Rationale for Ada 2005: 1 Object oriented model

John Barnes

John Barnes Informatics, 11 Albert Road, Caversham, Reading RG4 7AN, UK; Tel: +44 118 947 4125; email: jgpb@jbinfo.demon.co.uk

Abstract

This paper describes various important improvements to the object oriented model for Ada 2005.

First an alternative more traditional prefixed notation for calling operations has been introduced. A major improvement is that Java-like interfaces are introduced thereby permitting simple multiple inheritance; null procedures have also been introduced as a category of operation. Greater general flexibility is provided by allowing type extension at a more nested level than that of the parent.

There are also explicit features for overcoming nasty bugs which arise from confusion between overloading and overriding.

This is one of a number of papers concerning Ada 2005 which are being published in the Ada User Journal. An earlier version of this paper appeared in the Ada User Journal, Vol. 26, Number 1, March 2005. Other papers in this series will be found in later issues of the Journal or elsewhere on this website.

Keywords: rationale, Ada 2005.

1 Overview of changes

The WG9 guidance document [1] identifies very large complex systems as a major application area for Ada. It says

"The main purpose of the Amendment is to address identified problems in Ada that are interfering with Ada's usage or adoption, especially in its major application areas (such as high-reliability, long-lived real-time and/or embedded applications and very large complex systems). The resulting changes may range from relatively minor, to more substantial."

Object oriented techniques are of course important in very large systems in providing flexibility and extensibility. The document later asks the ARG to pay particular attention to

> Improvements that will remedy shortcomings in Ada. It cites in particular improvements in OO features, specifically, adding a Java-like interface feature and improved interfacing to other OO languages.

Ada 2005 does indeed make many improvements in the object oriented area. The following Ada Issues cover the relevant changes and are described in detail in this paper:

218 Accidental overloading when overriding

251 Abstract interfaces to provide multiple inheritance

252 Object.Operator notation

260 Abstract formal subprograms & dispatching constructors

284 New reserved words

310 Ignore abstract nondispatching ops during overloading

344 Allow nested type extensions

348 Null procedures

391 Functions with controlling results on null extension

396 The "no hidden interfaces" rule

400 Wide and wide-wide images

401 Terminology for interfaces

405 Progenitors and Ada.Tags

407 Terminology and semantics for prefix names

411 Equality for types derived from interfaces

417 Lower bound of functions in Ada.Tags etc

419 Limitedness of derived types

These changes can be grouped as follows.

First we discuss the fact that Ada 2005 has three new reserved words, **interface**, **overriding**, and **synchronized**. It so happens that these are all used in different aspects of the OO model and so we discuss them in this paper (284).

Then there is the introduction of the Obj.Op or prefixed notation used by many other languages (252, 407). This should make Ada easier to use, improve its image, and improve interfacing to other languages.

A huge improvement is the addition of Java-like interfaces which allow proper multiple inheritance (251, 396, 401, 411, 419). A related change is the introduction of null procedures as a category of operation somewhat like abstract operations (348).

Type extension is now permitted at a more nested level than that of the parent type (344). An important consequence is that controlled types no longer need to be declared at library level.

An interesting development is the introduction of generic functions for the dynamic creation of objects of any type of a class (260, 400, 405, 417). These are sometimes called object factory functions or just object factories.

Additional syntax permits the user to say whether an operation is expected to be overriding or not (218). This detects certain unfortunate errors during compilation which otherwise can be difficult to find at execution time. A small change to the overriding rules is that a function with a controlling result does not "go abstract" if an extension is in fact null (391). Finally, we discuss a minor but useful change to the overloading rules; in a sense this is not about OO at all since it concerns the rules for nondispatching operations but it is convenient to discuss it here (310).

There are in fact many other OO related improvements in Ada 2005 concerning matters such as access types, visibility, and generics. They will be described in later papers.

2 Reserved words

Ada 2005 has three further reserved words namely **interface**, **overriding**, and **synchronized**. Readers may recall that Ada 95 had six more reserved words than Ada 83 and the fact that this meant that some programs were incompatible and thus had to be rewritten loomed large in the minds of many commentators.

When new syntax for the introduction of interfaces was being discussed it was strongly felt that incompatibilities should be avoided and that any new syntax words should be unreserved. It was also noted that Interface was a popular identifier and that making it a reserved word would cause many programs to have to be rewritten.

However, it was soon realised that treating Interface as unreserved would have permitted sequences such as

```
type T is interface;
subtype Interface is T;
```

in which Interface is a subtype of the interface T. This would have been total madness. Some reviewers also had memories of PL/I in which words such as IF were not reserved so that one could write IF IF ... where the first IF is a syntax word and the second is a user identifier.

Accordingly it was decided that the new words would have to be reserved. No sensible alternative to **interface** could be thought of although it would be irritating for users who had packages called Interface – actually a brief survey revealed that most such packages had longer names such as Radar_Interface so that the problem was more apparent than real. The other new reserved words **overriding** and **synchronized** clearly present less of a problem since they are less likely to have been used as identifiers.

3 The prefixed notation

As mentioned in the Introduction, the Ada 95 object oriented model has been criticized for not being really OO since the notation for applying a subprogram (method) to an object emphasizes the subprogram and not the object. Thus given

```
package P is
  type T is tagged ... ;

  procedure Op(X: T; ... );
  ...
end P;
```

then we usually have to write

```
P.Op(Y, ... );                    -- subprogram first
```

in order to apply the operation to an object Y of type T whereas an OO person would expect to write

```
Y.Op( ... );                      -- object first
```

Some hard line OO languages such as Smalltalk take the view that everything is an object and that all activities are operations upon some object. Thus adding 2 and 3 can be seen as sending a message to 2 instructing 3 to be added to it. This is clearly an extreme view.

Older languages take the view that subprograms are dominant and that they act upon parameters which might be raw numbers such as 2 or denote objects such as a circle. Ada 95 primarily takes this view which reflects its Pascal foundation over 20 years ago. Thus if Area is a function which returns the area of a circle then we write

```
A := Area(A_Circle);
```

However, when we come to tasks and protected objects Ada takes the OO view in which the identity of the object comes first. Thus given a task Actor with an entry Start we call the entry by writing

```
Actor.Start( ... );
```

So Ada 95 already uses the object notation although it only applies to concurrent objects such as tasks. Other objects and, in particular, objects of tagged types have to use the subprogram notation.

A major irritation of the subprogram notation is that it is usually necessary to name the package containing the declaration of the subprogram thus

```
P.Op(Y, ... );                    -- package P mentioned
```

There are two situations when P need not be mentioned – one is where the procedure call is actually inside the package P, the other is where we have a use clause for P (and even that sometimes does not give the required visibility). But these are special cases.

In Ada 2005 we can replace P.Op(Y, ...); by the so-called prefixed notation

```
Y.Op( ... );                      -- package P never mentioned
```

provided that

- T is a tagged type,
- Op is a primitive (dispatching) or class wide operation of T,
- Y is the first parameter of Op.

The reason there is never any need to mention the package is that, by starting from the object, we can identify its type and thus the primitive operations of the type. Note that a class wide operation can be called in this way only if it is declared at the same place as the primitive operations of T (or one of its ancestors).

There are many advantages of the prefixed notation as we shall see but perhaps the most important is ease of maintenance from not having to mention the package containing the declaration of the operation. Having to name the package is often tricky because in complicated situations involving several levels of inheritance it may not be obvious where the operation is declared. This happens especially when operations are declared implicitly and when class-wide operations are involved. Moreover if we change the structure for some reason then operations might move.

As a simple example consider a hierarchy of plane geometrical object types. All objects have a position given by the two coordinates *x* and *y* (this is the position of the centre of gravity of the object). There will be other specific properties according to the type such as the radius of a circle. In addition there might be general properties such as the area of the object, its distance from the origin and moment of inertia about it centre.

There are a number of ways in which such a hierarchy might be structured. We might have a package declaring a root abstract type and then another package with several derived types.

```
package Root is
  type Object is abstract tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
  function Distance(O: Object) return Float;
end Root;

package body Root is
  function Distance(O: Object) return Float is
  begin
```

```
    return Sqrt(O.X_Coord**2 + O.Y_Coord**2);
  end Distance;
end Root;
```

This package declares the root type and two abstract operations Area and MI (moment of inertia) and a concrete operation Distance. We might then have

```
with Root;
package Shapes is
  type Circle is new Root.Object with
    record
      Radius: Float;
    end record;

  function Area(C: Circle) return Float;
  function MI(C: Circle) return Float;

  type Triangle is new Root.Object with
    record
      A, B, C: Float;                -- lengths of sides
    end record;

  function Area(T: Triangle) return Float;
  function MI(T: Triangle) return Float;

-- and so on for other types such as Square

end Shapes;
```

(In the following discussion we will assume that use clauses are not being used. This is quite realistic because many projects forbid use clauses.)

