OBJECT-ORIENTED PROGRAMMING IN SMALLTALK AND ADA

Ed Seidewitz
Code 554 / Flight Dynamics Analysis Branch
Goddard Space Flight Center
Greenbelt MD 20771
(301) 286-7631

Presented at the
1987 Conference on Object-Oriented Programming Systems, Languages and Applications
October 1987

Abstract

Though Ada and Modula-2 are not object-oriented languages, an object-oriented viewpoint is crucial for effective use of their module facilities. It is therefore instructive to compare the capabilities of a modular language such as Ada with an archetypal object-oriented language such as Smalltalk. The comparison in this paper is in terms of the basic properties of encapsulation, inheritance and binding, with examples given in both languages. This comparison highlights the strengths and weaknesses of both types of languages from an object-oriented perspective. It also provides a basis for the application of experience from Smalltalk and other object-oriented languages to increasingly widely used modular languages such as Ada and Modula-2.

1. Introduction

Procedural programming techniques concentrate on functions and actions. Object-oriented techniques, by contrast, attempt to clearly model the problem domain. The designers of Simula recognized the attractiveness of this concept for simulation and included specific constructs for object-oriented programming [Dahl 68]. Since then, several programming languages have been designed specifically for general-purpose object-oriented programming. The archetypal example is, perhaps, Smalltalk because the language is structured so completely around the object concept [Goldberg 83].

Ada* [DOD 83] and Modula-2 [Wirth 83] are not designed to be object-oriented programming languages. However, they do have certain object-oriented features which are descendants of Simula constructs. Further, object-oriented concepts have become extremely popular for design of Ada programs (e.g., see [Booch 83]). This paper compares and contrasts the strict object-oriented capabilities of Smalltalk with the object-oriented features of Ada. The comparison is in terms of the basic object-oriented properties of encapsulation, inheritance and binding. I have attempted to keep the main body of the paper fairly objective, reserving my more judgemental comments for the conclusion.

---

*Ada is a registered trademark of the US Government (Ada Joint Program Office)

2. Encapsulation

An object consists of some private data and a set of operations on that data. The intent of an object is to *encapsulate* the representation of a problem domain entity which changes state over time. *Abstraction* deals with how an object presents this representation to other objects, suppressing nonessential details. The stronger the abstraction of an object, the more details are suppressed by the abstract concept. The principle of *information hiding* states that such details should be kept secret from other objects, so as to better preserve the abstraction modeled by the object. Both Smalltalk and Ada directly support these basic encapsulation concepts for objects. In Smalltalk these features are the central structure of the language while in Ada they are added to a core language of ALGOL/Pascal heritage.

In Smalltalk, objects are always *instances* of a *class* which represents a set of problem domain entities of the same kind. All instances of a class provide the same interface (set of operations) to other objects. A class thus represents a single abstraction. The class definition provides implementations for each of the instance operations (*methods* in Smalltalk) and also defines the form of the internal memory of all instances.

A Smalltalk method is called by sending a *message* to the object, such as:

MyFinances receive: 25.50

The *protocol* of an object is the set of all messages that may be received by the object. A class itself has a protocol which usually includes a few messages to request creation of instances, e.g. "Finances new". Note that protocols are not really a part of the Smalltalk language proper, but are documentation of the abstraction represented by a Smalltalk class.

The basic object-oriented construct in Ada is the *package*. Unlike Smalltalk, objects can be defined directly in Ada without having any class. Further, Ada requires the definition of the interface of an object separately from the implementation of the object. This is done in a package specification. Ada uses a more traditional procedure call syntax for object operations.

Ada is a strongly typed language, so the type of every operation argument and return value must be declared. A package specification provides enough declarative information for compile-time syntax and type checking. Additional operation descriptions, such as in the Smalltalk protocol, can be provided by comments. Other code refers to package operations using a *qualified name*, e.g., "Finances.Receive". The *package body* gives the implementation of the package.

Example 1 -- Finances

Class Finances is a simple class of objects which represent financial accounts of income and debt (all examples are simplified and adapted from [Goldberg 83]). The protocol for this class is:

Finances class protocol

instance creation

initialBalance: amount	Begin a financial account with "amount" as the amount of money on hand.
new	Begin a financial account with 0 as the amount of money on hand.

