Using Inheritance for Heterogeneity

Mark Allen Weiss
School of Computer Science
Florida International University
University Park
Miami, FL 33199

Abstract

We describe the basics of using inheritance under Ada95. An example that illustrates *runtime binding* is provided. Basic knowledge of the language-independent concept of inheritance is assumed. Our goal is to illustrate the Ada95 syntax.

1 Review: Primitive Inheritance: Type Extension

Ada83 provided what could be viewed as a primitive form of inheritance. Consider the type declaration

```
type Position is new Integer;
```

The operations that can be performed on objects of type `Position` are exactly the same as those that could be performed on objects of type `Integer` prior to the declaration of `Position`[1]; in other words, we have inheritance. Furthermore, we can write additional routines that accept arguments of type `Position`, and even override the inherited routines.

This form of inheritance is limited because although we can add functionality to the type, we cannot alter its structure. The derived type has essentially the same structure as the original type. Consider the basic type `Student`. We might want to declare the new type `GradStudent` as

```
type GradStudent is new Student;
```

Already existing operations such as

```
procedure ViewCourses( S: Student );
```

will be meaningfully inherited, and we can add or alter routines specifically for graduate students, but we have no way of incorporating the fact that a graduate student requires extra data: a thesis title, thesis advisor, and an indication of whether the graduate student is on an assistantship. This requires a mechanism known as *type extension*.

2 Inheritance with Type Extension

To implement type extension in Ada95, the original type **must** be a *tagged type*. Taggedness is inherited, so we can continue deriving more types from the derived type. The syntax for tagged types is illustrated by the following:

[1] In other words, if we have a procedure `A` that accepts an `Integer` as a parameter, then procedure `A` is defined with `Position` as a parameter if `A` was declared *prior* to the declaration of `Position`.

```
type Student is tagged
  record
    CompleteName : String80; -- String80 is a fixed length String type
    Address : String80;
    SocialSecurityNum : Natural;
    Courses : Transcript;    -- Transcript is some composite type
  end record;
```

The new type is then derived as a `new` original type `with` additional components. For example,

```
type GradStudent is new Student with
  record
    Advisor : String80;
    ThesisTitle : String80;
    Support : AssistantShipInfo;
  end record;
```

Typically we would want both `Student` and `GradStudent` to be *private* (or limited private) types. This is done in a logical fashion: an incomplete type declaration is provided in the public section, and the complete declarations are placed in the private section. The incomplete declarations would be:

```
type Student is tagged private;
type GradStudent is new Student with private;
```

`GradStudent` now has seven components. In this example, we can deduce that all these components are accessible in the package by applying the standard Ada83 rules.[2]

Sometimes we have the case that a derived type adds no new data field to the base type. As we'll see in the next section, there are cases where we still want to use tagged types. In that case, we can replace

```
  record
    null;   // No additional fields
  end record;
```

with the shorthand

```
with null record;
```

3 Heterogeneous Structures

Perhaps the most interesting application of inheritance is the *heterogeneous data structure*. Here we would want an array (or list, or stack, or queue) of `Students`, but some of the `Student` objects might be graduate students, student-athletes, etc. We want the basic operations on the array to be independent of the exact type of student that is stored.

In general, we can not use an array of objects because the derived objects will be larger than the base objects (since they are extended with additional fields). Thus the general strategy involves the use of pointers to objects. What we generally do is supply an abstract type that can be used to derive all of the possible subtypes, and then work with pointers to the abstract type.

We'll illustrate the issues involved by writing a complete package specification, body, and test program that solves the following problem:

Read up to ten arbitrary shapes (circle, square, or rectangle) and output them sorted by area.

[2] If `Student` were a private (or limited private) type in a different package than the one that `GradStudent` is declared in, then these fields would be inaccessible in the package that defines `GradStudent`.

```
package Shape_Pack is
        -- Abstract class declaration
    type Shape is abstract tagged null record;

        -- The derived types
    type Circle is new Shape with private;
    type Rectangle is new Shape with private;
    type Square is new Rectangle with private;

        -- Abstract operations
    function Area( S: Shape ) return Float is abstract;
    procedure Print_Shape_Type( S: Shape ) is abstract;
    procedure Get( S: out Shape ) is abstract;

        -- Class-wide operations
    function "<"( Left, Right: Shape'class ) return Boolean;
    procedure Put( S: Shape'class );

        -- Circle dispatching operations
    function Area( S: Circle ) return Float;
    procedure Print_Shape_Type( S: Circle );
    procedure Get( S: out Circle );

        -- Rectangle dispatching operations
    function Area( S: Rectangle ) return Float;
    procedure Print_Shape_Type( S: Rectangle );
    procedure Get( S: out Rectangle );

        -- Square dispatching operations
    procedure Print_Shape_Type( S: Square );
    procedure Get( S: out Square );

private
        -- Elaboration of type extensions
    type Circle is new Shape with
      record
        Radius : Float;
      end record;

    type Rectangle is new Shape with
      record
        Length: Float;
        Width:  Float;
      end record;

    type Square is new Rectangle with null record;
end Shape_Pack;
```

