Working with Ada 9X Classes

Stéphane Barbey

Swiss Federal Institute of Technology in Lausanne
Software Engineering Laboratory
EPFL–DI–LGL
CH–1015 Lausanne
Switzerland

e-mail: *Stephane.Barbey @ di.epfl.ch*

Revised version, 18 November 1994

Abstract

Ada 9X refines the Ada 83 concept of class and introduces class-wide types to support run-time polymorphism. We study how the Ada definition of class compares to other definitions of this term. We examine how run-time polymorphism is introduced in Ada 9X, how it relates to the different concepts of subtype, and how it compares with other forms of polymorphism. We also discuss the different uses of class-wide types, mainly as a means for dynamic binding (dispatching). However, since Ada 9X has adhered to the philosophy of building blocks, class-wide entities can be used for other purposes than dispatching. These uses are idioms that every programmer should be aware of. Some of these idioms are presented here. We also examine how classes are integrated with generics.

Keywords

Ada 9X, Object-Oriented Programming, Class, Polymorphic types, Run-time polymorphism, Dispatching, Genericity, Class-wide programming, Subtype, Anchor, Mixin.

1. Introduction

One of the goals of the architects of Ada 83 [1] was to design a language which would promote reuse. This goal has been achieved by providing a powerful unit of modularity, the package, in which the interface and the implementation are separated, and a powerful means of parameterization, the generic program unit. The two concepts are well integrated in a strong and static type system. This architecture has led to safe programming practices and facilitated the construction of solid software components.

However, in the last decade, new software development methods, such as object-oriented technology and prototyping, have risen. The need for other means of reuse was felt, to ease the construction of components by refining existing components: programming by specialization/extension. Therefore one of the major requirements for the revision of Ada was to support this new kind of reuse by introducing inheritance and run-time polymorphism.

Inheritance is a mechanism to build a new component by only specifying the difference from an existing base component. Inheritance is added to Ada by enhancing the derivation mechanism, and introducing extensible types (tagged types).

Polymorphism, in its broadest definition, is the ability for an entity to have multiple forms. In programming languages, it describes the ability for a name to designate entities of different types, and implies the possibility to write subprograms, called polymorphic subprograms, which can act on those entities regardless of their specific type. As we will see later, there exist several variants of polymorphism, some of them being already present in Ada 83. The kind of polymorphism introduced in Ada 9X, run-time polymorphism, is intimately linked to inheritance. It is introduced by refining the notion of class with class-wide types. Although run-time polymorphism is orthogonal to inheritance, both those concepts come usually in pair to bring the full power of object-oriented programming.

The goal of this paper is to show what *class* and *class-wide* mean in Ada 9X, how programmers can take advantage of these concepts, and also some of their inherent dangers.

The structure of this paper is as follows:

In section 2, we will examine the usage of the term *class* as found in the new reference manual [2] by comparing it to the concept of class found in object-oriented and functional programming languages based on the mathematical definition of class. We will also introduce the concept of polymorphic type.

In section 3, we will show what the language-defined classes are, and how a programmer can define his own classes. We will also discuss the different uses of the word *subtype* in both revisions of Ada and in the object-oriented terminology.

In section 4, we will see how programmers can benefit from classes by programming with the new attribute 'Class, how this attribute can be used, and in which contexts it can appear. We will also examine some idioms of the language in which the attribute 'Class appears.

In section 5, we will examine the varieties of polymorphism offered by Ada 9X in the light of the class-wide types.

In section 6, we will examine some cases where the concept of class interacts with genericity.

2. Classes and types

A programmer studying the object-oriented features of Ada 9X [5], may be astonished by the use of the term *class*, i.e. *a set of types sharing common properties and closed under derivation*, and by the lack of a syntactic construct called class or object. However, as pointed out by G. Eckert in [14], there is no consensus in the object-oriented community to define the notion of class. Its definition ranges from a template from which objects can be instantiated —in programmer's jargon— to a collection of objects —in the database area. In most cases, classes are used as a means of classification, to describe families of entities that share properties, and can be handled in a similar fashion. The definition of Ada 9X is akin to that loose definition, but lies on firmer foundations:

- The revision team choose to keep the definition of *class* that appears in Ada 83 (§ 3.3) —although the term is not widely used in the reference manual [1]—, and to improve it by introducing user-defined classes through derivation. This definition of *class* is close to the mathematical usage of the term, and the same definition can be found in functional programming languages such as Haskell [15]. The idea behind this definition of *class* is to introduce classification by quantifying over sets of types on which some constraints are applicable, such as the types whose elements can be compared. The types that conform to these constraints are grouped in *type classes*, which correspond to *polymorphic types* [15], the properties of which are available for all elements whose type is a member of the class.
- Ada already has the right term for the Eiffel-[1] ([21], [22]) or C++-like[2] [26] definition of *class* as a template for objects: *type*. According to G. Booch [7], although the words *class* and *type* are usually interchangeable, class emphasizes the membership to a classification, whereas type emphasizes the importance of conformance to a protocol, which is what most of the above-mentioned programmers think of when they use the word *class*: the structural definition of the instances (objects), the means of generating instances, and the interface specification through the description of a set of operations. To suppress the ambiguity of this terminology, the designers of the revised language chose to differentiate the two terms and to keep the word *class* for classification purposes.

Moreover, whereas classes, types and modules share the same syntactic construct in the Simula-based object-oriented languages (like Eiffel) or extensions (like C++), Ada 9X makes a point of keeping the *package* as basic construct for modularity and encapsulation, and the *type* as construct for describing the properties of objects. It would have been misleading to introduce *class types* or *typed packages*, since both of these extensions would have introduced ad-hoc and redundant constructs in the language.

Ada is not the only language to keep this distinction. Other modern programming languages such as Modula-3 [9] and Oberon-2 [23] incorporate the same design choice. However, whereas these programming languages are built on flat namespaces, Ada is the only language to have enhanced the concept of modularity to take object-oriented systems into account for the management of the namespace with hierarchical libraries.