Finances instance protocol

transactions

receive: amount	Receive an amount of money.
spend: amount	Spend an amount of money.

inquiries

cashOnHand	Answer the total amount of money currently on hand.
totalReceived	Answer the total amount of money received so far.

totalSpent	Answer the total amount of money spent so far.

The implementation of the Finances class must include a method for each of the messages in the protocol. It also defines the names of a set of *instance variables* which represent the internal data of each class instance. The instance variables and the implementations of the methods are hidden from users of instances of the class. In the Smalltalk-80 system, the various parts of a class definition are accessed through an "interactive system browser." The textual description used here is based on the one used in [Goldberg 83]. The definition of class Finances is:

class name	Finances
superclass	Object
instance variable names	income debt

class methods

instance creation

```
initialBalance: amount
^super new setInitialBalance: amount

new
^super new setInitialBalance: 0
```

instance methods

transactions

```
receive: amount
income <- income + amount

spend: amount
debt <- debt + amount
```

inquiries

```
cashOnHand
^income - debt

totalReceived
^income

totalSpent
^debt
```

private

```
setInitialBalance: amount
income <- amount.
debt <- 0
```

Note that "super new" refers to the system method to create a new instance, "^" indicates returning a value and "<-" indicates assignment. Some examples of use of this class are:

```
MyFinances <- Finances initialBalance: 500.00.
MyFinances spend: 32.50.
MyFinances spend: foodCost + salesTax.
MyFinances receive: pay.
tax <- taxRate * (MyFinances totalReceived)
```

The specification for an Ada package Finances corresponding to the above Smalltalk protocol is:

```
package Finances is

  type MONEY is FLOAT;

  -- Initialization
  procedure Set (Balance : in MONEY);

  -- Transactions
  procedure Receive (Amount : in MONEY);
  procedure Spend (Amount : in MONEY);

  -- Inquiries
  function Cash_On_Hand return MONEY;
  function Total_Received return MONEY;
  function Total_Spent return MONEY;

end Finances;
```

The above specification for Finances really does not define a complete object in the Smalltalk sense. This is because a package is a static program module, and cannot be passed around as data. For an object to be passed as data in Ada it must have a *type*. A type is analogous to a Smalltalk class in that it represents a set of objects with the same set of operations and internal data. An object type is called a *private type* in Ada because the representation of the internal data is hidden. The specification for a private type FINANCES is:

```
package Finance_Handler is

  type FINANCES is private;
  type MONEY is FLOAT;

  -- Instance creation
  function Initial (Balance : MONEY)
    return FINANCES;

  -- Transactions
  procedure Receive
    ( Account   : in out FINANCES;
      Amount    : in MONEY );
  procedure Spend
    ( Account   : in out FINANCES;
      Amount    : in MONEY);

  -- Inquiries
  function Cash_On_Hand
    ( Account : FINANCES )
    return MONEY;
  function Total_Received
    ( Account : FINANCES )
    return MONEY;
  function Total_Spent
    ( Account : FINANCES )
    return MONEY;

private

  type FINANCES is
    record
      Income    : MONEY := 0.00;
      Debt      : MONEY := 0.00;
    end record;

end Finance_Handler;
```

Private types must be defined within packages. Package Finance_Handler specifies each of the operations on objects of type FINANCES, while the type itself defines the internal data for each object. The *private part* of the package contains the definition of type FINANCES in terms of other Ada type constructs. In this case, objects of type FINANCES are effectively declared to have two instance variables, as in the Smalltalk example. (The private part of a package is logically part of the package implementation, not the specification. It is included in the specification only so that the compiler can tell from the specification alone how much space to allocate for objects of private types.) The package Finance_Handler is in some ways similar to the *metaclass* of the Smalltalk class Finances. In Smalltalk, a metaclass is the class of a class. Both the metaclass and the handler package provide a framework for the definition of a class, and they also allow for the definition of class variables and class operations.

