Ada 95 as Implementation Language for Object-Oriented Designs

Stéphane Barbey

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: stephane.barbey @ di.epfl.ch
phone: +41 (21) 693.52.43 - fax: +41 (21) 693.50.79

Revised — 18 October 1995

ABSTRACT. In this paper, we show how Ada 95 can be used as an implementation language for object-oriented designs. We present a strategy to map Fusion class descriptions into Ada specifications, considering the various kinds of qualifiers that can be applied to attributes, and the various ways methods can be mapped. We also discuss issues such as naming conventions, mapping of operations, use of mixins and of generics. Finally, we show how bidirectional associations, that usually end up in a mutual dependency, can be implemented in Ada 95.

KEYWORDS. Object-oriented software development, Fusion, Object-oriented programming, Ada 95, Mixins, Genericity, Associations.

1. Introduction

The goal of this paper is to describe how Ada 95 [1] can be used as an implementation language for the designs of systems developed with an object-oriented method. For this paper, we have selected the Fusion method [13], which is currently used both in the industry —in application areas as various as printers, medical, test instruments, networking and MIS— and in academia, e.g. for the software engineering courses at EPFL (Swiss Federal Institute of Technology). However, most of the principles and ideas presented here can be useful in implementing designs developed from most other object-oriented development methods.

The basic principle of the Fusion method is to develop a system by detailing various models. Each model gives a view of an aspect of the system: the interaction with the agents, the relationships amongst the classes, the decomposition of the functionalities, the inter-object communication, etc. The ultimate step consists of gathering the information coming from all those models into class descriptions, which can then be mapped to various object-oriented programming languages.

This class description is independent from the implementation language, and, although the Fusion manual [13] only considers C++ and Eiffel, we will show how the class descriptions can easily and in a (semi-)automatic fashion be translated from the class description language into Ada 95. When appropriate, we will also make comparisons with implementation models of those other object-oriented languages.

However, implementing these class descriptions is not a simple task, and it goes further than just encapsulating types into packages. We will see how to implement the different aspects of an object-oriented system, such as attribute visibility, naming conventions, creation routines, class-wide generic subprograms, and so on by making the best use of the features introduced in Ada 95, such as tagged types, class-wide types, controlled types, or the new forms of generic parameters.

1.1 Plan

This paper is organized to give a systematic overview of the translation of the class descriptions into Ada 95.

In the second section, we give an overview of the Fusion method, introducing the various models and concepts that need to be implemented during the implementation phase.

In the third section, we give a general mapping strategy for programming classes and the different relationships among classes (inheritance, aggregation, association). We also consider the naming problems that can arise during the mapping of a class.

The fourth section focuses on attributes, with an emphasis on the different kinds of attributes and the different qualifiers that can be applied to them, giving information on their visibility or their lifetime.

The fifth section discusses the specification and the implementation of methods (operation for classes).

The two last sections deal with Ada-specific facets, showing

To appear in the Proceedings of TRI-Ada '95, November 5-10, 1995, Anaheim, California, USA. Printed with permission.

how some specific features of Ada can simplify the implementation of classes (for instance genericity), or how some problems specific to Ada can be solved (for instance the implementation of bidirectional associations).

1.2 Terminology

When using the word *class* in this paper, we refer to the notion of class used in the Fusion method. When speaking of classes in the Ada sense, we use the term *derivation class*.

The word *method* is used with two different meanings: either as an operation of a class, in the Fusion (and Smalltalk) terminology, or as a term to define the way in which some actions are carried out (e.g. the Fusion *method*, a *method* to map attributes). The use of this word should be clear from the context.

2. Fusion

Fusion [13] is a second-generation object-oriented development method, which covers all aspects of the software construction life cycle and includes strategies for verification and validation. It is called Fusion because it synthesizes the best features of the prominent object-oriented development methods: OMT/Rumbaugh [21], the Booch method [10], Objectory [16], and CRC [24]. Also, it includes some aspects coming from formal specification methods. The development of a system is based on a waterfall life cycle, but it could also be used for iterative development without much modification.

Throughout the whole development, a data dictionary is maintained to collect and check the consistency of the items introduced in the various models, together with some additional information, such as assertions on parts of the models or the initial values of the attributes.

2.1 Analysis

Fusion development starts with a phase of analysis, in which the developer elaborates the object model, the system interface and the interface model. The object model describes the different classes of the system, their attributes and their associations in a fashion similar to entity-relationship diagrams [12]. Among the relationships, one can find the traditional relationships found in other methods such as inheritance (subtyping), aggregation, and association.

For example, the banking system in figure 1 is composed of a class Bank, which manages several Accounts (thus the star in front of this class box). There are three kinds of accounts: simple accounts, checking accounts and saving accounts. Each account is owned by one Customer (which can in turn own several accounts). The bank keeps track of all transactions carried for each account (a transaction can involve one or several accounts), so that reports can be sent to the customer at the end of each month. Relationships can have attributes too. (Here, the relationship Owns holds the date when the Account was created and attributed to the Customer.)

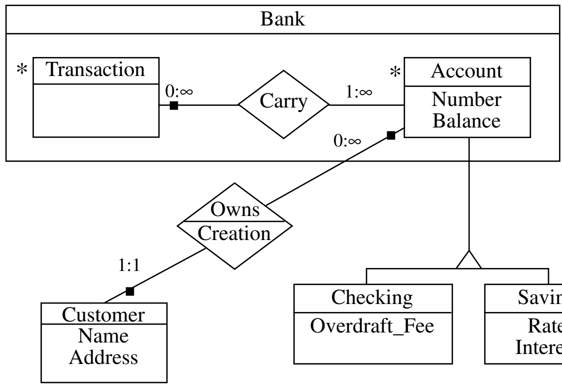


Fig. 1. Part of an object model for a banking system

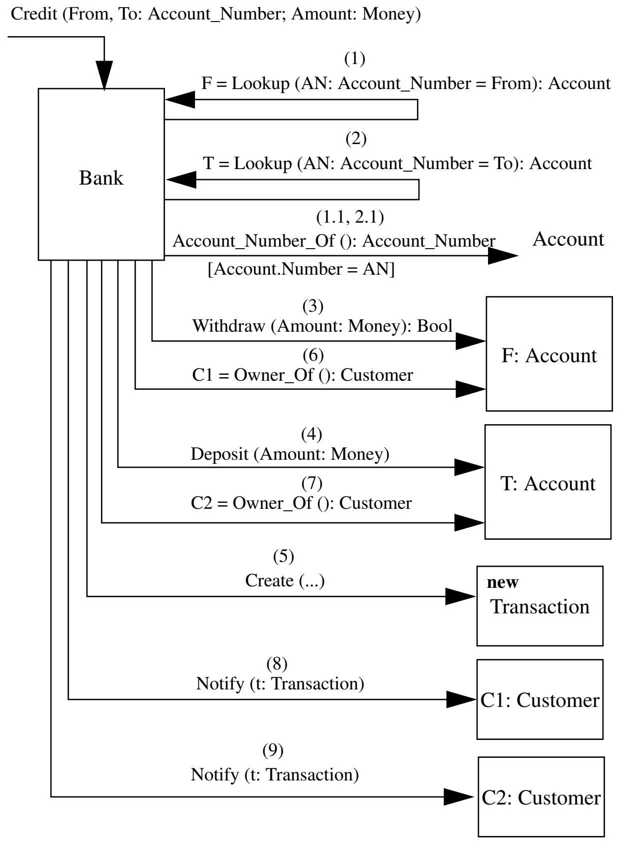
The system interface consists of a full description of the set of operations to which the system can respond, of the events that it can output, and of the list of agents that can interact with the system. The interface model consists of the description of a life cycle model and an operation model. The life cycle model defines the possible sequences of interaction in which a system can participate. It lists the various events they can send to and receive from the system, together with their arguments. The operation model defines the effect of each system operation. This description includes some formal semantics under the form of pre- and postconditions. However, the semantics of those conditions are not very rigorous, since their definition is not completely formalized.

