JOHNSON GRANT
IN-61-CR
243109
P-21

4

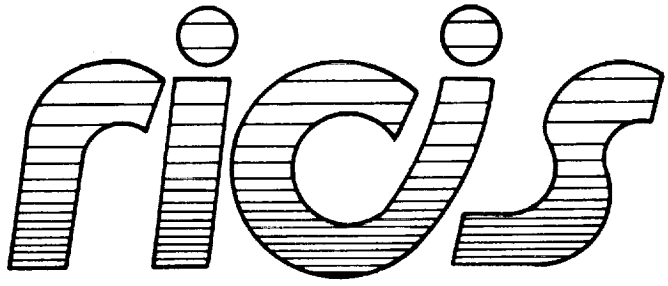
Object Oriented Programming Systems (OOPS) and Frame Representations, an Investigation of Programming Paradigms Final Report

David Auty

SofTech, Inc.

July 31, 1988

Cooperative Agreement NCC 9-16
Research Activity No. AI.9



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

Object Oriented Programming Systems (OOPS) and Frame Representations, an Investigation of Programming Paradigms Final Report

Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by David Auty of SofTech, Inc. Terry Feagin, Professor of Computer Science at the University of Houston-Clear Lake served as the technical representative for RICIS.

Funding has been provided by the Spacecraft Software Division, within the Mission Support Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between NASA Johnson Space Center and the University of Houston-Clear Lake. The technical monitor for this activity was Robert Shuler, Head, Systems Integration Section, System Development Branch, NASA/JSC.

FINAL REPORT ON
OBJECT ORIENTED PROGRAMMING SYSTEMS (OOPS) AND FRAME REPRESENTATIONS, AN INVESTIGATION OF PROGRAMMING PARADIGMS

Prepared for:

NASA/JSC/FR4
NASA Cooperative Agreement NCC 9-16

Submitted to:

Dr. Terry Feagin, Principal Investigator
University of Houston, Clear Lake
2700 Bay Area Boulevard
Houston, TX 77058-1096

RICIS Report AI.9

Prepared by:

David Auty, Principal Investigator
SofTech, Inc.
1300 Hercules Drive, Suite 105
Houston, TX 77058-2747

SofTech Document HO-003

Final Report

Object Oriented Programming Systems (OOPS) and Frame Representations, An Investigation of Programming Paradigms

July 31, 1988

Prepared for NASA/JSC/FR4
UHCL/RICIS Report AI.9
SofTech Document HO-003

INTRODUCTION

The project which led to the development of this report was initiated to research Object Oriented Programming Systems (OOPS) and Frame Representation systems, their significance and applicability, and their implementation in or relationship to Ada. "Object Oriented" is currently a very popular conceptual adjective. Object oriented programming, in particular, is promoted as a particularly productive approach to programming; an approach which maximizes opportunities for code reuse and lends itself to the definition of convenient and well-developed units. Such units are thus expected to be usable in a variety of situations, beyond the typical highly specific unit development of other approaches. Frame representation systems share a common heritage and similar conceptual foundations. Together they represent a quickly emerging alternative approach to programming.

Our approach is to first define our terms, starting with relevant concepts and using these to put bounds on what is meant by OOPS and Frames. From this we have pursued the possibilities of merging OOPS with Ada which will further elucidate the significant characteristics which make up this programming approach. Finally, we briefly consider some of the merits and demerits of OOPS as a way of addressing the applicability of OOPS to various programming tasks.

Definition of Terms:

As stated by [Seidewitz], there are three essential aspects to Object Oriented Programming which we address here: encapsulation, inheritance and a specific form of dynamic binding. A forth term often associated with OOPS is polymorphism. We will address this and other aspects which relax the rules of strong type checking under the discussion of dynamic binding. These concepts will serve to provide a framework within which we can better understand the notions of OOPS and Frames.

Encapsulation: The first concept of significance within our research was that of the encapsulation of data and processing which relates to that data. An object is best understood as such, as an encapsulation of code and data. Encapsulation is by no means unique to OOPS, but it is essential to the definition of an OOPS language.