Since the declaration of instance variables is in the private part of the specification of Finance_Handler, the package body only needs to define implementations for each of the specified operations:

```
package body Finance_Handler is

  -- Instance creation
  function Initial (Balance : MONEY)
    return FINANCES is
  begin
    return
        ( Income        => Balance,
          Debt          => 0.00      );
  end Finance_Handler;

  -- Transactions
  procedure Receive
    ( Account   : in out FINANCES;
      Amount    : in MONEY ) is
  begin
    Account.Income := Account.Income
                      + Amount;
  end Receive;

  procedure Spend
    ( Account   : in out FINANCES;
      Amount    : in MONEY ) is
  begin
    Account.Debt := Account.Debt
                    + Amount;
  end Spend;

  -- Inquiries
  function Cash_On_Hand
    ( Account : FINANCES )
    return MONEY is
  begin
    return
        Account.Income - Account.Debt;
  end Cash_On_Hand;
```

```
  function Total_Received
   ( Account : FINANCES )
   return MONEY is
  begin
   return Account.Income;
  end Total_Received;

  function Total_Spent
   ( Account : FINANCES )
   return MONEY is
  begin
   return Account.Debt;
  end Total_Spent;

end Finance_Handler;
```

Each FINANCES operation explicitly includes an Account of type FINANCES as one of its parameters. The instance variables of an Account are then accessed using a qualified notation such as "Account.Income". This access to instance variables is only allowed *within the body* of package Finance_Handler. Some examples of the use of type FINANCES are:

```
declare

  My_Finances
   : Finance_Handler.FINANCES
     := Finance_Handler.Initial
          (Balance => 500.00);

begin

  Finance_Handler.Spend
   ( Account  => My_Finances,
     Amount   => 32.50 );
  Finance_Handler.Spend
   ( Account  => My_Finances,
     Amount   => Food_Cost + Sales_Tax );
  Finance_Handler.Receive
   ( Account  => My_Finances,
     Amount   => Pay );
  Tax := Tax_Rate
           * Finance_Handler.Total_Received
                                (My_Finances);

end;
```

Packages in Ada allow the definition of objects as program modules or the definition of classes as private types. Packages cannot themselves be passed as data, but the instances of private types can. It is also possible in Ada to define classes of objects which cannot be passed as data. This is done using a *generic package* which serves as a template for instances of the class. For example, the earlier specification for package Finances can be made generic by simply adding the keyword **generic** at the beginning:

```
generic
package Finances is

  ...

end Finances;
```

Other packages can then be declared as *instantiations* of the generic package. For example:

```
declare

  package My_Finances is
    new Finances;

begin

  My_Finances.Receive (Amount => Pay);
  Cash := My_Finances.Cash_On_Hand;

end;
```

I will have more to say later on other important roles of generics in Ada.

3. Inheritance

A class represents a common abstraction of a set of entities, suppressing their differences. At a lower level of abstraction, some entities may differ from others. A *subclass* represents a subset of the entities of a class. A subclass *inherits* general abstract properties from its *superclass*, defining only the specific differences which appear at its lower level of abstraction. This technique of subclass inheritance allows the incremental building of application-specific abstractions from general abstractions.

Smalltalk directly supports the concept of subclassing and inheritance. In Smalltalk every class has a superclass, except for the system class Object which describes the similarities of all objects. Instances of a subclass are the same

as instances of the superclass except for differences explicitly stated in the subclass definition. The allowed differences are the addition of instance variables, the addition of new methods and the overriding of superclass methods. An instance of a subclass will respond to *at least* all of the same messages as instances of its superclass, though not necessarily in exactly the same way.

Ada does not provide direct support for subclassing or inheritance. However, the concept of inheritance can be used profitably within Ada, in some ways more generally than in Smalltalk. When defining a subclass in Ada, it is still necessary to declare *all* operations of that subclass, even those inherited from a superclass. Thus the specification of a subclass package will include all the operations of the superclass and possibly some additional ones. (This also results in a hiding of the use of inheritance reminiscent of the discussion in [Snyder 86].) In the body of the subclass package, inherited operations must be implemented as *call-throughs* to the operations of the superclass.

