510-61
136140

OBJECT-ORIENTED PROGRAMMING WITH MIXINS IN ADA

Ed Seidewitz
Goddard Space Flight Center
Code 552.2
Greenbelt MD 20771
(301)286-7631
eseidewitz@gsfcmail.nasa.gov

N93-17171

INTRODUCTION

My guess is that object-oriented programming will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it. And no one will know just what it is.

[Rentsch 82]

Recently, I wrote a paper discussing the lack of "true" object-oriented programming language features in Ada 83, why one might desire them in Ada and how they might be added in Ada 9X [Seidewitz 91]. The approach I took in this paper was to build the new object-oriented features of Ada 9X as much as possible on the basic constructs and philosophy of Ada 83. The object-oriented features proposed for Ada 9X [Ada9X 91b], while different in detail, are based on the same kind of approach.

Further consideration of this approach led me on a long reflection on the nature of object-oriented programming and its application to Ada. The results of this reflection, presented in this paper, show how a fairly natural object-oriented style can indeed be developed even in Ada 83. The exercise of developing this style is useful for at least three reasons:

1. It provides a useful style for programming object-oriented applications in Ada 83 until new features become available with Ada 9X;
2. It demystifies many of the mechanisms that seem to be "magic" in most object-oriented programming languages by making them explicit; and
3. It points out areas that are and are not in need of change in Ada 83 to make object-oriented programming more natural in Ada 9X.

In the next four sections I will address in turn the issues of object-oriented classes, mixins, self-reference and supertyping. The presentation is through a sequence of examples, similar to those in [Seidewitz 91]. This results in some overlap with that paper, but all the examples in the present paper are written entirely in Ada 83. I will return to considerations for Ada 9X in the last section of the paper.

CLASSES

An object *represents a component of...[a] software system... An object consists of some private memory and a set of operations...A crucial property of an object is that its private memory can be manipulated only by its operations...A* class *describes the implementation of a set of objects that all represent the same kind of system component.*

[Goldberg and Robson 83]

In Ada, an object is a variable or a constant that contains a value. The declared type of the object determines the set of possible values for the object and the set of operations that may be applied to the object. If this type is a private type, then the value of the object may only be changed through application of an operation. This corresponds to the object-oriented notion of a *class*.

Consider, for example, a simple financial account class implemented as a private type:

```
with Finance_Types; use Finance_Types;
package Finance is

   type ACCOUNT is limited private;

   procedure Open          (The_Account   : in out ACCOUNT;
                            With_Balance  : in     MONEY);
```

PRECEDING PAGE BLANK NOT FILMED

```
  procedure Deposit     (Into_Account  : in out ACCOUNT;
                          The_Amount    : in     MONEY);

  procedure Withdraw    (From_Account  : in out ACCOUNT;
                          The_Amount    : in     MONEY);

  function  Balance_Of  (The_Account   : ACCOUNT) return MONEY;

private

  type ACCOUNT is
    record
      Balance : MONEY := 0.00;
    end record;

end Finance;
```

The type Finance.ACCOUNT represents a class of account objects. The subprograms defined in package Finance are the allowable operations on objects of this class. The body of this package is straightforward. Note that for simplicity I will assume that a number of simple types (such as MONEY) are defined in a Finance_Types package.

The class defined by package Finance provides a simple but very general abstraction. In an object-oriented approach, such general classes are used as the basis for implementing more specialized classes. For example, a savings account is a specific kind of account that holds savings that earn interest. Other than some new operations associated with earning interest, a savings account is the same as the original general financial account. Thus we should be able to implement a savings account in terms of a general account:

```
with Finance_Types; use Finance_Types;
package Savings is

  type ACCOUNT is limited private;

  procedure Open           (The_Account  : in out ACCOUNT;
                             With_Balance : in     MONEY);

  procedure Set_Rate       (Of_Account   : in out ACCOUNT;
                             To_Rate      : in     RATE);

  procedure Deposit        (Into_Account : in out ACCOUNT;
                             The_Amount   : in     MONEY);

  procedure Withdraw       (From_Account : in out ACCOUNT;
                             The_Amount   : in     MONEY);

  procedure Earn_Interest  (On_Account   : in out ACCOUNT;
                             Over_Time    : in     INTERVAL);

  function  Balance_Of     (The_Account  : ACCOUNT) return MONEY;

  function  Interest_On    (The_Account  : ACCOUNT) return MONEY;

private

  type ACCOUNT is
    record
      Parent       : Finance.ACCOUNT;
      Rate         : RATE := 0.06;
      Interest     : MONEY := 0.00;
    end record;

end Savings;
```

While this may not seem to gain us a lot in this simple example, such incremental extension of abstractions is fundamental to object-oriented techniques. The class of financial accounts is said to be the *superclass* of the class of savings accounts. Each savings account (of type Savings.ACCOUNT) has a unique *parent* financial account (of type Finance.ACCOUNT).