2.2 Design

After analysis comes design. During design, the developer transforms the abstract models produced during analysis into software structures. In this phase, the developer must provide object interaction graphs, visibility graphs, inheritance graphs, and finally class descriptions. The object interaction graphs attribute each system operation described in the operation model to a class and describe a decomposition of their behaviour by distributing their functionality among the various classes of the system. The visibility graphs show how the system is structured to enable inter-object communication. The inheritance graphs complete the inheritance relationships already found during analysis by adding information on inheritance of implementation. In other words, during analysis, the supertype/subtype relationships are modeled, whereas in design the superclass/subclass relationships are found.

For example, the interaction graph of the operation Credit is given in figure 2. This operation withdraws an amount of money from one account and deposits it in another account, assuming that both accounts exist and that the withdrawal is authorized. (This authorization can depend on the type of account.) The operation is completed by keeping track of the transaction and notifying the owners of the account that the transaction was carried out.

It is decomposed in sending messages to either a particular object (Notify a customer, Create a transaction, etc.), or to the objects of a collection that fulfil a condition (find the account in the collection of accounts managed by the bank, the account number of which is From, etc.). The sequence in which methods are invoked is shown on top of the arrow describing the method invocation and summarized in text below the interaction graph:



Description:
operation Bank: Credit (From, To : Account_Number, Amount: Money)
Lookup two bank accounts whose numbers are From and To (1) (2).
If those two accounts exist,
    Withdraw Amount from the account whose number is From (3)
    If the withdrawal was successful, then
        Deposit the amount Amount in the account To (4)
        Create a transaction, and add it to the collections of transactions carried by the bank. (5)
Find the customers of the transaction (6,7), and Notify each of them that the transaction was carried out (8,9)

method Bank: Lookup (AN: Account_Number): Account
Lookup in the bank accounts of the bank the account whose number is AN (1.1, 2.1).

Fig. 2. Interaction graph for the operation Credit

Finally, the developer has to gather information coming from all these models and from the data dictionary to write a description of each class in the system. This description is the first step in coding the application. All information regarding the specification of each class is given: its various attributes, including their type and visibility information, its operations, including their various parameters and their result type.

For instance, the class Account, inspired by the example of Seidewitz [22] would look like the following:

```
type Money: delta 0.01 digits 15
type Account_Number: natural

class Account
   attribute Number:
                  bound exclusive Account_Number
   attribute Balance: bound exclusive Money
   attribute Owner: unbound shared Customer
   attribute Transactions:
                  exclusive bound col Transaction

   method Create (Initial_Amount: Money)
   method Deposit (Amount: Money)
   method Withdraw (Amount: Money): Bool
   method Balance (): Money
   method Owner_of (): Customer
   method Account_Number_Of (): Account_Number
   // other methods
endclass
```

Fig. 3. Class description of Account

The subclass Checking would be defined as:

```
class Checking isa Account
   attribute constant Overdraft_Fee:
      bound exclusive Money

   method Set_Fee (To_Fee: Money)
   method Withdraw (Amount: Money): Bool
endclass
```

Fig. 4. Class description of Checking

During the implementation phase, the programmer's job is to implement the class descriptions in the target language and code the behaviour of each method according to the descriptions of the interface model, the operation model, and the interaction graphs.

3. Mapping classes

3.1 General mapping strategy

The general strategy to map a class description into Ada 95 is to code it as an ADT, i.e. as the aggregation of

- a private tagged type (or a private type extension),
- possibly an access type (a general access-to-class-wide type) denoting its derivation class, so that access values designating objects of that derivation class can be created in a systematic way. The presence of this access type anticipates possible subtyping of this class in subsequent iterations, or when reusing the class, and allows for easy heterogeneous data structure handling,
- a set of (primitive) operations, and
- a package, to provide encapsulation.

The attributes are defined in the full type declaration, which appears in the private part of the package.

The operations defined in the class description are usually primitive operations, i.e. inherited by derivatives of the type. If they must be visible from any client of the class, they must be declared in the public part of the specification package. If some primitive operations must be visible only by the descendants of the class, they must be declared in the private part of the specification package (assuming the descendant is coded in a child package). Non-primitive operations can be declared in the package body, in which case they will only be usable by the class (and not inherited). In the interaction graphs, these private operations appear as method invocation whose client and server are objects of the same class. For example, on figure 2, the operation Lookup is an operation private to the class Bank: no other class is involved in this operation (neither as client nor as server).

Those three levels of encapsulation correspond respectively to the public, protected, and private encapsulation levels of C++. However, Ada allows the introduction of (C++-like) private operations without modification of the specification package. Thus, it avoids the recompilation of the specification of the class and of its clients, which can be a costly operation.

This strategy has been used extensively with Ada 83 (except for the ability to extend types). However, this strategy is not detailed enough to handle more complex descriptions, like the Fusion class descriptions. Furthermore, it does not take into account all the possibilities of Ada, such as generics. Generics do not appear in the standard Fusion notation, but some work has been carried out on extensions of the Fusion notation, for example the ION notation [2].

There can be exceptions to this general mapping strategy. For instance, closely related classes can be coded in the same package because they are related, or because they represent different aspects of the same class. For example, the two types used to implement sibling inheritance can be coded in the same package. (Sibling inheritance is the way multiple inheritance can be implemented in Ada 95 by binding two types using an access discriminant to the current instance of a type —see [6], §4.6.3). On the other hand, very big classes will probably be divided into several packages. For example, a plane in an air traffic control system may be an object too big to be programmed in a single package. However, in this case, the resulting class cannot easily be inherited from, because the operations not declared in the same package as the type are not primitive operations (and therefore are not inherited when deriving from the type).

3.2 Mapping aggregation

In the object model, aggregate classes appear inside their enclosing class. This notation is purely pragmatic —to avoid the unnecessary introduction of “has” associations. It has no consequence on the architecture of the implementation: aggregate classes can be coded in separate packages. These aggregate classes can also appear at different places in the object model, and be referenced by other classes than their enclosing class. Coding them inside the package of the enclosing class would not be practical, because their import would then require the import of the whole set of classes defined in the package.

