Safe and Secure Software

An Invitation to **Ada 2005**

5

Safe Object Oriented Programming

Courtesy of AdaCore
The GNAT Pro Company

John Barnes

OOP took programming by storm about twenty years ago. Its supreme merit is said to be its flexibility. But flexibility is somewhat like freedom discussed in the Introduction – the wrong kind of flexibility can be an opportunity that permits dangerous errors to intrude.

The key idea of OOP is that the objects dominate the programming and subprograms (methods) that manipulate objects are properties of objects. The other, older, view sometimes called Function-Oriented (or structured) programming, is that programming is primarily about functional decomposition and that it is the subprograms that dominate program organization, and that objects are merely passive things being manipulated by them.

Both views have their place and fanatical devotion to just a strict object view is often inappropriate.

Ada strikes an excellent balance and enables either approach to be taken according to the needs of the application. Indeed Ada has incorporated the idea of objects right from its inception in 1980 through the concept of packages which encapsulate types and the operations upon them, and tasks that encapsulate independent activities.

Object-Orientation versus Function-Orientation

We will look at two examples which can be used to illustrate various points. They are chosen for their familiarity which avoids the need to explain particular application areas. The examples concern geometrical objects (of which there are lots of kinds) and people (of which there are only two kinds, male and female).

Consider the geometrical objects first. For simplicity we will consider just flat objects in a plane. Every object has a position. In Ada we can declare a root object which has properties common to all objects thus

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

The word **tagged** distinguishes this type from a plain record type (such as Date in Chapter 3) and indicates that it can be extended. Moreover, objects of this type carry a tag with them at execution time and this tag identifies the type of the object. We are going to declare various specific object types such as Circle, Triangle, Square and so on in a moment and these will all have distinct values for the tag.

We can declare various properties of geometrical objects such as area and moment of inertia about the centre. Every object has such properties but they

vary according to shape. These properties can be defined by functions and they are declared in the same package as the corresponding type. We can start with

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

  function Area(Obj: Object) return Float is abstract;
  function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

We have declared the type and the operations as abstract. We don't actually want any objects of type Object and making it abstract prevents us from inadvertently declaring any. We want real objects such as a Circle, which have properties such as Area. If we did want to discuss a plain point without any areas then we should declare a specific type Point for this. The functions Area and Moment have been declared as abstract also. This ensures that when we declare a genuine type such as Circle then we are forced to declare concrete functions Area and Moment with appropriate code.

We can now declare the type Circle. It is best to use a child package for this

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

  function Area(C: Circle) return Float;
  function Moment(C: Circle) return Float;
end;

with Ada.Numerics;  use Ada.Numerics;          -- to give access to π
package body Geometry.Circles is
  function Area(C: Circle) return Float is
  begin
    return π * C.Radius**2;                    -- uses Greek letter π
  end Area;

  function Moment(C: Circle) return Float is
  begin
    return 0.5 * C.Area * C.Radius**2;
  end Moment;
end Geometry.Circles;
```

Note that the code defining the Area and Moment is in the package body. We recall from the chapter on Safe Architecture that this means that the code can be

changed and recompiled as necessary without forcing recompilation of the description of the type itself and consequently all those programs that use it.

We could then declare other types such as Square (which has an extra component giving the length of the side), Triangle (three components giving the three sides) and so on without disturbing the existing abstract type Object and the type Circle in any way.

The various types form a hierarchy rooted at Object and this set of types (a *class* in Ada terminology) is denoted by Object'Class. Ada carefully distinguishes between a specific type such as Circle and a class of types such as Object'Class. This distinction avoids confusion that can occur in other languages. If we subsequently define other types as extensions of the type Circle then we can then usefully talk about the class Circle'Class.

The function Moment declared above illustrates the use of the prefixed notation. We can write either of

```
C.Area                  -- prefixed notation
Area(C)                 -- functional notation
```

The prefixed notation emphasizes the object model, and indicates that we consider the object C to be the predominant entity rather than the function Area.

Suppose now that we have declared various objects, perhaps

```
A_Circle: Circle := (1.0, 2.0, Radius => 4.5);
My_Square: Square := (0.0, 0.0, Side => 3.7);
The_Triangle: Triangle := (1.0, 0.5, A => 3.0, B => 4.0, C => 5.0);
```

By way of illustration, we have used named notation for components other than the *x* and *y* coordinates which are common to all the types.

We might have a procedure to output the properties of a general object. We might write

