned relationships, associations are used to connect objects that have semantic relationships which do not influence their internal structure. The links of an association reflect the responsibilities of the associated objects. For example, an association called Author represents the connection between a Person and a Book. It is obvious that this connection has nothing to do with aggregation nor inheritance, but reflects the semantics defined between a book and the person who wrote it.

Readers may notice that this way of modelling object relationships offers a certain similarity with the entity-relationship model [Chen 76]. Both models have a notion of elements (entities vs. objects), that can be composed (of attributes vs. of attributes and objects), and of relationships between the elements (in our case associations). There are even enhanced entity-relationship models that introduce specializations and generalizations (for instance ERC+ [Parent & al. 88]). In the meanwhile, the entity-relationship model doesn't take into account the behaviour of objects, nor does it deal with encapsulation and visibility problems.

Even if the analysis and modelling of associations have been explored by the main authors of object-oriented design methods ([Coad & al. 90], [Rumbaugh & al. 91], [Shlaer & al. 92],...), their implementation is almost ignored. Although the majority of modern general-purpose programming languages provide mechanisms to implement aggregation and inheritance, it should be noted that none of them provide a simple means of handling associations.

Our intent is to show how associations can be implemented in such languages, not to try to define a new mechanism to translate associations. We propose a way to implement associations in a language that does not offer a built-in construct for that purpose.

As we do not consider the inheritance relationship, choosing Ada [Ada 83] for implementing our model has the advantage of providing us with its powerful mechanisms, such as genericity, encapsulation and strong typing. This way, we can steer clear of useless coupling, which should be avoided as much as possible. Moreover, the forthcoming revision of the language, called Ada 9X [ARM 93], corrects the lack of object-orientedness by introducing inheritance and polymorphism through type extension, dispatching and class-wide programming.

2. Classifying associations

In this section, we first give a definition of the associations and then an implementation-oriented taxonomy of the different kinds of associations.

The most obvious way to model associations is to translate the links into references. If two objects A and B are associated together, A contains a reference to B and B contains a reference to A. This way of translating associations is not satisfying. First, it implies circular references between objects, which is neither easy to implement nor to maintain. Next, it is not appropriate for associations relating one object to many others, or for optional associations.

Moreover, it violates the principle of independence between objects of a system. We wish to maintain this independence of the objects in order to increase the modularity of the system and the reusability of the objects and their classes. Were the links part of the objects, reusing a class would imply reusing all the classes it is associated with, even those which are not part of the problem domain to be developed. This approach violates the object-oriented paradigm.

Before giving a more convenient classification for the associations, we will give a few definitions.

2.1 The different kinds of associations

There are miscellaneous kinds of associations. We will concentrate on the most common ones, leaving aside those which rarely occur. Our model may be extended to cover all the different kinds of associations.

Within an association, each linked object plays a role which gives its responsibility in the association. This role is used to label the link between the object and the association. A cardinality is defined for each link. This cardinality characterizes the participation of an object in the association. For example, in the Author association, the role of a Person can be called *write*s while the role of the Book may be called *is_written_by*. The respective cardinalities are (0:n) and (1:n), meaning that any person may write zero to n books and that a book shall have at least one author.

The cardinality of the roles define the multiplicity of associations. We distinguish between one-to-one, one-to-many and many-to-many associations. We also distinguish optional and mandatory associations.

- The one-to-one associations connect one object to another. For example, the association Marriage connects two instances of the class Persons and the association Governments connects one instance of the class Governors and one instance of the class States.
- The one-to-many associations connect one object to several others. For example, an employee, instance of the class Employees, works in a factory, instance of the class Factories. But the same factory employs more than one employee.
- The many-to-many associations are those which connect several objects to several others. For example, planes land in airports. Several planes land in the same airport and several airports allow a plane to land.

Each of those associations are binary associations because they involve two links. There are also *n*-ary associations, which involve *n* links. For example there is a ternary association between clients, suppliers and products. Suppliers supply products to clients. Ternary associations cannot be modelled with three binary associations, because the resulting semantics is not equivalent.