Having declared some objects such as A_Circle and A_Triangle we can then apply the operations Area, Distance, and MI. In Ada 95 we write

```
A := Shapes.Area(A_Circle);
D := Shapes.Distance(A_Triangle);
M := Shapes.MI(A_Square);
```

Observe that the operation Distance is inherited and so is implicitly declared in the package Shapes for all types even though there is no mention of it in the text of the package Shapes. However, if we were using Ada 2005 and the prefixed notation then we could simply write

```
A := A_Circle.Area;
D := A_Triangle.Distance;
M := A_Square.MI;
```

and there is no mention of the package Shapes at all.

A clever friend then points out that by its nature Distance is the same for all types so it would be safer to avoid the risk of it getting changed by making it class wide. So we change the declaration of Distance in the package Root thus

```
  function Distance(O: Object'Class) return Float;
```

and recompile our program. But the Ada 95 version won't recompile. Why? Because class wide operations are not inherited. So there is only one function Distance and it is declared in the package Root. So all our calls of Distance have to be changed to

```
D := Root.Distance(A_Triangle);
```

However, if we had been using the prefixed notation then there would have been nothing to change.

Our manager might then read about the virtues of child packages and tell us to restructure the whole thing as follows

```
package Geometry is
  type Object is abstract ...

  ... -- functions Area, MI, Distance
end Geometry;

package Geometry.Circles is
  type Circle is new Object with
    record
      Radius: Float;
    end record;

  ... -- functions Area, MI
end Geometry.Circles;

package Geometry.Triangles is
  type Triangle is new Object with
    record
      A, B, C: Float;
    end record;

  ... -- functions Area, MI
end Geometry.Triangles;

-- and so on
```

This is of course a much more beautiful structure and avoids having to write Root.Object when doing the extensions. But, horrors, our assignments in Ada 95 now have to be changed to

```
A := Geometry.Circles.Area(A_Circle);
D := Geometry.Distance(A_Triangle);
M := Geometry.Squares.MI(A_Square);
```

But the lucky programmer using Ada 2005 can still write

```
A := A_Circle.Area;
D := A_Triangle.Distance;
M := A_Square.MI;
```

and have a refreshing coffee (or a relaxing martini) while we are toiling with the editor.

Some time later the program might be extended to accommodate triangles that are specialized to be equilateral. This might be done by

```
package Geometry.Triangles.Equilateral is
  type Equilateral_Triangle is new Triangle with private;
  ...
private
  ...
end;
```

This type of course inherits all the operations of the type Triangle. We might now realize that the object A_Triangle of type Triangle was equilateral anyway and so it would be better to change it to be of type Equilateral_Triangle. The lucky Ada 2005 programmer will only have to change the

declaration of the object but the poor Ada 95 programmer will have to change the calls on all its primitive operations such as

```
A := Geometry.Triangles.Area(A_Triangle);
```

to the corresponding

```
A := Geometry.Triangles.Equilateral.Area(A_Triangle);
```

Other advantages of the prefixed notation were mentioned in the Introduction. One is that it unifies the notation for calling a function with a single parameter and directly reading a component of the object. Thus we can write uniformly

```
X := A_Circle.X_Coord;
A := A_Circle.Area;
```

Of course if we were foolish and had a *visible* component Area as well as a function Area then we could not call the function in this way.

But now suppose we decide to make the root type private so that the coordinates cannot be changed inadvertently. Moreover we decide to provide functions to read them. So we have

```
package Geometry is
  type Object is abstract tagged private;
  function Area(O: Object) return Float is abstract;
  function MI(O: Object) return Float is abstract;
  function Distance(O: Object'Class) return Float;

  function X_Coord(O: Object'Class) return Float;
  function Y_Coord(O: Object'Class) return Float;

private
  type Object is tagged
    record
      X_Coord: Float;
      Y_Coord: Float;
    end record;

end Geometry;
```

Using Ada 95 we would now have to change statements such as

```
X := A_Triangle.X_Coord;
Y := A_Triangle.Y_Coord;
```

into

```
X := Geometry.X_Coord(A_Triangle);
Y := Geometry.Y_Coord(A_Triangle);
```

or (if we had not been wise enough to make the functions class wide) perhaps even

```
X := Geometry.Triangles.Equilateral.X_Coord(A_Triangle);
Y := Geometry.Triangles.Equilateral.Y_Coord(A_Triangle);
```

whereas in Ada 2005 we do not have to make any changes at all.

Another advantage mentioned in the Introduction is that when using access types explicit dereferencing is not necessary. Suppose we have

```
type Pointer is access all Geometry.Object'Class;
...
This_One: Pointer := A_Circle'Access;
```

In Ada 95 (assuming that X_Coord is a visible component) we have to write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(Geometry.Area(This_One.all));
```

whereas in Ada 2005 we can uniformly write

```
Put(This_One.X_Coord); ...
Put(This_One.Y_Coord); ...
Put(This_One.Area);
```

and of course this remains unchanged if we make the coordinates into functions whereas the Ada 95 statements will need to be changed.

There are other structural changes that can occur during program development which are much easier to cope with using the prefix notation. For example, a class wide operation might be moved. And in the case of multiple interfaces to be described in the next section an operation might be moved from one interface to another.

It is clear that the prefixed notation has significant benefits both in terms of program clarity and for program maintenance.

Other variations on the rules for the use of the notation were considered. One was that the mechanism should apply to untagged types as well but this was rejected on the grounds that it might add to rather than reduce confusion in some cases. In any event, untagged types do not have class wide types so they are intrinsically simpler.

It is of course important to note that the first parameter of an operation plays a special role since in order to take advantage of the prefixed notation we have to ensure that the first parameter is a controlling parameter. Treating the first parameter specially can appear odd in some circumstances such as when there is symmetry among the parameters. Thus suppose we have a set package for creating and manipulating sets of integers

```
package Sets is
  type Set is tagged private;
  function Empty return Set;
  function Unit(N: Integer) return Set;
  function Union(S, T: Set) return Set;
  function Intersection(S, T: Set) return Set;
  function Size(S: Set) return Integer;
  ...
end Sets;
```

then we can apply the function Union in the traditional way

```
A, B, C: Set;
...
C := Sets.Union(A, B);
```

The object oriented addict can also write

```
C := A.Union(B);
```

but this destroys the obvious symmetry and is rather like sending 3 to be added to 2 mentioned at the beginning of this discussion.

Hopefully the mature programmer will use the OO notation wisely. Maybe its existence will encourage a more uniform style in which the first parameter is always a controlling operand wherever possible. Of course it cannot be used for functions which are tag indeterminate such as

```
function Empty return Set;
function Unit(N: Integer) return Set;
```

since there are no controlling parameters. If a subprogram has just one parameter (which is controlling) such as Size then the call just becomes X.Size and no parentheses are necessary.

Note that the prefix does not have to be simply the name of an object such as X, it could be a function call so we might write

```
N := Sets.Empty.Size;                -- N = 0
M := Sets.Unit(99).Size;             -- M = 1
```

with the obvious results as indicated.

4 Interfaces

In Ada 95, a derived type can really only have one immediate ancestor. This means that true multiple inheritance is not possible although curious techniques involving discriminants and generics can be used in some circumstances

General multiple inheritance has problems. Suppose that we have a type T with some components and operations. Perhaps

```
type T is tagged
  record
    A: Integer;
    B: Boolean;
  end record;

procedure Op1(X: T);
procedure Op2(X: T);
```

Now suppose we derive two new types from T thus

```
type T1 is new T with
  record
    C: Character;
  end record;

procedure Op3(X: T1);

-- Op1 and Op2 inherited, Op3 added

type T2 is new T with
  record
    C: Colour;
  end record;

procedure Op1(X: T2);
procedure Op4(X: T2);

-- Op1 overridden, Op2 inherited, Op4 added
```

Now suppose that we were able to derive a further type from both T1 and T2 by perhaps writing

```
type TT is new T1 and T2 with null record;          -- illegal
```

This is about the simplest example one could imagine. We have added no further components or operations. But what would TT have inherited from its two parents?

There is a general rule that a record cannot have two components with the same identifier so presumably it has just one component A and one component B. But what about C? Does it inherit the character or the colour? Or is it illegal because of the clash? Suppose T2 had a component D instead of C. Would that be OK? Would TT then have four components?

And then consider the operations. Presumably it has both Op1 and Op2. But which implementation of Op1? Is it the original Op1 inherited from T via T1 or the overridden version inherited from T2? Clearly it cannot have both. But there is no reason why it cannot have both Op3 and Op4, one inherited from each parent.

The problems arise when inheriting components from more than one parent and inheriting different *implementations* of the same operation from more than one parent. There is no problem with inheriting the same specification of an operation from two parents.

These observations provide the essence of the solution. At most one parent can have components and at most one parent can have concrete operations – for simplicity we make them the same parent. But abstract operations can be inherited from several parents. This can be phrased as saying that this kind of multiple inheritance is about merging contracts to be satisfied rather than merging algorithms or state.

So Ada 2005 introduces the concept of an interface which is a tagged type with no components and no concrete operations. The idea of a null procedure as an operation of a tagged type is also introduced; this has no body but behaves as if it has a null body. Interfaces are only permitted to have abstract subprograms and null procedures as operations.

We will outline the ways in which interfaces can be declared and composed in a symbolic way and then conclude with a more practical example.

We might declare a package Pi1 containing an interface Int1 thus