3. Language- and user-defined classes

A class is a homogeneous set of types: its elements all share a common set of properties (which we will call the *properties of the class*). Ada has two kinds of classes, the language-defined classes, the shared properties of which are static and defined in the reference manual, and user-defined classes, the properties of which are defined by the programmer.

3.1 Language-defined classes

Several language-defined classes of types exist in Ada 83, such as the classes of scalar types, of task types,... A special syntax is provided by the language to declare types belonging to each of these classes. The language-defined classes appear in generic contracts, but are otherwise not explicitly used in programming techniques. They are however meaningful to describe the language. Ada 9X provides the same collection of language-defined classes, albeit refined to take the new semantics of the language into account. For example, the class of *private types* is not part of the language any more, since *private-ness* is now a property of a view of a type.

3.2 User-defined classes

In addition to these language-defined classes, Ada 83 programmers already had means to create their own classes, using type derivation. Type derivation is a mechanism used to let a new type (the derived type, also called descendant) inherit the data structure and the operations of some other type (the parent type). However, in Ada 83, type derivation was aimed at creating independent abstractions that "incidentally" share a common set of properties (implementation inheritance), i.e. it allowed to inherit the structure and behavior of a type, but there was no classification purpose

1. "Classes, defined as the modular units of software decomposition, also have another role: they serve as a basis for the types of Eiffel." [22]
2. "A class is a compound type containing a sequence of objects of various types, a set of functions for manipulating these objects and a set of restrictions on the access to these objects and functions." [26]

behind this mechanism [16]. Thus, type derivation has not been considerably used in Ada 83 programs.

To circumvent this problem, Ada 9X refines the type derivation mechanism by introducing *tagged types* and *type extension*. A tagged type is a record type the derivatives of which can be extended with additional components. Like in Ada 83, operations defined on the parent type —called primitive operations— are also defined for the derived type and may be redefined (overridden) if necessary. New primitive operations can also be added for the derived type, but no primitive operation can be removed, so that a type has at least the same features as its parent type. Every type derived from a tagged type is also a tagged type, and can therefore be extended too.

```
type Vehicle is tagged
   record
      Passengers: Natural;
      Speed: Float;
      Position: Location;
   end record;

procedure Set_Speed (V: in out Vehicle;
                     S: in Float);
function Speed_of (V: Vehicle) return Float;
procedure Repair (V: in out Vehicle);
procedure Move (V: in out Vehicle;
                L: in Location);
procedure Swap (L, R: in out Vehicle);


type Car is new Vehicle with
   record
      Plate_Number: String (1 .. 8);
   end record;

procedure Repair (C: in out Car);
   -- overridden
procedure New_Tires (C: in out Car);
   -- subclass-specific

   -- Move, Set_Speed, Speed_of and Swap
   -- are inherited without change


type Sports_Car is new Car with
   null record;


type Plane is new Vehicle with
   null record;
```

Fig. 1. A hierarchy of vehicles

In figure 1, a root type Vehicle has been declared, with five primitive operations; the type Car inherits from Vehicle, adds a component Plate_Number, overrides the operation Repair, and has a new primitive operation New_Tires. The type Sports_Car is derived from Car, but does not add new components, nor modify the inherited operations. The type Plane is derived from Vehicle in the same fashion.

All types derived and extended (directly or indirectly) from a particular tagged type belong to a *class* rooted at the latter type; thus, a class subsumes an open-ended hierarchy of types, including the union of their sets of values and operations. A tagged type is the root of a class, and defines the set of properties, values and operations, that will be common to all element of that class.

To distinguish between language- and user-defined classes, the new language reference manual designates user-defined classes as *derivation classes* instead of using *class* alone.

This type extension mechanism provides inheritance in Ada 9X.

3.3 Subtypes

Another problem of vocabulary that may arise is to distinguish between the different definitions of the concept *subtype*.

In the traditional object-oriented terminology, a particular type S is a *subtype* of T if S provides at least the functionality (*conformance of behavior*) of T [6]. S objects can always be used in any context in which a T object is expected. This is called Liskov's substitution principle [19].

In Ada 83, a subtype S of a type T characterizes a subset of the values of the type. The subset is determined by a constraint C on the type. A value belongs to the subtype S if it belongs to the type T and satisfies the constraint C.

In Ada 9X, a subtype has basically the same definition as in Ada 83. However, Ada 9X makes a stronger difference between types and subtypes. Only subtypes are nameable. Therefore, a type declaration declares a type—which has no name— and its first subtype, which is named after the identifier used in the declaration. This difference has no consequences for programmers, but it helps to describe the language in a more formal and precise way.

A derived type is not considered to be a subtype of the parent type, but defines a new type (and its first subtype), whereas, in the object-oriented terminology, it is considered as a subtype, since an object of the derived type may be used[1] anywhere an object of the parent type may be used because the derived type provides at least all the operations of the parent type.

These two definitions of *subtype* can therefore cause confusion in the context of derived types. This confusion arises from the two common definitions of "*providing the functionality of T"* in the object-oriented definition of *subtype*. Some authors ([6], [14]) limit the conformance of behavior to the assurance that the substitution cannot cause a typing error to occur at run-time. Other authors ([18], [20]) expect that the subtype relationship also ensures that the type and the subtype have the appropriate semantics, i.e. that a program that works correctly with objects of the type will have the same effect with objects of the subtype.

These levels of conformance can be summarized by the four compatibility levels of a descendant type (either a subtype

1. possibly with an explicit (but allowed) type conversion

or a derived type) towards its parent, identified by Wegner and Zdonik in [27]:

- Name compatibility
  All operations of a descendant are also defined for its parent, but they may be overridden in the descendant. Apart from the names, there is no other commitment between those operations (i.e. the profiles of the parameters may be different).

