Ada's Design Goals and Object–Oriented Programming

Jeffrey R. Carter
Boeing Information Services
7990 Boeing Court
Vienna, VA 22182–3999
(703) 827–2522
72030.677@compuserve.com

Abstract

Ada was designed to support and enforce software–engineering principles to promote the development of programs which are easy to read and understand. Ada 9X, the proposed revision to Ada 83, includes features for object–oriented programming, including inheritance and dispatching. Object–oriented programming is shown to emphasize ease of writing over readability and understandability, violating one of Ada's main design goals. A significant example demonstrates that object–oriented programming reduces readability and understandability.

Introduction

Ada was designed to support and enforce software–engineering principles to promote the development of programs which are easy to read and understand. The Ada Reference Manual for Ada 83 (ARM 83) explicitly states, "The need for langauges that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing" [1, 1.3(2)]. The draft Ada Reference Manual for Ada 9X (ARM 9X) repeats this design goal [2].

Ada 9X, the proposed revision to Ada 83, includes features for object–oriented programming (OOP), including inheritance (called type extension) and automatic run–time overload resolution (dispatching). This paper examines the effects of using object–oriented programming and the exent to which they achieve Ada's design goals.

Object–Oriented Programming

Ada 9X implements inheritance by means of the Ada–83 derived–type mechanism. Many experienced Ada users avoid derived types because they decrease readability and understandability: If the reader is not familiar with the parent type, the derived–type declaration does not provide any information about the type. To understand the new type, the reader must seek out the parent type declaration. This violates the software–engineering principle of locality, reducing readability and understandability. The use of derived types is frequently limited to situations in which the parent and derived type declarations are located near each other, such as when a change of representation is needed [3].

Ada 9X introduces tagged types; the full definition of a tagged type must be a record type declaration. A type derived from a tagged type may extend the type by adding additional record components. This implements inheritance, which the ARM 9X refers to as *type extension* [2]. A type derived from a tagged type is also a tagged type.

A tagged type T and all types derived from it define a type class. An operation on T may be called with an actual parameter of any type in the class, and will perform run–time overload resolution to ensure that the correct operation for the actual parameter is called. The ARM 9X refers to this automatic run–time overload resolution as *dispatching*.

The advantages of OOP are dispatching and inheriting the parent type's operations. In their absence, the developer must explicitly write operations for the new type, and write explicit dispatching operations for a classwide type to obtain run–time overload resolution. Since OOP requires less code to be written, we can see that it makes programs easier to write.

The disadvantage of OOP is the same as for derived types in Ada 83: reduced readability and understandability. With a derived type in Ada 83, there is, at least, a context clause which guides the reader, directly or indirectly, to the type's definition and operations. With OOP, as we will see, there may be no such explicit reference to the actual operation being invoked. OOP emphasizes ease of writing over readability and understandability. This is the opposite of Ada's explicit design goal.

The Alert Problem

Barnes presents the Alert Problem [4] (this paper also appears as an Ada–9X Project publication [5] and as Part I of the Ada–9X Rationale [6]). The Alert Problem is intended to show the advantages of OOP in producing readable, understandable code. Barnes' Ada–9X solution to the problem *is* better than his Ada–83 solution, but this is because his Ada–9X solution has a better design than his Ada–83 solution. We will see that an object–based design, implemented in Ada 83, is more readable than Barnes' object–oriented Ada–9X solution.

Barnes describes the Alert Problem as, "Our system represents the processing of alerts (alarms) in a ground mission control station. Alerts are of three levels of priority. Low level alerts are merely logged, medium level alerts cause a person to be assigned to deal with the problem and high level alerts cause an alarm bell to ring if the matter is not dealt with by a specified time. In addition, a message is displayed on various devices according to its priority" [4]. First, we will review an Ada 9X solution:

```
package Alert_System is
   type Alert_Handle is abstract tagged null record;

   procedure Handle (Alert : in out Alert_Handle) is abstract;
end Alert_System;

with Alert_System, Calendar;
package Low_Alert_System is
   type Low_Alert_Handle is new Alert_System.Alert_Handle with record
      Time    : Calendar.Time;
      Message : String (1 .. 80) := String'(1 .. 80 => ' ');
   end record;

   procedure Handle (Alert : in out Low_Alert_Handle);
end Low_Alert_System;

with Low_Alert_System;
package Medium_Alert_System is
   type Medium_Alert_Handle is new Low_Alert_System.Low_Alert_Handle
   with record
      Officer : String (1 .. 80) := String'(1 .. 80 => ' ');
   end record;

   procedure Handle (Alert : in out Medium_Alert_Handle);
end Medium_Alert_System;

with Medium_Alert_System, Calendar;
package High_Alert_System is
   type High_Alert_Handle is new Medium_Alert_System.Medium_Alert_Handle
   with record
```

```
      Alarm_Time : Calendar.Time;
   end record;

   procedure Handle (Alert : in out High_Alert_Handle);
end High_Alert_System;
```

OOP claims the additional advantage that a new alert type may be added without requiring recompilation of any units which "with" `Alert_System`:

```
with Alert_System;
package Emergency_Alert_System is
   type Emergency_Alert_Handle is new Alert_System.Alert_Handle with ...

   ...
end Emergency_Alert_System;
```

Consider the use of `Alert_System`. The ground–mission control system's main subprogram "withs" `Alert_System` and contains a call to `Alert_System.Handle`:

```
with Alert_System;
with Input;
procedure Ground_Mission_Control is
   type Alert_Ptr is access Alert_System.Alert_Handle'Class;

   Alert : Alert_Ptr;
   ...
begin -- Ground_Mission_Control
   ...
   Alert := Input.Get;
   ...
   Alert_System.Handle (Alert => Alert.all);
   ...
end Ground_Mission_Control;
```

Suppose that, after some time in service, it is necessary to modify the handling of high–priority alerts. None of the original developers of the system are available, so a developer with no experience of the system is assigned the modification. He quickly determines that the call to `Alert_System.Handle` is where alerts are handled, and directs his attention to package `Alert_System`. Here the trail runs dry, however, for this package provides him with no information about where to look next. Luck may play a significant role in determining how long it takes to find package `High_Alert_System`.

