Ada 9X : From Abstraction-Oriented to Object-Oriented

S. Tucker Taft
Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138
stt@inmet.com

Abstract

Ada 83 is an abstraction-oriented programming language. It supports the definition of abstract data types in modules called "packages," with a separate interface and implementation. The next revision of the language is now being prepared, and is designated Ada 9X. This revision will support full object-oriented programming. As part of designing the object-oriented features of Ada 9X, we had a choice whether to construct a conventional but essentially independent object-oriented capability in addition to the existing abstraction-oriented features of Ada 83, or to make an effort to integrate the existing abstraction support with the minimal set of additional capabilities necessary to support object-oriented programming. We have chosen the latter approach, and as part of this effort, we have tried to isolate exactly what capabilities distinguish a full object-oriented programming language from an abstraction-oriented programming language like Ada 83. Our conclusion is that the essential new capability of object-oriented programming is that a given abstraction may effectively have multiple implementations. The objects carry sufficient extra information at run-time to identify to which implementation they correspond. In Ada 9X, we call this run-time type identification the "tag" of the object, and the corresponding new language capability is called "tagged types."

Introduction

Ada 83[8] is an abstraction-oriented programming language. By this we mean it is a language that supports the definition of abstractions, each with a well-defined interface separated from its implementation. An abstraction (defined by a "package" in Ada) generally consists of one or more abstract data types (called "private types" in Ada), with corresponding operations for constructing, updating, and querying instances of the type. The implementations of the operations can be changed without disturbing the clients of the abstraction. The abstraction may or may not contain global state over and above the state represented within instances of the abstract data types. It if does, this global state is also generally managed internally, with operations provided for updating and querying the state, as appropriate.

Ada is now undergoing a revision, as part of the normal ANSI and ISO processes for periodically updating language standards. The revised language is currently designated Ada 9X[5], with balloting on the revised standard planned to start in late 1993.

Adding Object-Orientation

As part of this revision we are updating the language to be a true object-oriented language. This has turned out to be a challenging task, because many of the capabilities normally associated with object-orientation are already present in Ada 83, and rather than reinventing these capabilities in a different guise as part of a separate object-oriented extension to the language, we have chosen to enhance the existing language mechanisms. As might be expected, this poses a difficult integration problem, as we try to minimize redundancy while providing full object-oriented functionality in a seamless whole.