The conceptual adjective "object oriented" is centered on the notion of encapsulation. An object is often defined as being a representation of some aspect of the "problem space" (the system definition being addressed by a particular program design) which has some "state of existence" (a set of persistent data which characterizes the object) and some processing in reaction to changes within the system. The processing is usually thought of as being directly influenced by and influencing the object's state (its data). The power of such encapsulation is that it aids in partitioning the system and focusing the development effort. The extent that such an approach reduces the complexity of interaction between program elements is one measure of its merit.

Encapsulation is central to object orientation, but is only one part of what makes up an object oriented programming system. In fact it is only the combination of all three of the concepts being presented here which provides the essential power of OOPS. The encapsulation of data and processing, providing an object-oriented approach, is needed; a hierarchy of definitions, as derived from the early work on frame representation systems, is needed; and as described following, a special form of dynamic binding is needed to fully realize object oriented programming.

Inheritance: The second subject of discussion addresses a significant common characteristic of OOPS and Frames, that of inheritance of properties within a hierarchy of definitions. Both OOPS and Frames share the notion of characterizing something in terms of its common aspects, which it derives from a common heritage and which may be shared with many other entities, and those aspects which distinguish it from others.

For our purposes, this can be recognized as a process of information modeling. The process is that of the taxonomist, who attempts to properly categorize and organize items into a hierarchy which collects the pertinent common aspects into higher levels of definition and pushes distinguishing detail to the bottom elements. And, for our purposes, this will be considered the essential aspect of a frame based representation system.

The notion of a frame can be traced back to the originating work of [Minsky]. In this work, the frame was thought of as a vehicle for capturing a piece of knowledge necessary for interpreting a given situation. The common and simple organization of frames is the hierarchy as described above, so that further up in the hierarchy more common and generalized knowledge is represented, while further down in the hierarchy, more specialized and specific knowledge is represented.

While the concept of frames has developed and evolved in different directions since that time, a simplifying approach is to distinguish between systems which have as their central goal a knowledge data base or knowledge representation of some kind, and those which have a similar hierarchical structure but which have as their purpose more than just knowledge representation. In particular we distinguish between OOPS and Frames in this way. Frame representation systems allow for the establishment of a knowledge database, but require an auxiliary processing component, such as an inference engine for processing to be accomplished. OOPS, on the other hand, has as its fundamental purpose the organization of computational elements.

The focus of our project has been on programming paradigms and methods, thus the rest of this report will focus on OOPS without further consideration of Frames. There are interesting variations of frame representation systems, particularly in the area of attached processing which insures the correctness or integrity of the knowledge base, but these fall under the restricted model which separates the knowledge base from the processing components. The notion of hierarchical definitions serving as a modeling schema for knowledge within a system is an important contribution of frame representation systems, while variations have had less of an impact on programming methods.

Dynamic Binding: The last concept of importance then is that of a special form of dynamic binding. This dynamic binding distinguishes OOPS from other programming approaches and more general object oriented design approaches. The "specialized" dynamic binding which characterizes OOPS takes a specific approach to referencing supporting operations. Supporting operations here means the operations called within the body of procedure definition. In an OOPS language, an object which is passed in as a parameter is said to carry with it its own set of supporting operations. The compilation of an OOPS operation does not bind the supporting operation references to specific code, but rather generates a reference to the supporting operation which is associated with the actual parameter passed in.

Consider, for example, a sort routine. It performs a series of comparison operations and, for an internal sort, a series of swap operations (assignments within the array). In traditional languages if one has a sort procedure an array of one type, it must be rewritten and compiled as a separate procedure for a different type, even though the steps and the logical set of supporting operations are the same. In OOPS, one procedure defining the general algorithm and order of calls upon the supporting operations will do. When passed an array of integers, the integer supporting operations will be called. Similarly, when passed an array of floating point numbers, the floating point supporting operations will be called.

In an OOPS language, a given procedure or method can be compiled and put aside in a library. Then at some time in the future, if a new data type is defined which is "compatible" with the parameter type of the compiled procedure, that procedure can be called with the new data type. Compatible in this case means that the actual parameter carries with it the supporting operations which the procedure or method will call upon. In a traditional language only the data which can be passed to the supporting operations whose references are already compiled into the body of the procedure can be accepted. In an OOPS language, the replacement operations are carried with the new data item.