Example 2 -- Deductible Finances

The class DeductibleFinances is a subclass of the Finances class of Section 2. Instances of DeductibleFinances have the same functions as instances of Finances for receiving and spending money. However, they also keep track of tax deductible expenditures. The definition of DeductibleFinances specifies one new instance variable, four new instance methods and overrides two class methods:

class name	DeductibleFinances
superclass	Finances
instance variable names	deductibleDebt

class methods

instance creation

```
initialBalance: amount
^(super initialBalance: amount) zeroDeduction

new
^super new zeroDeduction
```

instance methods

transactions

```
spendDeductible: amount
self spend: amount deducting: amount.

spend: amount deducting: deductibleAmount
super spend: amount.
deductibleDebt <- deductibleDebt
                    + deductibleAmount
```

inquiries

```
totalDeduction
^deductibleDebt
```

private

```
zeroDeduction
deductibleDebt <- 0
```

Note that sending a message to "self" results in a call on one of an object's own methods, while sending a message to "super" results in a call on one of the methods of the superclass Finances.

Now consider an Ada type which defines a subclass of the FINANCES type of Section 2:

```
with Finance_Handler;
package Deductible_Finance_Handler is

  type DEDUCTIBLE_FINANCES is private;
  subtype MONEY is
    Finance_Handler.MONEY;

  -- Instance creation
  function Initial ( Balance : MONEY )
    return DEDUCTIBLE_FINANCES;

  -- Transactions
  procedure Receive
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY );
  procedure Spend
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY;
  Deductible_Amount : in MONEY := 0.00 );
  procedure Spend_Deductible
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY );
```

```
  -- Inquiries
  function Cash_On_Hand
   ( Account : DEDUCTIBLE_FINANCES )
   return MONEY;
  function Total_Received
   ( Account : DEDUCTIBLE_FINANCES )
   return MONEY;
  function Total_Spent
   ( Account : DEDUCTIBLE_FINANCES )
   return MONEY;
  function Total_Deduction
   ( Account : DEDUCTIBLE_FINANCES )
   return MONEY;

private

  type DEDUCTIBLE_FINANCES is
   record
    Finances : Finance_Handler.FINANCES;
    Deductible_Debt : MONEY := 0.00;
   end record;

end Finance_Handler;
```

Package Deductible_Finance_Handler has the new operations Spend_Deductible and Total_Deductions, and it has a modified Spend operation. The Spend procedure has a Deductible_Amount parameter with a *default value* of 0.00.

DEDUCTIBLE_FINANCES implements inheritance from FINANCES by using the instance variable Finances of type FINANCES. Inherited operations are then implemented as call-throughs to operations on Finances:

```
package body Deductible_Finance_Handler is

  -- Instance creation
  function Initial ( Balance : MONEY )
   return DEDUCTIBLE_FINANCES is
  begin
   return
( Finances => Finance_Handler.Initial(Balance),
  Deductible_Debt => 0.00                      );
  end Initial;

  -- Transactions
  procedure Receive
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY ) is
  begin
   -- INHERITED --
   Finance_Handler.Receive
    ( Account        => Account.Finances,
      Amount         => Amount           );
  end Receive;

  procedure Spend
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY;
  Deductible_Amount : in MONEY := 0.00 ) is
  begin
   Finance_Handler.Spend
    ( Account        => Account.Finances,
      Amount         => Amount           );
   Account.Deductible_Debt
       := Account.Deductible_Debt
                  + Deductible_Amount;
  end Spend;

  procedure Spend_Deductible
( Account : in out DEDUCTIBLE_FINANCES;
  Amount : in MONEY ) is
  begin
   Spend
    ( Account                => Account,
      Amount                 => Amount,
      Deductible_Amount      => Amount );
  end Spend_Deductible;

  -- Inquiries
  function Cash_On_Hand
   ( Account : DEDUCTIBLE_FINANCES )
  return MONEY is
  begin
   -- INHERITED.--
   return Finance_Handler.Cash_On_Hand
                        (Account.Finances);
  end Cash_On_Hand;
        .
        .
        .
  function Total_Deductions
   ( Account : DEDUCTIBLE_FINANCES )
  return MONEY is
  begin
   return Account.Deductible_Debt;
  end Total_Deductions;

end Deductible_Finance_Handler;
```

Unlike Smalltalk, implementing inheritance in Ada requires an extra level of operation call. Also, in Ada the subclass does not have direct access to the instance variables of the superclass. The superclass package presents the same abstract interface to subclass packages as to any other code. This tightens the encapsulation of the superclass abstraction. It also allows easy extension to *multiple inheritance* where a subclass may inherit operations from more than one superclass. Multiple inheritance simply requires multiple superclass instance variables with inherited operations calling-through to the appropriate superclass operations. In this case the new class is really a *composite* abstraction formed from more general *component* classes.