Now, three of the seven savings account operations (Open, Deposit and Withdraw) are syntactically and semantically the same as the corresponding financial account operations. Thus, we would like to *inherit* these financial account operations. Ada 83 has no direct way of doing this. Nevertheless, we can achieve the effect of inheritance for our present purposes by using *call-through* subprograms. For example, the Savings.Deposit operation can easily be implemented as follows:

```
procedure Deposit (Into_Account : in out ACCOUNT;
                   The_Amount   : in     MONEY) is
begin
  Finance.Deposit (Into_Account.Parent, The_Amount)
end Deposit;
```

The expense of such call-throughs may be minimized through the use of pragma `Inline`.

Three other savings account operations (`Set_Rate`, `Earn_Interest` and `Interest_On`) provide the incremental new functionality of the savings account *subclass*. These operations are implemented in terms of the additional components of the representation of type `Savings.ACCOUNT`. For example:

```
procedure Earn_Interest (On_Account : in out ACCOUNT;
                         Over_Time  : in     INTERVAL) is

  Balance : constant MONEY := Balance_Of (On_Account);

begin

  if Balance > 0.00 then
    On_Account.Interest := On_Account.Interest
                         + Balance*On_Account.Rate*Over_Time;
  end if;

end Earn_Interest;
```

Note that the `Balance_Of` operation used here is the subclass operation `Savings.Balance_Of`.

The remaining savings account operation, `Balance_Of`, is syntactically the same as the financial account operation, but it is semantically different. The balance of a savings account includes interest earned up to the present point in time:

```
function Balance_Of (The_Account : ACCOUNT) return MONEY is
begin

  return Finance.Balance_Of (The_Account.Parent)
       + The_Account.Interest;

end Balance_Of;
```

Note that while `Balance_Of` is not a call-through operation, the superclass operation `Finance.Balance_Of` is used in its implementation.

The usefulness of a superclass like the financial account class comes from the fact that it can provide a common basis for a *number* of subclasses. For example, a class of checking accounts may provide another subclass of financial accounts:

```
with Finance;
with Finance_Types; use Finance_Types;
package Checking is

  type ACCOUNT is limited private;

  procedure Open       (The_Account  : in out ACCOUNT;
                        With_Balance : in     MONEY);

  procedure Set_Fee    (Of_Account   : in out ACCOUNT;
                        To_Fee       : in     MONEY);

  procedure Deposit    (Into_Account : in out ACCOUNT;
                        The_Amount   : in     MONEY);

  procedure Withdraw   (From_Account : in out ACCOUNT;
                        The_Amount   : in     MONEY);

  function  Balance_Of (The_Account  : ACCOUNT) return MONEY;

private
```

```
  type ACCOUNT is
    record
      Parent            : Finance.ACCOUNT;
      Overdraft_Fee     : MONEY := 10.00;
    end record;

end Checking;
```

In this simple example, the only difference between checking accounts and financial accounts is that overdrawing a checking account is not permitted. Further, each overdraft attempt (i.e, a returned check) is penalized by deducting a fee from the account. Thus, the implementation of `Withdraw` must be changed for checking accounts:

```
procedure Withdraw (From_Account  : in out ACCOUNT;
                    The_Amount    : in     MONEY) is
begin

  if The_Amount <= Balance_Of(From_Account) then
    Finance.Withdraw (From_Account.Parent, The_Amount);
  else
    Finance.Withdraw (From_Account.Parent,
                      From_Account.Overdraft_Fee);
  end if;

end Withdraw;
```

The savings account and checking account subclasses of the financial account class may themselves act as superclasses for even more specialized classes. Thus, a general class may be the root of a quite extended class hierarchy. Each subclass in the hierarchy incrementally extends the capabilities of its superclass, while inheriting common functionality.

MIXINS

A mixin is...a subclass definition that may be applied to different superclasses to create a related family of modified classes.

[Bracha and Cook 90]

A superclass may be used as the base for many subclasses. However, as described so far, a subclass is tied to one superclass. For instance, savings accounts are based specifically on the class defined by package `Finance`. There may be other types of accounts to which we want to added interest-bearing functionality such as that defined for savings accounts. For example, an interest-bearing checking account is basically a checking account with interest-bearing functionality added to it (or, alternatively, a savings account with checking functionality added).