Dynamic binding works in concert with inheritance in the following way. An object may be defined in terms of a parent class definition giving its general characteristics, and some specific definitions which are unique to the object itself. The supporting operations may be defined by the parent class definition, or uniquely for the object itself. If there is a supporting operation defined at both levels, the object's unique definition overrides the parent class definition.

Dynamic binding and inheritance provide strong support for extendibility, and, consequently, for building upon existing software. A different aspect of

dynamic binding supports a flexible approach to dealing generically with different types of data. OOPS languages often make available a generic object type from which all other object types are derived. The generic object type is typically a consequence of the implementation, in which the generic object information is carried around for all objects in support of dynamic binding.

The generic object type can be used as a holder for objects of any type and thus can be used to create aggregates and collections of objects, both of arbitrary type and mixed type. In certain situations this can be beneficial when the purpose is to collect and organize, not to perform other type-specific operations. It is analogous to what can be done externally on private types in Ada, but the objects do not need to be of specific types.

Such loose collections and dynamic binding contribute to the polymorphic character of OOPS, which is the ability of procedures to be general, independent of type, or to act upon objects of differing types. This is not necessarily in conflict with the concepts of strong typing, although implementations often provide one to the exclusion of the other. OOPS systems vary most significantly in terms of their treatment of data typing and type checking constraints.

Relating OOPS to Ada:

We are now in a position to explore how OOPS relates to Ada by consideration of how one would implement the OOPS paradigms in the Ada language. We will present this by stepping through the same three concepts of OOPS discussed above and addressing them individually, an approach again drawn from [Seidewitz]. In fact to facilitate comparison, we will use two examples from that article. For each we will discuss in greater detail what capabilities each concept implies, then discuss alternatives and approaches to defining and translating an OOPS/Ada language.

Encapsulation: The first aspect of the system is simple encapsulation. This is an aspect which Ada supports well on its own. To merge OOPS with Ada, in fact, we must confine Ada's approach and adopt a notation which provides just the functionality of an OOPS language. Doing this provides simplification for the programmer, bringing the language closer to the specific form of encapsulation appropriate for OOPS processing.

To provide encapsulation all that is necessary is a language construct to combine the data and procedural elements. The construct should be treated, however, as a proper data type of the language, in the sense of acting as a template for the instantiation of multiple copies, allowing the definition of aggregate collections and in allowing the data objects to be passed to procedures as parameters.

There are two language features which come to mind when trying to implement this in Ada: records, as are commonly used for merging OOPS with languages like Pascal and C; and packages, which are unique to Ada and which already implement the encapsulation of data and procedures. We will rule out records in this case because there is no easy way to refer to executable units without incurring the significant overhead associated with tasking. Packages, however, while providing the desired encapsulation are not treated in the language as data types.

Thus no features of Ada map directly to the OOPS concept of encapsulation, although packages come very close. In fact, one might note that generic packages come even closer by providing the capability of instantiating multiple copies, but this is not a proper treatment as a data type; the other capabilities are missing.

The concept of encapsulation in OOPS is perhaps best represented in Ada as a particular form of package specification, containing a record type declaration which collects all of the data fields for the object and a listing of procedures and functions which correspond to the methods of the object. The record type can be used for creating instances of the data fields which would be true data objects, while the procedures and functions can be defined to provide operations on objects of this type.

If it is intended that the data fields be manipulated only by the methods of the object, the record type can be declared as a private type, or a private type can be used within a public record declaration to provide a mix of public and private fields. For defining OOPS objects, the package specification should contain nothing else. The package body should generally contain just the bodies of the method procedures and functions, although local data declarations would provide the functionality of class data fields offered in some OOPS languages. Other variations which extend the usage of package capabilities include the definition of class methods (procedures or functions in the package specification which do not take an object as one parameter but which may alter class data) and class initialization (package body).