Each kind of association can either be optional or mandatory for its members. An association is optional if the cardinality of one of its link is (0:1) or (0:n), which means that an object of the corresponding class is not compelled to take part in the association. For example, although all states have a governor, the government association is mandatory for states: each state must have a governor; while this association is optional for persons: not all persons are governors of a state.

An association can have attributes. The attributes of the associations, like the attributes of the objects depict the characteristics of the association. For example, the date of election can be an attribute of the Governments association.

2.2 The proposed taxonomy

The model chosen for an association will depend on two distinct criteria: the presence or the absence in the system of one of the classes involved in an association; and the indispensability of an association (optional or mandatory).

It may happen that a class has no responsibility in the system and therefore does not belong to the domain of the application. Such a class does not need to be designed as a class of the system. However, such a class could have been identified as taking part in an association. For instance, consider the class Persons in an application managing salaries. It is interesting in such an application to consider the work of each person and to define specializations accordingly. For the Governors class, specialization of the Persons class, it may be useful to internally know which state they govern, although we actually have no use of a State class outside of governors. This is an association between the Governors and the States.

For the one-to-one or one-to-many associations, if one of the classes is missing from the application to develop, it can simply be modelled as an attribute of the present class. In the Governments example, the association between the governors and the state they govern can be given by an attribute State of the class Governors.

This model is straightforward for all one-to-one associations and for one-to-many associations if the missing class is on the one-side of the associations. If the missing class is on the many-side of the association, the attribute must be modelled using a container class, such as a set or a list, etc. For example, in the Works association between employees and factories, if the Employees class is absent from the application the association can be modelled as a list of employees in the Factories class.

We can model the optional associations like the mandatory associations, or use inheritance. For example the association Works between persons and a factory is an optional association for persons if we consider that not all persons work. If we define the association Works between the class Persons and the class Factories without taking into account the optional property of the association, modelling the association as an attribute is not elegant because the attribute will be defined for all the instances of the class, even those which do not take part in the association. In the example, all persons, even children, will have an attribute Factory.

Using inheritance for this case means that we create a subclass such that only the instances of the subclass will participate in the association. The attribute which will model the association is put in the subclass, as if it were an extension of the superclass. In our example, we create a class Employees, subclass of the class Persons, such that the association Works will be defined only for the Employees class. This model also has some drawbacks. It is the case when an object of the superclass participates later in the association. For example, an instance of the class Persons taking her first work should become instance of the class Employees. Unfortunately, there is actually no mechanism allowing an object to change its class dynamically.

In the general case, we will model the associations as classes. Modelling associations as classes has the advantage of giving the possibility to naturally define attributes for the associations. For example, we can create a class Governments such as each instance of the class is an association between a governor and a state. The attribute Election_Date is therefore an attribute of the class Governments.

3. Implementation of the models in Ada

In this section we give a practical approach for coding classes in Ada. Then we give the Ada mechanism for the implementation of the three aforementioned models.

3.1 Implementing classes in Ada

Ada has no special construct for classes. However, it offers all the needed mechanisms. A class can be implemented as a package exporting a type and its associated operations.

The specification of the package is the interface of the class and the body its implementation. The type exported by the package is a record type which components are the attributes, unless the value of the type is atomic. The operations defined for the instances of the class are the operations implicitly defined by the declaration of the type and the subprograms explicitly declared in the specification and having at least one parameter or the result of that type. An instance of the class is created by the declaration of an Ada object (variable or constant) of the type exported by the package.

Neither the package nor the type exported by the package constitute the class, but both of them. The name of the package is used in a context clause when the class must be visible or as prefix in an expanded name. The name of the type is used to refer to the class in all other cases.

3.2 Implementing the association as an attribute

Every Ada programmer knows how to implement an attribute by adding a component to a record type. We must nevertheless precise the mechanism used if the association cannot be modelled as an atomic value or if the association has attributes itself. Consider the Works example. If the Employees class is not part of the system it can be implemented as an attribute. However, if the employee cannot be modelled as an atomic object, i.e. a single value, a type Employee_Type has to be defined. Employee_Type can be considered as a local class of the Factories class.

Moreover, if we now want to stress the date of the beginning of contract, we must define an attribute for the association. Adding this attribute as another component of Factory_Type would not work because the date of the beginning of contract qualifies the entry of each employee and not the factory. Instead of this, we must put the attribute in the Employee_Type. Therefore, this date will be known for each employee.