- Signature compatibility
  Signature compatibility implies name compatibility. Moreover all operations of a descendant must also be defined for its parent and each descendant operation must have a profile which is type-compatible with the type specified by the parent.

- Behavioral compatibility
  Behavioral compatibility implies signature compatibility. Moreover all operations of a descendant must also be defined for its parents and the operations must have compatible behaviors. Syntax is specified by signatures and semantics for example by interpretations of equational axioms.

- Inheritance with cancellation
  This relationship, inheritance with cancellation, allows the descendant to delete as well as modify operations of the parent. It may be useful to define exceptions. For instance, an ostrich is a bird that cannot fly. This scheme cannot be found in any Ada construct.

Since, in Ada 9X, the conformance of a tagged derived type to its parent is only defined by the specification and not by the implementation, a tagged derived type is signature-compatible with its parent, and may possibly be behavior-compatible, although it is not enforced by the language.

The conformance of an *untagged* derived type to its parent is only name-compatible: when overriding a subprogram it is possible to change the parameter subtypes or the parameter modes.

Since the overloading rules prevent an operation with a parameter of a given type to be overloaded by an operation with the same profile but for the substitution of the type by one of its subtypes, a type and its Ada subtype must share the same behavior for any operation of the type. A type and its Ada subtypes are therefore behavior-compatible. However, the achieved level of conformance is not *true behavioral compatibility*, but *partial compatibility* — corresponding operations on corresponding parameters give corresponding results whenever the result is defined for the subtype. For example, 6+7 is undefined for the subtype Index declared as

```
subtype Index is Integer range 1..10;
```

but is certainly defined for the type Integer [27].

From these discussions, we can conclude that:

- tagged derived types are subtypes in the loose object-oriented definition, that only takes typing errors into account. In the definition that also expects compatible behavior, derived types cannot be considered as object-oriented subtypes, because the behavior of the derived operations can vary;

- untagged derived types are not subtypes in any of the object-oriented definitions because of the possibility to modify the subtype and possibly cause typing errors;

- Ada subtypes are object-oriented subtypes in both definitions. To solve the problem of specifying partial compatibility, B. Liskov suggests in [20] a way to specify a compatible behavior when the subtype is more constrained than its parent in the values it can take.

4. Programming with the 'Class attribute

Although type extension is a powerful mechanism, it only allows the construction of new abstractions by refinement of an existing abstraction. Type extension is not sufficient to provide the full power of expression of object-oriented programming because this mechanism does not offer a way to handle in a uniform way all the objects in a (derivation) class.

There is therefore a need for a language construct to be able to denote objects of different types: the so-called polymorphic types introduced in section 2. In Ada 9X, polymorphic types are called *class-wide types*. A class-wide type is implicitly associated with each tagged type; it denotes the derivation class rooted at the tagged type. A class-wide type is denoted by the attribute 'Class applied to the name of a tagged type. Class-wide types can be used in most contexts where a type can be used, i.e. it can appear in object declaration, as a subprogram formal type parameter,... Explicitly declared types are called *specific types* to stress the difference with class-wide types. Class-wide types are only defined for tagged types.

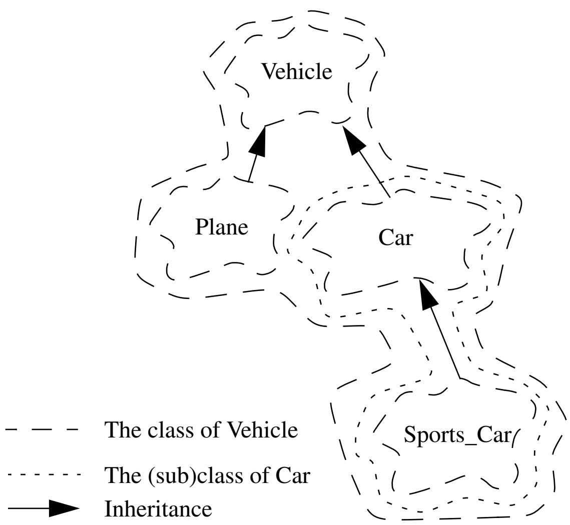

Fig. 2. The classes Vehicle and Car

In figure 1 and figure 2, Vehicle, Car, Sports_Car and Plane are four specific types defining four derivation classes, respectively denoted by Vehicle'Class, Car'Class, Sports_Car'Class and Plane'Class. The specific types Vehicle, Car, Sports_Car and Plane belong to the derivation class Vehicle'Class. Car is the root of the derivation class Car'Class. Car'Class is a subclass of Vehicle'Class, i.e. a subset of Vehicle'Class closed under derivation. The properties of Car'Class are a superset of the properties of Vehicle'Class, containing all the properties of Vehicle plus the properties specific to Car.

All descendants of Car (e.g. Sports_Car) belong to the class Car, but the ancestors and siblings of Car, e.g. Vehicle and Plane, do not.

Each specific type in a derivation class is identified by a *tag*, held by each object belonging to the type, thus the name tagged type. This information allows to find at run-time the specific implementations of an object's operations, i.e. to identify its specific type.

Usually, declaring an object of a class-wide type (a variable, a constant, a formal parameter,...) means that the declared object can belong to any type in the specified class, and in this case, class-wide types assume the role of polymorphic types. These cases will be studied in section 4.1. However, in some cases, class-wide entities are declared even though an object of a specific type is expected. Theses idiomatic uses of class-wide types will be shown in section 4.2.

4.1 Common class-wide types usages

4.1.1 Class-wide entities

Instead of being tied to a specific type, variables and formal parameters can be of a class-wide type, or can access (point to) objects of a class-wide type; all these cases correspond to *class-wide entities*.

