A Naming Convention for Classes in Ada 9X

J-P. Rosen
ADALOG
27, avenue de Verdun
92170 VANVES
FRANCE
Tel: +33 1 46 45 51 12
Fax: +33 1 46 45 52 49
E-m: rosen@enst.fr

Abstract:
This paper discusses a problem encountered when using generics for faceted OOP in Ada 9X, and proposes a general naming scheme for classes that provides a convenient and readable solution to this problem.

Note: Throughout this paper, we refer to Ada 9"X", since at the time of writing, there is still uncertity on the last bit of the last digit of "X". We hope that when published, the "X" will have become "4".

I. Classes in Ada 9X

A "class" in the sense of OOP is built in Ada 9X as a package declaring a tagged type and primitive subprograms for this type. We will call this type the "main" type. Nothing prevents from declaring other types in the same package, and although generally not recommended, this may be useful to declare "secondary" types that serve for the definition of the main tagged type. A typical class will look like this:

```
package Employee_Class is
    type Employee_Number is range 1..10_000;
    function New_Id return Employee_Number;

    type Employee(Id : Employee_Number) is tagged private;
-- The "main" type

-- Operations on Employee
private
    ...
end Employee_Class;
```

Although it is possible to derive from Employee in the same package, it is generally better to keep the structure where *one package = one class*. In order to provide a different form of employee - say the computer scientists-, one will declare:

```
with Employee_Class; use Employee_Class;
package Computer_Scientist_Class is
    type Computer_Scientist is new Employee with private;

-- Redefinition or addition of operations
private
    ...
end Computer_Scientist_Class;
```

Sometimes, it is necessary to create some new classes by adding properties that are orthogonal to the derivation tree. To avoid the issue of arguing whether Ada has multiple inheritance or not[2], we propose to call this "faceted programming": you add new facets to an existing class. In other languages, multiple inheritance is used for faceted programming, while in Ada it is achieved through the use of generics. Suppose we want to add a facet for the employees with optionnal benefits. This can be applied to any existing class of employee. A generic package to transform any employee into an employee-with-benefits will look like:

```
with Employee_Class; use Employee_Class;
generic
    type Origin is new Employee with private;
package With_Benefits_Facet is
    type Employee_With_Benefits is new Origin with private;

    -- Operations to handle benefits
private
    ...
end With_Benefits_Facet;
```

From this on, it is easy to create Computer-Scientists-With-Benefits as:

```
with Computer_Scientist_Class; use Computer_Scientist_Class;
with With_Benefits_Facet;
package Computer_Scientists_With_Benefits_Class is
    new With_Benefits_Facet(Computer_Scientist);
```

II. The naming problem

In the previous example, we used a natural and straightforward naming convention: all packages serving as classes had their names ending with "_Class", while packages serving as facets had their names ending in "_Facet"; the rest of the name corresponded to the entity being represented. The main type of the package had the same name as the package (the entity's name).

There are several problems with this naming scheme. The first one is general for those who do not use the **use** clause; declarations of objects are quite awkward, like:

```
Myself: Computer_Scientist_Class.Computer_Scientist
                                    (Computer_Scientist_Class.New_Id);
```

[2] An issue that boils down to depending on the definition of multiple inheritance.

Secondly, if a package declares more than one type, there is no obvious way to tell the main type from secondary types. Of course, the main type will normally reflect the package name, but this does not allow for a convenient automatic search.

Finally, there is a much more serious problem. In a generic instantiation, you can choose the name of the instantiation itself, but not the names of entities declared within the instantiated package. Therefore, the main type of *any* instantiation of With_Benefits_Facet will be called "Employee_With_Benefits". The problem can be lessenned by using full name notation, but the basic issue remains:

```
Person_1 :
    Computer_Scientists_With_Benefits_Class.Employee_With_Benefits
                                        (Computer_Scientist_Class.New_Id);
-- NOT very readable

use Computer_Scientists_With_Benefits_Class, Employee_Class;
Person_2 : Employee_With_Benefits(New_Id);
-- MISLEADING: The fact it is a computer scientist does not appear
```

Since the problem comes from an inadequate type name, we first tried to provide a different name by defining a subtype with the right identifier. Of course, we didn't want that subtype declaration to be on the user's side, so we had to embed all this stuff into a package:

```
with Computer_Scientist_Class; use Computer_Scientist_Class;
with With_Benefits_Facet;
package Computer_Scientists_With_Benefits_Class is
    package Auxiliary is
       new With_Benefits_Facet(Computer_Scientist);
    subtype Computer_Scientist_With_Benefits is
       Auxiliary.Employee_With_Benefits;
end Computer_Scientists_With_Benefits_Class;
```

Although it does provide the "right" name for the main type, this solution has severe drawbacks; there is no "good" name for the instantiation (we called it *auxiliary*, but it is not very satisfying). Moreover, although the type Computer_Scientist_With_Benefits is directly visible, the rest of the package, and especially the operations, are not. The user will therefore have to either use a **use** clause for Auxiliary (but many projects will not allow it), or have Auxiliary named in every call, which is very cumbersome, since that package is merely an artifact.