The implementation of the Factories class as described above is shown on figure 1. We suppose in the example that we have already implemented an abstract data type for sets: Set_G.

```
with Calendar;
package Factories is
   type Factory_Type is private;
   procedure Initialize
      (The_Factory: out Factory_Type;
       Name: in String);
   procedure Add
      (The_Factory: in out Factory_Type;
       Name: in String;
       Salary: in Natural;
       Beginning_of_Contract: in Calendar.Time);
   -- Other subprograms defined for the factories.
private
   type Employee_Type is
      record
         Name: String (1..20);
         Salary: Natural;
         Beginning_of_Contract: Calendar.Time;
      end record;
   package Set_of_Employees is
      new Set_G (Employee_Type);
   type Factory_Type is
      record
         Name: String (1..20);
         Employees: Set_of_Employees.Set_Type;
      end record;
end Factories;
```

Figure 1. Implementing the Works association as an attribute of the Factories class

As we can see in this example, it is quite easy to find the employees working in a given factory but it is not possible to find the factory where an employee works. We will admit that this last request is not interesting, as the Employees class is not a class of the system.

3.3 Implementing optional associations using inheritance

Implementing inheritance, as defined in our model, does not present too much of a problem in Ada because it does not involve dynamic binding. We will not go into details about the way to implement inheritance in Ada. It is out of the scope of this paper. For more information about the implementation of inheritance in Ada, please refer to [Hirasuna 92] and [Seidewitz 87].

As we stated before, inheritance can be used for an optional association when it is modelled as an attribute. We propose two implementations for this case. In the one case, a record type is defined such that one of its components is an object of the parent type and the other component is the association. This implementation will be preferred if the Persons class already exists and should not be modified for maintenance reasons. In the employees example, the Employees class is a subclass of the Persons class. The package for the Employees class is given by the figure 2:

```
with Persons; -- Exports a type Person_Type.
package Employees is
   type Employee_Type is private;
   procedure Initialize
      (The_Employee: out Employee_Type;
       Person: Persons.Person_Type;
       Factory: in String);
   -- Other subprograms defined for the employees.
private
   type Employee_Type is
      record
         Person: Persons.Person_Type;
         Factory: String (1..20);
      end record;
end Employees;
```

Figure 2. Implementation of the subclass Employees based on the superclass Persons

In the other case, a variant type is used such that the discriminant of the type differentiates the superclass and its subclasses. The full type of the Persons class is given in figure 3.

```
   type Activity_Type is (Salaried, Freelance, Retired, Non_Working);
   type Person_Type (Activity: Activity_Type) is
      record
         Name: Name_Type;
         Birth_Date: Calendar.Time;
         case Activity is
            when Salaried =>
               Factory: String (1..20);
            when others =>
               null;
         end case;
      end record;
```

Figure 3. Implementation of Employees as a variant of Persons

This implementation will be preferred when all the variants of persons should be handled in a uniform way. Note that the attributes of the associations can be managed as described in section 3.2.

3.4 Implementing the association as a class

In all other cases, the association will be implemented as a class (i.e. as explained in section 3.1). The type exported by the package is a record type of which components are the attributes of the association and the links to the associated objects. If the associated objects have an identifier, this identifier will serve as link. If the system has a by-reference semantics, the access variables designating the associated objects will serve as links. Otherwise, i.e. if the system has a by-value semantics and the objects have no identifier, we will have to obtain references to the associated objects for the links. Of course, we cannot copy the objects themselves, to preserve the integrity of the system.

This implementation allows us to easily find the associated objects knowing the association.