3.3 Mapping associations

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, the association called Owns represents the connection between a Customer and an Account. This connection reflects the semantics defined between an account and the customer who owns it.

Associations do not appear in the design because they are replaced by visibility links on the interaction graphs. That is, they only appear as messages (e.g. on figure 2, Owner_Of), which can be used to find an object associated with another object. Thus, they do not constitute a software structure. However, in specific cases we shall examine in section 7, they must be coded as classes.

3.4 Mapping inheritance

During analysis, Fusion introduces inheritance as a means for subtyping, i.e. an object of the subtype can appear everywhere an object of the supertype is allowed. Fusion supports partitioned and non-partitioned subtyping, respectively indicated in the object model by a full and an empty triangle. In partitioned inheritance, the domain of the subtypes must cover all the derivation class, i.e. no object belongs to the parent type. In Ada, this can be enforced by making the parent type an abstract type. For example, on figure 1, a checking and a saving account can appear everywhere a simple account is allowed.

During the last phase of design, Fusion allows inheritance for implementation, to enhance reuse. This is mainly a case of non-partitioned inheritance.

Subtyping inheritance should be made visible to the clients of the class, and performed in the visible part of a child package of the parent type’s package. That way, the Ada code reflects the inheritance relationship directly in the software, and all information regarding the representation of the class are available to implement the subtype. Implementation inheritance should be performed in the private part of the package (possibly in a child unit too), to disallow meaningless (and dangerous) subtyping (see figure 5).

```
package Parent.Derived is

  type Derived_Type is
    new Parent_Type
      with private;
  -- the operations of
  -- Parent_Type are
  -- visible from here
  ...
private

  type Derived_Type is
    new Parent_Type
      with record
        ...
      end record;
  -- the (private) structure
  -- of Parent_Type is
  -- visible from here

end Parent.Derived;
```

subtyping inheritance

```
package Examples is

  type Example_Type is
    tagged private;
  -- the type is extended and
  -- can be extended, but
  -- its parent is not visible
  ...

private

  type Example_Type is
    new Implementation_Type
      with record
        ...
      end record;
  -- the operations of
  -- Implementation_Type
  -- are visible from here

end Examples;
```

implementation inheritance

Fig. 5. Subtyping inheritance vs. implementation inheritance

Multiple inheritance is allowed in Fusion. We will not give a strategy to map multiple inheritance, which is only needed in sporadic cases of multiple classification, since this topic is extensively discussed in the rationale ([6], §4.6). However, we will show some cases where multiple inheritance is used in the design, but can be avoided during the implementation. Also, we have been working on the design patterns of Gamma's *et al.* catalogue [14], and found it interesting that amongst the 23 patterns described in this catalogue, only two use multiple inheritance. In both cases, the authors note that there a more elegant solution exist, that does not require multiple inheritance.

3.5 Naming conventions

Several naming problems arise when converting a class description into Ada 95. These problems must be solved with practical and systematic naming conventions, that allow for (semi-)automatic code generation from the class descriptions.

Since two implementation entities —a package and a type— stand for a single design entity —a class—, a naming problem arises: the identifier of the class cannot name both the package and the type. This problem is of course not new to Ada 95, and it has already been discussed several places in the literature. Two main approaches have been proposed:

- keeping the name of the class for the package and using a generic identifier for the type, and
- using a Polish-inspired notation, by prefixing or suffixing the name of the class with an indication that distinguishes the identifier as a type name or a package name.

For Ada 95 specifically, two similar naming conventions have already been proposed for the first approach, e.g. the Roman-9X [11] and the Rosen [20] convention. Both conventions use the name of the class to identify the package, and use the identifier "Object" (respectively "Instance") to name the type. This naming convention is also used in the Ada to CORBA IDL mapping [15]. However, we disagree with this convention for several reasons:

- A name should describe a thing and not a property, so entities should be named as accurately as possible. Emphasis should be put on the type rather than on the package [7].
- This convention makes it difficult to put several tagged types in one package, whereas this may be a convenient thing to do, as mentioned in section 3.1.
- It is relatively absurd to name a type "Object" or "Instance". It makes the program harder to read, because the name of the package has to appear with each use of the type outside the package.
- This convention makes it harder to take advantage of use clauses, or can be confusing in type conversions, which arise more often in object-oriented programming than in traditional structured programming.

Consequently, we propose to use the second approach. Several conventions are possible. The most common is to name the type after the name of the class and add the suffix _Type. The package is named after the class, or, if possible, after a plural form of the name of the class. If the translation is automated, the singular form can be more suitable, especially if, in the developer's language, the plural form of a name is not systematically found from the singular form, as is the case in English (e.g., mouse becomes mice, or worse, sheep remains sheep). This convention has been used for years in the Ada community for example for the Ada 83 Booch components [9] or for the LGL free software components [23].

The second problem arises from the lack of a name for the controlling operand in the Fusion class descriptions. In the class descriptions, the controlling parameter (i.e. the object on which the operation is invoked) is anonymous and does not appear in a method description, a name must be chosen for it. Three alternatives exist to solve this naming issue:

- to use a generic identifier, like "Self", "Target", or "Controller",
- to prefix the name of the class (for example with "The_", e.g. The_Account)
- to use a more meaningful name with a semantic content.

This last option is of course more sound, but the two first conventions may be better options if the translation is automated.

Finally, programmers should be aware that the Fusion class descriptions are case sensitive, whereas Ada programs, applying sound readability principles, are not. This problem can often arise since Fusion encourages the use of mixed case for class names and the use of the same name in lower case to designate objects of that type. When using the Polish-like convention mentioned above to name the package and the type, this last problem can be easily solved.

However, we are aware that those naming problems are mostly a question of taste, and no ultimate solution is likely to be found.

4. Mapping attributes

4.1 Kinds of attributes

Two kinds of attributes appear in the class descriptions: data attributes and object attributes.

4.1.1 Data attributes

Data attributes are attributes of the standard data types (like Character, Integer, String, Float, user-defined enumeration types, etc.) or of any user-defined data type (in our example Money or Account_Number). The data attributes are variables and have a name, but no identity, i.e. it is impossible to distinguish two data attributes holding the same value. In the class Account, Balance is a data attribute of the user-defined type Money. Data attributes are represented as components of the record representing the class.

4.1.2 Object Attributes

An object attribute holds a reference to an object, for example to an enclosed object (in the case of aggregation), or more generally to any associated object. In the class Account, the owner of the account is an object attribute: it references an object of the class Customer. Instead of holding a reference to one specific object, Fusion also allows an object attribute to hold a collection of references, for example the transactions carried out on one account in the Account class description.

The object attributes are at the heart of communications in an object-oriented system. Such a system is basically a set of connected objects that work by passing messages to each other (i.e. by invoking operations): a client object sends a message to a server object, which can in turn return a result message (e.g. a function result, an out parameter, an exception, etc.). References to server objects exist in one of two forms, according to their "reference lifetime":

- A reference is *dynamic* if it only exists during the execution of an operation, and can be forgotten at the end of this operation.