Figure 1: Package specification for Shape hierarchy

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;

package body Shape_Pack is
    function "<"( Left, Right: Shape'class ) return Boolean is
    begin
        return Area( Left ) < Area( Right );
    end "<";

    procedure Put( S: Shape'class ) is
    begin
        Print_Shape_Type( S ); Put( " of area " ); Put( Area( S ), exp=>0 );
    end Put;

    function Area( S: Circle ) return Float is
    begin
        return 3.1415927 * S.Radius * S.Radius;
    end Area;

    function Area( S: Rectangle ) return Float is
    begin
        return S.Length * S.Width;
    end Area;

    procedure Print_Shape_Type( S: Circle ) is
    begin
        Put( "Circle" );
    end Print_Shape_Type;

    procedure Print_Shape_Type( S: Rectangle ) is
    begin
        Put( "Rectangle" );
    end Print_Shape_Type;

    procedure Print_Shape_Type( S: Square ) is
    begin
        Put( "Square" );
    end Print_Shape_Type;

    procedure Get( S: out Circle ) is
    begin
        Get( S.Radius );
    end Get;

    procedure Get( S: out Rectangle ) is
    begin
        Get( S.Length ); Get( S.Width );
    end Get;

    procedure Get( S: out Square ) is
        Side : Float;
    begin
        Get( Side ); S.Length := Side; S.Width := Side;
    end Get;
end Shape_Pack;
```


Figure 2: Package body for shape hierarchy

3.1 Abstract Base Classes: The Shape Hierarchy

We begin by examining the specification for our shape hierarchy package, Shape_Pack. The code is shown in Figure 1. The package consists of four types. We have an *abstract* shape as the root of the hierarchy. Because Shape is abstract, we cannot declare any objects of type Shape; we can only declare objects of a derived type. Derived from Shape are both Circle and Rectangle. Square is derived from Rectangle, since a square IS-A rectangle (whose length and width are equal). The private section gives the type extensions for Circle, Rectangle, and Square.

The remainder of the package specification details what can be done to a shape. First we have three abstract operations, Area, Print_Shape_Type, and Get. Because they are abstract, these operations are not defined for objects of type Shape. Since Shape objects cannot exist, this does not seem meaningful. The meaning is that since these operations are not defined, undefined operations are inherited for the derived types. The derived types (if they are not themselves abstract) **must** provide a definition for these operations, or the compiler will complain that no operation is defined for a particular derived type. Only abstract types may have abstract operations. We can see that the package will provide definitions for the abstract operations for each derived type.

You may wonder what happened to the Area operation for type Square. Did we forget to provide this operation? Here the answer is no: the Area provided for Rectangle is inherited.

The only other declarations are the so-called *class-wide operations*. For instance, operator "<" takes two parameters of type Shape and compares them. If the specified type was merely Shape, Left and Right would be matched only by a Shape object (which is clearly impossible, since Shape is abstract). By using Shape'class each parameter is matched by any member of the Shape hierarchy. Because we can conveniently describe the "<" function for any shape (by calling an Area function), we only need to write one "<" function.

The body of the package is shown in Figure 2. As we can see, the class-wide operations Put and "<" are easily implemented in terms of other functions. The abstract operations Area, Print_Shape_Type, and Get must be provided for each derived type. As mentioned before, Area for Square is defined by virtue of inheritance from Rectangle.

3.2 Using The Shape Hierarchy

Now that we have a package that supports the required shapes, we can implement a procedure that reads the shapes, calls a sorting routine, and outputs the shapes. This is shown in Figure 3.

As we mentioned earlier, Shape objects can't exist, so we can't have an array of shapes. Instead we have an array of pointers to shapes. We do this by declaring a new type ShapePtr that is a pointer to a class-wide Shape. This means that it can point at any derived type of Shape. Notice carefully that for this to be true, it is necessary to supply the 'class attribute.

Once we've done that, we can declare the type ShapePtrArray, which is an unconstrainted array type to store the pointers, and we can instantiate a sorting package and declare the array object and a counter. The comparison function is simple: we just compare the dereferenced pointers using the class-wide "<" function defined for shapes.

The function Get_Shape, shown separately in Figure 4 is used to prompt for input, to dynamically allocate the appropriate object, and then to initialize and return a pointer to it.

4 Exercises

These exercises are just some ideas of what could be given as programming assignments.

1. Add triangle to the shape hierarchy. From triangle, specialize an equilateral triangle.
2. Instantiate a linked list with type ShapePtr. Perform some insertions and then print the list.
3. Repeat the previous exercise using a sorted linked list.
4. For each of the following examples implement and test a minimal inheritance hierarchy.
   (a) A student hierarchy with graduate students, student athletes, and students with double majors.