It is not difficult to see that an OOPS/Ada notation can be defined, which, with a relatively simple translator, can be used to generate this particular form of Ada code. Figure one provides one example. Of course this does not yet address inheritance or dynamic binding.

```
-- The following is a proposed OOPS/Ada
-- declaration that can be translated to an Ada
-- equivalent.
Type Money is Float;

Class Finances is
  Assets : Money := 0.0;
  Debt   : Money := 0.0;

  Initial
  Function New_Account
    (Balance     : in Money) Return Finances;

-- Allowable Transactions on a Finances Object.

  Procedure Receive (Amount : in Money);
  Procedure Spend   (Amount : in Money);

-- Allowable Inquiries on a Finances Object.

  Function Cash_On_Hand    Return Money;
  Function Total_Received  Return Money;
  Function Total_Spent     Return Money;

End Finances_handler;
```

```
-- Here is true Ada code which does the same
-- thing:

Type Money is Float;

Package Finances_Handler is

  Type Finances is
    Record
      Assets : Money;
      Debt   : Money;
    end Record;

  Function New_Account
    (Balance  : in Money) Return Finances;
  Procedure Receive (obj     : in out Finances;
                     Amount  : in     Money);
  Procedure Spend   (obj     : in out Finances;
                     Amount  : in     Money);
  Function  Cash_On_Hand
     (obj : in Finances)   Return Money;
  Function Total_Received
     (obj : in Finances)   Return Money;
  Function Total_Spent
     (obj : in Finances)   Return Money;
End Finances_Handler;
```

Figure 1. Encapsulation in OOPS/Ada

Inheritance: Adding the feature of inheritance complicates the situation, but not entirely beyond the scope of what Ada can do. At the programming level, inheritance provides a shorthand for saying that the subclass has all of the same fields and methods as the superclass, and then has additional or overriding field and method definitions. While again, there is no exact equivalent in Ada, a form of Ada usage can provide similar results.

The concept of inheritance requires two aspects of implementation, one for the data fields and another for the methods. For the data fields, assuming the approach recommended for encapsulation, the record type declaration which collects all fields for the subclass can include a field referencing the record type of the superclass. This requires an additional level of referencing when accessing the superclass fields (for the subclass record and the superclass record in addition to the field itself), but provides the essential functionality.

For the methods, two approaches are possible. One is to know the superclass package name and reference the superclass method directly, passing it the superclass record contained within the subclass record. The second approach is to define a procedure (or function, but we will refer to just procedures for simplicity) within the subclass package, one for each of the superclass methods, which will accept the subclass record. The body of this procedure will simply extract the superclass record and call the superclass procedure with it. This latter approach has the advantage of treating a class as a self contained definition and not requiring detailed knowledge of the inheritance hierarchy, but introduces a bit of runtime inefficiency.

Given the above notation for supporting simple encapsulation, it is a small extension to provide for inheritance. In this case the translator can accept the notation of the second approach, yet implement the first approach for handling methods, allowing for the treatment of the class as self contained and yet avoiding the inefficiency. This requires the translator to know the inheritance hierarchy and look up superclass package names. Figure two provides an example of the notation and the equivalent Ada code.

Figures three and four provide the bodies of the respective specifications and some code samples displaying how these definitions would be used. In addition to the declarative unit for class definitions, the OOPS/Ada notation introduces a type constructor which allows for declaring objects of a certain class type, and a new operator ("^") which indicates object field reference or method invocation.

```
-- Example #2 of a class declaration using
-- inheritance.

Subclass of
  Finances
Class Deductible_Finances is

  Deductible_Debt : Money;

  Replacement Initial
  Function New_Account
     (Balance     : in Money)
     Return Deductible_Finances;

  Replacement
  Procedure Spend
     (Amount             : in Money;
      Deductible_Amount  : in Money);

  Function Total_Deduction Return Money;

End Deductible_Finances;
```

```
-- Example #2 in true Ada

With Finances_Handler;
Package Deductible_Finances_Handler is

  Type Deductible_Finances is
    Record
      Parent_fields : Finances_Handler.Finances;
      Deductible_Debt : Finances_Handler.Money;
    end Record;

  Function New_account (Balance : in Money)
    Return Deductible_Finances;

  Procedure Spend
     (obj : in out Deductible_Finances;
      Amount  : in Money;
      Deductible_Amount : in Money);

  Function Total_Deduction
     (obj : in Deductible_Finances)
    Return Money;

End Deductible_Finances_Handler;
```