The main drawback of this approach is that the Ada typing system does not recognize subclassing. In Ada all private types are distinct. Even though the type DEDUCTIBLE_FINANCES is logically a subclass of type FINANCES, the type DEDUCTIBLE_FINANCES is *not* a subtype of type FINANCES. It is not possible, for instance, to pass an instance of type DEDUCTIBLE_FINANCES to a procedure expecting an argument of type FINANCES. The Ada compiler would see this as a type inconsistency. A partial solution to this involves the use of the Ada generic facility, and will be discussed later in Section 4. However, the problem cannot be fully overcome in Ada, and [Meyer 86] clearly shows that true inheritance is more powrful than genericity.

4. Binding

The Smalltalk message passing mechanism operates dynamically. When a message is sent to a Smalltalk object, the method to respond to that message is looked-up at run-time in the object's class (and possibly superclasses). Further, Smalltalk variables are not typed, so they may contain objects of any class. Thus it is generally not possible to determine statically exactly what method in what class will respond to a message. Messages are *dynamically bound* to methods at run-time. If an object cannot respond to a message, there is a run-time error.

The use of dynamic binding gives the programmer great freedom to create general code. Any object can be used in an instance variable or as an argument in a message as long as it can respond to the messages sent to it. Another use of dynamic binding in Smalltalk is with the "pseudo-variable" "self" which is used by an object to send messages to itself. When a message is sent to an object, "self" is set to the object to which the message is sent. The dynamic binding of messages sent to "self" allows a class to call on methods that are really defined in a subclass.

Unlike Smalltalk, Ada is a strongly typed language. This means that all variables and parameters must be declared to be of a single specific type. This allows an Ada compiler to check statically that only values of the correct type are being assigned to variables and used as arguments. The Ada compiler can also always determine exactly what operation from what package (if any) is being invoked by a given call. Operation calls are thus *statically bound* to the proper operation. Undefined operation calls are always discovered at compile-time.

A way around this involves the use of generics. In addition to their role in creating classes of packages, generics also allow a package to be *parameterized* with type and subprogram parameters. This feature can be used to declare a package which can use *any* class with certain needed operations. Generic facilities can also be used to allow a class to defer the implementation of some operations to subclasses.

Example 3 -- Sample Space

The class SampleSpace represents random selection without replacement from a collection of items. It has the following protocol:

SampleSpace class protocol

instance creation

data: aCollection	Create an instance such that aCollection is the sample space.

SampleSpace instance protocol

accessing

next	Answer the next element chosen at random from the sample space, removing it from the space.
next: anInteger	Answer an ordered collection of anInteger number of selections from the sample space.

testing

isEmpty	Answer whether any items remain to be sampled.
size	Answer the number of items remaining to be sampled.

This protocol does *not* specify exactly what kind of collection must be used for the sample space. The class definition is:

class name	SampleSpace
superclass	Object
instance variable names	data rand

class methods

instance creation

```
data: aCollection
^super new setData: aCollection.
```

instance methods

accessing

```
next
| item |
self isEmpty ifTrue:
[self error 'no values exist in the sample space'].
item <- data at:
        (rand next * data size) truncated + 1.
data remove: item.
^item

next: anInteger
| aCollection |
aCollection
      <- OrderedCollection new: anInteger.
anInteger timesRepeat:
  [aCollection addLast: self next].
^aCollection
```

testing

```
isEmpty
^self size = 0

size
^data size
```

private

```
setData: aCollection
data <- aCollection.
rand <- Random new
```

Note that local variables in methods are listed between vertical bars at the beginning of the method. Also, the definition of SampleSapce uses an instance of the Smalltalk system class Random to generate random numbers. In the methods for "next" and "size", SampleSpace sends the messages "at:", "size" and "remove:" to the instance variable "data" which holds the collection of sample space items. This means that any object which can respond to "at:", "size" and "remove:" can serve as the collection. This object could be an instance of a Smalltalk system class such as Array, or it could be an instance of a user-defined class. An example of the use of SampleSpace is shuffling a deck of cards:

class name	CardDeck
superclass	Object
instance variable names	cards

```
.
.
.

shuffle
| sample |
sample <- SampleSpace data: cards.
cards <- sample next: cards size
```