Class-wide variables are variables the specific type of which can be set at run-time amongst the types of a class. However, once set (by elaboration), the specific type will freeze the variable's specific type for the whole life-time of the variable. The rationale behind this rule is that

- like the string type, class-wide types are unconstrained, i.e. a class-wide type declaration does not carry enough information to create an object;
- strong typing is preserved, i.e. no object can "freely float" into the derivation class. Each object belongs to a specific type and there is a strong type distinction thanks to the run-time tag;
- finally, efficiency considerations impose that an object must not change its size once allocated.

When declaring a variable of a class-wide type, this specific type must be provided by a mandatory initial value, that can be either an object of a specific type in the class, or a dynamically computed expression.

```
C: Car;
V1: Vehicle'Class := C;
   -- initialization by an object of a
   -- specific type

function Some_Vehicle return Vehicle'Class;
V2: Vehicle'Class := Some_Vehicle;
   -- initialization by a dynamically
   -- computed expression
```

Fig. 3. Class-wide entities declaration and initialization

The declaration of V1 is unlikely to be written in that form: since the initial value is known, it makes more sense to declare an object of a specific type than an object of a class-wide type (but there are exceptions, as we will see in section 4.1.4).

Although objects of class-wide types are not mutable, it is possible to make a variable cover at run-time a whole class by using access-to-class-wide objects (by-reference semantics).

```
type Vehicle_Ref is access all Vehicle'Class;
V_Ref: Vehicle_Ref;
```

V_Ref can *designate* any object of the class Vehicle, i.e. a Vehicle, Car, Plane, or any of their descendants:

```
V_Ref := new Car'(...);
V_Ref := new Plane'(...);
```

This feature is also consistent with strong typing, since each denoted object is of one specific type.

The strength of Ada against other object-oriented programming languages is to make such a strong difference between polymorphic and specific types. Some object-oriented languages, like Smalltalk, only have by-reference semantics and only provide constructs equivalent to access-to-class-wide types, preventing the user to limit the type of objects to specific types. Other object-oriented programming languages, like C++ or Eiffel, have by-value and by-reference semantics. However, in these languages, variables are monomorphic[1], and pointers (references) are polymorphic. This approach results in a confusion of polymorphism with access types, while preventing polymorphic variables. The type system of Ada is more orthogonal, and avoids such disorder, while introducing by-value and by-reference semantics. It provides polymorphic and monomorphic variables and references. The polymorphism of variables is limited to their elaboration, where they are constrained to a specific type. That does hardly restrict the power of expression of Ada, since most uses of polymorphic variables are as subprograms parameters, which are elaborated at every call. Moreover, Ada does not pollute the concept of access type. The entities that can designate types in a class are clearly separated from those that must designate objects of specific types.

1. Eiffel's variables are the objects of expanded classes.

4.1.2 Class-wide subprograms

A formal parameter of a subprogram or a function result type can also be of a class-wide type. Every actual parameter belonging to the class will match with the formal parameter. Such a subprogram is not bound to one specific type but to all types in a given class, and that is why we call it a *class-wide subprogram*. Here is an example:

```
procedure Service (V: in out Vehicle'Class);
```

Any actual parameter belonging to the class-wide type Vehicle'Class, to the specific type Vehicle, or to any of its descendants (here Car, Sports_Car and Plane) will match the formal parameter.

There are therefore two methods for associating an operation to a class: the first one is to program a specific primitive operation for each specific type in the class (by defining a primitive operation and possibly overriding it for each type), the second is to create a class-wide operation (i.e. a unique subprogram with a class-wide type). The advantage of using class-wide subprograms is that they have no special restriction on their place of declaration, unlike the primitive operations that must be declared in the proximity of the type (i.e. before a freezing point —a place in the program where the representation of a tagged type must be fully determined, including the declaration of its primitive operations).

Class-wide operations can also be used to write subprograms that are general to a class, and cannot be isolated for each specific type.

```
function Same_Speed (L, R: Vehicle'Class)
   return Boolean is
begin
   return Speed_of (L) = Speed_of (R);
   -- two dispatching calls (see below)
end Same_Speed;
```

Fig. 4. Class-wide subprogram

The above procedure cannot be programmed for each specific type of the class Vehicle'Class. This algorithm is bound to the class.

4.1.3 Dispatching

Although there is no structural difference between an object of a tagged type T and an object of the associated type T'Class holding a value of type T, there is an important difference in the behavior of these two objects: objects of a class-wide types are dispatching (dynamic binding), while objects of specific types are statically bound.

In the presence of a class-wide actual parameter, it is generally impossible to select at compile-time the implementation of an overridden operation. But at run-time, it is possible to *dispatch* to the implementation according to the tag of the actual parameter; *binding* of the appropriate code to an operation call is therefore *dynamic* and can vary for each execution of one same call statement. In Ada 9X, primitive operations of tagged types are potentially dispatching and are therefore called *dispatching operations*. If it is impossible to bind a call to such an operation at compile-time, it is a dispatching call; a dispatching call has at least one class-wide actual parameter or result type. Occurrence of dispatching is therefore not a property of the operation, but is linked to the presence of at least one class-wide entity, called a *controlling operand*. Dispatching typically occurs in the body of a subprogram with a class-wide parameter:

```
procedure Service(V:in out Vehicle'Class) is
begin
   Repair (V);
   -- Dispatching call: the implementation is
   -- chosen according to the tag of the
   -- actual parameter designated by V.
end Service;
```

Fig. 5. Dispatching

Another case of dispatching is the use of a class-wide variable, and this is an opportunity to stress how different it is from an "ordinary" call:

```
declare
   H1: Vehicle;
   H2: Vehicle'Class := ...;
begin
   Repair(H1);-- Static binding

   Repair (H2);-- Dynamic binding
end;
```

Fig. 6. Static vs. dynamic binding

The first call is a case of static binding: the compiler knows the specific type of H1. Therefore it can select an implementation. The second call is a case of dynamic binding: H2 may be a specific Vehicle, Car, Sports_Car, or Plane. Thus the compiler cannot select an implementation. The appropriate implementation is selected at run-time. Note that the second call can possibly be optimized into a static call if the tag can be determined at compilation-time (for example from the initial value of H2).