  A dynamic reference can be coded either as a reference local to the operation, or as a parameter of the message, according to its lifetime (i.e. whether its lifetime is contained in the execution of the operation or not). For examples, the references T, F, C1 and C2 in figure 2 are dynamic references and are not coded as attributes of the class Bank, but as references local to the operation Credit.

- A reference is *permanent* if it must persist between different calls.

  A permanent reference must be held by the client object as an object attribute, or possibly as a reference local to the package in which the client is declared, depending on whether the reference is shared between all the instances of the class or not. (This is the same difference as between static and non-static members in C++.) For example, the bank must hold a reference to the collection of its accounts and its transactions, an account must keep track of who its owner is, etc.

From the designer's point of view, the information held by an object attribute is the identity of the referenced object. From an implementer's point of view, an object attribute can be coded in any way that enables access to the referenced object (for example by holding an identifier that allows looking up the referenced object in a table). However, an object attribute is usually coded as a component of the tagged record representing the class, i.e. either the referenced object itself or an access value to that object. The representation of object attributes can be more complex, for example if the developed system is a distributed application.

4.2 Attribute qualifiers

The Fusion method qualifies attributes with several "visibility" qualifiers that explain how the attributes (and the objects they can reference) are related to the enclosing object or the potential clients of the object.

attribute ::= [**attribute**] mutability Name **:**
sharing binding type

mutability ::= **constant** | **variable**

sharing ::= **shared** | **exclusive**

binding ::= **bound** | **unbound**

type ::= [**col**] Name $_{Class}$ | Name $_{DataType}$

Fig. 6. Syntax of an attribute definition and of the attribute qualifiers (reserved words appear in bold; the Name is the identifier of the reference)

- Server visibility (sharing qualifier)

  The server visibility relationship establishes whether, when invoking an operation, a server object is *exclusively* referenced by one client, or if it can be *shared* among many clients. At different times, the server object can be referenced by different clients, which means that during an invocation, the exclusivity of a reference need not hold. (Sharing has not the same sense as in concurrent programming: it only specifies whether an object can be designated by several references at the same time or not. Since Fusion is aimed at sequential programs, the protection of the object is implicit.)

- Server binding (binding qualifier)

  The server binding relationship deals with the lifetime of referenced objects. A referenced object is *bound* when the referenced object does not outlive its client, and *unbound* if it does.

- Reference mutability (mutability qualifier)

  Reference mutability specifies whether or not a reference can designate different objects or if it will constantly designate the same object. Thus, a reference is *variable* if modification is allowed, and *constant* otherwise. Fusion does not make a distinction between mutability within a derivation class and mutability within a type. This last issue is important in Ada, because Fusion assumes no difference between a specific and a class-wide type, whereas in Ada, the user has the possibility of choosing between those two options. Thus, we will augment the Fusion syntax of the mutability qualifier by adding **specific**.

  mutability ::= [**specific**] **constant** | **variable**

Fig. 7. Modified mutability qualifier

The absence of **specific**[1] (the default) means that the reference is class-wide and can designate any object in the whole derivation class rooted at the type. For a non-specific constant attribute, this reference cannot change during the lifetime of the object. The presence of the **specific** qualifier means that the attribute is of the specific type only. The mutability qualifier is a property of the reference, whereas the sharing and binding qualifiers are properties of the referenced object. It is possible for

[1] In a previous version of this paper, we used **class-wide** instead of **specific**. However, this solution is more elegant since it preserves the original semantics of Fusion. (i.e. class-wide by default)

a designer who does not want to modify the design language, to consider that all references are class-wide. However, as we will see below, this idea restricts the use of object values to map references.

No default values for attributes appear in the class description. Such values can appear in the data dictionary. The developer will possibly have to seek them there to complete the Ada specification.

4.3 Mapping the attributes according to their qualifiers

Attributes can be coded as object values (i.e. directly embedded into the object), as access values (a pointer to the object), or as functions.

The Fusion method recommends all object attributes to be coded as access values. Besides the arguments of bounding and sharing, which are valid in Ada and that we will see below, Coleman *et al.* [13] give this advice for efficiency reasons: to avoid the overhead of passing object values as parameters (implying a copy of the whole object on the stack). This reason is not valid in Ada, since tagged types are by-reference types.

However, Coleman *et al.* [13] also give reasons to map the object attributes as object values rather than access values:

- to access the data or the referenced object directly rather than through an indirection,
- to avoid extra free store allocation, and guarantee the synchronization of the lifetime,
- to avoid dynamic binding (dispatching), when invoking operations on the object attribute, in the case where the specific type of the component is known.

This last argument is not valid in Ada 95, because it is based on the assumption that in object-oriented programming languages, access values are dispatching and objects values are not, while in Ada 95, this distinction is made on the basis of whether the type is class-wide, regardless of its representation.

In some cases an attribute can also be coded as a function, if it is a calculated attribute, for example if its value is deduced from the values of other attributes.

4.3.1 Binding

There are basically two ways to implement a bounded attribute:

- as an object value of the enclosing object; this is the easiest and most convenient strategy,
- as a controlled access value, in which case the attribute (not the referenced object) must be controlled[2] and its lifetime terminated when finalizing the enclosing object. The attribute being controlled does not imply that the enclosing object itself is controlled: objects of a controlled type are initialized, adjusted and finalized even if they are components of a non-controlled composite type. For example, in the example on figure 8, the type Example_Type is not controlled. However the component Controlled_Component, which holds the object attribute is controlled. The Initialize operation can be implemented to create and bind the object referenced by the access value, and the Finalize operation is implemented so as to deallocate the memory (e.g. an unchecked deallocation), so that the attribute Component lifetime ends at the same time as the object to which it is bound.

```
class Example isa Base_Example
   attribute Component:
      exclusive bound Component_Type
endclass

with ...; use ...;
package Examples is

   type Example_Type is
      new Base_Example_Type
         with private;

private
   type Controlled_Component is
      new Controlled with
         record
            Component: Access_to_Component_Type;
         end record;
   procedure Initialize
      (C: in out Controlled_Component);
   procedure Adjust (C: in out Controlled_Component);
   procedure Finalize
      (C: in out Controlled_Component);

   type Example_Type is
      new Base_Example_Type with
         record
            Component: Controlled_Component;
         end record;
   -- the attribute is controlled, but neither
   -- its type, nor the enclosing type are.

end Examples;
```

Fig. 8. Component of a controlled type

This is a typical case where multiple inheritance is needed in other languages (to combine Controlled and Example_Type), but is not required in Ada 95.

References to unbound objects must be mapped as access values. An object attribute cannot be bound to more than one enclosing object.

4.3.2 Sharing

There are also two ways to share an attribute. If the attribute is implemented as an access value, it is automatically sharable. If it is an object value or a data attribute, it must be coded as an aliased[3] component. However, while an access value can be either bound or unbound, an object value is inherently bound. Thus, it is not possible to have unbound

[2] A controlled type is a type derived from Ada.Finalization.*[Limited_]*Controlled. Three user-definable primitive operations are defined for such a type: Initialize, which is invoked immediately after the normal default initialization of a controlled object, Finalize, which is invoked immediately before finalization of any of its components, and —for non-limited types— Adjust, which is automatically invoked as the last step of an assignment (see [1], §7.6).

shared components implemented as object values. Access values must be used for that purpose.