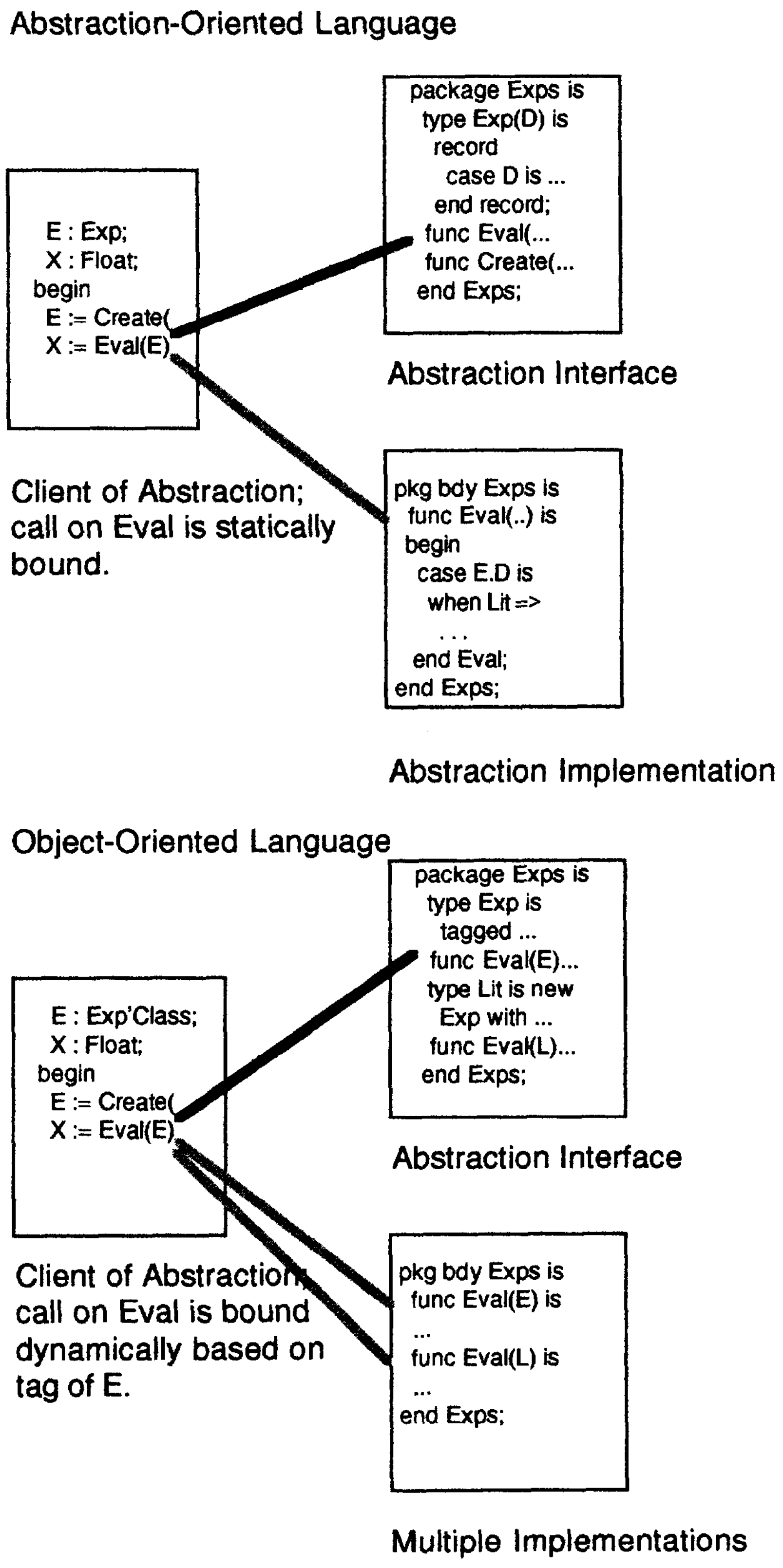
Other efforts to add object-oriented capabilities to existing programming languages have generally started from languages without built-in support for user-defined abstract data types. The original object-oriented language, Simula-67[1], was an outgrowth of Algol 60, which did not even support the definition of record types. It was the nested block structure of Algol 60 that provided the starting point for the class construct of Simula-67. Similarly, C++[7] was developed as an extension of C, which provides user-defined data structures (structs), but no built-in mechanism for data abstraction. With these languages, both abstraction and full object-orientation were incorporated in a single set of new features, thereby blurring the distinction between abstraction-orientation and object-orientation.

In fact, many of the benefits associated with object-orientation are due to its support for abstractions. If a language already supports abstractions, like Ada 83, extending the language to support object-orientation is a quite different process. One could of course ignore the existing facilities, and build an independent object-oriented "corner" of the language. But the net result would be a bigger, more complex, and less usable language. A user would have to make an explicit shift from using the preexisting abstraction facilities, to using the new object-oriented facilities.

For Ada 9X, we have taken a different approach. Rather than adding a new "class construct" or "object type" to the language, we have chosen to directly enhance the existing record type, private type, and "derived" type capabilities. Ada 83 is unusual in being an abstraction-oriented language that already provides a limited form of inheritance, where a new abstract data type may be defined in terms of a preexisting abstract data type. However, the facility is quite limited, because a "derived" data type is limited to having exactly the same set of internal data components as its "parent" data type. Only the operations of the type may be extended or overridden as part of deriving from the type. Furthermore, the parent type and the derived type are totally distinct types -- there are no operations other than explicit conversion that can treat the different derivatives of a given parent type equivalently. Essentially these different derivatives represent independent abstractions, even though they happen to have a common structure and a common set of operations.