A call with a class-wide access parameter is also dispatching:

```
... Repair (V_Ref.all) ...
   -- V_Ref can designate any object in the
   -- class Vehicle.
```

Programmers should note that Ada 9X provides a fine tuning of dispatching, i.e. it is accomplished on a call-by-call basis instead of on a by-operation basis, reducing the overhead sometimes involved by run-time polymorphism.

4.1.4 Conversions to a class-wide type

In some cases, the programmer will need to convert an object to a class-wide type, or to rename it, in order to change the view of the object. This conversion is a *view conversion* and is usually not meant to specify that the converted object suddenly may belong to any type in the class, since each object carries a tag and has therefore a specific type. This conversion can have two meanings: to enforce dispatching or to give access to properties.

The first case, *to enforce dispatching*, is needed for *redispatching* purposes. A derived type can always keep a primitive operation of its parent type, i.e. not provide its own implementation. However, if the implementation of these non-overridden operations depends on the behavior of overridden operations, it is sometimes suitable to base the implementation of the non-overridden operation on the behavior of the overridden operation. For example, imagine that the implementation of Move, which is inherited as is by Car (i.e. not overridden) needs to call Speed_of, which is overridden for Car. Care must be taken, not to call the implementation of Vehicle but to call the implementation of the specific type in current use, thus to convert the object to a class-type to enforce dispatching.

```
procedure Move (V: in out Vehicle;
                L: in Location) is
begin
   ... Speed_of (Vehicle'Class (V))...;
   -- redispatching
end Move;
```

Fig. 7. Enforcing dispatching with a view type conversion

In the body of the implementation for Vehicle, the conversion of the parameter V to the type Vehicle'Class provides redispatching to the correct implementation of Speed_of, depending on the actual tag of V.

The following alternate implementation based on the renames statement produces the same effect and can prove to be more readable.

```
procedure Move (V: in out Vehicle;
                L: in Location) is
   V_C:Vehicle'Class renames Vehicle'Class(V);
begin
   ... Speed_of (V_C) ...; -- redispatching
end Move;
```

Fig. 8. Enforcing dispatching with a renaming declaration

Redispatching is a dangerous operation since it is not apparent in the profile of the operation, and it modifies the behavior of a non-overridden operation at derivation time. Redispatching can therefore hinder the testing of classes and should absolutely give way to a comment following the declaration of the operation[4].

The second case, *to give access to properties*, occurs when an object of a class-wide type is known to belong to a subclass. In that case, it is possible to convert the object to any subclass to which it belongs (view conversion), making visible the properties of the subclass besides the properties of the class. The membership operator **in** is used to check that the converted object belongs to the expected class, and that the conversion will therefore not raise an exception (Constraint_Error).

In figure 9, if we know for sure that V is a car or one of its derivatives, we will change the tires, an operation which is only provided for the class Car'Class, thus the need for a conversion.

```
procedure Service(V:in out Vehicle'Class) is
begin
   ...
   if V in Car'Class then
      New_Tires (Car'Class (V));
      -- conversion to provide New_Tires
   end if;
   ...
end Service;
```

Fig. 9. Giving access to properties

This programming practice based on membership operator is sometimes considered to be a failure to apply polymorphic behavior effectively, since it leads to sequences of tests that polymorphism is supposed to avoid. There are however some notable exceptions in which membership operators are welcome, the most appropriate being to provide specific operations to an object extracted from a heterogeneous data structure.

4.2 Idiomatic class-wide types usages

In addition to the cases where class-wide types are used as polymorphic types, there are idiomatic cases where objects are declared of a class-wide type although it is perfectly clear that these objects will belong to exactly one known specific type.

In some other cases, class-wide types are used instead of specific types because the rules of the languages forbid the specific type to appear.

4.2.1 To provide an anchor for local declarations

When deriving a type, all the occurrences of the parent type in the profile of the inherited operations are substituted with the derived type, but the types of the local declarations are not modified by the derivation. However, a local object sometimes needs to be of the same specific type as the controlling operand of the operation, for example because this local object will have to carry a copy of the controlling operand.

Such local objects can be declared with an "anchored declaration"—to follow the Eiffel terminology. An anchored declaration carries a provision for automatic redefinition in a descendant of the class. In Ada 9X, an anchored declaration is achieved by the declaration of an object of a class-wide type constrained to the specific type of the controlling operand.

```
procedure Swap (L, R: in out Vehicle) is
   T: Vehicle'Class := L;
begin
   T := Vehicle'Class(L);
   Vehicle'Class(L) := Vehicle'Class(R);
   Vehicle'Class(R) := T;
end Swap;
```

Fig. 10. Anchor declaration

In figure 10, the variable T is declared to be of the class-wide type Vehicle'Class, and simultaneously constrained to be of the specific type of L.

When Car is derived from Vehicle, its inherited subprogram Swap has the following specification:

```
procedure Swap (L, R: in out Car)
```

If the subprogram is not overridden, its implementation will be a call (with view conversion) to the implementation of Swap for Vehicle.

Since the specific type of T is anchored to the specific type of L, T will be a Car. Therefore, the implementation of Swap in figure 10 need not be overridden to work for all the derivatives of Vehicle.

This implementation is interesting in two respects. First, it shows an example of anchored type, and an implementation of the procedure Swap that works for all the specific types in the class. It also shows an example of dispatching for the assignment statement. Although a statement, the assignment is partly defined like an operation which is sensitive to class-wide operands. If the target operand of an assignment is of a class-wide type, the implementation of the assignment will be selected at run-time, according to the specific type of the target. In figure 10, the assignments are dispatching, so that they work for all specific types that may inherit the operation Swap.