```
procedure Print(Obj: Object'Class) is
begin
  Put("Area is ");  Put(Obj.Area);          -- dispatching call of Area
  ...                                       -- and so on
end Print;
```

and then

```
Print(A_Circle);
Print(My_Square);
```

The procedure Print can take any item in the class Object'Class. Within the procedure, the call to Area is dynamically bound and calls the function Area appropriate to the specific type of the parameter Obj. This always works safely

since the language rules are such that every possible object in the class Object'Class is of a specific type derived ultimately from Object and will have a function Area. Note that the type Object itself was abstract and so no geometrical object of that type can be declared – accordingly it does not matter that the function Area for the type Object is abstract and has no code – it could never be called anyway.

In a similar way we might have types concerning persons. Consider

```
package People is
  type Person is abstract tagged
    record
      Birthday: Date;
      Height: Inches;
      Weight: Pounds;
    end record;

  type Man is new Person with
    record
      Bearded: Boolean;          -- whether he has a beard
    end record;

  type Woman is new Person with
    record
      Births: Integer;           -- how many children she has borne
    end record;

  ...      -- various operations
end People;
```

Since there is no possibility of any additional types of persons we could describe them by using a variant record, which is more in the line of function-oriented programming. Thus

```
type Gender is (Male, Female);

type Person (Sex: Gender) is
  record
    Birthday: Date;
    Height: Inches;
    Weight: Pounds;
    case Sex is
      when Male =>
        Bearded: Boolean;
      when Female =>
        Births: Integer;
    end case;
  end record;
```

and we might then declare various operations on this version of the type Person. Each operation would have to have a case statement to take account of the two sexes.

This might be considered rather old fashioned and inelegant. However, it has its own considerable advantages.

If we need to add another **operation** in the Object-Oriented formulation then the whole structure will need to be recompiled – each type will need to be revisited in order to implement the new operation. If we need to add another **type** (such as a Pentagon) then the existing structure can be left unchanged.

In the case of the Function-Oriented formulation, the situation is completely reversed (basically we simply interchange the words type and operation).

If we need to add another **type** in the Function-Oriented formulation then the whole structure will need to be recompiled – each operation will need to be revisited to implement the new type (by adding another branch to its case statement). If we need to add another **operation** then the existing structure can be left unchanged.

The Object-Oriented approach has often been lauded as so much safer than Function-Oriented programming because there are no case statements to maintain. This certainly is true but sometimes the maintenance is harder if new operations are added because they have to be added individually for every type.

Ada offers both approaches and both approaches are safe in Ada.

Overriding indicators

One of the dangers of Object-Oriented programming occurs with overriding inherited operations. When we add a new type to a class we can add new versions of all the appropriate operations. If we do not add a new operation then that of the parent is inherited.

The danger is that we might attempt to add a new version but spell it incorrectly

```
function Aera(C: Circle) return Float;
```

or get a parameter or result wrong

```
function Area(C: Circle) return Integer;
```

In both cases the existing function Area is not overridden but a totally new operation added. And then when a class-wide operation dispatches to Area it will call the inherited version rather than the one that failed to override it. Such

bugs can be very difficult to find – the program compiles quietly and seems to run but just produces curious answers.

(Actually, Ada has already provided a safeguard here because we declared Area for Object as abstract and this is a further defensive measure. But if we had a second generation or had not had the wisdom to make Area abstract then we would be in trouble.)

In order to guard against such mistakes we can write for example

```
overriding
function Area(C: Circle) return Float;
```

and then if we make an error we will not get a new operation but instead the program will fail to compile. On the other hand, if we did truly want to add a new operation then we could assert that also by

```
not overriding
function Aera(C: Circle) return Float;
```

Such overriding indicators are always optional, largely for compatibility with earlier versions of Ada.

Languages such as C++ and Java provide less assistance in this area and consequently subtle errors can remain undetected for some time.

Dispatchless programming

In safety-critical programming, the dynamic selection of code is sometimes forbidden. Safety is enhanced if we can prove that the flow of control follows a strict pattern with, for example, no dead code. Traditionally this means that we have to use a more function-oriented approach, with visible if statements and case statements to select the appropriate flow path.

Although dynamic dispatching is at the heart of much of the power of Object-Oriented programming, other object-oriented features (chiefly code reuse through inheritance) are valuable. Thus we might value the ability to extend types and thereby share much coding but declare specific named operations where no dynamic behavior is required. We might also wish to use the prefixed notation which has a number of advantages.

Ada has a facility known as pragma Restrictions which enables a programmer to ensure that specific features of Ada are not used in a particular program. In this case we write