If you compare this relatively primitive kind of inheritance with a true object-oriented capability, the critical difference is that with object-orientation, one can take advantage of the common structure and operations of types related by inheritance, and treat them as simply different implementations of the same abstraction, rather than as separate abstractions. Or to put it another way: in an abstraction-oriented languages, a given abstraction interface has exactly one implementation, thereby allowing full static binding between clients of the interface and this implementation; with object-orientation, a given interface may have many implementations, and each object carries around sufficient information (a run-time "tag" of some sort) to identify its implementation, allowing a dynamic binding between clients and the appropriate implementation of any given operation of the interface. This distinction between the static binding of abstraction-oriented languages and the dynamic binding of object-oriented languages is illustrated graphically in Figure 1.

It is worth noting that in the computer science literature on data abstractions, the possibility of multiple implementations of a given abstraction has always been recognized[3,4]. However, when abstraction facilities were incorporated into conventional compiled languages, a single


Abstraction-Oriented Language
Abstraction Interface
Client of Abstraction; call on Eval is statically bound.
Abstraction Implementation
Object-Oriented Language
Abstraction Interface
Client of Abstraction; call on Eval is bound dynamically based on tag of E.
Multiple Implementations

```
  E : Exp;
  X : Float;
begin
  E := Create(
  X := Eval(E)
```

```
package Exps is
 type Exp(D) is
  record
   case D is ...
  end record;
 func Eval(...
 func Create(...
end Exps;
```

```
pkg bdy Exps is
 func Eval(..) is
 begin
  case E.D is
   when Lit =>
    . . .
  end Eval;
end Exps;
```

```
  E : Exp'Class;
  X : Float;
begin
  E := Create(
  X := Eval(E)
```

```
package Exps is
 type Exp is
  tagged ...
  func Eval(E)...
 type Lit is new
  Exp with ...
  func Eval(L)...
 end Exps;
```

```
pkg bdy Exps is
 func Eval(E) is
 ...
 func Eval(L) is
 ...
end Exps;
```

Figure 1

implementation per interface was typically adopted for pragmatic reasons[2]. This is illustrated by the languages CLU[6], Ada 83, and Modula[9]. It was really C++ that was the first main-stream systems programming language that recognized that the dynamic binding inherent in having objects identify their own implementation could be provided while preserving high performance. In fact, once it was recognized that this "dynamic binding" was essentially equivalent to the explicit case analysis performed on variant records, it became clear that an object-oriented approach to managing related variants of the same abstraction could actually be more efficient than the traditional methods.

In retrospect, it seems clear that variant records with the attendant explicit case analysis represented a somewhat awkward way-station on the path between the simple numeric-array-oriented processing of the early Fortran days, and the complex information-oriented processing of today. In a similar vein, the very strong type checking inherent in abstraction-oriented languages like Ada 83, which developed somewhat as a reaction to the very weak type checking of the early variants of Fortran, now appears as an overreaction. One needs the flexibility of treating related types all as a single type when desired, presuming they have some degree of commonality in their interface. Although a variant record can be used for this purpose, it rapidly becomes unwieldy as the number of variants grows large, and adding a new variant becomes a maintenance nightmare. Having a separate type for each variant is a much more natural, open-ended solution. But once you go this route, you need some way to bring the distinct types back together for that processing which can be common across the types. This inevitably requires that objects of such types carry some kind of tag to allow a predominantly common algorithm to dispatch back to type-specific processing when appropriate.

With this as a backdrop, it now becomes relatively clear what capabilities are needed to go from an abstraction-oriented language like Ada 83 to a full object-oriented language. First of all, we must enhance the ability to derive one type from another, by allowing not only the extending or overriding of operations, but also the extension of the type with additional data components. Secondly, we need a way to refer to a set of related types when defining the type of a parameter to an operation, or the type designated by a pointer (called an "access value" in Ada). Finally, we need a way for objects to identify their "underlying" type, so that when all we know statically (at compile-time) is what set of types ("class of types") it belongs to, at run-time we can determine the specific type of the object so as to dispatch to the particular type-specific implementation of a given primitive operation of the abstract data type.