```
package Pi1 is
  type Int1 is interface;
  procedure Op1(X: Int1) is abstract;
  procedure N1(X: Int1) is null;
end Pi1;
```

Note the syntax. It uses the new reserved word **interface**. It does not say **tagged** although all interface types are tagged. The abstract procedure Op1 has to be explicitly stated to be abstract as usual. The null procedure N1 uses new syntax as well. Remember that a null procedure behaves as if its body comprises a single null statement; but it doesn't actually have a concrete body.

The main type derivation rule then becomes that a tagged type can be derived from zero or one conventional tagged types plus zero or more interface types. Thus

```
type NT is new T and Int1 and Int2 with ... ;
```

where Int1 and Int2 are interface types. The normal tagged type if any has to be given first in the declaration. The first type is known as the parent so the parent could be a normal tagged type or an interface. The other types are known as progenitors. Additional components and operations are allowed in the usual way.

The term progenitors may seem strange but the term ancestors in this context was confusing and so a new term was necessary. Progenitors comes from the Latin progignere, to beget, and so is very appropriate.

It might have been thought that it would be quite feasible to avoid the formal introduction of the concept of an interface by simply saying that multiple parents are allowed provided only the first has components and concrete operations. However, there would have been implementation complexities with the risk of violating privacy and distributed overheads. Moreover, it would have caused maintenance problems since simply adding a component to a type or making one of its abstract operations concrete would cause errors elsewhere in the system if it was being used as a secondary parent. It is thus much better to treat interfaces as a fundamentally new concept. Another advantage is that this provides a new class of generic parameter rather neatly without complex rules for instantiations.

If the normal tagged type T is in a package Pt with operations Opt1, Opt2 and so on we could now write

```
with Pi1, Pt;
package PNT is
  type NT is new Pt.T and Pi1.Int1 with ... ;
  procedure Op1(X: NT);                    -- concrete procedure
  --  possibly other ops of NT
end PNT;
```

We must of course provide a concrete procedure for Op1 inherited from the interface Int1 since we have declared NT as a concrete type. We could also provide an overriding for N1 but if we do not then we simply inherit the null procedure of Int1. We could also override the inherited operations Opt1 and Opt2 from T in the usual way.

Interfaces can be composed from other interfaces thus

```
type Int2 is interface;
...
type Int3 is interface and Int1;
...
type Int4 is interface and Int1 and Int2;
...
```

Note the syntax. A tagged type declaration always has just one of **interface**, **tagged** and **with** (it doesn't have any if it is not a tagged type). When we derive interfaces in this way we can add new operations so that the new interface such as Int4 will have all the operations of both Int1 and Int2 plus possibly some others declared specifically as operations of Int4. All these operations must be abstract or null and there are fairly obvious rules regarding what happens if two or more of the ancestor interfaces have the same operation. Thus a null procedure overrides an abstract one but otherwise repeated operations must have profiles that are type conformant and have the same convention.

We refer to all the interfaces in an interface list as progenitors. So Int1 and Int2 are progenitors of Int4. The first one is not a parent – that term is only used when deriving a type as opposed to composing an interface.

Note that the term ancestor covers all generations whereas parent and progenitors are first generation only.

Similar rules apply when a tagged type is derived from another type plus one or more interfaces as in the case of the type NT which was

```
type NT is new T and Int1 and Int2 with ... ;
```

In this case it might be that T already has some of the operations of Int1 and/or Int2. If so then the operations of T must match those of Int1 or Int2 (be type conformant etc).

We informally speak of a specific tagged type as implementing an interface from which it is derived (directly or indirectly). The phrase "implementing an interface" is not used formally in the definition of Ada 2005 but it is useful for purposes of discussion.

Thus in the above example the tagged type NT must implement all the operations of the interfaces Int1 and Int2. If the type T already implements some of the operations then the type NT will automatically implement them because it will inherit the implementations from T. It could of course override such inherited operations in the usual way.

The normal "going abstract" rules apply in the case of functions. Thus if one operation is a function F thus

```
package Pi2 is
  type Int2 is interface;
  function F(Y: Int2) return Int2 is abstract;
end Pi2;
```

and T already has such a conforming operation

```
package PT is
  type T is tagged record ...
  function F(X: T) return T;
end PT;
```

then in this case the type NT must provide a concrete function F. See however the discussion at the end of this paper for the case when the type NT has a null extension.

Class wide types also apply to interface types. The class wide type Int1'Class covers all the types derived from the interface Int1 (both other interfaces as well as normal tagged types). We can then dispatch using an object of a concrete tagged type in that class in the usual way since we know that any abstract operation of Int1 will have been overridden. So we might have

```
type Int1_Ref is access all Int1'Class;
NT_Var: aliased NT;
Ref: Int1_Ref := NT_Var'Access;
```

Observe that conversion is permitted between the access to class wide type Int1_Ref and any access type that designates a type derived from the interface type Int1.

Interfaces can also be used in private extensions and as generic parameters.

Thus

```
  type PT is new T and Int2 and Int3 with private;
  ...
private
  type PT is new T and Int2 and Int3 with null record;
```

An important rule regarding private extensions is that the full view and the partial view must agree with respect to the set of interfaces they implement. Thus although the parent in the full view need not be T but can be any type derived from T, the same is not true of the interfaces which must be such that they both implement the same set exactly. This rule is important in order to prevent a client type from overriding private operations of the parent if the client implements an interface added in the private part.

Generic parameters take the form

```
generic
  type FI is interface and Int1 and Int2;
package ...
```

and then the actual parameter must be an interface which implements all the ancestors Int1, Int2 etc. The formal could also just be **type** FI **is interface**; in which case the actual parameter can be any interface. There might be subprograms passed as further parameters which would require that the actual has certain operations. The interfaces Int1 and Int2 might themselves be formal parameters occurring earlier in the parameter list.

Interfaces (and formal interfaces) can also be limited thus

```
type LI is limited interface;
```

We can compose mixtures of limited and nonlimited interfaces but if any one of them is nonlimited then the resulting interface must not be specified as limited. This is because it must implement the equality and assignment operations implied by the nonlimited interface. Similar rules apply to types which implement one or more interfaces. We will come back to this topic in a moment.

There are other forms of interfaces, namely synchronized interfaces, task interfaces, and protected interfaces. These bring support for polymorphic, class wide object oriented programming to the real time programming arena. They will be described in a later paper.

Having described the general ideas in somewhat symbolic terms, we will now discuss a more concrete example.

Before doing so it is important to emphasize that interfaces cannot have components and therefore if we are to perform multiple inheritance then we should think in terms of abstract operations to read and write components rather than the components themselves. This is standard OO thinking anyway because it preserves abstraction by hiding implementation details.

Thus rather than having a component such as Comp it is better to have a pair of operations. The function to read the component can simply be called Comp. A procedure to update the component might be Set_Comp. We will generally use this convention although it is not always appropriate to treat the components as unrelated entities.

Suppose now that we want to print images of the geometrical objects. We will assume that the root type is declared as

```
package Geometry is
  type Object is abstract tagged private;
  procedure Move(O: in out Object'Class; X, Y: Float);
  ...
private
  type Object is abstract tagged
    record
      X_Coord: Float := 0.0;
      Y_Coord: Float := 0.0;
    end record;
  ...
end;
```

The type Object is private and by default both coordinates have the value of zero. The procedure Move, which is class wide, enables any object to be moved to the location specified by the parameters.

Suppose also that we have a line drawing package with the following specification