Figure 2. Inheritance in OOPS/Ada

```
Class Body Deductible_Finances is

  Function New_Account
     (Balance     : in Money)
     Return Deductible_Finances is
  Begin
     Return ( ^^New_Account (Balance),
              -- the superclass function
              Deductible_Debt => 0.00 );
  end New_Account;

  Procedure Spend
     (Amount             : in Money;
      Deductible_Amount  : in Money) is
  Begin
    Self^^Spend(Amount);  -- the superclass's
    ^Deductible_Debt :=
       ^Deductible_Debt + Deductible_Amount;
  end Spend;

  Function Total_Deduction    Return Money is
  Begin
    Return ^Deductible_Debt;
  end Total_Deductions;

end Deductible_Finances_handler;
```

```
Package Body Deductible_Finances_Handler is

  Function New_Account
    ( Balance     : in     Money )
    Return Deductible_Finances is
  Begin
    Return
      (Finances_Handler.New_account (Balance),
       Deductible_Debt => 0.00 );
  end New_Account;

  Procedure Spend
     (obj : in out  Deductible_Finances;
      Amount  : in  Money;
      Deductible_Amount : in  Money) is
  begin
    Finances_Handler.Spend
       (obj.parent_Fields,  Amount);
    obj.Deductible_Debt
       := obj.Deductible_Debt+Deductible_Amount;
  end Spend;

  Function Total_Deduction
    (obj : in Deductible_Finances)
    Return Money is
  begin
    Return obj.Deductible_Debt;
  end Total_Deductions;

end Deductible_Finances_Handler;
```

Figure 3. Class Bodies

```
Declare
  Jane_Doe : Object ( Deductible_Finance,
    New_Account ( Balance => 500.00) );
End;
Procedure Transaction_Set
  (Jane_Doe : Object ( Deductible_Finance)) is
begin
  -- Buy a red dress for work purposes

  Jane_Doe^Spend
    (Amount=> 69.95, Deductible_Amount=> 69.95);

  -- Do Lunch with an associate

  Jane_Doe^Spend ( Amount => 3.50 + 1.00,
        Deductible_Amount => 3.50 + 1.00 );
              -- Fettucini & Garlic_Bread,

  -- Receive Pay Check

  Jane_Doe^Receive ( Amount => 500.00 );

  -- Calculate Taxes

  Taxes := 0.20 * (Jane_Doe^Total_Received
                   - Jane_Doe^Total_Deduction);

end Transaction_Set;
```

```
Use Deductible_Finances_Handler;
Declare
  Jane_Doe : Deductible_Finances
          := New_Account (Balance => 500.00);
End;
Procedure Transaction_Set
  (Jane_Doe : Deductible_Finances) is
begin
  -- Buy a red dress for work purposes
  Spend (obj     => Jane_Doe,
         Amount => 69.95,
         Deductible_Amount => 69.95);

  -- Do Lunch with an associate
  Spend (obj     => Jane_Doe,
         Amount => 3.50 + 1.00,
         Deductible_Amount => 3.50 + 1.00);

  -- Receive Pay Check
  Finances_Handler.Receive
      ( obj     => Jane_Doe.Finances,
        Amount => 500.00 );

  -- Calculate Taxes
  Taxes :=
     0.20 * ( Finances_Handler.Total_Received
              (obj => Jane_Doe.Finances)
  -Deductible_Finances_Handler.Total_Deduction
              (obj => Jane_Doe) );
end Transaction_Set;
```

Figure 4. Class Use in OOPS/Ada

Dynamic Binding: The last feature to add is that of dynamic binding. Unfortunately, dynamic binding is a much more fundamental characteristic of a language than the two previous features and thus is more difficult to implement as a translation into existing features of the Ada language.

For accessing supporting operations, dynamic binding requires that a set of supporting operations be associated with each object and that a mechanism be implemented to call these supporting operations. Ada provides no direct way within the language to call one of a set of procedures selected at runtime (i.e. arrays of procedures or procedure parameters). The options available are to implement a complex mechanism involving intermediate procedures and case statements, or to use a combination of tasking features. In either case the resulting overhead is burdensome.

In Ada, the closest approximation to such dynamic operations binding is with Generics. We have already seen how generics fail to properly support the implementation of encapsulation. Their use in this case is quite a bit different. A more abstract perspective of dynamic binding indicates that all that is required is to be able to handle a new data type declaration with a previously defined set of operations. The Generic mechanism of Ada supports this by allowing the declaration of a new set of operations based on a previously compiled generic template. While the syntax implies that a whole new set of operations is being created, and most implementations actually follow this model, for the restricted usage of generics required in this case the mechanism of dynamic binding could be the underlying implementation.