The Ada 9X Approach

In Ada 9X, these capabilities are all provided by a modest enhancement of the existing record, private, and derived type facilities. Record and private types can be identified as "tagged" types. This indicates that their objects will be self-identifying, allowing tagged types related by derivation to be grouped into "classes" for parameter passing and pointer manipulation, while preserving the ability to dispatch on the specific type of an object in such a class at a later point. When deriving from a tagged type, one can add components, either privately (a private extension), or publicly (a record extension). Private types can have public extensions, and vice-versa, thereby allowing a single type with some public and some private components.

Ada 9X Example

These capabilities are best illustrated by example:

```
package Expressions is
    -- This package defines a class of types
    -- used to represent an arithmetic
    -- expression, as might be used
    -- by a simple desk calculator program.
```

```
type Expression is tagged null record;
    -- This is the root of the class of
    -- types that will represent
    -- expressions
function Evaluate(E : Expression)
  return Float is <>;
    -- This is a primitive operation of
    -- the type; no implementation will
    -- be provided for the root type (it
    -- is called an "abstract" operation).

type Literal is new Expression with
    -- This is a simple, visible extension
    -- of the root Expression type,
    -- used to represent the "leaves" of
    -- an expression tree.
    record
        Value : Float;
    end record;
function Evaluate(E : Literal)
  return Float;
    -- The inherited Evaluate function is
    -- overridden; the implementation
    -- will appear in the package body.

type Expr_Ptr is
  access Expression'Class;
    -- This is a pointer type; its values
    -- can point to objects of any type
    -- in the class rooted at Expression.
    -- This class is represented by the
    -- "class-wide" type
    -- Expression'Class.

type Binary_Operator is
  new Expression with private;
    -- This type is the root of a subclass
    -- of types, one for each
    -- binary operator.
function Evaluate(Bop : Binary_Operator)
  return Float is <>;
function Create_Binary_Op(
    Op_Name : String;
    Left_Opnd, Right_Opnd : Expr_Ptr)
  return Expr_Ptr;
        -- This creates an instance of an
        -- appropriate derivative of
        -- Binary_Operator, determined
        -- by the operator name.

type Unary_Operator is
  new Expression with private;
    -- This is the root of the subclass of
    -- types used to represent
    -- invocations of unary operators
function Evaluate(Uop : Unary_Operator)
  return Float is <>;
function Create_Unary_Op(
    Op_Name : String;
    Right_Opnd : Expr_Ptr)
  return Expr_Ptr;
        -- This creates an instance of an
        -- appropriate derivative of
        -- Unary_Operator, determined
        -- by the operator name.

Invalid_Operator_Name : exception;
    -- This exception is raised if the
    -- operator name is unrecognized,
    -- or requires a different number of
    -- operands than are provided
    -- by the Create routine.
private
  -- These full type declarations are
  -- hidden from clients of this package
  type Binary_Operator is
    new Expression with
      -- The operator name will be inherent
      -- in each particular derivative
      record
          Left_Opnd : Expr_Ptr;
          Right_Opnd : Expr_Ptr;
      end record;

  type Unary_Operator is
    new Expression with
      -- The operator name will be inherent
      -- in each particular derivative
      record
          Right_Opnd :  Expr_Ptr;
      end record;
end Expressions;
```

Types and Classes

The above package represents the interface to the Expression class of types. It is worthwhile noting at this point certain unique characteristics of the Ada 9X object-oriented facilities. Because we are building on a language that is very strongly type-checked, we choose to retain an explicit distinction between a "specific" type like Expression, Literal, Binary_Operator, or Unary_Operator, and the set of types ("class" of types) rooted at such a type. This distinction is blurred in most object-oriented approaches, with the same name referring to a single type or a set of types depending on context.

In Ada 9X, if one wants to define a parameter or an access type that permits references to objects of any type in a class, one must explicitly specify the "class-wide" type, such as Expression'Class. This class-wide type stands for the type Expression, plus all of its derivatives, direct or indirect (Literal, Binary_Operator, etc.) Similarly, a class-wide type Binary_Operator'Class stands for Binary_Operator and its derivatives (such as Add_Op and Subtract_Op defined below in the body of package Expressions). Binary_Operator'Class can be thought of as a subclass of Expression'Class, in that it represents a subset of the types represented by Expression'Class.