```
package Line_Draw is
   type Printable is interface;
   type Colour is ... ;
   type Points is ... ;
   procedure Set_Hue(P: in out Printable; C: in Colour) is abstract;
   function Hue(P: Printable) return Colour is abstract;
   procedure Set_Width(P: in out Printable; W: in Points) is abstract;
   function Width(P: Printable) return Points is abstract;

   type Line is ... ;
   type Line_Set is ... ;

   function To_Lines(P: Printable) return Line_Set is abstract;

   procedure Print(P: in Printable'Class);

private
   procedure Draw_It(L: Line; C: Colour; W: Points);

end Line_Draw;
```

The idea of this package is that it enables the drawing of an image as a set of lines. The attributes of the image are the hue and the width of the lines and there are pairs of subprograms to set and read these properties of any object of the interface Printable and its descendants. These operations are of course abstract.

In order to prepare an object in a form that can be printed it has to be converted to a set of lines. The function To_Lines converts an object of the type Printable into a set of lines; again it is abstract. The details of various types such as Line and Line_Set are not shown.

Finally the package Line_Draw declares a concrete procedure Print which takes an object of type Printable'Class and does the actual drawing using the slave procedure Draw_It declared in the private part. Note that Print is class wide and is concrete. This is an important point. Although all primitive operations of an interface must be abstract this does not apply to class wide operations since these are not primitive.

The body of the procedure Print could take the form

```
procedure Print(P: in Printable'Class) is
   L: Line_Set := To_Lines(P);
   A_Line: Line;
begin
   loop
      -- iterate over the Line_Set and extract each line
      A_Line := ...
      Draw_It(A_Line, Hue(P), Width(P));
   end loop;
end Print;
```

but this is all hidden from the user. Note that the procedure Draw_It is declared in the private part since it need not be visible to the user.

One reason why the user has to provide To_Lines is that only the user knows about the details of how best to represent the object. For example the poor circle will have to be represented crudely as a polygon of many sides, perhaps a hectogon of 100 sides.

We can now take at least two different approaches. We can for example write

```
with Geometry, Line_Draw;
package Printable_Geometry is
  type Printable_Object is
                    abstract new Geometry.Object and Line_Draw.Printable with private;
  procedure Set_Hue(P: in out Printable_Object; C: in Colour);
  function Hue(P: Printable_Object) return Colour;
  procedure Set_Width(P: in out Printable_Object; W: in Points);
  function Width(P: Printable_Object) return Points;
  function To_Lines(P: Printable_Object) return Line_Set is abstract;

private
  ...
end Printable_Geometry;
```

The type Printable_Object is a descendant of both Object and Printable and all concrete types descended from Printable_Object will therefore have all the operations of both Object and Printable. Note carefully that we have to put Object first in the declaration of Printable_Object and that the following would be illegal

```
type Printable_Object is
                abstract new Line_Draw.Printable and Geometry.Object with private;   --illegal
```

This is because of the rule that only the first type in the list can be a normal tagged type; any others must be interfaces. Remember that the first type is always known as the parent type and so the parent type in this case is Object.

The type Printable_Object is declared as abstract because we do not want to implement To_Lines at this stage. Nevertheless we can provide concrete subprograms for all the other operations of the interface Printable. We have given the type a private extension and so in the private part of its containing package we might have

```
private
  type Printable_Object is abstract new Geometry.Object and Line_Draw.Printable with
    record
      Hue: Colour := Black;
      Width: Points := 1;
    end record;
end Printable_Geometry;
```

Just for way of illustration, the components have been given default values. In the package body the operations such as the function Hue are simply

```
  function Hue(P: Printable_Object) return Colour is
  begin
    return P.Hue;
  end;
```

Luckily the visibility rules are such that this does not do an infinite recursion!

Note that the information containing the style components is in the record structure following the geometrical properties. This is a simple linear structure since interfaces cannot add components. However, since the type Printable_Object has all the operations of both an Object and a Printable, this adds a small amount of complexity to the arrangement of dispatch tables. But this detail is hidden from the user.

The key point is that we can now pass any object of the type Printable_Object or its descendants to the procedure

```
procedure Print(P: in Printable'Class);
```

and then (as outlined above) within Print we can find the colour to be used by calling the function Hue and the line width to use by calling the function Width and we can convert the object into a set of lines by calling the function To_Lines.

And now we can declare the various types Circle, Triangle, Square and so on by making them descendants of the type Printable_Object and in each case we have to implement the function To_Lines.

The unfortunate aspect of this approach is that we have to move the geometry hierarchy. For example the triangle package might now be

```
package Printable_Geometry.Triangles is
  type Printable_Triangle is new Printable_Object with
    record
      A, B, C: Float;
    end record;
  ... -- functions Area, To_Lines etc
end;
```

We can now declare a Printable_Triangle thus

```
A_Triangle: Printable_Triangle := (Printable_Object with A => 4.0, B => 4.0, C => 4.0);
```

This declares an equilateral triangle with sides of length 4.0. Its private Hue and Width components are set by default. Its coordinates which are also private are by default set to zero so that it is located at the origin. (The reader can improve the example by making the components A, B and C private as well.)

We can conveniently move it to wherever we want by using the procedure Move which being class wide applies to all types derived from Object. So we can write

```
A_Triangle.Move(1.0, 2.0);
```

And now we can make a red sign

```
Sign: Printable_Triangle := A_Triangle;
```

Having declared the object Sign, we can give it width and hue and print it

```
Sign.Set_Hue(Red);
Sign.Set_Width(3);
Sign.Print;                                    -- print thick red triangle
```

As we observed earlier this approach has the disadvantage that we had to move the geometry hierarchy. A different approach which avoids this is to declare printable objects of just the kinds we want as and when we want them.

So assume now that we have the package Line_Draw as before and the original package Geometry and its child packages. Suppose we want to make printable triangles and circles. We could write

```
with Geometry, Line_Draw;  use Geometry;
package Printable_Objects is
  type Printable_Triangle is new Triangles.Triangle and Line_Draw.Printable with private;
  type Printable_Circle is new Circles.Circle and Line_Draw.Printable with private;

  procedure Set_Hue(P: in out Printable_Triangle; C: in Colour);
  function Hue(P: Printable_Triangle return Colour;
  procedure Set_Width(P: in out Printable_Triangle; W: in Points);
```

```
    function Width(P: Printable_Triangle) return Points;
    function To_Lines(T: Printable_Triangle) return Line_Set;

    procedure Set_Hue(P: in out Printable_Circle; C: in Colour);
    function Hue(P: Printable_Circle) return Colour;
    procedure Set_Width(P: in out Printable_Circle; W: in Points);
    function Width(P: Printable_Circle) return Points;
    function To_Lines(C: Printable_Circle) return Line_Set;
  private

    type Printable_Triangle is new Triangles.Triangle and Line_Draw.Printable with
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;

    type Printable_Circle is new Circles.Circle and Line_Draw.Printable with
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;
  end Printable_Objects;
```

and the body of the package will provide the various subprogram bodies.

Now suppose we already have a normal triangle thus

```
A_Triangle: Geometry.Triangles.Triangle := ... ;
```

In order to print A_Triangle we first have to declare a printable triangle thus

```
Sign: Printable_Triangle;
```

and now we can set the triangle components of it using a view conversion thus

```
Triangle(Sign) := A_Triangle;
```

And then as before we write

```
Sign.Set_Hue(Red);
Sign.Set_Width(3);
Sign.Print_It;                          -- print thick red triangle
```

This second approach is probably better since it does not require changing the geometry hierarchy. The downside is that we have to declare the boring hue and width subprograms repeatedly. We can make this much easier by declaring a generic package thus

```
with Line_Draw;  use Line_Draw;
generic
  type T is abstract tagged private;
package Make_Printable is
  type Printable_T is abstract new T and Printable with private;

  procedure Set_Hue(P: in out Printable_T; C: in Colour);
  function Hue(P: Printable_T) return Colour;
  procedure Set_Width(P: in out Printable_T; W: in Points);
  function Width(P: Printable_T) return Points;

private
  type Printable_T is abstract new T and Printable with
```

```
      record
        Hue: Colour := Black;
        Width: Points := 1;
      end record;
   end;
```

This generic can be used to make any type printable. We simply write

```
package P_Triangle is new Make_Printable(Triangle);
type Printable_Triangle is new P_Triangle.Printable_T with null record;
function To_Lines(T: Printable_Triangle) return Line_Set;
```

The instantiation of the package creates a type Printable_T which has all the hue and width operations and the required additional components. However, it simply inherits the abstract function To_Lines and so itself has to be an abstract type. Note that the function To_Lines has to be especially coded for each type anyway unlike the hue and width operations which can be the same.

We now do a further derivation largely in order to give the type Printable_T the required name Printable_Triangle and at this stage we provide the concrete function To_Lines.

We can then proceed as before. Thus the generic makes the whole process very easy – any type can be made printable by just writing three lines plus the body of the function To_Lines.

Hopefully this example has illustrated a number of important points about the use of interfaces. The key thing perhaps is that we can use the procedure Print to print anything that implements the interface Printable.

Earlier we stated that it was a common convention to provide pairs of operations to read and update properties such as Hue and Set_Hue and Width and Set_Width. This is not always appropriate. Thus if we have related components such as X_Coord and Y_Coord then although individual functions to read them might be sensible, it is undoubtedly better to update the two values together with a single procedure such as the procedure Move declared earlier. Thus if we wish to move an object from the origin (0.0, 0.0) to say (3.0, 4.0) and do it by two calls