```
pragma Restrictions(No_Dispatch);
```

and this ensures that no use is made of the construction X'Class which in turn means that no dispatching calls are possible.

Note that this exactly matches the requirements of SPARK which we mentioned in the Introduction is often used for critical software. SPARK permits type extension but does not permit class-wide types and operations.

If we do specify the restriction No_Dispatch then the implementation is able to reduce the code overheads typically associated with OOP. There is of course no need to generate a dispatch table for each type. (A dispatch table is a look-up table that contains the addresses of the various specific operations for the type.) Moreover, there is also no need to store a tag in every record structure.

There are other less obvious benefits as well. In full OOP some of the predefined operations such as equality are dispatching and so the code overheads associated with them are also avoided. The net result is that the use of the pragma minimizes the need for the justification of deactivated code (code that is present in the executable and that can be traced back to specific requirements, but which will never be executed) for level A certification.

Interfaces and multiple inheritance

Some have looked upon multiple inheritance as a Holy Grail – an objective against which languages should be judged. This is not the place to digress on the history of various techniques that have been used. Rather we will summarize the key problems.

Suppose that we were able to inherit arbitrarily from two parent types. Recall that fabulous book *Flatland* written by Edwin Abbott (the second edition was published in 1884). It is a satire on class structure (in the sociological, not the programming sense) and concerns a world in which people are flat geometrical objects. The working classes are triangles, the middle classes are other polygons. The aristocracy are circles. Curiously, all females are two-sided and thus simply a line segment.

So using the two classes Objects and Persons introduced above, we could conceive of representing the inhabitants of Flatland by a type derived from both such as

```
type Flatlander is new Geometry.Object and People.Person;
```

The question now arises as to what are the properties inherited from the two parent types? We might expect a Flatlander to have components X_Coord and Y_Coord inherited from Object and also a Birthday inherited from Person, although Height and Weight might be dubious for a two-dimensional person. And certainly we would expect an operation such as Area to be inherited because clearly a Flatlander has an area and indeed a moment of inertia.

But we see potential problems in the general case. Suppose both parent types have an operation with the same identifier. This would typically arise with operations of a rather general nature such as Print, Make, Copy and so on. Which one is inherited? Suppose both parents have components with the same identifier. Which one do we get? These problems particularly arise if both parents themselves have a common ancestor.

Some languages have provided multiple inheritance and devised somewhat lengthy rules to overcome these difficulties (C++ and Eiffel for example). Possibilities include using renaming, mentioning the parent name for ambiguous entities, and giving precedence to the first parent type in the list. Sometimes the solutions have the flavor of unification for its own sake – one person's unification is often another person's confusion. The rules in C++ give plenty of opportunities for the programmer to make mistakes.

The difficulties are basically twofold: inheriting components and inheriting the *implementation* of operations from more than one parent. But there is generally no problem with inheriting the *specification* of operations. This solution was adopted by Java and has proved successful and is also the approach used by Ada.

So the Ada rule is that we can inherit from more than one type thus

```
type T is new A and B and C with
  record
    ...             -- additional components
  end record;
```

but only the first type in the list (A) can have components and concrete operations. The other types must be what are known as *interfaces* which are essentially abstract types without components and all of whose operations are abstract or null procedures. (The first type could be an interface as well.)

We can reformulate the type Object as an interface as follows

```
package Geometry is
  type Object is interface;

  procedure Move(Obj: in out Object;
                 New_X, New_Y: in Float) is abstract;
  function X_Coord(Obj: Object) return Float is abstract;
  function Y_Coord(Obj: Object) return Float is abstract;
  function Area(Obj: Object) return Float is abstract;
  function Moment(Obj: Object) return Float is abstract;
end Geometry;
```

Observe that the components have been deleted and replaced by further operations. The procedure Move enables an object to be moved – that is it sets

both the *x* and *y* coordinates and the functions X_Coord and Y_Coord return its current position.

Note that the prefixed notation means that we can still access the coordinates by for example A_Circle.X_Coord and The_Triangle.Y_Coord just as when they were visible components.

So now when we declare a concrete type Circle we have to provide implementations of all these operations. Perhaps