Rather than recoding essentially the same interest-bearing functionality each time it is needed, we can capture this functionality in a generic package that takes a specific superclass as a parameter:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with function Balance_Of (The_Account : SUPERCLASS) return MONEY is <>;

package Interest is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent        : SUPERCLASS;
      Extension     : MIXIN;
    end record;

  procedure Set_Rate      (Of_Account  : in out ACCOUNT;
                           To_Rate     : in     RATE);

  procedure Earn_Interest (On_Account  : in out ACCOUNT;
                           Over_Time   : in     INTERVAL);

  function  Balance_Of    (The_Account : ACCOUNT) return MONEY;
  function  Interest_On   (The_Account : ACCOUNT) return MONEY;

private

  type MIXIN is
    record
      Rate          : RATE := 0.06;
      Interest      : MONEY := 0.00;
    end record;

end Interest;
```

A generic package such as this is called a *mixin* because it provides an increment of functionality which may be "mixed-into" any superclass that has the operations required to fill in the generic parameters. Typically, mixins are used within a framework of *multiple inheritance*. For example, we can reconstruct the savings account class by inheriting from *both* the financial account class and an appropriate instantiation of the interest mixin:

```
with Finance, Interest;
with Finance_Types; use Finance_Types;
package Savings is

  type ACCOUNT is limited private;

  procedure Open         (The_Account   : in out ACCOUNT;
                          With_Balance  : in     MONEY);


  ...

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Savings_Interest is
    new Interest (Finance.ACCOUNT, Finance.Balance_Of);

  type ACCOUNT is new Savings_Interest.ACCOUNT;

end Savings;
```

The `Parent` component of the `ACCOUNT` type defined in mixin `Interest` is used to inherit from the parent superclass `Finance` via a call-through. For example:

```
procedure Open (The_Account  : in out ACCOUNT;
                With_Balance : in     MONEY) is
begin
  Finance.Open (The_Account.Parent, With_Balance);
end Open;
```

The record type `Savings_Interest.ACCOUNT` is defined as a visible, rather than a private, type in the mixin to allow access to the `Parent` component. Note that it would not be possible to replace this use of a visible record component with a function that returns the parent object, because we need to use the parent as an **in out** parameter. The type `MIXIN` is never used itself outside of the mixin package.

Call-through subprograms are also needed to inherit from the mixin instantiation `Savings_Interest`. This is because the equivalent derived subprograms obtained from the derived type definition of `Savings.ACCOUNT` are hidden by the operations declared in the package specification, and in Ada 83 there must be a full subprogram body for each of these declarations. For example:

```
function  Interest_On (The_Account : ACCOUNT) return MONEY is
begin
  return Savings_Interest.Interest_On
         (Savings_Interest.ACCOUNT(The_Account));
end Interest_On;
```

Having introduced the concept of mixins, we can, of course, also create a mixin that embodies the overdraft functionality of the checking account class:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with procedure Withdraw  (From_Account : in out SUPERCLASS;
                            The_Amount   : in     MONEY) is <>;

  with function Balance_Of (The_Account  : SUPERCLASS) return MONEY is <>;

package Draft is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent          : SUPERCLASS;
      Extension       : MIXIN;
    end record;

  procedure Set_Fee  (Of_Account   : in out ACCOUNT;
                      To_Fee       : in     MONEY);

  procedure Withdraw (From_Account : in out ACCOUNT;
                      The_Amount   : in     MONEY);

private

  type MIXIN is
    record
      Overdraft_Fee   : MONEY := 10.00;
    end record;

end Draft;
```

Even our original financial account class can be converted to a mixin:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

package Monetary is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent       : SUPERCLASS;
      Extension    : MIXIN;
    end record;

  procedure Open        (The_Account  : in out ACCOUNT;
                          With_Balance : in     MONEY);

  ...

  function  Balance_Of  (The_Account   : ACCOUNT) return MONEY;

private

  type MIXIN is
    record
      Balance      : MONEY := 0.00;
    end record;

end Monetary;
```

Of course, this mixin does not require any superclass functionality to implement its operations. However, use of the mixin construct allows monetary account functionality to be mixed into any class.

The use of mixins causes traditional class hierarchies to collapse into pieces. Each piece is a mixin that provides a well-defined increment of functionality. We can then form specific classes from these pieces by instantiating a number of mixins and inheriting all necessary functionality from them. To provide a definite starting point for this process, we can define a *root* class that basically does nothing more than provide an empty record to which we can add mixins:

```
package Root is

  type CLASS is limited private;

private

  type CLASS is
    record
      null;
    end record;

end Root;
```

While this root class seems a bit pointless, the concept will prove useful in the next section.

At last we are ready to construct an interest-bearing checking account class without rewriting any savings account or checking account functionality. To do this, we simply mix together interest, draft and monetary account functionality. All Interest_Bearing_Checking.ACCOUNT operations are implemented as call-throughs to various mixin operations. Thus, from the three mixins Monetary, Interest and Draft, we can easily construct an interest-bearing checking account class, as well as reconstructing our original financial, savings and checking account classes.

Of course, in the actual Interest_Bearing_Checking package, the three mixin generics must be instantiated in a specific sequential order. In the present case, we must first establish the basic monetary account functionality, then mix in interest and draft functionality. This results in the following implementation:

```
with Root, Monetary, Interest, Draft;
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open          (The_Account   : in out ACCOUNT;
                            With_Balance  : in     MONEY);

  ...