```
Obj.Set_X_Coord(3.0);          -- first change X
Obj.Set_Y_Coord(4.0);          -- then change Y
```

then it seems as if it was transitorily at the point (3.0, 0.0). There are various other risks as well. We might forget to set one component or accidentally set the same component twice.

Finally, as discussed earlier, null procedures are a new kind of subprogram and the user-defined operations of an interface must be null procedures or abstract subprograms – there is of course no such thing as a null function.

(Nonlimited interfaces do have one concrete operation and that is predefined equality; it could even be overridden with an abstract one.)

Null procedures will be found useful for interfaces but are in fact applicable to any types. As an example the package Ada.Finalization now uses null procedures for Initialize, Adjust, and Finalize as described in the Introduction.

We conclude this section with a few further remarks on limitedness. We noted earlier that an interface can be explicitly stated to be limited so we might have

```
type LI is limited interface;      -- limited
type NLI is interface;             -- nonlimited
```

An interface is limited only if it says limited (or synchronized etc). As mentioned earlier, a descendant of a nonlimited interface must be nonlimited since it must implement assignment and

equality. So if an interface is composed from a mixture of limited and nonlimited interfaces it must be nonlimited

```
type I is interface and LI and NLI;              -- legal
type I is limited interface and LI and NLI;      -- illegal
```

In other words, limitedness is never inherited from an interface but has to be stated explicitly. This applies to both the composition of interfaces and type derivation. On the other hand, in the case of type derivation, limitedness is inherited from the parent provided it is not an interface. This is necessary for compatibility with Ada 95. So given

```
type LT is limited tagged ...
type NLT is tagged ...
```

then

```
type T is new NLT and LI with ...        -- legal, T not limited
type T is new NLT and NLI with ...       -- legal, T not limited
type T is new LT and LI with ...         -- legal, T limited
type T is new LT and NLI with ...        -- illegal
```

The last is illegal because T is expected to be limited because it is derived from the limited parent type LT and yet it is also a descendant of the nonlimited interface NLI.

In order to avoid certain curious difficulties, Ada 2005 permits **limited** to be stated explicitly on type derivation. (It would have been nice to insist on this always for clarity but such a change would have been too much of an incompatibility.) If we do state **limited** explicitly then the parent must be limited (whether it is a type or an interface).

Using **limited** is necessary if we wish to derive a limited type from a limited interface thus

```
type T is limited new LI with ...
```

These rules really all come down to the same thing. If a parent or progenitor (indeed any ancestor) is nonlimited then the descendant must be nonlimited. We can state that in reverse, if a type (including an interface) is limited then all its ancestors must be limited.

An earlier version of Ada 2005 ran into difficulties in this area because in the case of a type derived just from interfaces, the behaviour could depend upon the order of their appearance in the list (because the rules for parent and progenitors are a bit different). But in the final version of the language the order does not matter. So

```
type T is new NLI and LI with ...        -- legal, not limited
type T is new LI and NLI with ...        -- legal, not limited
```

But the following are of course illegal

```
type T is limited new NLI and LI with ...    -- illegal
type T is limited new LI and NLI with ...    -- illegal
```

There are also similar changes to generic formals and type extension – Ada 2005 permits **limited** to be given explicitly in both cases.

5 Nested type extension

In Ada 95 type extension of tagged types has to be at the same level as the parent type. This can be quite a problem. In particular it means that all controlled types must be declared at library level because the root types Controlled and Limited_Controlled are declared in the library level package Ada.Finalization. The same applies to storage pools and streams because again the root types Root_Storage_Pool and Root_Stream_Type are declared in library packages.

This has a cumulative effect since if we write a generic unit using any of these types then that package can itself only be instantiated at library level. This enforces a very flat level of programming and hinders abstraction.

The problems can actually be illustrated without having to use controlled types or generics. As a simple example consider the following which is adapted from a text book [3]. It manipulates lists of colours and we assume that the type Colour is declared somewhere.

```
package Lists is
  type List is limited private;
  type Iterator is abstract tagged null record;
  procedure Iterate(IC: in Iterator'Class; L: in List);
  procedure Action(It: in out Iterator; C: in out Colour) is abstract;
private
  ...
end;
```

The idea is that a call of Iterate calls Action (by dispatching) on each object of the list and thereby gives access to the colour of that object. The user has to declare an extension of Iterator and a specific procedure Action to do whatever is required on each object.

Some readers may find this sort of topic confusing. It might be easier to understand if we look at the private part and body of the package Lists which might be

```
private
  type Cell is
    record
      Next: access Cell;                    -- anonymous type
      C: Colour;
    end record;

  type List is access Cell;
end;

package body Lists is
  procedure Iterate(IC: in Iterator'Class; L: in List) is
    This: access Cell := L;
  begin
    while This /= null loop
      Action(IC, This.C);                   -- dispatching call
                                            -- or IC.Action(This.C);
      This := This.Next;
    end loop;
  end Iterate;
end Lists;
```

Note the use of the anonymous access types which avoid the need to have an incomplete declaration of Cell in the private part.

Now suppose we wish to change the colour of every green object to red. We write (in some library level package)

```
type GTR_It is new Iterator with null record;

procedure Action(It: in out GTR_It; C: in out Colour) is
begin
```

```
    if C = Green then C := Red; end if;
  end Action;

  procedure Green_To_Red(L: in List) is
    It: GTR_It;
  begin
    Iterate(It, L);                          -- or It.Iterate(L);
  end Green_To_Red;
```

This works but is not ideal. The type GTR_It and the procedure Action should not be declared outside the procedure Green_To_Red since they are really only part of its internal workings. But we cannot declare the type GTR_It inside the procedure in Ada 95 because that would be an extension at an inner level.

The extra facilities of the predefined library in Ada 2005 and especially the introduction of containers which are naturally implemented as generic units forced a reconsideration of the reasons for restricting type extension in Ada 95. The danger of nested extension of course is that values of objects could violate the accessibility rules and outlive their type declaration. It was concluded that type extension could be permitted at nested levels with the addition of just a few checks to ensure that the accessibility rules were not violated.

So in Ada 2005 the procedure Green_To_Red can be written as

```
  procedure Green_To_Red(L: in List) is
    type GTR_It is new Iterator with null record;

    procedure Action(It: in out GTR_It; C: in out Colour) is
    begin
      if C = Green then C := Red; end if;
    end Action;

    It: GTR_It;
  begin
    Iterate(It, L);                          -- or It.Iterate(L);
  end Green_To_Red;
```

and all the workings are now wrapped up within the procedure as they should be.

Note incidentally that we can use the notation It.Iterate(L); even though the type GTR_It is not declared in a package in this case. Remember that although we cannot add new dispatching operations to a type unless it is declared in a package specification, nevertheless we can always override existing ones such as Action.

This example is all quite harmless and nothing can go wrong despite the fact that we have performed the extension at an inner level. This is because the value It does not outlive the execution of the procedure Action.

But suppose we have a class wide object Global_It as in the following

```
with Lists; use Lists;
package body P is

  function Dodgy return Iterator'Class is
    type Bad_It is new Iterator with null record;

    procedure Action(It: in out Bad_It; C: in out Colour) is
    begin
      ...
    end Action;
```

```
    It: Bad_It;
  begin
    return It;
  end Dodgy;

  Global_It: Iterator'Class := Dodgy;
begin
  Global_It.Action(Red_For_Danger);        -- dispatches
end P;
```

Now we are in deep trouble. We have returned a value of the local type Bad_It, assigned it as the initial value to Global_It and then dispatched on it to the procedure Action. But the procedure Action that will be called is the one inside Dodgy and this does not exist anymore since we have left the function Dodgy. So this must not be allowed to happen.

So various accessibility checks are required. There is a check on the return from a function with a class wide result that the value being returned does not have the tag of a type at a deeper level than that of the function itself. So in this example there is a check on the return from the function Dodgy; this fails and raises Program_Error so all is well.

There are similar checks on class wide allocators and when using T'Class'Input or T'Class'Output. Some of these can be carried out at compile time but others have to be checked at run time and they also raise Program_Error if they fail.

Moreover, in order to implement the checks associated with T'Class'Input and T'Class'Output two additional functions are declared in the package Ada.Tags; these are

```
function Descendant_Tag(External: String; Ancestor: Tag) return Tag;

function Is_Descendant_At_Same_Level (Descendant, Ancestor: Tag) return Boolean;
```

The use of these will be outlined in the next section.

6 Object factory functions

The Ada 95 Rationale (Section 4.4.1) [2] says "We also note that object oriented programming requires thought especially if variant programming is to be avoided. There is a general difficulty in finding out what is coming which is particularly obvious with input–output; it is easy to write dispatching output operations but generally impossible for input." In this context, variant programming means messing about with case statements and so on.

The point about input–output is that it is easy to write a heterogeneous file but not so easy to read it. In the simple case of a text file we can just do a series of calls of Put thus