```
with Shape_Pack; use Shape_Pack;
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
with Sorters;

procedure Shape_Test is
    type ShapePtr is access Shape'class;
    type ShapePtrArray is array( natural range <> ) of ShapePtr;
    function "<"( Left, Right: ShapePtr ) return Boolean;

    package Sort is new Sorters( ShapePtr, Integer, ShapePtrArray, "<" );
        use Sort;

    MaxShapes : constant Integer := 10;
    NumberOfShapes : Integer := 0;
    TheShapes : ShapePtrArray( 1..MaxShapes );

    function "<"( Left, Right: ShapePtr ) return Boolean is
    begin
        return Left.all < Right.all;
    end "<";

    function Get_Shape return ShapePtr is separate;

begin
        -- Read the shapes
    begin
        while NumberOfShapes < MaxShapes loop
            TheShapes( NumberOfShapes + 1 ) := Get_Shape;
            NumberOfShapes := NumberOfShapes + 1;
        end loop;
    exception
        when End_Error => Put( "Read only " );
                          Put( NumberOfShapes );
                          Put_Line( " items." );
    end;

        -- Sort the shapes (actually we are sorting the pointers)
    Insertion_Sort( TheShapes( 1..NumberOfShapes ) );

        -- Output the result
    for i in 1..NumberOfShapes loop
        Put( TheShapes( i ).all );
        New_Line;
    end loop;

end Shape_Test;
```

Figure 3: Main procedure for shape example

```
separate( Shape_Test )

    function Get_Shape return ShapePtr is
        Shape_Type : character;
        S : ShapePtr;
    begin
            -- Prompt for and read the type of shape
        Put( "Enter Shape Type: c, r, or s: " );
        Get( Shape_Type );

            -- Allocate a specific shape and prompt for its dimensions
        if Shape_Type = 'c' then
            S := new Circle;
            Put( "Enter radius: " );
        elsif Shape_Type = 'r' then
            S := new Rectangle;
            Put( "Enter length and width: " );
        elsif Shape_Type = 's' then
            S := new Square;
            Put( "Enter side: " );
        else
            raise Data_Error;
        end if;

            -- Get the dimensions and return the pointer to the shape
        Get( S.all );
        return S;

    exception
        when Data_Error =>
            Put_Line( "Bad data!! Skipping a line and retrying" );
            Skip_Line;
            return Get_Shape;

    end Get_Shape;
```

Figure 4: Get_Shape function in Shape_Test

(b) A bank account hierarchy with checking accounts, savings accounts, money market accounts.

(c) A date hierarchy (supporting data arithmetic) using both the standard Roman calendar and some alternate (such as a lunar calendar).

(d) A hierarchy that supports bibliography citations. Types of citations include books, journals, conference papers, and book chapters. See the end of chapter references or examples of how these types of citations differ.

(e) A hierarchy that supports basic sports statistics. Sports include baseball and hockey; for baseball differentiate between pitchers and non-pitchers. For hockey, differentiate between goalies and non-goalies.

5 Acknowledgement

The shape example is taken from Chapter 14 of my textbook *Efficient C Programming, A Practical Approach* (Prentice-Hall, 1995); the idea of using `ShapePtr` is described in Chapter 4 of my textbook *Algorithms, Data Structures, and Problem Solving with C++* (Benjamin/Cummings, 1996).