By making this explicit distinction between specific types and their associated class-wide types, Ada 9X gives the programmer full control over static versus dynamic binding to operations. It is only operations on a class-wide operand, such as the dereference of a class-wide pointer (like Expr_Ptr) that result in a run-time dispatch to an "appropriate" implementation of a given primitive operation (like Evaluate). Operands of a specific type result in a normal statically bound call. This distinction makes it easier to identify, and hence document, those places where (re)dispatching occurs, which is essential when defining derivatives of a type that inherit some of the primitive operations, but choose to override others. If the implementation of one primitive operation involves a redispatch to the implementation of a second primitive, overriding this second primitive has an indirect effect on the first. The Ada 9X approach ensures that these indirect effects are readily visible in the source, and deserving of documentation.

Completing the Example

Here is a simple test program that illustrates how the interface to the Expressions abstraction is used, and includes a call on Evaluate that involves run-time dispatch (dynamic binding):

```
with Expressions; use Expressions;
procedure Test_Expr is
        -- Simple test for the expression
        -- abstraction
    X : Float;
    E : Expr_Ptr;
begin
    -- Build an expression tree
    -- for (12.0 - 7.0) + 10.0
    E := Create_Binary_Op("+",
        Create_Binary_Op("-",
            new Literal'(Value=>12.0),
            new Literal'(Value=>7.0)),
        new Literal'(Value=>10.0));

    -- Dispatch to the appropriate Evaluate
    -- routine, based on the tag of E.all (it
    -- should identify the adding operator).
    X := Evaluate(E.all);
    if X /= 15.0 then
        raise Program_Error;
            -- Something is amiss
    end if;
end Test_Expr;
```

Here is a possible implementation of the Expressions package:

```
package body Expressions is

    -- First we define the body for the
    -- Evaluate for literals
    function Evaluate(E : Literal)
     return Float is
    begin
        return E.Value;
    end Evaluate;

    -- Next we define some derivatives of
    -- Binary_Operator

    type Add_Op is
     new Binary_Operator with null record;
        -- This is a null record extension of
        -- Binary_Operator, since the
        -- components in Binary_Operator
        -- are sufficient for Add_Op
    function Evaluate(Aop : Add_Op)
     return Float is
    begin
    return Evaluate(Aop.Left_Opnd.all) +
        Evaluate(Aop.Right_Opnd.all);
        -- The above calls on Evaluate
        -- dispatch at run-time to the
        -- "appropriate" implementation,
        -- based on the tags of the
        -- objects designated by the left
        -- and right operand pointers.
end Evaluate;

type Subtract_Op is
 new Binary_Operator with null record;
function Evaluate(Sop : Subtract_Op)
 return Float is
begin
    return Evaluate(Sop.Left_Opnd.all) -
        Evaluate(Sop.Right_Opnd.all);
        -- The above calls involve run-
        -- time dispatching, as in the
        -- version of Evaluate for
        -- Add_Ops.
end Evaluate;

. . . -- etc. for other binary operators;
    -- a similar approach would be used
    -- for unary operators.

-- This defines an access-to-function
-- type, whose values point to an
-- appropriate creation function
type Bin_Op_Creator is
 access function(
    Left_Opnd, Right_Opnd : Expr_Ptr)
 return Expr_Ptr;

-- Here we define a data structure for
-- associating operator names
-- with a function for creating a
-- particular derivative of
-- Binary_Operator

type Bin_Op_Name_Record(
 Name_Length : Positive);
type Bin_Op_Name_Ptr is
 access Bin_Op_Name_Record;
type Bin_Op_Name_Record(
 Name_Length : Positive) is
    record
        Next : Bin_Op_Name_Ptr;
            -- link on chain
        Creator : Bin_Op_Creator;
            -- creation function
        Op_Name : String(1..
         Name_Length);
                -- operator name
    end record;

First_Bin_Op : Bin_Op_Name_Ptr := null;
    -- Singly-linked list of registered
    -- binary operators

procedure Register_Binary_Op(
    Op_Name : String;
    Creation_Function : Bin_Op_Creator)
is
begin
    -- Prepend operator to list of
    -- registered binary operators
    First_Bin_Op :=
     new Bin_Op_Name_Record'(
        Name_Length => Op_Name'Length,
        Next => First_Bin_Op,
          -- Chain onto head of list
        Creator => Creation_Function,
        Op_Name => Op_Name);
end Register_Binary_Op;

function Create_Binary_Op(
        Op_Name : String;
        Left_Opnd : Expr_Ptr;
        Right_Opnd : Expr_Ptr) is
    Op_Ptr : Binary_Op_Name_Ptr :=
     First_Binary_Op;
begin
    -- Look for operator in list of
    -- registered binary ops
    while Op_Ptr /= null loop
        if Op_Ptr.Op_Name = Op_Name
          then
            -- Found it, call its creation
            -- function
            return
             Op_Ptr.Creator(Left_Opnd,
                Right_Opnd);
        end if;
        Op_ptr := Op_Ptr.Next;
    end loop;
    -- Not a registered binary operator,
    -- raise an exception
    raise Invalid_Operator_Name;
end Create_Binary_Op;

-- Here we define the creation functions,
-- one for each binary operator:

function Create_Add_Op(
    Left_Opnd : Expr_Ptr;
    Right_Opnd : Expr_Ptr)
```

```
      return Expr_Ptr is
     begin
         return new Add_Op'(
             Left_Opnd, Right_Opnd);
     end Create_Add_Op;

     function Create_Sub_Op(
         Left_Opnd : Expr_Ptr;
         Right_Opnd : Expr_Ptr)
      return Expr_Ptr is
     begin
         return new Subtract_Op'(
             Left_Opnd, Right_Opnd);
     end Create_Sub_Op;

 begin
     -- Now register the various binary
     -- operators
     Register_Binary_Op(
         Op_Name => "+",
         Creation_Function =>
             Create_Add_Op'Access);
     Register_Binary_Op(
         Op_Name => "-",
         Creation_Function =>
             Create_Sub_Op'Access);
     . . . -- etc. for other binary operators.
         -- Unary operators would be handled
         -- in a similar fashion
 end Expressions;
```