```
Put ("John is ");  Put(21, 0);  Put(" years old.");
```

But text input is not so easy unless we know the order of the items in the file. If we don't know the order then we really have to read the wretched thing a line at a time and then analyse the lines.

Ada 95 includes a mechanism for doing this relatively easily in the case of tagged types and stream input–output. Suppose we have a class of tagged types rooted at Root with various derived specific types T1, T2 and so on. We can then output a sequence of values X1, X2, X3 of a variety of these types to a file identified by the stream access value S by writing

```
Root'Class'Output(S, X1);
Root'Class'Output(S, X2);
Root'Class'Output(S, X3);
...
```

The various calls first write the tag of the specific type and then the value of the type. The tag corresponding to the type T1 is the string External_Tag(T1'Tag). Remember that External_Tag is a function in the predefined package Ada.Tags.

On input we can reverse the process by writing something like

```
declare
   X: Root'Class := Root'Class'Input(S);
begin
   Process(X);                              -- now process the object in X
```

The call of Root'Class'Input first reads the external tag and then dispatches to the appropriate function Tn'Input according to the value of the tag. The function reads the value and this is now assigned as the initial value to the class wide variable X. We can then do whatever we want with X by perhaps dispatching to a procedure Process which deals with it according to its specific type.

This works in Ada 95 but it is all magic and done by smoke and mirrors inside the implementation. The underlying techniques are unfortunately not available to the user.

This means that if we want to devise our own stream protocol or maybe just process some values in circumstances where we cannot directly use dispatching then we have to do it all ourselves with if statements or case statements. Thus we might be given a tag value and separately some information from which we can create the values of the particular type. In Ada 95 we typically have to do something like

```
The_Tag: Ada.Tags.Tag;
A_T1: T1;                                   -- series of objects of each
A_T2: T2;                                   -- specific type
A_T3: T3;
...
The_Tag := Get_Tag( ... );                  -- get the tag value
if The_Tag = T1'Tag then
   A_T1 := Get_T( ... );                    -- get value of specific type
   Process(A_T1);                           -- process the object
elsif The_Tag = T2'Tag then
   A_T2 := Get_T( ... );                    -- get value of specific type
   Process(A_T2);                           -- process the object
elsif
   ...
end if;
```

We assume that Get_T is a primitive function of the class rooted at Root. There is therefore a function for each specific type and the selection in the if statements is made at compile time by the normal overload rules. Similarly Process is also a primitive subprogram of the class of types.

This is all very tedious and needs careful maintenance if we add further types to the class.

Ada 2005 overcomes this problem by providing a generic object constructor function. Its specification is

```
generic
   type T (<>) is abstract tagged limited private;
   type Parameters (<>) is limited private;
   with function Constructor(Params: access Parameters) return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
                              (The_Tag: Tag; Params: access Parameters) return T'Class;
```

```
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);
```

This generic function works for both limited and nonlimited types. Remember that a nonlimited type is allowed as an actual generic parameter corresponding to a limited formal generic type. The generic function Generic_Dispatching_Constructor is Pure and has convention Intrinsic.

Note carefully the formal function Constructor. This is an example of a new kind of formal generic parameter introduced in Ada 2005. The distinctive feature is the use of **is abstract** in its specification. The interpretation is that the actual function must be a dispatching operation of a tagged type uniquely identified by the profile of the formal function. The actual operation can be concrete or abstract. Remember that the overriding rules ensure that the specific operation for any concrete type will always have a concrete body. Note also that since the operation is abstract it can only be called through dispatching.

In this example, it therefore has to be a dispatching operation of the type T since that is the only tagged type involved in the profile of Constructor. We say that T is the controlling type. In the general case, the controlling type does not itself have to be a formal parameter of the generic unit but usually will be as here. Moreover, note that although the operation has to be a dispatching operation, it is not primitive and so if we derive from the type T, it will not be inherited.

Formal abstract subprograms can of course be procedures as well as functions. It is important that there is exactly one controlling type in the profile. Thus given that TT1 and TT2 are tagged types then the following would both be illegal

```
with procedure Do_This(X1: TT1; X2: TT2) is abstract;      -- illegal
with function Fn(X: Float) return Float is abstract;       -- illegal
```

The procedure Do_This is illegal because it has two controlling types TT1 and TT2. Remember that we can declare a subprogram with parameters of more than one tagged type but it can only be a dispatching operation of one tagged type. The function Fn is illegal because it doesn't have any controlling types at all (and so could never be called in a dispatching call anyway).

The formal function Constructor is legal because only T is tagged; the type Parameters which also occurs in its profile is not tagged.

And now to return to the dispatching constructor. The idea is that we instantiate the generic function with a (root) tagged type T, some type Parameters and the dispatching function Constructor. The type Parameters provides a means whereby auxiliary information can be passed to the function Constructor.

The generic function Generic_Dispatching_Constructor takes two parameters, one is the tag of the type of the object to be created and the other is the auxiliary information to be passed to the dispatching function Constructor.

Note that the type Parameters is used as an access parameter in both the generic function and the formal function Constructor. This is so that it can be matched by the profile of the attribute Input whose specification is

```
function T'Input(Stream: access Root_Stream_Type'Class) return T;
```

Suppose we instantiate Generic_Dispatching_Constructor to give a function Make_T. A call of Make_T takes a tag value, dispatches to the appropriate Constructor which creates a value of the specific tagged type corresponding to the tag and this is finally returned as the value of the class wide type T'Class as the result of Make_T. It's still magic but anyone can use the magic and not just the magician implementing stream input–output.

We can now do our abstract problem as follows

```
function Make_T is
   new Generic_Dispatching_Constructor(Root, Params, Get_T);

...
declare
  Aux: aliased Params := ... ;
  A_T: Root'Class:= Make_T(Get_Tag( ... ), Aux'Access);
begin
  Process(A_T);                                   -- dispatch to process the object
end;
```

We no longer have the tedious sequence of if statements and the calls of Get_T and Process are dispatching calls.

The previously magic function T'Class'Input can now be implemented in a very natural way by something like

```
function Dispatching_Input is
   new Generic_Dispatching_Constructor(T, Root_Stream_Type'Class, T'Input);

function T_Class_Input(S: access Root_Stream_Type'Class) return T'Class is
  The_String: String := String'Input(S);                -- read tag as string from stream
  The_Tag: Tag := Descendant_Tag(The_String, T'Tag);    -- convert to a tag
 begin
  -- now dispatch to the appropriate function Input
  return Dispatching_Input(The_Tag, S);
end T_Class_Input;

for T'Class'Input use T_Class_Input;
```

The body could of course be written as one giant statement

```
return Dispatching_Input(Descendant_Tag(String'Input(S), T'Tag), S);
```

but breaking it down hopefully clarifies what is happening.

Note the use of Descendant_Tag rather than Internal_Tag. Descendant_Tag is one of a few new functions introduced into the package Ada.Tags in Ada 2005. Streams did not work very well for nested tagged types in Ada 95 because of the possibility of multiple elaboration of declarations (as a result of tasking and recursion); this meant that two descendant types could have the same external tag value and Internal_Tag could not distinguish them. This is not an important problem in Ada 95 as nested tagged types are rarely used. In Ada 2005 the situation is potentially made worse because of the possibility of nested type extension.

The goal in Ada 2005 is simply to ensure that streams do work with types declared at the same level and to prevent erroneous behaviour otherwise. The goal is not to permit streams to work with the nested extensions introduced in Ada 2005. Any attempt to do so will result in Tag_Error being raised.

Note that we cannot actually declare an attribute function such as T'Class'Input by directly using the attribute name. We have to use some other identifier such as T_Class_Input and then use an attribute definition clause as shown above.

Observe that T'Class'Output can be implemented as

```
procedure T_Class_Output(S: access Root_Stream_Type'Class; X: in T'Class) is
begin
  if not Is_Descendant_At_Same_Level(X'Tag, T'Tag) then
    raise Tag_Error;
```

```
   end if;
   String'Output(S, External_Tag(X'Tag));
   T'Output(S, X);
end T_Class_Output;

for T'Class'Output use T_Class_Output;
```

Remember that streams are designed to work only with types declared at the same accessibility level as the parent type T. The call of Is_Descendant_At_Same_Level, which is another new function in Ada 2005, ensures this.

We can use the generic constructor to create our own stream protocol. We could in fact replace T'Class'Input and T'Class'Output or just create our own distinct subsystem. One reason why we might want to use a different protocol is when the external protocol is already given such as in the case of XML.

Note that it will sometimes be the case that there is no need to pass any auxiliary parameters to the constructor function in which case we can declare