In order to make inherited subprograms work without redefinition, it is a good programming practice to always make the assignment statement dispatching.

Note that the declaration of T and the first statement of Swap can be merged into a single declaration:

```
   T: Vehicle'Class := Vehicle'Class(L);
```

4.2.2 To declare non-inherited operations

It may sometimes be useful to declare operations of a tagged type which are not primitive operations, i.e. that will not be inherited [11].

This is especially useful for constructor operations because these operations can usually not be overridden for a descendant in their inherited shape. (In most cases, new parameters are needed to take the extension into account.) This can be a burden for primitive operations that return a result of a tagged type, because these operations must be overridden in the derived type, or will be considered as abstract, making the derived type itself abstract.

A solution to that problem is to declare constructor operations, the return type of which is of a class-wide type.

In figure 11, Create has a result type of the class-wide type Vehicle'Class, although its implementation will only return objects of the specific type Vehicle. Since it is not a primitive operation of Vehicle, it will not be inherited, thus suppressing the need for any descendant to override it with an implementation containing the same set of parameters.

```
type Vehicle is tagged private;
function Create (
      Nb_Passengers: Natural;
      Speed: Float;
      Position: Location)
   return Vehicle'Class;
```

Fig. 11. A constructor operation

4.2.3 To avoid multimethods

A multimethod is a rare form of operation definition found in the programming languages CLOS [17] and Cecil [12], in which an operation can be primitive of more than one tagged type.

Multimethods are not legal in Ada 9X, since it is forbidden for an operation to be a primitive of more than one tagged type. Thus, in figure 12, the declaration of the operation Add is illegal.

```
package Graph_Handler is
   type Vertex is tagged private;
   ...
   type Graph is tagged limited private;
   procedure Add
      (The_Vertex: in Vertex;
       In_The_Graph: in out Graph); -- illegal
private
   ...
end Graph_Handler;
```

Fig. 12. Multimethods are illegal in Ada

To solve that problem, one of the incriminated types in the profile of such an operation must be replaced by the equivalent class-wide type.

```
package Graph_Handler is
   type Vertex is tagged private;
   ...
   type Graph is tagged limited private;
   procedure Add
      (The_Vertex: in Vertex'Class;
       The_Graph: in out Graph); -- legal
private
   ...
end Graph_Handler;
```

Fig. 13. Multimethods (solved) by constraining a parameter to a class-wide type

The operations must be programmed in terms of chained dispatch (i.e. by programming Add with dispatching calls to primitive operations of Vertex) to ensure that extensions of both types are taken into account.

5. Classes and polymorphism

Although class-wide types provide a support for run-time polymorphism in Ada 9X, they are by no means the only ways to achieve polymorphism. Cardelli and Wegner [10] have identified several kinds of polymorphism, most of which are present in Ada (figure 14).

Universal polymorphism is a kind of polymorphism in which polymorphic subprograms work on a possibly infinite number of types, by taking advantage of a common structure. On the other hand, ad hoc polymorphism only works on a limited number of types, that may be completely unrelated. Universal polymorphism could therefore be considered as the true form of polymorphism while ad hoc polymorphism is only a façade of polymorphism.

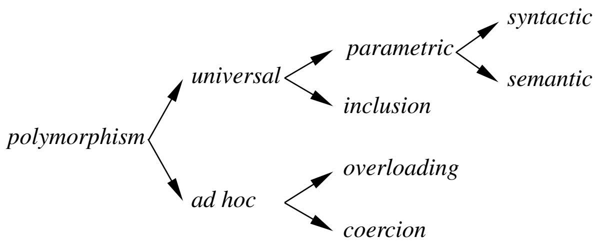

Fig. 14. Varieties of polymorphism

5.1 Ad hoc polymorphism

Overloading is an ad hoc form of inheritance well known to Ada practitioners, in which a name can designate different subprograms which may share a common semantics in the eyes of the programmer ("Open" a file, a document, "Add" two integer number, two real numbers,...), but have completely different behavior, and their signatures can only be name-compatible. Overloaded operations must be considered as a set of monomorphic functions rather than a polymorphic operation.

Coercion is also an ad hoc form of polymorphism in which operands are implicitly converted to the type defined for the needed operation. In Ada, where implicit conversions are discouraged, coercion polymorphism is rather uncommon, but can still be found with literals.

5.2 Universal polymorphism

Universal inclusion polymorphism is the form of universal polymorphism that corresponds to the set of different implementations of an operation in a user-defined class. Such polymorphism is called inclusion polymorphism because an object belonging to a subclass simultaneously belongs to the classes in which the subclass is included. Thus, every object of a subclass can be used in a place where an object of the class is expected. Inclusion polymorphism is the most flexible and potentially the most dangerous form of polymorphism, because the selection of the implementation of an operation is undecidable at compile-time and must occur at run-time. It becomes impossible to proceed to a static analysis of a program. This can sensibly complicate the testing of object-oriented software [4].

"In *universal parametric polymorphism*, the same object or function can be used uniformly in different type contexts without change, coercions, or any kind of run-time tests or special encoding of representation" [10]. This kind of polymorphism comes in two fashions in Ada 9X: the syntactic and the semantic fashion.

The *syntactic fashion* corresponds to the generic units. This polymorphism is only syntactic, because, to be used, the generic units must be instantiated with actual parameters that must be clearly designated at compile-time. Thus, there is no run-time polymorphism and every call to a subprogram in a generic unit can be statically bound. Generic instantiations of a generic units can be considered as a set of monomorphic units. It differs however from overloading, because all units in the set will share a common behavior, parameterized by generic actual parameters.