Generics and OOP

If you look at the implementations given above for the various operators, you should notice that the definitions tend to be quite repetitive. This provides an opportunity to show how the generic facilities of Ada are integrated with the tagged type features. The following implementation is essentially equivalent to the above, but encapsulates in a generic the type extension, the implementation of the operations, and the registration all in one place, allowing it to be instantiated repeatedly, once for each operator:

```
package body Expressions is
    -- This version of the body illustrates
    -- the combination of tagged type
    -- capabilities and the generic
    -- capabilities of Ada 9X

    type Bin_Op_Creator is
     access function(
         Left_Opnd, Right_Opnd : Expr_Ptr)
      return Expr_Ptr;
             -- as above

    procedure Register_Binary_Op(
        Op_Name : String;
        Creation_Function : Bin_Op_Creator)
    is
    begin
        . . . -- as above
    end Register_Binary_Op;

    generic
        Op_Name : String;
        with function Operate(
            Left, Right : Float) return Float;
    package Define_Bin_Op is
        type Bin_Op is new Binary_Operator
          with null record;
        function Evaluate(Bop : Bin_Op)
          return Float;
        function Create(
          Left_Opnd, Right_Opnd : Expr_Ptr)
          return Expr_Ptr;
        Create_Ptr : constant Bin_Op_Creator
          := Create'Access;
    end Define_Bin_Op;

    package body Define_Bin_Op is
        function Evaluate(Bop : Bin_Op)
          return Float is
        begin
            -- Apply the Operate function to
            -- the value of the operands
            return Operate(
             Evaluate(Bop.Left_Opnd.all),
             Evaluate(Bop.Right_Opnd.all));
                -- These two dispatch based
                -- on the tag of the operands
        end Evaluate;
        function Create(
         Left_Opnd, Right_Opnd : Expr_Ptr)
         return Expr_Ptr is
        begin
            -- Create and return a pointer to
            -- an instance of the type
            return new Bin_Op'(
                Left_Opnd, Right_Opnd);
        end Create;
    begin
        -- Register operator as part of
        -- instantiation
        Register_Binary_Op(
            Op_Name, Create_Ptr);
      end Define_Bin_Op;

      -- Given the above generic, we can now
      -- define and register a binary operator
      -- by a single instantiation.
      package Define_Add_Op is
       new Define_Bin_Op(
          Op_Name => "+",
          Operate => "+");

      package Define_Subtract_Op is
       new Define_Bin_Op(
          Op_Name => "-",
          Operate => "-");

      . . . -- etc. for other operators;  a similar
          -- generic could be defined and
          -- then instantiated for each
          -- unary operator.
   end Expressions;
```