An Ada generic Sample_Space package needs a COLLECTION type and At, Size and Remove

operations. A specification for this package is:

```
generic

  type COLLECTION_TYPE is private;
  type ELEMENT_TYPE is private;

  with function At
    ( Collection : COLLECTION_TYPE;
      Index      : POSITIVE )
    return ELEMENT_TYPE;
  with function Size
    ( Collection : COLLECTION_TYPE )
    return ELEMENT_TYPE;
  with procedure Remove
    ( Collection : in out COLLECTION_TYPE;
      Element    : in ELEMENT_TYPE );

package Sample_Space is

  Empty : exception;

  type ELEMENT_LIST is
    array (NATURAL range <>)
      of ELEMENT_TYPE;

  -- Initialization
  procedure Set
    ( Data       : in COLLECTION_TYPE );

  -- Accessing
  function Next return ELEMENT_TYPE;
  function Next ( Number : NATURAL )
    return ELEMENT_LIST;

  -- Testing
  function Is_Empty return BOOLEAN;
  function Size return NATURAL;

end Sample_Space;
```

Package Sample_Space uses the generic facility both to parameterize itself and to allow a class of objects (as discussed in Section 2). It would also have been possible to define a generic Sample_Space_Handler package with a SAMPLE_SPACE type. This would have allowed sample spaces to be passed as data, an ability which is not really needed for the present example.

The body of Sample_Space is:

```
with Random;
package body Sample_Space is

  -- Instance variable
  Sample_Data : COLLECTION_TYPE;

  -- Initialization
  procedure Set
    ( Data       : COLLECTION_TYPE ) is
  begin
    Sample_Data := Data;
  end Set;

  -- Accessing
  function Next return ELEMENT_TYPE is
    Item : ELEMENT_TYPE;
  begin
    if Is_Empty then
      raise Empty;
    end if;
    Item := At ( Sample_Data, Index =>
          NATURAL((Random.Value*Size)+1) );
    Remove
      ( Collection       => Sample_Data,
        Element          => Item );
    return Item;
  end Next;

  function Next ( Number : NATURAL )
    return ELEMENT_LIST is
    List : ELEMENT_LIST(1 .. Number);
  begin
    for I in 1 .. Number loop
      List(I) := Next;
    end loop;
    return List;
  end Next;

  -- Testing
  function Is_Empty return BOOLEAN is
  begin
    return (Size = 0);
  end Is_Empty;

  function Size return NATURAL is
  begin
    return Size(Sample_Data);
  end Size;

end Sample_Space;
```

The Sample_Space package body assumes the availability of a package Random to generate random numbers. Sample_Space could then be

used to shuffle an instance of private type CARD_DECK:

```
with Sample_Space;
package body Card_Deck_Handler is

    ...

  package Sample is new Sample_Space
  ( COLLECTION_TYPE  => CARD_DECK,
    ELEMENT_TYPE     => CARD_TYPE,
    At               => Card,
    Size             => Deck_Size,
    Remove           => Remove_Card );

    ...

  procedure Shuffle
    ( Cards : in out CARD_DECK ) is
  begin
    Sample.Set (Data => Cards);
    Cards := CARD_DECK
              (Sample.Next(Deck_Size(Cards)));
  end Shuffle;

    ...

end Card_Deck_Handler;
```

Generic package Sample_Space is a template for a general class of sample spaces. Since a COLLECTION_TYPE must be specified when Sample_Space is instantiated, each instance of this class can only handle a *single* type of collection for sampling. Thus an Ada compiler can still perform static type checking for each instantiation of generic packages.