```
type Params is null record;
Aux: aliased Params := (null record);
```

Another example can be based on part of the program Magic Moments in [3]. This reads in the values necessary to create various geometrical objects such as a Circle, Triangle, or Square which are derived from an abstract type Object. The values are preceded by a letter C, T or S as appropriate. The essence of the code is

```
Get(Code_Letter);
case Code_Letter is
   when 'C' => Object_Ptr := Get_Circle;
   when 'T' => Object_Ptr := Get_Triangle;
   when 'S' => Object_Ptr := Get_Square;
   ...
end case;
```

The types Circle, Triangle, and Square are derived from the root type Object and Object_Ptr is of the type **access** Object'Class. The function Get_Circle reads the value of the radius from the keyboard, the function Get_Triangle reads the values of the lengths of the three sides from the keyboard and so on.

The first thing to do is to change the various constructor functions such as Get_Circle into various specific overridings of a primitive operation Get_Object so that we can dispatch on it.

Rather than just read the code letter we could make the user type the external tag string and then we might have

```
function Make_Object is
   new Generic_Dispatching_Constructor(Object, Params, Get_Object);

...
S: String := Get_String;
...
Object_Ptr := new Object'(Make_Object(Internal_Tag(S), Aux'Access));
```

but this is very tedious because the user now has to type the external tag which will be an implementation defined mess of characters. Observe that the string produced by a call of Expanded_Name such as

```
OBJECTS.CIRCLE
```

cannot be used because it will not in general be unique and so there is no reverse function. (It is not generally unique because of tasking and recursion.) But Expanded_Name is useful for debugging purposes.

In these circumstances the best way to proceed is to invent some sort of registration system to make a map to convert the simple code letters into the tag. We might have a package

```
with Ada.Tags; use Ada.Tags;
package Tag_Registration is
  procedure Register(The_Tag: Tag; Code: Character);
  function Decode(Code: Character) return Tag;
end;
```

and then we can write

```
Register(Circle'Tag, 'C');
Register(Triangle'Tag, 'T');
Register(Square'Tag, 'S');
```

And now the program to read the code and then make the object becomes simply

```
Get(Code_Letter);
Object_Ptr := new Object'(Make_Object(Decode(Code_Letter), Aux'Access));
```

and there are no case statements to maintain.

The really important point about this example is that if we decide at a later date to add more types such as 'P' for Pentagon and 'H' for Hexagon then all we have to do is register the new code letters thus

```
Register(Pentagon'Tag, 'P');
Register(Hexagon'Tag, 'H');
```

and nothing else needs changing. This registration can conveniently be done when the types are declared.

The package Tag_Registration could be implemented trivially as follows by

```
package body Tag_Registration is
  Table: array (Character range 'A' .. 'Z') of Tag := (others => No_Tag);
  procedure Register(The_Tag: Tag; Code: Character) is
  begin
    Table(Code) := The_Tag;
  end Register;

  function Decode(Code: Character) return Tag is
  begin
    return Table(Code);
  end Decode;
end Tag_Registration;
```

The constant No_Tag is a value of the type Tag which does not represent an actual tag. If we forget to register a type then No_Tag will be returned by Decode and this will cause Make_Object to raise Tag_Error.

A more elegant registration system could be easily implemented using the container library which will be described in a later paper.

Note that any instance of Generic_Dispatching_Constructor checks that the tag passed as parameter is indeed that of a type descended from the root type T and raises Tag_Error if it is not.

In simple cases we could in fact perform that check for ourselves by writing something like

```
   Trial_Tag: Tag := The_Tag;
loop
   if Trial_Tag = T'Tag then exit; end if;
   Trial_Tag := Parent_Tag(Trial_Tag);
   if Trial_Tag = No_Tag then raise Tag_Error; end if;
end loop;
```

The function Parent_Tag and the constant No_Tag are further items in the package Ada.Tags whose specification in Ada 2005 is

```
package Ada.Tags is
   pragma Preelaborate(Tags);

   type Tag is private;
   No_Tag: constant Tag;

   function Expanded_Name(T: Tag) return String;
   ...                    -- also Wide and Wide_Wide versions
   function External_Tag(T: Tag) return String;
   function Internal_Tag(External: String) return Tag;
   function Descendant_Tag(External: String; Ancestor: Tag) return Tag;
   function Is_Descendant_At_Same_Level(Descendant, Ancestor: Tag) return Boolean;
   function Parent_Tag(T: Tag) return Tag;

   type Tag_Array is (Positive range <>) of Tag;
   function Interface_Ancestor_Tags(T: Tag) return Tag_Array;

   Tag_Error: exception;
private
   ...
end Ada.Tags;
```

The function Parent_Tag returns No_Tag if the parameter T of type Tag has no parent which will be the case if it is the ultimate root type of the class. As mentioned earlier, two other new functions Descendant_Tag and Is_Descendant_At_Same_Level are necessary to prevent the misuse of streams with types not all declared at the same level.

There is also a function Interface_Ancestor_Tags which returns the tags of all those interfaces which are ancestors of T as an array. This includes the parent if it is an interface, any progenitors and all their ancestors which are interfaces as well – but it excludes the type T itself.

Finally note that the introduction of 16- and 32-bit characters in identifiers means that functions also have to be provided to return the images of identifiers as a Wide_String or Wide_Wide_String. So we have functions Wide_Expanded_Name and Wide_Wide_Expanded_Name as well as Expanded_Name. The lower bound of the strings returned by these functions and by External_Tag is 1 – Ada 95 forgot to state this for External_Tag and Expanded_Name!

7 Overriding and overloading

One of the key goals in the design of Ada was to encourage the writing of correct programs. It was intended that the structure, strong typing, and so on should ensure that many errors which are not detected by most languages until run time should be caught at compile time in Ada. Unfortunately the introduction of type extension and overriding in Ada 95 produced a situation where careless errors in subprogram profiles lead to errors which are awkward to detect.

The Introduction described two typical examples. The first concerns the procedure Finalize. Consider

```
with Ada.Finalization;  use Ada.Finalization;
package Root is
  type T is new Controlled with ... ;
  procedure Op(Obj: in out T; Data: in Integer);
  procedure Finalise(Obj: in out T);
end Root;
```

We have inadvertently written Finalise rather than Finalize. This means that Finalize does not get overridden as expected and so the expected behaviour does not occur on finalization of objects of type T.

In Ada 2005 we can prefix the declaration with **overriding**

```
  overriding
  procedure Finalize(Obj: in out T);
```

And now if we inadvertently write Finalise then this will be detected during compilation.

Similar errors can occur in a profile. If we write

```
package Root.Leaf is
  type NT is new T with null record;
  overriding                                   -- overriding indicator
  procedure Op(Obj: in out NT; Data: in String);
end Root.Leaf;
```

then the compiler will detect that the new procedure Op has a parameter of type String rather than Integer.

However if we do want a new operation then we can write

```
  not overriding
  procedure Op(Obj: in out NT; Data: in String);
```

The overriding indicators can also be used with abstract subprograms, null procedures, renamings, instantiations, stubs, bodies and entries (we will deal with entries in the paper on tasking). So we can have

```
overriding
procedure Pap(X: TT) is abstract;

overriding
procedure Pep(X: TT) is null;

overriding
procedure Pip(Y: TT) renames Pop;

not overriding
procedure Poop is new Peep( ... );

overriding
procedure Pup(Z: TT) is separate;

overriding
procedure Pup(X: TT) is
begin ... end Pup;
```

We do not need to apply an overriding indicator to both a procedure specification and body but if we do then they naturally must not conflict. It is expected that overriding indicators will typically only be given on specifications but they would be appropriate in the case of a body standing alone as in the example of Action in the previous section. So we might have

```
procedure Green_To_Red(L: in List) is
  type GTR_It is new Iterator with null record;

  overriding
  procedure Action(It: in out GTR_It; C: in out Colour) is
  begin
    if C = Green then C := Red; end if;
  end Action;
...
```

The overriding indicators are optional for two reasons. One is simply for compatibility with Ada 95. The other concerns awkward problems with private types and generics.

Consider

```
package P is
  type NT is new T with private;
  procedure Op(X: T);
private
```

Now suppose the type T does not have an operation Op. Then clearly it would be wrong to write

```
package P is
  type NT is new T with private;          -- T has no Op
  overriding                              -- illegal
  procedure Op(X: T);
private
```

because that would violate the information known in the partial view.

But suppose that in fact it turns out that in the private part the type NT is actually derived from TT (itself derived from T) and that TT does have an operation Op.