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Checking_Finance is           -- Basic financial account
    new Monetary (Root.Class);

  package Checking_Interest is          -- Mix in interest functionality
    new Interest (Checking_Finance.ACCOUNT, Checking_Finance.Balance_Of);

  procedure Withdraw (From_Account     : in out Checking_Interest.ACCOUNT;
                      The_Amount       : in     MONEY);
  -- call-through to Finance.Withdraw

  package Checking_Draft is             -- Mix in overdraft fee functionality
    new Draft (Checking_Interest.ACCOUNT,
               Withdraw,
               Checking_Interest.Balance_Of);

  type ACCOUNT is new Checking_Draft.ACCOUNT;
                                        -- Private type representation

end Interest_Bearing_Checking;
```

Note that all the mixins are instantiated in the private part of the specification. Each instantiation uses the type and subprograms from the previous instantiation as arguments. The intermediate procedure Withdraw for type Checking_Interest.ACCOUNT is necessary because the instantiated mixin Checking_Interest only provides the interest-related operations on Checking_Interest.ACCOUNT. It is implemented as simply a call-through to Finance.Withdraw.

When an object of a given class is created its state components include those of the class and all its superclasses and it can perform operations of the class and its superclasses on the component state. References to "self" in operations of a superclass refer to the composite object on behalf of which the operation is to be performed.

[Wegner 87]

In the interest-bearing checking account package at the end of the last section, the `Interest` mixin was instantiated before the `Draft` mixin. It would seem that we could equally well have instantiated them in the opposite order:

```
with Root, Monetary, Interest, Draft;
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open        (The_Account   : in out ACCOUNT;
                          With_Balance  : in     MONEY);

  ...

  function  Interest_On (The_Account   : ACCOUNT) return MONEY;

private

  package Checking_Finance is          -- Basic financial account
    new Monetary (Root.Class);

  package Checking_Draft is           -- Mix in overdraft fee functionality
    new Draft (Checking_Finance.ACCOUNT, Checking_Finance.Withdraw,
               Checking_Finance.Balance_Of);

  function Balance_Of (The_Account : in Checking_Draft.ACCOUNT)
    return MONEY;
  -- call-through to Finance.Balance_Of

  package Checking_Interest is         -- Mix in interest functionality
    new Interest (Checking_Draft.ACCOUNT, Balance_Of);

  type ACCOUNT is new Checking_Interest.ACCOUNT;
                                       -- Private type representation

end Interest_Bearing_Checking;
```

Unfortunately, it turns out that this introduces a subtle error, as follows:

- In the new implementation, the `Draft` mixin is instantiated before the `Interest` mixin, using the `Checking_Finance.Balance_Of` operation.
- The implementation of the `Withdraw` operation in the `Draft` mixin uses the `Balance_Of` operation given as a generic formal superclass operation to determine if there is an overdraft. In this case, the actual subprogram used is `Checking_Finance.Balance_Of`, which does not add in any earned interest.
- The `Interest_Bearing_Checking.Withdraw` operation is inherited from the instantiation `Checking_Draft` of the `Draft` mixin, so as to include the overdraft functionality. This means that accumulated interest is ignored when checking for an overdraft. This is clearly unfair to the customer!

The problem is that we do not really want to use the *superclass* `Balance_Of` operation in the `Draft` mixin instantiation. Rather, we need to use the `Balance_Of` operation from the composite *subclass* being constructed. However, we cannot use the subclass type `Interest_Bearing_Checking.ACCOUNT` in the instantiation of the `Draft` mixin, because that type cannot be fully defined yet. Thus, we must instead be sure to instantiate the `Interest` mixin first, so that the interest-bearing functionality is mixed into the `Balance_Of` operation before `Draft` is instantiated.

Such order dependencies are at best annoying sources of potential errors. At worst, they can introduce circular dependencies that make it impossible to mix together certain mixins. To avoid this, we need a mechanism that allows mixins to call subclass operations in addition to superclass operations. Following the parameterization approach that led us to mixins in the first place, we can include a second generic formal type parameter in mixins to represent the subclass.

For example, we want the Draft mixin to use the subclass Balance_Of operation:

```
with Finance_Types; use Finance_Types;
generic

  type SUPERCLASS is limited private;

  with procedure Withdraw  (From_Account : in out SUPERCLASS;
                            The_Amount   : in     MONEY) is <>;

  type SUBCLASS is limited private;

  with function Balance_Of (The_Account  : SUBCLASS) return MONEY is <>;

  with function Self       (Parent       : SUPERCLASS) return SUBCLASS is <>;

package Draft is

  type MIXIN is limited private;
  type ACCOUNT is
    record
      Parent        : SUPERCLASS;
      Extension     : MIXIN;
    end record;

  procedure Set_Fee  (Of_Account   : in out ACCOUNT;
                      To_Fee       : in     MONEY);

  procedure Withdraw (From_Account : in out ACCOUNT;
                      The_Amount   : in     MONEY);

  function  Self     (This_Account : ACCOUNT) return SUBCLASS;

private

  type MIXIN is
    record
      Overdraft_Fee  : MONEY := 10.00;
    end record;

end Draft;
```

The Withdraw operation for this mixin is then implemented as follows:

```
procedure Withdraw (From_Account : in out ACCOUNT;
                    The_Amount   : in     MONEY) is
begin

  if The_Amount <= Balance_Of(Self(From_Amount)) then
    Finance.Withdraw (From_Account.Parent, The_Amount);
  else
    Finance.Withdraw (From_Account.Parent,
                      From_Account.Extension.Overdraft_Fee);
  end if;

end Withdraw;
```

Note the use of the function Self to convert an object of type Draft.ACCOUNT to the appropriate object of type SUBCLASS. These odd little Self functions are the key to this approach. They allow us to use the subclass operations as required.

The question is, of course, how can we implement such a Self function? Strangely enough, we can implement it in terms of the superclass Self function given as a generic formal parameter:

```
function Self (This_Account : ACCOUNT) return SUBCLASS is
begin
  return Self (This_Account.Parent);
end Self;
```

Obviously, this passing of the buck must end someplace. It ends with the root class, which we reimplement as follows:

```
generic

  type SUBCLASS is limited private;

package Root is

  type CLASS is limited private;

  procedure Initialize (The_Object  : in out CLASS;
                        To_Self     : in     SUBCLASS);

  function  Self       (This_Object : CLASS) return SUBCLASS;

private

  type CLASS is
    record
      Self : SUBCLASS;
    end record;

end Root;
```

Thus the mystery is resolved: the `Self` functions all ultimately access a `Self` component defined in the root class.

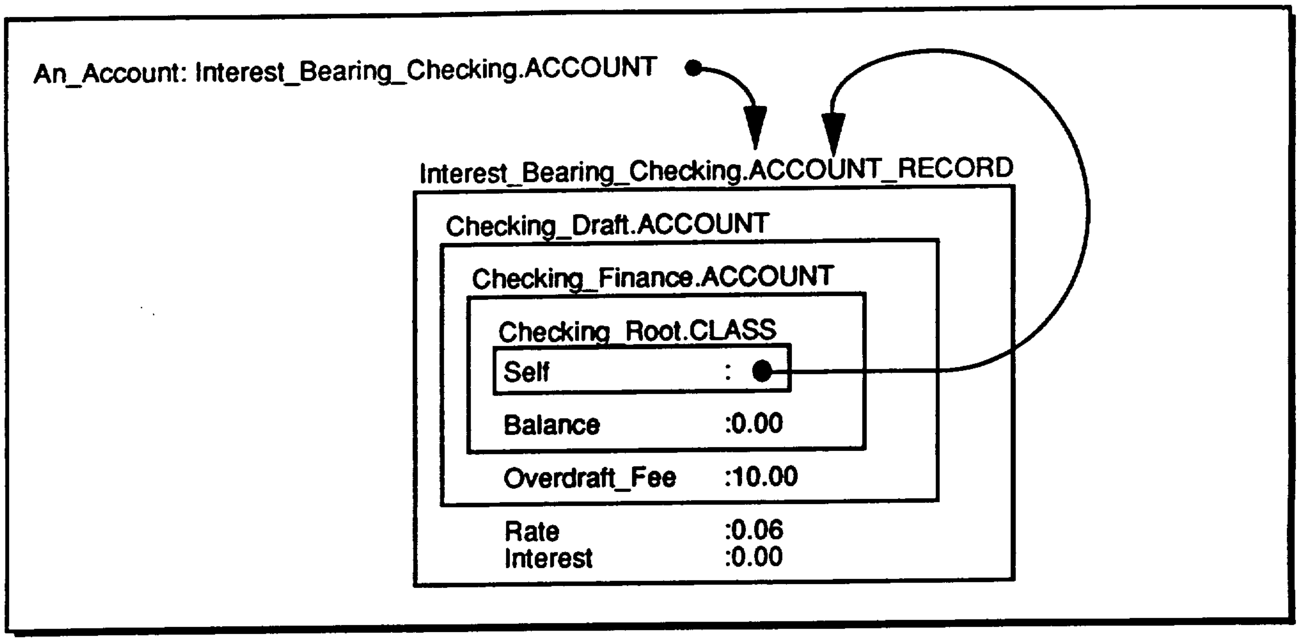
Now, the astute reader may have noticed that we have introduced a strange sort of circularity here. The representation of any class built on the root class will include a component of the subclass type. However, when we finish constructing a class from the root class and mixins, the result is the very subclass with which we need to instantiate the root class to begin with! To achieve this circularity, we must require that the subclass type be an access type. The `Self` component is then intended to be a *pointer* back to the complete, composite subclass object. (Actually, access types are also needed to allow the `Self` functions to work properly with subclass procedures that would otherwise have `in out` parameters.)

With inclusion of subclass parameters in mixins, we can now correctly implement the interest-bearing checking account class using either order of mixin instantiation:

```
with Finance_Types; use Finance_Types;
package Interest_Bearing_Checking is

  type ACCOUNT is limited private;

  procedure Open     (The_Account  : in out ACCOUNT;
                      With_Balance : in     MONEY);

  procedure Close    (The_Account  : in out ACCOUNT);

  procedure Set_Rate (Of_Account   : in     ACCOUNT;
                      To_Rate      : in     RATE);

  ...

private

  type ACCOUNT_RECORD;
  type ACCOUNT is access ACCOUNT_RECORD;

end Interest_Bearing_Checking;
```

An advantage of implementing a private type as an access type is that the details of the type representation can be deferred to the package body by using an incomplete type definition for ACCOUNT_RECORD in the private part of the specification. The use of an access type also allows the use of `in` rather than `in out` parameters in procedures such as `Set_Rate`, which is necessary for the use of `Self` functions.

Circular type definition is also achieved using the incomplete type definition for ACCOUNT_RECORD. The circle is closed by completing the definition of ACCOUNT_RECORD after all the mixin instantiations in the package body. The figure on the next page shows the structure of an `Interest_Bearing_Checking.ACCOUNT` object resulting from the following implementation:



```
with Root, Monetary, Interest, Draft;
package body Interest_Bearing_Checking is

  package Checking_Root is
    new Root (SUBCLASS => Interest_Bearing_Checking.ACCOUNT);

  use Checking_Root;
  package Checking_Finance is
    new Monetary
      (SUPERCLASS     => Checking_Root.CLASS,
       SUBCLASS       => Interest_Bearing_Checking.ACCOUNT);

  use Checking_Finance;
  package Checking_Draft is
    new Draft
      (SUPERCLASS     => Checking_Finance.ACCOUNT,
       SUBCLASS       => Interest_Bearing_Checking.ACCOUNT);

  function Balance_Of (The_Account : in Checking_Draft.ACCOUNT)
    return MONEY;
  -- call-through to Finance.Balance_Of

  use Checking_Draft;
  package Checking_Interest is
    new Interest
      (SUPERCLASS     => Checking_Draft.ACCOUNT,
       SUBCLASS       => Interest_Bearing_Checking.ACCOUNT);

  type ACCOUNT_RECORD is new Checking_Interest.ACCOUNT;

  ...

end Interest_Bearing_Checking;
```

(Note that to simplify the instantiations, I have taken advantage of the box defaults on the generic formal subprogram parameters of the mixins.)

A disadvantage of using an access type is that interest-bearing checking accounts must be explicitly allocated. We can do this as part of the Open operation:

```
procedure Open (The_Account   : in out ACCOUNT;
                With_Balance  : in     MONEY) is
begin

  if The_Account /= null then
    Close (The_Account);
  end if;

  The_Account := new ACCOUNT_RECORD;
  Checking_Root.Initialize
    (The_Object => The_Account.Parent.Parent.Parent,
     To_Self    => The_Account);
  Checking_Finance.Open (The_Account.Parent.Parent, With_Balance);

end Open;
```

Note the use of the root `Initialize` operation to set the `Self` component. The figure on the previous page shows the structure of nested records and self reference that results from the allocation and initialization of an `Interest_Bearing_Checking.ACCOUNT` object.

We also need to provide a way to deallocate interest-bearing checking accounts:

```
procedure Free is new Unchecked_Deallocation (ACCOUNT_RECORD, ACCOUNT);

procedure Close (The_Account  : in out ACCOUNT) is
begin
  Free(The_Account);
end Close;
```

All the rest of the interest-bearing checking account operations are inherited from one or the other of the mixin instantiations.

SUPERTYPES

Subtyping is a substitutability relationship, i.e., an instance of a subtype can stand in for an instance of its supertype. How the subtype is implemented is totally irrelevant; all that matters is that it have the right behavior so that it can be substituted.

[Lalonde and Pugh 91]

Typically, the customer of a bank will have several accounts at that bank. Each bank account may be, say, a savings account, a checking account or an interest-bearing checking account. To manage all the bank accounts of one customer, we would like to create a bank account type that is the *supertype* of the types that represent the various classes of accounts. We could then create lists of bank accounts, define bank account operations, etc.

As discussed in the previous sections, each class is implemented in Ada by a private type that is distinct from all other class types. Nevertheless, we can still explicitly create a bank account supertype:

```
with Savings, Checking, Interest_Bearing_Checking;
with Finance_Types;  use Finance_Types;
package Bank is

  type ACCOUNT_TYPE is (SAVINGS, CHECKING, INTEREST_CHECKING);

  type ACCOUNT (Kind : ACCOUNT_TYPE := SAVINGS) is
    record
      case Kind is
        when SAVINGS           => A_Savings_Account   : Savings.ACCOUNT;
        when CHECKING          => A_Checking_Account  : Checking.ACCOUNT;
        when INTEREST_CHECKING => An_Interest_Checking_Account
                                    : Interest_Bearing_Checking.ACCOUNT;
      end case;
    end record;

  procedure Open        (The_Account  : in out ACCOUNT;
                         With_Balance : in     MONEY);

  procedure Close       (The_Account  : in out ACCOUNT);
  procedure Deposit     (Into_Account : in out ACCOUNT;
                         The_Amount   : in     MONEY);

  procedure Withdraw    (From_Account : in out ACCOUNT;
                         The_Amount   : in     MONEY);

  function  Balance_Of (The_Account  : ACCOUNT) return MONEY;

end Bank;
```

The type Bank.ACCOUNT defines a supertype with *subtypes* Bank.ACCOUNT(SAVINGS), Bank.ACCOUNT(CHECKING) and Bank.ACCOUNT(INTEREST_CHECKING). Each subtype corresponds to one of the classes defined in previous sections. Note that a private type is unnecessary here, because we wish to be able to freely convert between the Bank.ACCOUNT subtypes and the class types.

The five operations defined in package Bank reflect the operations that are common to all the account types. Semantically, we wish each supertype operation to mirror the implementation of the appropriate subtype operation. For example, the statement:

```
Bank.Withdraw (From_Account => A, The_Amount => X);
```

should be equivalent to either Savings.Withdraw, Checking.Withdraw or Interest_Bearing_Checking.Withdraw, depending on the subtype of A. Since the subtype of A can, in general, only be determined at run-time, we are effectively asking that Bank.Withdraw be *dynamically bound* to the appropriate subtype operation.

We can achieve the effect of dynamic binding in Ada by implementing the bank account operations as *dispatching* or *case-selection* subprograms. For example:

```
procedure Withdraw (From_Account  : in out ACCOUNT;
                    The_Amount    : in     MONEY) is
begin

  case Kind is
    when SAVINGS =>
      Savings.Withdraw (From_Account.A_Finance_Account, The_Amount);

    when CHECKING =>
      Checking.Withdraw (From_Account.A_Checking_Account, The_Amount);

    when INTEREST_CHECKING =>
      Interest_Bearing_Checking.Withdraw
        (From_Account.An_Interest_Checking_Account, The_Amount);

  end case;

end Withdraw;
```

Once we have the bank account supertype, we can create *polymorphic* data structures and operations that can handle all kinds of bank accounts. For example:

```
type CUSTOMER_ACCOUNTS is array(POSITIVE range <>) of Bank.ACCOUNT;

function Total_Assets_Of (The_Accounts : CUSTOMER_ACCOUNTS) return MONEY is

  Total : MONEY := 0.00;

begin

  for I in The_Accounts'range loop
    Total := Total + Bank.Balance_Of (The_Accounts(I));
  end loop;

  return Total;