An attribute is made (bound) exclusive by making it a non-aliased (private) component of the object. (It then becomes possible to make copies of the attribute, but the copies will have distinct identities, and the object will not be shared.) Unbound exclusive attributes are mapped as access values.

Shared attributes must be visible to the clients of the class, while the exclusive components must not be visible. At the same time, shared and exclusive attributes can coexist in a class description. This causes a problem in Ada 95, since the components of a root type declaration or added by a given type extension are all either public or private. There are three solutions to this problem:

- separate the type in a private part type and a public part type,
- use discriminants,
- provide access and update operations for the shared attributes.

The first solution involves decomposing the class description into two types, the first being a public abstract type (from which no object can be created), that holds the shared components, and the second a private type, that hides the exclusive components.

For example:

```
with Customers, Finance;
use Customers, Finance;
package Accounts is

   -- the public part type:
   type Account_Public_Type is abstract tagged
      record
         Owner: Customer_Ref;
         -- a shared unbound attribute
      end record;
   -- (no primitive operations)

   -- the private part type:
   type Account_Type is
      new Account_Public_Type with
         private;
   type Account_Ref is
      access all Account_Type'Class;

   function Create (Initial_Amount: Money_Type)
      return Account_Type'Class;
   procedure Deposit
      (The_Account: in out Account_Type;
       Amount: in Money_Type);
   procedure Withdraw
      (The_Account: in out Account_Type;
       Amount: in Money_Type;
       Successful: out Boolean);
   function Balance (The_Account: Account_Type)
      return Money_Type;
   ... -- other operations

private

   type Account_Type is
      new Account_Public_Type with
         record
            Number: Account_Number;
            Balance: Money_Type;
            -- exclusive bound attributes
         end record;

end Accounts;
```

Fig. 9. Public and private attributes of a class

All methods appearing in the class description are coded as primitive operations of the private part type, which is really the type that implements the class and that clients should use.

There are several reasons why making attributes public is usually not a good idea:

- the representation of the class may change, and attributes could be mapped differently in different iterations over the Fusion models;
- the semantics of the class may change, for example to become a protected class (in the Ada 95 sense, see the rationale [6], § 9.6.1 on how protected and tagged types can be combined), in which case the direct reference to attributes would become forbidden.

The second solution is to declare any shared components discriminants of the type, as in the following:

```
with Customers, Finance;
use Customers, Finance;
package Accounts is

   type Account_Type (Owner: Customer_Ref) is
      tagged private;
   type Account_Ref is
      access all Account_Type'Class;
   -- various operations as on figure 9

private

   type Account_Type (Owner: Customer_Ref) is
      new Account_Public_Type with
         record
            Number: Account_Number;
            Balance: Money_Type;
         end record;

end Accounts;
```

Fig. 10. Shared attributes as discriminants

This strategy can only be applied when the mutability qualifier of the shared attributes is constant (because discriminants cannot be modified). Also the discriminant values must be known when declaring objects.

This is why we recommend using the third solution to get visibility on the shared components: the traditional method of providing access and update operations. These operations can potentially be inlined, so that no run-time overhead is added to the execution of the system when accessing the attribute.

[3] An aliased object (or component) is one that can be designated by an access value. This can be done by using the reserved word aliased in its declaration. The attribute Access, when applied to an object, returns an access value that designates that object (see [1], §3.9).

4.3.3 Mutability

The mutability qualifier is not perfectly mappable in Ada 95, because there is no way to declare a component as a constant, unless it is a shared attribute, in which case it can be coded as a discriminant (see above). For attributes non modelled as discriminants, it is therefore the responsibility of the programmer to check that no changes are made to a constant reference after its initialization. (The constant mutability qualifier is not applied to the object, but to the reference itself.)

Specific and class-wide references are implemented by using specific types and class-wide types respectively. Class-wide references are always mapped using access values, because record components of a class-wide type are not allowed (because such a type has no fixed, nor known maximal length.).

4.3.4 Summary

Here is a summary (in pseudo-Ada) of the strategies exposed above. (Naturally, anonymous access types are not allowed as written below.)

	bound	unbound

exclusive	`Attribute: Type;`
	
	`(in the private part)`
		`Attribute:`
		`access all` `Type;`
		`(in the private part)`

shared	`Attribute:`
	`aliased` `Type;`
	`or as a controlled component`
	`(in a public part or with an access and an update operation)`
		`Attribute:`
		`access all` `Type;`
		
		`(in a public part or with an access and an update operation)`

Fig. 11. Attribute qualifiers for specific mutability

	bound	unbound

exclusive	`Attribute:`
	`access all` `Type'Class;`
	`(in the private part)`
		`Attribute:`
		`access all` `Type'Class;`
		`(in the private part)`

shared	`Attribute:`
	`access all` `Type'Class;`
	`(in a public part or with an access and an update operation)`
		`Attribute:`
		`access all` `Type'Class;`
		`(in a public part or with an access and an update operation)`

Fig. 12. Attribute qualifiers for class-wide mutability

4.4 A strategy for collections

In addition to being a predefined type or an object type, the type that appears in a attribute definition can also be a collection of either data or object attributes. In this case, the developer will have to check through the object interactions graphs to see which of the available library component supporting collection abstractions (e.g. heap, stack, queue, list, tree, table) is suitable for implementing the attribute.

4.5 Mutual dependencies

Although this issue is not raised in the Fusion method, an Ada programmer should think of checking for mutual dependencies: mutual dependencies are allowed among classes in Fusion, whereas Ada does not allow mutual dependencies among specification packages. Many mutual dependencies can usually be avoided during the design phase.

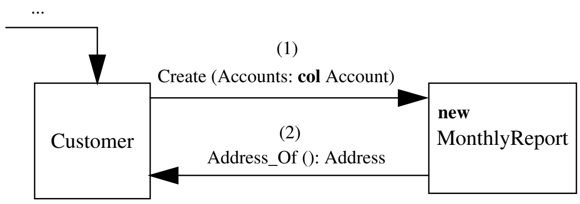


Fig. 13. Mutual dependencies between Customer and MonthlyReport

For example, in figure 13, there is a mutual dependency between Customer and MonthlyReport, because, when creating a monthly report, the class MonthlyReport must get a piece of information —Address— back from the Customer (and needs a reference to customer for that purpose). This mutual dependency can be avoided by giving the address of the customer directly as an argument to the method Create. Thus MonthlyReport need not hold a reference to the class Customer[4]

Some solutions also exist if the design cannot be changed. In figure 14, we show another example: each city knows of the country to which its belong, and each country has a capital city. Here, Ada allows mutual dependencies by decomposing the structure of one of the two classes in two interrelated types: one in the package specification (plus an incomplete type to hold the dependency), and one in the package body.

```
package Cities is
  type City_Type is
    tagged private;
  type City_Ref is access all
    City_Type'Class;
  -- operations on City_Type

private
  type Dependency_Type;
  type Dependency_Ref is
    access Dependency_Type;

  type City_Type is tagged
    record
      Name: String (1..20);
      Dependency:
        Dependency_Ref;
    end record;
end Cities;

with Countries;
use Countries;
package body Cities is
  type Dependency_Type is
    record
      Country: Country_Ref;
    end record;
end Cities;
```

```
with Cities; use Cities;
package Countries is
  type Country_Type is
    tagged private;
  type Country_Ref is access
    all Country_Type'Class;
  -- operations on Country_Type

private

  type Country_Type is tagged
    record
      Name: String (1..20);
      Capital: City_Ref;
    end record;

end Countries;
```

Fig. 14. Mutual dependency between City and Country

4. While this solution seems obvious, it was the main source of mutual dependencies in some student projects, because this design style is encouraged by certain examples of the Fusion manual [13].