```
private
  type NT is new TT with ...              -- TT has Op
end P;
```

In such a case it turns out in the end that Op is in fact overriding after all. We can then put an overriding indicator on the body of Op since at that point we do know that it is overriding.

Equally of course we should not specify **not overriding** for Op in the visible part because that might not be true either (since it might be that TT does have Op). However if we did put **not overriding** on the partial view then that would not in itself be an error but would simply constrain the full view not to be overriding and thus ensure that TT does not have Op.

Of course if T itself has Op then we could and indeed should put an overriding indicator in the visible part since we know that to be the truth at that point.

The general rule is not to lie. But the rules are slightly different for **overriding** and **not overriding**. For **overriding** it must not lie at the point concerned. For **not overriding** it must not lie anywhere.

This asymmetry is a bit like presuming the prisoner is innocent until proved guilty. We sometimes start with a view in which an operation appears not to be overriding and then later on we find that it

is overriding after all. But the reverse never happens – we never start with a view in which it is overriding and then later discover that it was not. So the asymmetry is real and justified.

There are other similar but more complex problems with private types concerning implicit declarations where the implicit declaration turns up much later and is overriding but has no physical presence on which to hang the indicator. It was concluded that by far the best approach to these problems was just to say that the overriding indicator is always optional. We cannot expect to find all the bugs in a program through syntax and static semantics; the key goal here is to provide a simple way of finding most of them.

Similar problems arise with generics. As is usual with generics the rules are checked in the generic itself and then rechecked upon instantiation (in this case for uses within both the visible part and private part of the specification). Consider

```
generic
  type GT is tagged private;
package GP is
  type NT is new GT with private;
  overriding                              -- illegal, GT has no Op
  procedure Op(X: NT);
private
```

This has to be illegal because GT has no operation Op. Of course the actual type at instantiation might have Op but the check has to pass both in the generic and in the instantiation.

On the other hand saying **not overriding** is allowed

```
generic
  type GT is tagged private;
package GP is
  type NT is new GT with private;
  not overriding                          -- legal, GT has no Op
  procedure Op(X: NT);
private
```

However, in this case we cannot instantiate GP with a type that does have an operation Op because it would fail when checked on the instantiation. So in a sense this imposes a further contract on the generic. If we do not want to impose this restriction then we must not give an overriding indicator on the procedure Op for NT.

Another situation arises when the generic formal is derived

```
generic
  type GT is new T with private;
package GP is
  type NT is new GT with private;
  overriding                              -- legal if T has Op
  procedure Op(X: NT);
private
```

In this case it might be that the type T does have an operation Op in which case we can give the overriding indicator.

We might also try

```
generic
  type GT is tagged private;
  with procedure Op(X: GT);
```

```
package GP is
  type NT is new GT with private;
  overriding                                   -- illegal, Op not primitive
  procedure Op(X: NT);
private
```

But this is incorrect because although GT has to have an operation corresponding to Op as specified in the formal parameter list, nevertheless it does not have to be a primitive operation nor does it have to be called Op and thus it isn't inherited.

It should also be observed that overriding indicators can be used with untagged types although they have been introduced primarily to avoid problems with dispatching operations. Consider

```
package P is
  type T is private;
  function "+" (Left, Right: T) return T;
private
  type T is range 0 .. 100;                    -- "+" overrides
end P;
```

as opposed to

```
package P is
  type T is private;
  function "+" (Left, Right: T) return T;
private
  type T is (Red, White, Blue);                -- "+" does not override
end P;
```

The point is that the partial view does not reveal whether overriding occurs or not – nor should it since either implementation ought to be acceptable. We should therefore remain silent regarding overriding in the partial view. This is similar to the private extension and generic cases discussed earlier. Inserting **overriding** would be illegal on both examples, while **not overriding** would be allowed only on the second one (which would constrain the implementation as in the previous examples). Again, it is permissible to put an overriding indicator on the body of "+" to indicate whether or not it does override.

It is also possible for a subprogram to be primitive for more than one type (this cannot happen for more than one tagged type but it can happen for untagged types or one tagged type and some untagged types). It could then be overriding for some types and not overriding for others. In such a case it is considered to be overriding as a whole and any indicator should reflect this.

The possibility of having a pragma which would enforce the use of overriding indicators (so that they too could not be inadvertently omitted) was eventually abandoned largely because of the private type and generic problem which made the topic very complicated.

Note the recommended layout, an overriding indicator should be placed on the line before the subprogram specification and aligned with it. This avoids disturbing the layout of the specification.

It is hoped that programmers will use overriding indicators freely. As mentioned in the Introduction, they are very valuable for preventing nasty errors during maintenance. Thus if we add a further parameter to an operation such as Op for a root type and all type extensions have overriding indicators then the compiler will report an error if we do not modify the operators of all the derived types correctly.

We now turn to a minor change in the overriding rules for functions with controlling results.

The reader may recall the general rule in Ada 95 that a function that is a primitive operation of a tagged type and returns a value of the type, must always be overridden when the type is extended. This is because the function for the extended type must create values for the additional components. This rule is sometimes phrased as saying that the function "goes abstract" and so has to be overridden if the extended type is concrete. The irritating thing about the rule in Ada 95 is that it applies even if there are no additional components.

Thus consider a generic version of the set package of Section 3

```
generic
  type Element is private;
package Sets is
  type Set is tagged private;
  function Empty return Set;
  function Unit(E: Element) return Set;
  function Union(S, T: Set) return Set;
  function Intersection(S, T: Set) return Set;
  ...
end Sets;
```

Now suppose we declare an instantiation thus

```
package My_Sets is new Sets(My_Type);
```

This results in the type Set and all its operations being declared inside the package My_Sets. However, for various reasons we might wish to have the type and its operations at the current scope. One reason could just be for simplicity of naming so that we do not have to write My_Sets.Set and My_Sets.Union and so on. (We might be in a regime where use clauses are forbidden.) An obvious approach is to derive our own type locally so that we have

```
package My_Sets is new Sets(My_Type);
type My_Set is new My_Sets.Set with null record;
```

Another situation where we might need to do this is where we wish to use the type Set as the full type for a private type thus

```
  type My_Set is private;
private
  package My_Sets is new Sets(My_Type);
  type My_Set is new My_Sets.Set with null record;
```

But this doesn't work nicely in Ada 95 since all the functions have controlling results and so "go abstract" and therefore have to be overridden with wrappers thus

```
function Union(S, T: My_Set) return My_Set is
begin
  return My_Set(My_Sets.Union(My_Sets.Set(S), My_Sets.Set(T)));
end Union;
```

This is clearly a dreadful nuisance. Ada 2005 sensibly allows the functions to be inherited provided that the extension is visibly null (and that there is no new discriminant part) and so no overriding is required. This new facility will be much appreciated by users of the new container library in Ada 2005 which has just this style of generic packages which export tagged types.

The final topic to be discussed concerns a problem with overloading and untagged types. Remember that the concept of abstract subprograms was introduced into Ada 95 largely for the purpose of tagged types. However it can also be used with untagged types on derivation if we do not want an

operation to be inherited. This often happens with types representing physical measurements. Consider

```
type Length is new Float;
type Area is new Float;
```

These types inherit various undesirable operations such as multiplying a length by a length to give a length when of course we want an area. We can overcome this by overriding them with abstract operations. Thus

```
function "*" (L, R: Length) return Length is abstract;
function "*" (L, R: Area) return Area is abstract;
function "*" (L, R: Length) return Area;
```

We have also declared a function to multiply two lengths to give an area. So now we have two functions multiplying two lengths, one returns a length but is abstract and so can never be called and the other correctly returns an area.

Now suppose we want to print out some values of these types. We might declare a couple of functions delivering a string image thus

```
function Image(L: Length) return String;
function Image(L: Area) return String;
```

And then we decide to write

```
X: Length := 2.5;
...
Put_Line(Image(X * X));                    -- ambiguous in 95
```

This fails to compile in Ada 95 since it is ambiguous because both Image and "*" are overloaded. The problem is that although the function "*" returning a length is abstract it nevertheless is still there and is considered for overload resolution. So we don't know whether we are calling Image on a length or on an area because we don't know which "*" is involved.

So declaring the operation as abstract does not really get rid of the operation at all, it just prevents it from being called but its ghost lives on and is a nuisance.

In Ada 2005 this is overcome by a new rule that says "abstract nondispatching subprograms are ignored during overload resolution". So the abstract "*" is ignored and there is no ambiguity in Ada 2005.

Note that this rule does not apply to dispatching operations of tagged types since we might want to dispatch to a concrete operation of a descendant type. But it does apply to operations of a class-wide type.

References

[1] ISO/IEC JTC1/SC22/WG9 N412 (2002) *Instructions to the Ada Rapporteur Group from SC22/WG9 for Preparation of the Amendment*.

[2] *Ada 95 Rationale* (1995) LNCS 1247, Springer-Verlag.

[3] J. G. P. Barnes (1998) *Programming in Ada 95*, 2nd ed., Addison-Wesley.