end Total_Assets_Of;
```

The function defined above finds the total assets a customer has in his accounts, regardless of what kinds of accounts they are.

It is important to note that to be included in a supertype, a class need only provide implementations for all the operations defined for the supertype. The ways in which various subtype classes implement these operations do not have to be related at all. For example, the `Bank.ACCOUNT` supertype is constructed from a number of classes implemented by various combinations of the mixins `Monetary`, `Interest` and `Draft`. These classes thus share some common implementation, but this is not at all important to the construction of the supertype.

Thus, supertypes and superclasses are really distinct concepts. Looking at it another way, the supertype provides a set of dispatching operations for those operations which are common to all its subtypes, regardless of how those operations may be implemented by the subtype classes or how the subtypes may be represented. A supertype that is constructed in this way from a given list of subtype classes is said to be the *union type* of those classes. Thus we have constructed a bank account supertype that is the union of the savings, checking and interest-bearing checking account classes.

It was noted earlier that the use of mixins causes a collapse of the original class hierarchy. Using union types, however, we can still form a type hierarchy by appropriately grouping classes. As well as the `Bank.ACCOUNT` union type, such a *type* hierarchy for account classes could include the union of the savings and interest-bearing checking account classes (an investment supertype treating interest-bearing checking accounts as savings accounts) and the union of the checking and interest-bearing checking account classes (a cash account supertype treating interest-bearing checking accounts as checking accounts). Note how it is possible for a class to be included in more than one union type.

CONSIDERATIONS FOR ADA 9X

There is a recognized need for improving Ada's support for data abstraction, and the construction of programs from pre-existing components.

[Ada9X 91a]

The mixin-based style described in this paper combines the benefits of object-oriented mixins with the advantages of explicit parameterization through generics. With superclass and subclass parameterization, mixins are completely independent software components that can be mixed and matched in many combinations. This leads to a powerful paradigm known as parameterized programming that promotes highly reusable code (see, for example, [Goguen 84; Seidewitz and Stark 91]).

Unfortunately, as the reader can see from the examples in this paper, this style is awkward in places with Ada 83. In particular, the following areas especially need to be addressed in Ada 9X:

1. There needs to be a way to create a subclass type by simple extension of a class type and to parameterize this extension with a mixin. The proposed Ada 9X record extension mechanism [Ada9X 91b] fills this need admirably well.

2. There needs to be a simpler way to achieve self-reference during the combination of mixins. This need seems to be filled by the proposed mechanism in Ada 9X to allow type extensions as generic formal type parameters [Ada9X 91b]. This would probably necessitate the use of nested generics to allow the mixin type to be an extension of the `SUPERCLASS` type parameter and the `SUBCLASS` type parameter to be an extension of the mixin type. Such a construction would, however, eliminate the need for `Self` functions.

3. There needs to be a mechanism for constructing supertypes without having to explicitly code dispatch operations. Ada 9X does provide an automatic dispatching capability using "tagged records" [Ada9X 91b]. However, this capability can only be used if the subtypes are implemented as subclasses (type extensions) of the supertype. This perpetuates the confusion of superclass and supertype.

Thus the proposed object-oriented features for Ada 9X largely support the mixin style described in this paper. Unfortunately, the tagged record mechanism confuses type extension and dispatching. This is analogous to the equation of superclasses and supertypes in most typed object-oriented programming languages (such as C++ [Stroustrup 86]).

Requiring supertypes to be superclasses is inconvenient when we are using generic mixins to construct classes. and wish to create a type hierarchy after the fact. Perhaps a better model for Ada 9X would be the "abstract type" mechanism of the languages Emerald [Black et al. 87] and POOL-I [America and van der Linden 90]. Even with the currently proposed Ada 9X features, however, a generics-based approach to mixins, such as that presented in this paper, could be an important contribution of Ada 9X back to the object-oriented programming community.

ACKNOWLEDGEMENT

I would like to thank my colleague Mike Stark for a number of good suggestions that greatly improved the clarity of presentation of this paper.

REFERENCES

Ada9X 91a	*DRAFT Mapping Rationale Document*, Ada 9X Project Report, February 1991
Ada9X 91b	*Ada 9X Mapping Document*, Draft Ada 9X Project Report (2 volumes), August 1991
America and van der Linden 90	Pierre America and Frank van der Linden, "A Parallel Object-Oriented Language with Inheritance and Subtyping", *Proceedings of the Conference on Object-Oriented Programming System, Languages, and Applications / European Conference on Object-Oriented Programming, SIGPLAN Notices*, October 1990
Black et al. 87	Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy and Larry Carter, "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, January 1987
Bracha and Cook 90	Gilad Bracha and William Cook, "Mixin-Based Inheritance", *Proceedings of the Conference on Object-Oriented Programming System, Languages, and Applications / European Conference on Object-Oriented Programming, SIGPLAN Notices*, October 1990
Goguen 84	Joseph A. Goguen, "Parameterized Programming", *IEEE Transactions on Software Engineering*, September 1984
Goldberg and Robson 83	Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983
Lalonde and Pugh 91	Wilf LaLonde and John Pugh, "Subclassing ≠ Subtyping ≠ Is–a", *Journal of Object-Oriented Programming*, January 1991
Rentsch 82	"Object-Oriented Programming", *SIGPLAN Notices*, September 1982
Seidewitz 91	Ed Seidewitz, "Object-Oriented Programming through Type Extension in Ada 9X", *Ada Letters*, March/April 1991
Seidewitz and Stark 91	Ed Seidewitz and Mike Stark, "An Object-Oriented Approach to Parameterized Software in Ada", *Proceedings of the Eighth Washington Ada Symposium*, June 1991
Stroustrup 86	Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986
Wegner 87	Peter Wegner, "The Object-Oriented Classification Paradigm", in *Research Directions in Object-Oriented Programming*, ed. by Bruce Shriver and Peter Wegner, The MIT Press, 1987