OOP vs. Readability

Jeffrey R. Carter
PragmAda Software Engineering
1540 Coat Ridge Road
Herndon, VA 22070-2728
72030.677@compuserve.com

Introduction

A previous paper discussed the relationship between the object-oriented-programming (OOP) feature known in Ada 95 as dispatching and Ada's expressed design goal of emphasizing ease of reading and understanding over ease of writing [1]. A comparison of Barnes' alert-system example [2] using Ada-95 dispatching and using Ada-83 composition showed that dispatching emphasized ease of writing over ease of reading, while composition emphasized ease of reading over ease of writing. This led to the conclusion that the goals of dispatching conflict with Ada's expressed design goals.

Dispatching is half of OOP; the other half is inheritance, correctly called type extension in Ada 95. This paper examines the relationship between inheritance and Ada's expressed design goal of emphasizing ease of reading and understanding over ease of writing.

Object-oriented design (OOD), as introduced by Booch [3], emphasizes the software-engineering principles of locality, encapsulation, and information hiding. Ada 83 supports such OOD well; using OOD and Ada results in easy-to-read programs that exhibit low coupling and high cohesion. OOD is independent of OOP: Fully object-oriented designs may be implemented using composition methods familiar to Ada-83 users without any need for OOP features (inheritance and dispatching). OOP features may likewise be used without OOD.

Alert System

Let's start with the alert system used in the discussion of dispatching. Consider the package that defines high-level alerts:

```
with Medium_Alert_System;
with 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;
```

`High_Alert_Handle` is defined using inheritance. If inheritance supports Ada's expressed design goals, then this should be easy to read and understand, in which case it should be easy to answer such questions as: What are the visible components of the type? What are the visible operations on the type?

In fact, we cannot answer such questions without examining the type's parent type, `Medium_Alert_Handle`:

```
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;
```

We still can't answer our basic questions about `High_Alert_Handle`. Continuing our search, we examine `Low_Alert_System`:

```
with Alert_System;
with 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;
```

Finally, we examine `Alert_System`:

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

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

Now we know that `High_Alert_Handle` has four visible components, `Time`, `Message`, `Officer`, and `Alarm_Time`; and a single visible operation, `Handle`. However, we had to look at four packages to determine this. This is a clear violation of the software-engineering principle of locality, which holds that one should be able to obtain all the information about something in one place. Locality is important in achieving low coupling. While it is possible to violate the principle of locality without using inheritance, it is impossible to use inheritance effectively without violating locality.

Inheritance is similar to Ada 83's derived-type mechanism, and Ada 95 implements type extension as a variation of type derivation. Unless the parent type is a predefined type, or defined in the same declarative region, derived types have the same failing as inheritance, which led many Ada users to limit the use of derived types to situations where the parent type meets these criteria. Such limitations could also be applied to inheritance.

Bank-Account Example

We can see the same problem with a variation of Balfour's bank-account example [4]. To understand

```
with Account_Handler;
with Checking_Handler;
package Checking_With_Interest_Handler is
   type Interest_Value is delta 0.00001 digits 8 range 0.0 .. 1.0;

   type Checking_With_Interest_Account is new
      Checking_Handler.Checking_Account
   with record
      Rate : Interest_Value := 0.0;
   end record;

   procedure Open (Account : in out Checking_With_Interest_Account;
                   Amount  : in     Account_Handler.Money_Value;
                   Fee     : in     Account_Handler.Money_Value;
                   Rate    : in     Interest_Value);

   procedure Earn_Interest
      (Account : in out Checking_With_Interest_Account);
end Checking_With_Interest_Handler;
```

we have to look at

```
with Account_Handler;
package Checking_Handler is
   type Checking_Account is new Account_Handler.Bank_Account with record
      Fee : Account_Handler.Money_Value := 0.0;
   end record;

   procedure Open (Account : in out Checking_Account;
                   Amount  : in     Account_Handler.Money_Value;
                   Fee     : in     Account_Handler.Money_Value);

   procedure Withdraw (Account : in out Checking_Account;
                       Amount  : in     Account_Handler.Money_Value);
end Checking_Handler;
```

and

```
package Account_Handler is
   type Money_Value is delta 0.01 digits 13 range
      -100_000_000.00 .. 100_000_000.00;

   type Bank_Account is tagged record
      Open    : Boolean     := False;
      Balance : Money_Value := 0.0;
   end record;

   procedure Open (Account : in out Bank_Account; Amount : in Money_Value);

   procedure Deposit (Account : in out Bank_Account;
                      Amount  : in     Money_Value);

   procedure Withdraw (Account : in out Bank_Account;
                       Amount  : in     Money_Value);

   function Balance (Account : Bank_Account) return Money_Value;
end Account_Handler;
```

Only now do we know that `Checking_With_Interest_Account` has four visible components, `Open`, `Balance`, `Fee`, and `Rate`; and seven visible operations, three named `Open`, and one each named `Deposit`, `Withdraw`, `Balance`, and `Earn_Interest`.

Conclusion

OOD emphasizes the software-engineering principles of locality, encapsulation, and information hiding. Using OOD with composition results in easy-to-read programs that exhibit low coupling and high cohesion. It is curious that OOP does not support OOD.

Inheritance gains us something: We had to write less code using inheritance than we would have to write to implement these examples using composition, so using inheritance makes code easier to write. The price for this benefit is reduced readability. Using composition gains us something: The result is easy to read and understand. The price for this benefit is that we write more code.

There are always tradeoffs in language design. Ada's expressed design goal, of emphasizing ease of reading and understanding over ease of writing, means such tradeoffs should be decided in favor of ease of reading against ease of writing. This was clearly not done in the case of inheritance. The goals of inheritance and dispatching, and so of OOP, conflict with Ada's expressed design goals.

References

1. Carter, J., "Ada's Design Goals and Object-Oriented Programming," *Ada Letters*, 1995 Jan/Feb
2. *Ada 95 Rationale*, Intermetrics, Inc., 1995
3. Booch, G., *Software Engineering with Ada*, Benjamin/Cummings, 1983
4. Balfour, B., "Ada 9X: An Object Oriented View," Washington Ada Symposium tutorial, 1994