As is illustrated above, object-oriented facilities are complemented by generic facilities; they do not take their place. Both the run-time polymorphism inherent in class-wide types with dispatching on object tags, and the compile-time polymorphism inherent in generic templates and parameter substitution, are critical to building sophisticated yet maintainable abstractions.

Summary

Adding object-oriented facilities to Ada 83 represents both a unique opportunity and a challenging integration task. In contrast to C++ and Simula-67, Ada 9X is building on a modern abstraction-oriented language, that already has support for modularization (packages), abstract data types (private types), compile-time polymorphism (generic templates and instantiation), and run-time exception handling. We considered simply adding an object-oriented "corner" to the language. But instead, we chose to identify the minimal set of enhancements that would provide full support for object-oriented programming, while remaining consistent and integrated with the existing abstraction facilities. In so doing, we believe we identified the critical capability that distinguishes true object-oriented programming languages from abstraction-oriented programming languages, namely the ability to have effectively multiple implementations for a single abstraction. Each object identifies its particular implementation, allowing a dynamic binding between the client of an abstraction and the appropriate implementation of a given operation.

References

1. Dahl, D.J., Myhrhaug, B. and Nygaard, K. *The Simula 67 Common Base Language.* Norwegian Computing Center, Oslo, 1970.

2. Dijkstra, E. Notes on structured programming. In *Structured Programming*, O.J. Dahl et al., Eds., Academic Press, New York, 1972.

3. Guttag, J. Abstract data types and the development of data structures. *Communications of the ACM 20*, 6 (June 1977), 396-404.

4. Guttag, J.V., Horowitz, E., and Musser, D.R. Abstract data types and software validation. *Communications of the ACM 21*, 12 (Dec. 1978), 1048-1064.

5. Intermetrics, Inc. *Ada 9X Draft Reference Manual* Version 3.0 (June 1993).

6. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8 (Aug. 1977), 564-576.

7. Stroustrup, B. *The C++ Programming Language*, 2nd Ed., Addison-Wesley, Reading, MA, 1991.

8. US DoD. *Military standard Ada programming language.* ANSI/MIL-STD 1815A. American National Standards Institute, 1983.

9. Wirth, N. Modula: A language for modular multiprogramming. *Software--Practice and Experience* (Jan. 1977), 3-35.

```
\documentstyle[11pt,twocolumn]{article}
\columnsep=0.33in
\topmargin=-0.25in
\textheight=8.75in
\oddsidemargin=-0.25in
\evensidemargin=-0.25in
\textwidth=7.0in
% No page numbers:
\pagestyle{empty}

\begin{document}

% Have title and abstract with narrow interline spacing:
\baselineskip 12pt

\title{
    {\LARGE How to Get Your Paper Accepted at OOPSLA}}

\author{{\bf Alan Snyder}\\
        \\
       {\it Hewlett-Packard Laboratories}\\
       {\it P.O. Box 10490}\\
       {\it Palo Alto, CA 94303-0969}}

% The following suppresses the date:
\date{}
\maketitle
% No page number on title page (the global empty declaration takes effect
% on second page):
\thispagestyle{empty}

% In order to get room for the copyright notice at the bottom of the left
% column of the first page, use the '\newpage' command which starts a new
% column, rather than a new page when in two-column format. Insert the
% \newpage command where appropriate in your text or abstract.

\begin{abstract}
  Abstract body.
\end{abstract}

% Regular interline spacing:
\baselineskip 14pt

\section{Introduction}

Body of the paper.

\end{document}
```