The dynamic binding and lack of typing in Smalltalk allow an instance of a subclass to be used anyplace an instance of its superclass may be used. As mentioned at the end of Section 3 the Ada type system does not allow this because it views all private types as distinct and incompatible. The above generic technique can help with this problem, also. A generic package (or other program unit) which is parameterized by the types and operations it needs will be able to use any type with the necessary operations. Thus if the private type representing some class can be plugged into a generic, then a subclass type can also be plugged into that same generic. However, the generic must be instantiated *separately* for each type. There is no easy way in Ada have a true *polymorphic* procedure, that is, a *single* procedure with an argument which accepts values of different types.

5. Conclusion

Smalltalk and Ada are based on quite different philosophies. Smalltalk is designed to make it easier to program and to incrementally build and modify systems. Ada, on the other hand, purposefully places certain additional obligations on the programmer so that the final system will be more reliable and more maintainable. The Ada philosophy takes a much more life-cycle-oriented approach, recognizing that most costly phase of software development is maintenance, not coding.

If the languages have such different bases, then why consider using object-oriented ideas for Ada? The answer is that object-oriented concepts really apply to more than just programming. In Ada circles, these concepts are usually applied to design [Booch 83, Seidewitz 86a, Seidewitz 86b]. The object-oriented viewpoint is crucial to designing for effect use of Ada's package facility. Further, the object-oriented approach can be a general way of thinking about software systems which can be applied from system specification through testing. This fits in quite well with the Ada life-cycle philosophy [Booch 86, Stark 87].

Still, Ada has some unfortunate drawbacks for object-oriented programming, especially in its lack of support for inheritance. As an object-oriented programming language Smalltalk is in many ways clearly superior to Ada. However, as a life-cycle software engineering language Ada has great advantages. Static strong typing is crucial to increasing the reliability of software. Even with a good testing methodology, large amounts of code will not be thoroughly tested because it is only executed in rare combinations of situations. But when a system is running continuously for years, any errors that remain in these sections of code will almost certainly occur. This is especially true for the embedded real-time systems which were Ada's original mandate. In Ada, all sections of code are checked by the compiler, and many errors can be caught before the testing phase due to static type checking and static operation binding.

It is possible to support inheritance and even polymorphism within a statically typed language (as in, for example, Eiffel [Meyer 86, Meyer 87]). Inheritance might be added to Ada without too much change to the design of the language. Incorporation of polymorphism would be much more difficult, and probably require a philosophical change in the Ada language design. However, even with these deficiencies for object-oriented programming, Ada still provides a useful vehicle for applying object-oriented concepts throughout the software development life-cycle.

Much of the above discussion also applies to other modular languages such as Modula-2 (though Modula-2 does not directly support genericity). As these languages become more and more widely used it will be increasingly important to apply to them the experience in object-oriented software development gained from Smalltalk and other object-oriented languages.

References

[Booch 83]
Grady Booch. Software Engineering with Ada, Benjamin/Cummings, 1983.

[Booch 86]
Grady Booch. "Object-Oriented Development," IEEE Transactions on Software Engineering, February 1986.

[Dahl 68]
O-J Dahl. Simula 67 Common Base Language, Norwegian Computing Center, Oslo, Norway, 1968.

[DOD 83]
US Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.

[Goldberg 83]
Adele Goldberg and David Robson. Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Meyer 86]
Bertrand Meyer. "Genericity versus Inheritance," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, November 1986.

[Meyer 87]
Bertrand Meyer. "Eiffel: Programming for Reusability and Extendability," SIGPLAN Notices, February 1987.

[Seidewitz 86a]
Ed Seidewitz and Mike Stark. "Towards an Object-Oriented Software Development Methodology," Proc. of the 1st Intl. Conf. on Ada Applications for the Spac Station, June 1986.

[Seidewitz 86b]
Ed Seidewitz and Mike Stark. General Object-Oriented Software Development, Goddard Space Flight Center, SEL-86-002, August 1986.

[Snyder 86]
Alan Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, November 1986.

[Stark 87]
Mike Stark and Ed Seidewitz. "Towards a General Object-Oriented Ada Life-Cycle," Proc. of the Joint 4th Washington Ada Symposium / Fifth Nat. Conf. on Ada Technology, March 1987.

[Wirth 83]
Niklaus Wirth. Programming in Modula-2, Springer-Verlag, 1983.