On figure 4 is an example of implementation of the class Governments if the governors and the states have no identifier.

```
with Calendar;
generic
  type Governor_Type is private;
  type State_Type is private;
package Governments_G is
  type Government_Type is private;
  procedure Initialize
    (The_Government: out Government_Type;
     A_Governor: in Governor_Type;
     A_State: in State_Type);
     Election_Date: in Calendar.Time;
  -- Other subprograms defined for the governments.
private
  type Governor_Ref_Type is
    access Governor_Type;
  type State_Ref_Type is
    access State_Type;
  type Governement_Type is
    record
      Governor: Governor_Ref_Type;-- link to a Governor object
      State: State_Ref_Type;     -- link to a State object
      Election_Date: Calendar.Time;
    end record;
end Governments_G;
```

Figure 4. Implementation of Governments as a class

In this example, we are in the case of a by-value semantics system without known identifiers for the objects. In order to initialize the Governor and State components of the record Government_-Type, we will have to find the references to the associated objects. This can be done by the two functions shown on figure 5, internal to the class and only used for the initialization of an association.

```
function To_Governor_Ref_Type is
  new Unchecked_Conversion (System.Address, Governor_Ref_Type);
function To_State_Ref_Type is
  new Unchecked_Conversion (System.Address, State_Ref_Type);
```

Figure 5. Finding references to the associated objects

These two functions are certainly not the most elegant items in our proposal but they are necessary because we cannot have the objects themselves in the definition of the type exported by the package. Fortunately, this conversion is not necessary in Ada 9X as we will see in section 3.5.

This implementation does not allow us to find an association given one associated object. In order to satisfy such request we shall keep track of all association instances. Thus we define in the class association itself tables which will hold all the information needed to seek an association instance given one of the associated objects. The keys of those tables are the links to the associated objects and the values related to the keys are the corresponding associations in which the referenced object takes part.

In the example on figure 6 the implementation of part of the body of the class Governments is given. We suppose that we have already implemented an abstract data type for associative tables: Table_of_Dynamic_Keys_and_Static_Values_G. The implementation of this abstract data type is out of the scope of this paper. For more information about it, the reader can refer to [Genillard & al. 89].

```
function "<" (Left, Right: Governor_Ref_Type)
  return Boolean;
package Governor_Tables is
  new Table_of_Dynamic_Keys_and_Static_Values_G
    (Key_Type => Governor_Ref_Type,
     Less => "<",
     Equals => "=",
     Value_Type => Government_Type);
Governments_Indexed_by_Governors: Governor_Tables.Table_Type;

function "<"(Left, Right: State_Ref_Type)
  return Boolean;
package State_Tables is
  new Table_of_Dynamic_Keys_and_Static_Values_G
    (Key_Type => State_Ref_Type,
     Less => "<",
     Equals => "=",
     Value_Type => Government_Type);
Governments_Indexed_by_States: State_Tables.Table_Type;
```

Figure 6. Implementation of the associative tables

With this mechanism, it is easy to find the association knowing an object as is shown on figure 7. The function Government_of gives the government for a governor:

```
function Government_of (The_Governor: Governor_Type)
  return Government_Type is
begin
  return Governor_Tables.Related_Value
    (Governments_Indexed_by_Governors,
     To_Governor_Ref_Type (The_Governor'Address));
end Government_of;
```

Figure 7. Implementation of a function allowing to find an association from the associated objects

The subprogram Related_Value returns a value stored into a table (Governments_Indexed_by_-Governors), given a key (the address of the associated object).

3.5 Implementation in Ada 9X

Ada 9X —the revised definition of Ada— includes among other things support for object-oriented programming, data-oriented synchronization and hierarchical library units. Moreover, the requirements for system programming includes dynamic references. This requirement is satisfied by the addition of general access types, which values can also designate global objects, instead of only objects dynamically created using allocators (new) in Ada 83.

In this section, we will only give the implementation in Ada 9X of the case of an association modelled as a class for a by-value semantics system. This can be done using access discriminants, to solve accessibility and scope problems, and the new Access attribute, that returns a reference to an object (thus suppressing the need for System.Address and unchecked programming, as shown on figure 5). The figure 8 gives the implementation of the Governments class.

```
with Calendar;
generic
  type Governor_Type is private;
  type State_Type is private;
package Governments_G is
  type Government_Type
       (Governor: access Governor_Type;
        State: access State_Type) is
    private;
  -- the constant components Governor and State are
  -- directly accessible (using the selector notation)
  function Election_Date (Government: Government_Type)
    return Calendar.Time;
  -- returns date of election of the governor as head of state

  -- other operations on type Governement_Type
private
  type Government_Type
       (Governor: access Governor_Type;
        State: access State_Type) is
    record
      Election_Date: Calendar.Time;
    end record;
end Governments_G;

with Governors, States;
package Governments is new Governments_G
  (Governors.Governor_Type, States.State_Type);

with Governments, States, Governors;
procedure Usage is
  a_Governor: Governors.Governor_Type := ...;
  a_State: States.State_Type := ...;
  a_Government: Governments.Government_Type
                  (A_Governor'Access,
                   A_State'Access);
begin
  ...
end Usage;
```