However, this solution is not completely satisfying, because the primitive operations of the one class partly modelled in the package body cannot include parameters of the other type.

Another solution is shown in figure 15. It is very similar to the previous solution, in the sense that it also consists of modelling a class with two types: an abstract type, that the mutually dependent class can reference, and a concrete type in a child unit. This time, it is also possible for operations to include references to the mutually dependent type.

```
package Cities is
  type City_Type is abstract
    tagged null record;
  type City_Ref is access
    all City_Type'Class;
end Cities;

with Countries; use Countries;
package Cities.Real_Cities is
  type Real_City_Type is
    new City_Type with
      private;
  function Country_Of
      (C: Real_City_Type)
    return Country_Ref;

private

  type Concrete_City_Type is
    new City_Type with
      record
        Name: String (1..20);
        Country: Country_Ref;
      end record;
end Cities.Real_Cities;
```

```
with Cities; use Cities;
package Countries is
  type Country_Type is
    tagged private;
  type Country_Ref is access
    all Country_Type'Class;
  function Capital_Of
      (C:Country_Type)
    return City_Ref;

private

  type Country_Type is tagged
    record
      Name: String (1..20);
      Capital: City_Ref;
    end record;
end Countries;
```

Fig. 15. Mutual dependency between City and Country

Since the abstract type City_Type cannot have instances, the attribute Capital will of course only designate objects of Real_City_Type.

In section 7 we give another solution to this problem for the case of bidirectional associations.

5. Mapping methods

Mapping methods into primitive operations is quite straightforward, because the syntax and the semantics of the method definition is very close to the Ada syntax for subprograms, except for the controlling operand, which in Ada must be named.

method ::= **method** Name ArgList [: Type = expression]

Fig. 16. syntax of method declarations

Fusion specifies that expressions must be given to specify the behaviour of each method with a result type. However, it does not precise whether side effects are allowed in expressions.

The only other issues the programmer must be aware of are that:

- a mode (**in**, **out**, **in out**) must be selected for each parameter of an operation, including the controlling operand. This issue can be solved by examining the semantics of each defined operation.
- the mapping of methods with a result type does not automatically lead to an Ada function. If the controlling operand or any parameter is modified by the method, then the method can be mapped as a procedure with an **out** parameter for holding the result of the method. As long as no change of identity is required from one of the **in out** parameters, it can also remain a function by using **access** parameters instead of **in out** parameters.

  For example, the method Withdraw on figure 3 cannot be implemented as a function, because the anonymous parameter of type Account is modified by the method. Thus, in figure 9, it is coded as a procedure, with two additional parameters: the controlling operand —the **in out** parameter The_Account— and the **out** parameter Successful that holds the result of the function.
- the type of a parameter must be the related class-wide type and not the specific type if the parameter can belong to the derivation class rooted at the specific type instead of just the specific type.

  For example, every parameter of type Account in the class descriptions (except for the controlling operand, of course) should be mapped to the class-wide type Account_Type'Class, so that checking and saving accounts can be freely substituted for the simple account.

This third rule also holds for coding some cases of dynamic references. For example, the references F and T in figure 2 can reference any type of account and must be coded as being of type Account_Type'Class instead of just Account_Type. Failing to do so has dramatic consequences for the validity of the code, because the methods called for these references would be the methods defined for the root type Account, and would not dispatch to the possibly overridden operations of the reference. In this specific case, withdrawing money from a checking account with an insufficient balance would be rejected even if the overdraft is allowed, because the implementation of Withdraw for the class Account would always be invoked instead of the overridden implementation provided in the class Checking.

5.1 Creation methods

Although the operation model of Fusion defines how to create objects (for example, the creation of an object Transaction is explicit on figure 2.), it does not define the creation methods in a systematic way. It is therefore the responsibility of the programmer to make sure that all objects of the system can properly be created and initialized by providing the adequate creation routine(s) in the form of functions that take the initial values of the attributes of a created object and return the object. While performing this work, the developer should make sure that:

- the creation routines are not inherited, because they usually have no semantics in the context of a descendant. Strategies to avoid inheriting an operation include the use of a class-wide return type instead of a specific return type, as explained in [3], and declare creators separately, for example in a subunit or a child package of the package that contains the definition of the class.

- if the creation routine must absolutely be invoked in order to create an object, the type should be an indefinite type[5], which will force all objects to be created by a call to a creation method, or by the copying of a preexisting object. However, records are not allowed to have components of an indefinite type. Thus, using indefinite types compels the use of object values to code the object attributes of those types. More details on indefinite types are given in [18].

If some attributes cannot be initialized with a default expression, then those components should be controlled one way or another (see section 3.3.2) and initialized during the implicit invocation of the Initialize operation.

6. Using generics

Although generic units are not part of the description language, they remain an important building block in the construction of Ada specifications. They can be of great help in better modularizing systems and in building better abstractions.

6.1 Using generics to suppress coupling

The foremost use of generics in the translation of object-oriented designs into implementation is to enhance modularization, and thus reuse. Instead of coupling a class to another class, and introducing a strong coupling between the entities of the system, it is possible to abstract the properties needed by a class in its generic parameters, so that classes can be developed and tested individually. Such generic subsystems have the properties of reusable frameworks.

6.2 Using generics to suppress inheritance

Genericity can also be used as a substitute for inheritance, as long as no heterogeneous collections are involved [19].

6.3 Using generics to suppress protocol classes

Generics can be used to suppress protocol classes, i.e. abstract classes that only define the set of properties that an object should have to be used in a certain context. One example of this is a protocol class Sortable (see figure 17), that only defines the property of having the “less_than” operator, and a class Sorted_List that stores only objects of the class Sortable.

Classes like Sortable are sometimes called *functionoid*, because they do not constitute objects, but functions to be considered as objects, and are of great use in “pure” object-oriented languages, where all entities must be coded as classes. However, in Ada, this design can easily and elegantly be replaced with a generic package, by integrating the protocol Sortable into the generic parameters of the package Sorted_List.