`Input.Get` is where the system detects alerts. Since there is no need to modify how this is done, we may assume that the developer does not examine `Input`. How `Get` works is not relevant to this discussion.

The apparent attractiveness of this approach is created by presenting a small amount of code in one place. We can easily read and understand the code presented above. In reality, the system is large, and each of the package specifications given above is in a separate file. The system may have hundreds of packages and millions of statements. OOP does not provide the explicit references needed to understand the alert–handling part of the system.

Ignoring Barnes' ad hoc Ada–83 implementation, consider an object–based design. Being object based, the design is similar to the object–oriented design of the Ada–9X version given above, but does not rely on inheritance and dispatching. We can implement an object–based design in Ada 83:

```
with Calendar;
package Low_Alert_System is
```

```
   type Low_Alert_Handle is record
      Time    : Calendar.Time;
      Message : String (1 .. 80) := String'(1 .. 80 => ' ');
   end record;

   procedure Handle (Alert : in out Low_Alert_Handle);
end Low_Alert_System;

with Low_Alert_System;
package Medium_Alert_System is
   type Medium_Alert_Handle is record
      Low_Alert : Low_Alert_System.Low_Alert_Handle;
      Officer   : String (1 .. 80) := String'(1 .. 80 => ' ');
   end record;

   procedure Handle (Alert : in out Medium_Alert_Handle);
end Medium_Alert_System;

with Medium_Alert_System, Calendar;
package High_Alert_System is
   type High_Alert_Handle is
      Medium_Alert : Medium_Alert_System.Medium_Alert_Handle;
      Alarm_Time   : Calendar.Time;
   end record;

   procedure Handle (Alert : in out High_Alert_Handle);
end High_Alert_System;

with Low_Alert_System, Medium_Alert_System, High_Alert_System;
package Alert_System is
   type Priority_Id is (Low, Medium, High);

   type Alert_Handle (Priority : Priority_Id) is record
      case Priority is
      when Low =>
         Low_Alert : Low_Alert_System.Low_Alert_Handle;
      when Medium =>
         Medium_Alert : Medium_Alert_System.Medium_Alert_Handle;
      when High =>
         High_Alert : High_Alert_System.High_Alert_Handle;
      end case;
   end record;

   procedure Handle (Alert : in out Alert_Handle);
end Alert_System;
```

This technique, known as *composition*, is the opposite of the OOP approach: We create specific types first, and the classwide type last. As OOP devotees will point out, this involves writing more code, using case statements, and changing the specification of `Alert_System` to add a new alert type.

The main program is very similar to the 9X solution:

```
with Alert_System;
with Input;
procedure Ground_Mission_Control is
   type Alert_Ptr is access Alert_System.Alert_Handle;
```

```
      Alert : Alert_Ptr;
      ...
   begin -- Ground_Mission_Control
      ...
      Alert := Input.Get;
      ...
      Alert_System.Handle (Alert => Alert.all);
      ...
   end Ground_Mission_Control;
```

As before, the modifier would find the call to `Alert_System.Handle`, which would guide him to package `Alert_System`. In this case, however, `Alert_System`'s context clause guides him to package `High_Alert_System`. The existence of `High_Alert_System` in `Alert_System`'s context clause is the explicit reference which makes this version easier to read and understand than the OOP version.

Barnes' example notwithstanding, given equally good designs, OOP produces a less readable system. This is because OOP emphasizes ease of writing over readability and understandability, violating Ada's explicit design goal. The composition technique used to implement the Ada–83 solution includes explicit references missing from the OOP solution. This technique requires writing more code, and performing more recompilation when the abstraction changes, but in return significantly increases readability and understandability.

Conclusion

Object–oriented programming, including inheritance and dispatching, is currently the most popular software–development concept. Including it in Ada 9X is probably good marketing strategy. However, as Ada devotees, we championed Ada when C was the most popular language, and object–based development techniques when structured techniques were the most popular development techniques, because we believed object–based Ada was better from the total–system–cost perspective. With 80% of a system's cost coming after delivery, features, such as OOP, which violate Ada's explicit design goal of emphasizing ease of reading and understanding over ease of writing are technically indefensible.

References

1. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL–STD–1815A, 1983
2. Ada 9X Mapping/Revision Team, *Ada 9X Reference Manual*, Draft Version 5.0, Intermetrics, Inc., 1994
3. Cohen, N., *Ada as a Second Language*, McGraw–Hill, 1986
4. Barnes, J., "Introducing Ada 9X," *Ada Letters*, 1993 Nov/Dec
5. Barnes, J., *Introducing Ada 9X*, Ada 9X Project Report, Ada 9X Project Office, 1993
6. Ada 9X Mapping/Revision Team, *Ada 9X Rationale*, Draft Version 5.0, Intermetrics, Inc., 1994