Figure 5 outlines how this would work if we extended the example of Figures 1-4 to include a new class (New_Finances) which provides the same external specification as the class finances. In an OOPS language an object of the new type could be passed to procedure transaction_set without complications. Transaction_set calls upon a set of procedures provided by the actual parameter passed. Dynamic Binding requires only that the same set of supporting operations be provided.

In Ada, transaction_set would have to be compiled as a generic with the procedures provided by the Finances class as generic procedure parameters. In this way transaction_set could be instantiated for any set of actual procedures which collectively implement a given class. While this approach is sufficiently general to handle the intent of dynamic binding it quickly becomes quite cumbersome, with both excessive compile-time and run-time inefficiencies with most if not all implementations of Ada.

In Figure 5 the OOPS/Ada procedure Transaction_Set is redefined as an Ada generic procedure based on its use of a class derived object type as a formal parameter. This is followed by the OOPS/Ada code and its Ada equivalent to introduce and work with a new class definition.

```
Class New_Finances is
-- same protocol as Class Deductible_Finances
-- (with inherited components)
End New_Finances;


Declare
  John_Doe : Object (New_Finances);
Begin
  Transaction_Set (John_Doe);
End;
```

```
Generic
  Type Deductible_Finances is limited private
  With Procedure Spend ...
  With Procedure Receive ...
  With Function  Total_Received ...
  With Function  Total_Spent ...
Procedure Transaction_Set
  (John_Doe : in Deductible_Finances) is
Begin
-- same body as before
End Transaction_Set;

Package New_Finances_Handler is
  Type New_Finances is Record...
-- like specification for Deductible_Finances_
-- Handler with inherited components
End New_Finances_handler;

New_Finances_Transaction_Set
  is new transaction_Set
     (Deductible_Finances =>
        New_Finances_handler.New_Finances
      ...);
Declare
  John_Doe : New_finances_handler.New_Finances;
Begin
  New_Finances_Transaction_set (John_Doe);
End;
```


Figure 5. Handling New Types

Applicability:

If one steps back from these implementation details and asks when is OOPS most applicable for system development in differing application domains, a different perspective emerges. While throughout this report we have been considering what distinguishes OOPS from other approaches, we now consider how these differences affect the applicability of the approach.

It is important to recognize that while the "object oriented" methods may be applied throughout the life-cycle of software development, OOPS refers specifically to a class of programming languages. From this perspective the issues of applicability of OOPS are issues of software development, after much of the design has been done, i.e., the choice of OOPS vs. some more traditional language is mostly a coding decision. At this stage in development the concerns are, for example, of code reuse, reliability, test and integration effort and efficiency.

From the earlier sections of this report it is clear that while there are differences in the fundamental capabilities of different OOPS languages, the OOPS approach in general addresses the modularity and interconnection of units, not the fundamental data types and operations. Thus while one can talk more specifically about different OOPS languages, in general only issues of modularity can be discussed.

OOPS has as its great appeal the potential for code reuse. As was illustrated above with the sort example, a given algorithm once coded has a far wider potential for reuse. Any subsequently defined data type can be passed in, so long as it has the necessary supporting operations defined, without having to modify the algorithm or recompile its code.

Note however, that this potential for reuse has its savings in the elimination of coding time but does not eliminate the need for detailed unit testing of the module in the particular environment of its reuse. With traditional approaches to testing, once a unit has been developed and has been through unit testing, it is assumed correct for most any use. Unit and integration testing provide a bottom-up and top-down assessment of correctness, respectfully, which complement each other. While top-down testing provides assurance that the basic system level performance is correct, it cannot hope to test every path or circumstance for correctness. Unit testing, on the other hand, provides a more fundamental check on the correctness of a given unit, under a variety of conditions in which it might be called, providing a better check on the unusual circumstances, but not on the overall correctness of the system.

OOPS changes the nature of bottom-up unit testing. In more traditional languages, when a unit is tested, the supporting operations which the unit calls upon are part of that testing. They are not changed by a different set of actual parameters. For the same assurance of correctness, OOPS requires a unit-level testing for each set of actual parameter types, or more likely, a unit-level testing for each application of the unit.