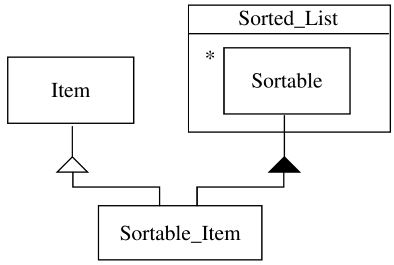


Fig. 17. Object model showing the protocol class Sortable

In this case too, while multiple inheritance would be used in other programming languages (this specific example being taken out of the Eiffel class library), Ada 95 can do the same thing with its own building blocks.

6.4 Specifying and implementing “inherited” generic subprograms

A problem that developers can have when implementing some operations or classes as generic units is that the generic subprograms are not defined to be inheritable operations, and thus are not inherited by the descendants of a formal type.

To circumvent that problem, it is necessary to consider generic methods as being operations of a derivation class rather than of the specific type, and to implement them in terms of possibly new (private) operations that the subclasses will have to override.

For example, in the following Stack example, the iterator Traverse takes a class-wide parameter, to show that it belongs to the whole type hierarchy rooted at Stack_Type, and it is implemented in terms of the (abstract) operation Item, that must be overridden by the concrete descendants of Stack_Type (Bounded_Stack_Type, Managed_Stack_Type, etc.).

```
generic
   type Item_Type is private;
package Stacks is
   type Stack_Type is abstract tagged private;

   procedure Push (Stack: in out Stack_Type;
      Item: in Item_Type) is abstract;
   procedure Pop (Stack: in out Stack_type;
      Item: out Item_Type) is abstract;
   function Size (Stack: Stack_Type) return Natural;

   function Item (Stack: in Stack_Type;
         Number: in Natural)
      return Item_Type is abstract;

   generic
      with procedure Action (Item: in Item_Type);
   procedure Traverse (Stack: in Stack_Type'Class);

private

   type Stack_Type is abstract tagged
      record
```

[5] An indefinite type is a type whose objects cannot be declared without an explicit initial value. Indefinite types include types with unknown discriminants (**type** T (<>) **is** ...), unconstrained array types, etc.

```
         Size: Natural := 0;
      end record;

end Stacks;

package body Stacks is

   function Size (Stack: Stack_Type)
      return Natural is ...

   procedure Traverse (Stack: in Stack_Type'Class) is
   begin
      for Cursor in 1..Size (Stack) loop
         Action (Item (Stack, Cursor));
      end loop;
   end Traverse;

end Stacks;
```

Fig. 18. A generic class-wide program with a (private) utility subprogram Item

For the sake of the example, the implementation of this iterator is not the most efficient one. Furthermore, the primitive operation Item should be a private operation of the type Stack_Type (i.e. declared in the private part of Stacks). However, Ada 95 prevents us from declaring abstract operations in the private part of a package.

6.5 Using generics to program mixins

Finally, generics are also useful to create mixins [5]. Mixin classes do not appear explicitly in the various models of Fusion. The developer will have to find them by himself, by exploring the object model. All classes that are inherited by more than one class, but are not referenced by any other class in the object model can be designed as mixins.

A good hint to find mixins is to look for classes whose name is an adjective or a past participle. For example, in figure 17, which models the management of a pet shop, some dogs and some cats can be raced. In the object model, this property can appear as either a superclass for raced cats and dogs, as an association (a cat has zero or one race), or as a property common to non-related classes (a common attribute Race).

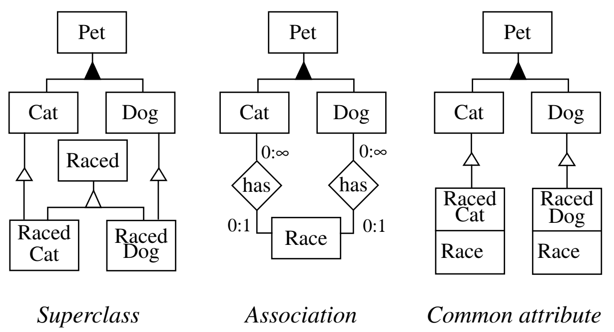


Fig. 19. Three object models leading to a mixin class Raced

In the three cases, “raced” can be coded as a mixin class, as long as it is not referenced directly by a class other than its descendants. That is, there are no object attributes holding a reference to Race (but there are attributes that reference Raced_Cat or to Raced_Dog).

```
with ...; use ...;
generic
   type Animal is new Pet with private;
package Raced is
   type Raced_Animal is
      new Animal with private;
   function Description (R: Raced_Animal)
      return String;
   -- gives information on the animal,
   -- including its race
private
   type Raced_Animal is
      new Animal with
         record
            R: Race;
         end record;
end Raced;
```

Fig. 20. A mixin package for the class Raced

Programmers should be aware that the fact that mixin classes do not exist if not connected to an ancestor type does not mean that there is no way they can used as building blocks. For example, it is possible to build utility packages that go on top of mixins using the Ada 95 feature of formal package parameters.

```
with Raced;
generic
   with package Raced_Pet is new Raced (<>);
package Utility_for_Raced_Pet is
   ... -- makes use of the operation
      -- Description defined in Raced.
end Utility_for_Raced_Pets
```

Fig. 21. A utility package for a mixin class

7. Implementing bidirectional associations

The most obvious way to model associations is to translate the links into references. This is how we have translated unidirectional associations (in which only a member of the association is interested in knowing its associate(s)). However, this way of translating associations is not satisfying for all purposes:

- If the association is bidirectional (both members of the association are interested in knowing their associate(s), for example, a customer needs to know the accounts he has opened at the bank, while the bank must trace the owners of all its accounts, to send them reports) it implies mutual references between objects, which cannot be fully achieved in Ada.
- It is not appropriate for associations relating one object to many others, or for optional associations. (In the optional case, a possible strategy consists of creating a subclass for the associated object and adding the reference in that subclass.)
- 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. When the links are part of the class, reusing this class

implies reusing all the classes it is associated with, even those which are not part of the problem domain to be developed.

To handle bidirectional associations, we propose to model the bidirectional associations as a class. The type exported by the package is a record type whose components are the attributes of the association and the links to the associated objects. The identity of the associated objects will serve as links, generally access discriminants designating the associated objects. Of course, we cannot use the objects themselves as links, to preserve the integrity of the system. In the example below, the package Ownership_G is made generic to satisfy the independence principle, on the class Account and Customer, although this is not necessary, but it could be replaced by a with clause.

```
with Calendar;
generic
   type Account_Type is tagged private;
   type Customer_Type is tagged private;
package Ownerships_G is
   type Ownership_Type
         (Account: access Account_Type'Class;
          Customer: access Customer_Type'Class) is
       limited private;
   -- the constant components Account and Customer are
   -- directly accessible (using the selector
   -- notation)
   function Opening_Date
         (Ownership: Ownership_Type)
      return Calendar.Time;
   -- returns date of election of the Account
   -- as head of Customer

   -- other operations on type Ownership_Type
private
   type Ownership_Type
         (Account: access Account_Type;
          Customer: access Customer_Type) is
      limited
         record
            Creation: Calendar.Time;
         end record;
end Ownerships_G;

with Accounts, Customers, Ownerships_G;
package Ownerships is new Ownerships_G
    (Accounts.Account_Type,
     Customers.Customer_Type);

with Ownerships, Customers, Accounts;
use Ownerships, Customers, Accounts;
procedure Usage is
   a_Account: aliased Account_Type := ...;
   a_Customer: aliased Customer_Type := ...;
   a_Ownership: Ownership_Type
                     (A_Account'Access,
                      A_Customer'Access);
begin
   ...
end Usage;
```

Fig. 22. Bidirectional association modelled as a class

The instances of the type Ownership_Type can then be stored in a collection, with selector functions to find an associated object when given the other one. More details on that topic can be found in [17].