In order to provide access to operations, we changed the subtype into a derived type::

```
with Computer_Scientist_Class; use Computer_Scientist_Class;
with With_Benefits_Facet;
package Computer_Scientists_With_Benefits_Class is
    package Auxiliary is
       new With_Benefits_Facet(Computer_Scientist);
    type Computer_Scientist_With_Benefits is new
       Auxiliary.Employee_With_Benefits with null record;
end Computer_Scientists_With_Benefits_Class;
```

This solution does provide direct access to operations; but the drawbacks are also important. First, the new type is an extension (although a null extension) of the original type; this means that

if you want to use an aggregate as a parameter in some subprogram call, you must provide it as an extension aggregate with a null extension part; not exactly an elegant solution. Moreover, nothing can prevent the user from using the type and operations declared by Auxiliary directly: of course, the project manager can forbid this, but it is impossible to enforce it by language rules[3]. Finally, if Auxiliary declares other types or objects, they must also be manually re-exported.

III. A convenient naming convention

Given the previous discussion, we wanted a way to provide convenient access to faceted programming that would not involve special effort on the designer's side nor special rulings on the user's side. This was achieved by the following naming scheme:

- All class packages are named after the object they represent, without any "_Class" appended.
- All facet packages are named after the facet they represent, with "_Facet" appended.
- In both cases, the main type is *always* named "Instance".
- By the same token, the declaration of the main type is *always* followed by the declaration of a subtype for the corresponding class wide type; this subtype is *always* named "class"[4].

With this convention, our previous packages become:

```
package Employee is
    type Employee_Number is range 1..10_000;
    function New_Id return Employee_Number;

    type Instance(Id : Employee_Number) is tagged private;
    subtype Class is Instance'CLASS;

-- Operations on Employee.Instance
private
    ...
end Employee;

with Employee; use Employee;
package Computer_Scientist is
    type Instance is new Employee.Instance with private;
    subtype Class is Instance'CLASS;

-- Redefinition or addition of operations
private
    ...
end Computer_Scientist;
```

[3] And it is our firm belief that there *must* be a way of enforcing project rules.

[4] Remember: "class" is not a reserved word in Ada 9X, although some might regret it...

```
with Employee; use Employee;
generic
    type Origin is new Employee.Instance with private;
package With_Benefits_Facet is
    type Instance is new Origin with private;
    subtype Class is Instance'Class;

    -- Operations to handle benefits
private
    ...
end With_Benefits_Facet;

with Computer_Scientist; use Computer_Scientist;
with With_Benefits_Facet;
package Computer_Scientists_With_Benefits is
    new With_Benefits_Facet(Computer_Scientist.Instance);
```

Now, declarations of objects or subprograms become very explicit and readable:

```
Myself: Computer_Scientist.Instance(New_Id);

procedure Print_any_Employee (Who : Employee.Class);
```

Since facet generics follow the same general naming scheme, there is no difference in names, whether the package has been explicitely written or obtained from generic instantiation:

```
Him : Computer_Scientist_With_Benefits.Instance(New_Id);
```

Of course, this works well as long as you use full names. What happens in the case of a **use** clause? *It works*! Because *all* main types are named the same, they will *always* hide one another[5], and the compiler will therefore enforce the rule that every use of an instance must explicitely tell what it is an instance of..

IV. Conclusion

The proposed naming scheme is simple, and allows uniform and readable naming of OOP classes and entities. It simplifies greatly the use of facet generics, and has the additionnal benefit of clearly identifying instances, classes, and the main type in a class package. The correct usage of the names is automatically checked by the compiler.

We hope that it will be useful in promoting the use of OOP with Ada 9X..

Final note:

We are very proud to have used an example of OOP that does *not* refer to graphic objects nor insects...

[5] Unless you use only one class. But using classification with only one class is of so little value that we think it's not worth bothering about this case...