Consider the case of a sort routine written in OOPS. An object, which represents a uniform collection of data and has a comparison operation defined for that data, may be passed in for sorting. However, if the comparison

operation itself is inconsistent or in any way fails to provide a single correct ordering of the data items, the sort algorithm may not even complete, let alone perform the sort correctly. This is an example of a functional dependency which is difficult to impossible to check for. A clear understanding and careful documentation is needed in addition to careful testing in each application.

As for efficiency, there are method invocation implementation approaches which perform much of the required processing pre-runtime, leaving only a few additional instructions above that required for a more traditional procedure call in most cases. Thus the issue in efficiency need not focus on the overhead of method invocation. A more subtle issue has to do with the difficulty of compiler optimizations. The dynamic binding of OOPS certainly precludes the possibility of inline code generation and optimizations which depend upon the good behavior of called procedures. For these reasons, OOPS may not be the first choice for low-level system code development.

The fact that OOPS provides alternatives in the organization and modularity of a system suggests that OOPS is most valuable at the higher levels of coding within a system, where the supporting operations are likely to be higher abstractions themselves and where the more general abstractions are more likely to be called from differing places in the system. At the higher levels in system, it is more likely that an operation will be inherited in the definition of a necessary refinement. At the lower levels of a system, the inability to optimize and the lack of any consideration for system level details are factors against the adoption of OOPS.

Other than these considerations there are few relevant aspects of OOPS, which are implementation independent, which would argue for or against its use. Of course, in a real-time application one would be concerned for the real-time support aspects of a given implementation, and there are many if not most implementations of OOPS which would be inappropriate. These implementations fail, however, because of factors which are not inherent in the OOPS concept itself.

Summary

In pursuing this project we have covered a lot of ground, much of which is only glossed over in this report. The list of references includes many important works upon which most of this paper is based. From these we have reached a certain understanding of what OOPS is and what the elements are upon which it is based.

Two areas remain for further consideration in the adoption of OOPS paradigms. The first has already been introduced, that of testing and reliability. This is an issue facing software reuse in any form, but it is of particular concern in the mechanisms of generics for Ada and dynamic binding for OOPS. The second concern has not been addressed in this report, but will need to be addressed before OOPS can be fully integrated into system development. This is how OOPS should interact with concurrency and multi-tasking.

In consideration of how Ada compares with OOPS capabilities we have shown that Ada provides much of what is needed, although only through preprocessing a notational extension would these capabilities be directly recognized in the language. The disadvantage of Ada is that while generics provide much of what is provided by dynamic binding, generics are a feature which must be added to other features, unlike dynamic binding which is built in and automatic. In addition, the typical implementation of generics results not in code reuse, but rather in a form of automated code generation. This is one more instance of requirements for flexible and sophisticated implementation support for generics in Ada.

In general, there is much to merit the object oriented approach to both design and code development. Like other high-level approaches to coding, object orientation can imply paradigms for the use of existing Ada features. It is not uncommon for such approaches to require cooperation of the compiler and other support software to provide efficient implementation. OOPS is an important approach to programming, one that deserves the further development in terms of the above issues and in terms of its support in Ada.

References

[Seidewitz] Seidewitz, Ed. "Object-Oriented Programming in Smalltalk and Ada", OOPSLA '87.

[Minsky] Minsky, Marvin. "A Framework for Representing Knowledge" in The Psychology of Computer Vision, edited by Patrick Henry Winston, McGraw-Hill, 1975.

[Meyer] Meyer, Bertrand. "Genericity Versus Inheritance", OOPSLA '86

[Snyder] Snyder, Alan. "Encapsulation and Inheritance in Object Oriented Programming Languages", OOPSLA '86.

[Cox] Cox, Brad. Object Oriented Programming - An Evolutionary Approach, Addison Wesley, 1986.

[Schmucker] Schmucker, Kurt J. Object-Oriented Programming for the Macintosh, Hyden, 1986.