Class-wide subprograms are of *universal parametric semantic polymorphism*, since the same implementation can be invoked for any object in the class.This is the purest form of polymorphism. Unlike overloading, inclusion polymorphism and universal parametric syntactic polymorphism, it does not provide polymorphism through a set of different implementations selected at compile-time or at run-time. A class-wide subprogram consists of a unique implementation that can be uniformly invoked for all objects of a type in the class.

6. Classes in generics

Classes also have a key role to play in generics, although this role is centered on the membership of types to a class, rather than to the membership of objects: the generic formal types define the class of types to which a type must belong to be used in instantiating a generic unit.

Ada 9X provides a syntax for generic formal type parameters of most of the language-defined classes (record and task types being noticeable exceptions), ensuring that the actual parameter will have a set of properties which the generic unit's developer can take into account. For example, the following generic formal part:

```
generic
   type I is range <>;
```

ensures that the generic actual type will be a member of the class of signed integer types, and that it will have all the properties of signed integer types: an assignment statement, a set of mathematical operators, the relational operators, the attributes defined for scalar and discrete types,...

Ada 9X introduces two new kinds of formal generic types that are of interest for this paper: the generic formal derived types and the generic formal tagged types. These generic formal types allow to specify that a generic actual type parameter must belong to a user-defined class.

6.1 Generic formal tagged private types

The generic formal tagged (possibly limited) private types can be used to pass to a generic unit any tagged type. For example, the following generic formal part:

```
generic
   type T is tagged private;
```

ensures that the generic actual type will be a non-limited tagged type.

A generic formal tagged private type can therefore act a "universal class" of tagged types. Generic tagged private types are very similar to the generic formal (untagged) types. However, the properties of tagged types can be exploited in the generic unit:

- the generic actual type can be extended (derived) in the generic unit;
- the class-wide type defined for the generic actual type also stands at disposal.

Generic formal tagged types are useful for creating mixins ([3], [8], [25]), which offer most of the functions of multiple inheritance without introducing its problems. Conceptually, a mixin is an abstract subclass, an added-value that can be added to any tagged type. For instance, "being insured" is a property that can be applied to several classes of items. The following figure shows how to implement this pattern in Ada 9X terms.

```
with Money; use Money;
generic
   type Item is tagged private;
package Insured_Mixin is
   type Insured_Item is new Item with private;

   function Reimbursed_Value
      (Insured: Insured_Item)
      return Money_Type;
   -- ...(other operations)

private
   type Insured_Item is new Item
      with record
         Police_Number: Positive;
         Insured_Value: Money_Type;
         Premium: Money_Type;
         -- ...
      end record;
end Insured_Mixin;
```

Fig. 15. A mixin for the subclass of insured items

The new type Insured_Item is derived from a generic formal tagged type Item, which matches any tagged type. Therefore, this package is very general and one can easily create a new abstraction, Insured_Car, in which the property "being insured" is mixed to the properties of Cars, as illustrated in figure 16. In addition to all the properties of a normal car, an Insured_Car has all the properties of insured items (Reimbursed_Value,...).

```
package Insured_Cars is
   new Insured_Mixin (Car);

My_Car: Insured_Cars.Insured_Item;
```

Fig. 16. Mixin of Insured and Car

However, since any tagged type can be passed as actual parameter, mixins can be used to add properties to any tagged type regardless of its properties, at the risk of involuntarily overriding an operation [13].

```
generic
   type T is private;
package Mixin_G is

   type Mixin is new T with private;
   procedure P (M: in Mixin);

private
   ...
end Mixin_G;

...

type T is tagged ...;
procedure P (X: in T);

package Mixin_T is new Mixin_G (T);
```

Fig. 17. Involuntary overriding of properties

In figure 17, when creating the type Mixin_T.Mixin, the operation P of the type T is overridden by the operation P declared in the package Mixin_G. This overriding may or may not have been desired. Most of the time, primitive operations of a mixin type are intended to be new, since a mixin has no knowledge of its parent type. Most of the overriding that may occur in a mixin are accidental and end in a conflict between the parent type and the mixin. The visible part of the generic must therefore be carefully examined by clients of a mixin package to prevent any conflict resulting from blind reuse.

6.2 Generic formal derived types

The generic formal derived types can be used to instantiate a generic unit with a type (tagged or untagged) in a specific user-defined class. Generic formal derived types come in two fashions, depending on whether the actual derived type shall be tagged or not. The generic formal tagged derived type fashion is very similar to the generic formal tagged types, except that it constrains the actual type to belong to a given class instead of being just tagged, and ensures that all the primitive operations of the formal type (i.e. the operations of the class rooted at the formal derived type) will be available. The programmer can benefit from these operations and use them to program the generic unit. The implementations of these operations will be the implementations of the actual type.

For example, the following generic formal part:

```
generic
   type Item is new Vehicle with private;
```

can be matched by Vehicle or any type derived directly or indirectly from Vehicle (i.e. any type in the class Vehicle). Therefore, all the properties of Vehicle will also be available for Item. Generic formal derived types can prove to be useful to program constrained mixins. If the previous generic formal part was substituted for the generic formal part of the mixin in figure 15, vehicles only could be insured.

Generic formal untagged derived types should have the same purpose as generic formal tagged derived types, except that those parameters do not provide the implementation of the actual type, but always the implementation of the root type.

```
type Angle is new Integer;
function "+" (Left, Right: Angle)
   return Angle;

generic
   type NI is new Integer;
procedure Double_G(X: in out NI);

procedure Double_G(X: in out NI) is
begin
   X := X+X;  -- always use the implementation
              -- of "+" defined for Integer.
end Double_G;
```

Fig. 18. Reemergence of the predefined operator "+" in a generic

In figure 18, the procedure Double_G will use the implementation of the plus operation defined for the type Integer, even if this subprogram was overridden for the actual parameter for NI.

This subtle distinction in the semantics of both generic formal derived types takes its roots in the subtype conformance issues introduced in the section 3.3. Since untagged derived types are name-compatible and not signature-compatible, the generic contract could be broken by a derived type the subprograms of which are in contradiction with the profile of the operations of the parent type.