8. Summary and conclusion

A good test of an object-oriented programming language is to see whether it can easily implement an object model defined by a popular object-oriented development method, such as Objectory, OMT or Fusion. This paper has shown that the building blocks that Ada 95 provides are usable and allow the programmer to create classes in an elegant fashion: packages for encapsulation, tagged types and type derivation for inheritance, class-wide types for mutability, access values, aliased objects and controlled types for other attribute qualifiers.

We have also shown that features such as generics can add reusability to object-oriented programming by helping suppress unnecessary coupling and providing mixin inheritance. We have also shown that multiple inheritance is not necessary in most cases where it would be needed in other object-oriented programming languages.

Of course, this work is not complete: it does only cover the translation of class descriptions. It does not deal with the coding of the bodies of the operations, nor error handling or the global behaviour of the system (described in the lifecycle model.) These issues are however rather straightforward in Ada, given the strategies for mapping class descriptions described above.

In the future, we plan to implement a tool, written in Ada 95, that handles the data dictionary and from which we will be able to generate Ada specifications of class descriptions in a (semi-)automatic fashion. Simultaneously, we will work on the test of those class descriptions, by first introducing more formalism in the definition of the semantics (for example using the object-oriented specification language CO-OPN/2 [8]). We aim to generate to generate automatic test sets and an oracle for the Fusion classes coded to Ada [4].

9. Acknowledgements

Gary Dismukes, Robert Duff, and Gabriel Eckert own my gratitude for their careful reading of earlier versions of this paper and their thoughtful suggestions.

Also, I would like to thank the anonymous referees for their helpful comments, and Dorothea Beringer for her advice on interaction graphs. The section on the implementation of associations is directly derived from work performed with Catherine Jean-Pousin, to whom I am also grateful.

Moreover, the author thanks Stef van Vlierberghe and Shawn Hannan for the comments that led to the revision of this paper.

The work reported in this paper is supported by the Swiss National Science Foundation (Nationalfonds), grant number 21-36038.92.

10. References

[1] Ada 9X Mapping/Revision Team. *Ada 95 Reference Manual*. Intermetrics, Inc., Feb. 15 1995. ISO 8652:1995 (E).

[2] C. Atkinson and M. Izygon. ION – a notation for the graphical depictions of object-oriented programs. RICIS Report RB.02a.1, Research Institute for Computing and Information Systems, University of Houston, Clear Lake, USA, 1995.

[3] S. Barbey. Working with Ada 9X classes. In C. B. Engle, Jr., editor, *TRI-Ada 1994 Conference*, pages 129–140, Baltimore, Maryland, USA, Nov. 6-11 1994. Also available as Technical Report (EPFL-DI-LGL No 94/65).

[4] S. Barbey. Testing Ada 95 object-oriented programs. In *Proceedings of Ada Europe '95*, Oct. 1995. (to appear).

[5] S. Barbey, M. Ammann, and A. Strohmeier. Open issues in testing object-oriented software. In K. Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257– 267, Basel, Switzerland, Oct. 17-20 1994. vdf Hochschulverlag AG an der ETH Zürich. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[6] J. Barnes, B. Brosgol, K. Dritz, O. Pazy, and B. Wichmann. *Ada 95 Rationale*. Intermetrics, Inc., Cambridge, MA, USA, Feb. 1995.

[7] S. Berner. Semantic naming convention and software quality. In K. Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 56–65, Basel, Switzerland, 1994. vdf Hochschulverlag AG an der ETH Zürich.

[8] O. Biberstein and D. Buchs. CO-OPN/2, an object-oriented specification language based on hierachical algebraic nets. In R. Wieringa and R. Fenstra, editors, *Proceedings of ISCORE '94*, pages 47–62, Sept. 1994. Also available as technical report (EPFL-DI-LGL No 94/76).

[9] G. Booch. *Software components with Ada : structures, tools and subsystems*. (Benjamin/Cummings series in Ada and software engineering). Benjamin/Cummings, 1987.

[10] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin-Cummings, second edition, 1994.

[11] G. J. Cernosek. Roman-9X: a technique for representing object models in Ada 9X notation. In C. B. Engle, Jr., editor, *TRI-Ada '93 Conference Proceedings*, pages 385–406, Seattle, Washington, Sept. 18-23 1993.

[12] P. P. Chen. The entity-relationship model: towards a unified view of data. *ACM TODS*, 1(1), 1976.

[13] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development The Fusion Method*. Object-Oriented Series. Prentice Hall, 1994.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addisson-Wesley, 1994.

[15] V. Giddings, editor. IDL => Ada language mapping specification. OMG request for comment submission, June 27 1995. OMG Document #95.5.16 (available at URL http://conf4.darpa.mil/corba-ada/).

[16] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addisson Wesley, 1994. Revised printing.

[17] C. Jean-Pousin and S. Barbey. Implementing associations with Ada. In *Software Engineering & its Applications 1993*, pages 149–158, Paris, France, Nov. 15-19 1993. EC2. Also available as Technical Report (EPFL-DI-LGL No 93/31).

[18] M. Kempe. Abstract data types are under full control with Ada 9X. In C. B. Engle, Jr., editor, *TRI-Ada'94*, pages 141–152, Baltimore, Maryland, USA, Nov. 6-11 1994. Also available as Technical Report (EPFL-DI-LGL No 94/66).

[19] J.-P. Rosen. What orientation should Ada objects take? *Communications of the ACM*, 35(11):71–76, Nov. 1992.

[20] J.-P. Rosen. A naming convention for classes in Ada 9X. *ACM Ada Letters*, XV(2):54–58, Mar.-Apr. 1995.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[22] E. Seidewitz. Object-oriented programming with mixins in Ada. *ACM Ada Letters*, XII(2):76–90, Mar.-Apr. 1992.

[23] A. Strohmeier. Use of a software component library in student projects. In A. Finkelstein and B. Nuseibeh, editors, *ACM/IEEE International Workshop on Software Education (ICSE)*, pages 319–326, Sorrento, Italia, 1994.

[24] R. Wirfs-Brock, B. Wilkerson, and R. Wiener. *Designing object-oriented software*. Prentice-Hall International, 1990.

Biography

Stéphane Barbey graduated from the C.S. Dept. of EPFL in 1992. He is a research assistant at the Software Engineering Laboratory of EPFL since 1992. His domains of research include object-oriented technology, especially the coding and testing phases of the software life cycle. He is a member of the Swiss delegation to the Ada working group of ISO (WG9), and takes part in the translation of the Ada 95 reference manual into French.

His home page is located at http://lglwww.epfl.ch/Team/SB.