```
package Geometry.Circles is
  type Circle is new Object with private;                    -- partial view

  procedure Move(C: in out Circle; New_X, New_Y: in Float);
  function X_Coord(C: Circle) return Float;
  function Y_Coord(C: Circle) return Float;
  function Area(C: Circle) return Float;
  function Moment(C: Circle) return Float;

  function Radius(C: Circle) return Float;
  function Make_Circle(X, Y, R: Float) return Circle;

private
  type Circle is new Object with                             -- full view
    record
      X_Coord, Y_Coord: Float;
      Radius: Float;
    end record;
end Geometry.Circles;

package body Geometry.Circles is
  procedure Move(C: in out Circle; New_X, New_Y: in Float) is
  begin
    C.X_Coord := New_X;
    C.Y_Coord := New_Y;
  end Move;

  function X_Coord(C: Circle) return Float is
  begin
    return C.X_Coord;
  end X_Coord;

  -- and similarly Y_Coord and Area and Moment as before
  -- also functions Radius and Make_Circle
end Geometry.Circles;
```

We have made the type Circle private so that all the components are hidden. Nevertheless the partial view reveals that it is derived from the type Object and

so must have all the properties of the type Object. Note how we also add functions to create a circle and to access the radius component.

So the essence of programming with interfaces is that we have to implement the properties promised. It is not so much multiple inheritance of existing properties but multiple inheritance of contracts to be satisfied.

Returning now to Flatland, we can declare

```
package Flatland is
  type Flatlander is abstract new Person and Object with private;

  procedure Move(F: in out Flatlander; New_X, New_Y: in Float);
  function X_Coord(F: Flatlander) return Float;
  function Y_Coord(F: Flatlander) return Float;

private
  type Flatlander is abstract new Person and Object with
    record
      X_Coord, Y_Coord: Float := 0.0;          -- at origin by default
      ... -- any new components we wish
    end record;
end;
```

and the type Flatlander will inherit the components Birthday etc of the type Person, any operations of the type Person (we didn't show any above) and the abstract operations of the type Object. However, it is convenient to declare the coordinates as components since we need to do that eventually and we can then override the inherited abstract operations Move, X_Coord and Y_Coord with concrete ones. Note also that we have given the coordinates the default value of zero so that any flatlander is by default at the origin.

The package body is

```
package body Flatland is
  procedure Move(F: in out Flatlander; New_X, New_Y: Float) is
  begin
    F.X_Coord := New_X;
    F.Y_Coord := New_Y;
  end Move;

  function X_Coord(F: Flatlander) return Float is
  begin
    return F.X_Coord;
  end X_Coord;

  -- and similarly Y_Coord
end Flatland;
```

Making Flatlander abstract means that we do not have to implement all the operations such as Area just yet. And finally we could declare a type Square suitable for Flatland (when originally written the book was published anonymously and the author designated as A Square) as follows

```
package Flatland.Squares is
  type Square is new Flatlander with
    record
      Side: Float;
    end record;

  function Area(S: Square) return Float;
  function Moment(S: Square) return Float;
end Flatland.Squares;

package body Flatland.Squares is

  function Area(S: Square) is
  begin
    return S.Side**2;
  end Area;

  function Moment(S: Square) is
  begin
    return S.Area * S.Side**2 / 6.0;
  end Moment;

end Flatland.Squares.
```

and all the operations are thereby implemented. By way of illustration we have made the extra component Side of the type Square directly visible but we could have used a private type. So we can now declare Dr Abbott as

```
A_Square: Square := (Flatlander with Side => 3.00);
```

and he will have all the properties of a square and a person. Note the extension aggregate which takes the default values for the private components and gives the additional visible component explicitly.

There are other important properties of interfaces that can only be touched upon in this overview. An interface can have a null procedure as an operation. A null procedure behaves as if it has a null body – that is, it can be called but does nothing. If two ancestors have the same operation then a null procedure overrides an abstract operation with the same parameters and results. If two ancestors have the same abstract operation with equivalent parameters and results then these merge into a single operation to be implemented. If the parameters and results are different then this results in overloading and both operations have to be implemented. In summary the rules are designed to minimize surprises and maximize the benefits of multiple inheritance.

North American Headquarters
104 Fifth Avenue, 15th floor
New York, NY 10011-6901, USA
tel +1 212 620 7300
fax +1 212 807 0162
sales@adacore.com
www.adacore.com

European Headquarters
46 rue d'Amsterdam
75009 Paris, France
tel +33 1 49 70 67 16
fax +33 1 49 70 05 52
sales@adacore.com
www.adacore.com