Addendum I

On Data Typing and Operations Bindings

In untyped languages, the definition of data and operations are kept separate from each other, in the sense that operations are defined in terms which are independent of the actual interpretation of the data. Thus, for example, a floating point divide may be performed on any double-word value whether that value was originally assigned as a double-length integer, fixed point value, character string or in fact as a floating point number. A typed language adds the notion that values have associated with them an interpretation which defines their meaning. The interpretation can be generalized as the data's type. With this definition of data type is associated a certain set of operations which make sense for that data type. A strongly typed language imposes restrictions so that only proper operations are performed on values, based on required declarations for all data objects which specify their type.

Where dynamic binding makes the most significant difference is in the definition of new operations, i.e., procedures or methods, which accept parameters. Most definitions of new operations make some significant assumptions about what operations may be carried out on the parameters. In a strongly-typed language, the type of the parameter is declared in the formal declaration of the operation (i.e., the operation's external interface). Thus only actual parameters of the specified type are permitted. In this case the declaration of the parameter's type formally defines the set of allowed operations which the body of the new operation may call upon. Whether or not the parameter types are formally defined, however, a new operation definition is a composition of other operations, in a certain order, and with iteration and branching, etc. to arrive at its intended processing conclusion.

In a "traditional" language (e.g. FORTRAN, Pascal, C), the body of a new operation contains references to supporting operations which are statically bound when the definition is compiled. The binding is either to built-in operations of the underlying machine, or to a specific subroutine of instructions for which an address can be determined prior to execution. So long as the actual parameters passed at runtime contain the right data type (an aspect guaranteed by a strongly type language) the operation can be assumed to generate the correct result.

A Program Derived from Example 3 of [Seidewitz] in OOPS/Ada and Ada

```
Class Collection ( Max_Size : positive ) is

  -- Max_Size is a discriminant for this class

  -- implements a loose (mixed types) collection
  -- of objects which maintains order.  Access
  -- is by index based on insertion order

  Function  Add     (item : in Object)
    Return Natural;
  Procedure Replace(index: in Positive,
                    item : in Object);
  Function  At      (index: in Positive)
    Return Object;
  Function  Size    Return Natural;

Private

  current_size : ... ;
  Data         : ... ;

End Collection;

-----
With Class Collection;
Class Sample_Set ( Max_Size : Positive ) is

  Procedure Set
    (new_data : in Object (Collection));
  Function  Sample Return Object;
  Function  Samples(n         : in Natural )
    Return Object (Collection);

  Empty : Exception;

Private

  Data : Object (Collection (Max_Size));
  Remaining_Size : Natural;

End Sample_Set;

-----

With Random;
Class Body Sample_Set is
. . .
  Function Sample Return Object is
    Item  : Object;
    Index : Natural;
  Begin

    If ^Remaining_Size = 0 Then
      Raise Empty;
    End if;

    Index := Natural ((Random.Value
                       * ^Remaining_Size) + 1 );
    Item  := data^at(Index);

    Data^Replace
      (Index, Data^At (^Remaining_Size));

    ^Remaining_Size := ^Remaining_Size - 1;
    Return Item;
  End Sample;
. . .
End Sample_Set;
```

```
-- Code which uses Sample_Set :
Subclass of
  Collection
Class Card_Deck is

  Procedure Renew;
  Function  Deal Return Card;
  Procedure Return (returnee : card);
  Procedure Shuffle;

Private

  Type suit is (spade, heart, diamond, club);
  Type face is (ace, king, queen, jack,
       c10, c9, c8, c7, c6, c5, c4, c3, c2 );
  Type card is
    record
      a : suit;
      b : face;
    end record;

  Deck  : object (collection (max_Size => 52));
  -- filled with cards in procedure renew
  top,
  bottom: deck_index;

End Card_Deck;

With Class Collection, Sample_Set;
Class Body Card_Deck is

  Procedure renew is
     temp : object (collection (max_size => 52);
  begin
     for i in suit loop
       for j in face loop
         temp^add ( object (card'(i,j)) );
       end loop;
     end loop;

     ^deck  := temp;
     ^top   := 1;
     ^bottom:= 52;
  end renew;

  ...

  Procedure Shuffle is
     temp : Object (Sample_Set(max_size => 52));
  Begin
     temp^Set ( ^deck );
     ^deck := temp^Samples (n => 52);
  End Shuffle;

End Card_Deck;
```