Figure 8. Implementing and using associations with access type discriminants (Ada 9X)

As we can see on figure 8, Ada 9X allows to implement more elegantly than Ada 83 the way to find the references of objects for which we don't have a reference at hand.

4. Conclusion

Currently the associations have not been investigated thoroughly enough in the object-oriented world, albeit researchers in the database field have long understood the utility of this concept in the modelling of systems. We believe that it is essential to study associations throughout the software life cycle of object systems.

This paper covers an implementation of associations in Ada. Associations present non trivial problems, it is therefore important to examine them in detail. The model we propose allows, once the analysis and design phases are achieved, to effectively implement these associations.

The proposed solutions allow to translate all the semantics attached to associations while preserving the integrity of the developed system. To be able to implement those associations we have first given a definition of the associations and then a taxonomy of the different kinds of associations. Then, according to this taxonomy, we proposed a way to translate those associations, and we gave the corresponding implementation in general in Ada 83 and for a particular case in Ada 9X.

In the future, we shall investigate the new mechanisms offered by Ada 9X, especially the features for object-oriented programming. We assert that inheritance (via type extension) will allow us to implement more elegantly and probably unify the two approaches presented in this paper.

5. References

[Ada 83] Karl A. Nyberg Ed.; The Annotated Ada Reference Manual; ANSI/MIL-STD-1815A-1983, Grebyn Corporation, 1991.

[ARM 93] Ada 9X Mapping/Revision Team; Annotated Ada 9X Reference Manual; draft version 3.0, July 1993.

[Booch 91] Grady Booch; Object Oriented Design with applications; Benjamin Cummings, 1991.

[Chen 76] P. P. Chen; The Entity-Relationship Model: towards a unified view of data; ACM TODS, Vol. 1, Nb.1, 1976.

[Coad & al. 90] Peter Coad, Edward Yourdon; Object-Oriented Analysis; Second Edition, Yourdon Press, 1990.

[Coad & al. 91] Peter Coad, Edward Yourdon; Object-Oriented Design; Prentice-Hall International Editions, 1991.

[Date 86] C. J. Date; An Introduction to Database Systems; Addisson-Wesley, 1986

[Genillard & al. 89] Christian Genillard, Norbert Ebel, Alfred Strohmeir; Rationale for the Design of Reusable Abstract Data Types Implemented in Ada; ACM Ada Letters; Vol. IX, No 2, Mar/Apr 1989.

[Hirasuna 92] Michael Hirasuna; Using Inheritance and Polymorphism with Ada in Government Sponsored Contracts; ACM Ada Letters, Vol. XII, No 2, Apr 1992.

[Meyer 88] Bertrand Meyer; Object Oriented Software Construction; Prentice Hall, 1988.

[Parent & al. 88] Christine Parent, Stefano Spaccapietra; About entities, complex objects and object-oriented data models; Information System Concepts - An In-depth Analysis, Proceedings of an IFIP WG 8.1 Working Conference, Naur, October 18-20, 1989, E.D. Falkenbert Ed., North Holland, 1989.

[Rumbaugh & al. 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederic Eddy, William Lorensen; Object-Oriented Modeling and Design; Prentice-Hall, 1991.

[Seidewitz 87] Ed Seidewitz; Object-Oriented Programming in Smalltalk and Ada; OOPSLA'87 Proceedings.

[Shlaer & al. 92] Sally Shlaer, Stephen J. Mellor; Object Lifecycle, Modeling the World in States; Yourdon Press Computing Series, 1992

[Wirfs-Brock & al. 90] Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener; Designing Object-Oriented Software; Prentice-Hall, 1990.