Nevertheless, this difference in the behavior of generic formal derived types is disturbing and error-prone. The programmers should be warned of the potential effects of using these parameters.

7. Summary and conclusion

The way run-time polymorphism is introduced in Ada 9X —by adding class-wide types— is original and well-integrated in the language. The term *class* is kept for classification purposes, its meaning is precise, and does not interfere with existing concepts such as modularity. It is also perfectly suited with its different uses. Moreover, it helps to remove the ambiguity specific- vs. polymorphic type. The implicit differentiation between static binding/dynamic dispatching on a call-by-call basis allows a fine tuning of dispatching. It reduces the overhead induced by dispatching to the sole cases where dispatching is needed, and should lead to quality code.

The idiomatic uses described in section 4.2 are perfectly in accordance with the philosophy of building blocks that prevailed during the design of Ada 9X, i.e. to provide a set of small multi-purpose features rather than a set of ad-hoc constructions.

We have also seen that *polymorphism* is not only brought by class-wide types. Ada offers many different building blocks to create general components, especially generics, and the use of these building blocks still corresponds to good software engineering practices. Programmers should examine what kind of polymorphism better suits their needs instead of blindly selecting run-time polymorphism [24].

The integration of classes with generics provide a powerful mechanism, mixins, although the difference amongst tagged and untagged generic formal derived types is non-intuitive and may lead to errors. It is important for Ada programmers to be aware of these idioms.

A good understanding of what's behind Ada 9X classes will without any doubt be a key factor in a successful programming with Ada 9X.

8. Acknowledgments

The author would like to thank Gabriel Eckert and Magnus Kempe for their useful comments and suggestions for improvements in this paper. This paper benefits from examples developed in collaboration with Alfred Strohmeier and Magnus Kempe for a tutorial on Ada 9X.

The author would also thank Stef van Vlierberghe and Angel Alvarez for the numerous 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.

9. References

[1] —. *Reference Manual of The Ada Programming Language*. Washington, D.C.: Department of Defense, Ada Joint Program Office, Feb. 1983.

[2] Ada 9X Mapping/Revision Team. *Ada 9X Reference Manual (Draft Version 5.0)*. Intermetrics, Inc., 733 Concord Avenue, Cambridge, Massachusetts 02138, MA, USA, June 1994. IR-MA-1363-3.

[3] S. Barbey, M. Kempe, and A. Strohmeier. Advanced object-oriented programming with Ada 9X. In *TRI-Ada 1994 Tutorials*, Baltimore, Maryland, USA, Nov. 6-11 1994.

[4] S. Barbey and A. Strohmeier. The problematics of testing object-oriented software. In M. Ross, C. Brebbia, G. Staples, and J. Stapleton, editors, *SQM '94 Second Conference on Software Quality Management*, volume 2, pages 411–426, Edinburgh, Scotland, UK, July 26-28 1994.

[5] J. Barnes, B. Brosgol, K. Dritz, O. Pazy, and B. Wichmann. *Ada 9X Rationale (Draft Version 5.0)*. Intermetrics, Inc., Cambridge, MA, USA, June 1994.

[6] G. Blair, J. Gallagher, D. Hutchinson, and D. Sheperd, editors. *Object-Oriented Languages, Systems and Applications*. Pitman, 1991.

[7] G. Booch. *Object-Oriented Analysis and Design with*

Applications. Benjamin-Cummings, second edition, 1994.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *ECOOP/OOPSLA '90 Conference Proceedings*, pages 303–311, Ottawa, Canada, Oct. 1990. ACM SIGPLAN, ACM Press. Published as Volume 25, Number 10 of SIGPLAN Notices.

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–42, Aug. 1992.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, Dec. 1985.

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

[12] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92 Conference*, volume 615, pages 33–56, Utrecht, The Netherlands, July 1992. Springer Verlag.

[13] G. Dismukes and E. Ploedereder. Accidental overriding considered evil. Comment to the MRT 94-3955.a (ftp ajpo.sei.cmu.edu:/public/ada9x/mrtcomments/ 94.0302), Mar. 1992.

[14] G. Eckert. Types, classes and collections in object-oriented analysis. In *ICRE '94 First International Conference on Requirements Engineering*, pages 32–39, Colorado Springs, Colorado, USA, Apr. 18-22 1994.

[15] P. Hudak, S. P. Jones, and P. Wadler (Ed.). Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):section R, May 1992.

[16] J.D. Ichbiah, J.G.P. Barnes, R.J. Firth, and M. Woodger. *Rationale for the Design of the Ada Programming Language*. 1986.

[17] G. Kiczales, J. des Riviers, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[18] G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *ECOOP/OOPSLA '90 Conference Proceedings, Ottawa, Canada*, volume 25 of *SIGPLAN Notices*, pages 212–223. ACM SIGPLAN, ACM Press, Oct. 1990.

[19] B. Liskov. Data abstraction and hierarchy. In L. Power and Z. Weiss, editors, *OOPSLA '87 Addendum to the Proceedings*, pages 17–34, Orlando, Florida, May 1987. ACM Press. Published as Volume 23, Number 5 of SIGPLAN Notices.

[20] B. Liskov and J. M. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP '93*, volume 707 of *Lecture Notes in Computer Sciences*, pages 118–141, Kaiserslautern, Germany, July 1993. Springer Verlag.

[21] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[22] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, 1992.

[23] H. Mössenböck and N. Wirth. The programming language Oberon-2. Technical report, Institut fur Compustersysteme, ETH Zurich, Jan. 1992.

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

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

[26] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[27] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *Proceedings ECOOP '88*, volume 322 of *Lecture Notes in Computer Sciences*, pages 55–77, Oslo, Norway, Aug. 15-17 1988. Springer Verlag.