5-1-1990

What does modula-2 need to fully support object oriented programming?

Joseph Bergin

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT

Number 30, May 1990

What Does Modula — 2 Need to Fully Support Object Oriented Programming?

JOSEPH BERGIN
Computer Science
Pace University, New York, NY

STUART GREENFIELD
Division of Computer Science and Mathematics
Marist College, Poughkeepsie, NY

PACE UNIVERSITY

Joseph Bergin is Professor of Computer Science at Pace University with his office on the New York campus. Before coming to Pace he had been Professor of Computer Science at Marist College. He holds a doctorate in Mathematics from Michigan State University. His research interests lie in the area of programming languages, especially object oriented languages. He is currently working on a textbook titled Data Abstraction and Object Pascal, portions of which he plans to classroom test this Fall in CS131.

Stuart Greenfield is Assistant Professor of Computer Science at Marist College. He holds a degree in electrical engineering and is working on a doctorate in computer science. With one book already to his credit, Invitation to Modula-2, and another on compiler construction underway; his research interests are also in programming languages. At this point he, too, is especially interested in object oriented languages.

"What Does Modula-2 Need to Fully Support Object Oriented Programming?" appeared initially in SIGPLAN Notices (a monthly publication of the ACM Special Interest Group on Programming Langauges): Volume 23, Number 3; March, 1988; pages 73-82. A year later it was re-printed by the editors of the Journal of Object Oriented Programming: Volume 1, Number 6; March/April, 1989; pages 31-38.

What Does Modula-2 Need to Fully Support Object Oriented Programming?

Joseph Bergin
Computer Science
Pace University, New York, NY

Stuart Greenfield
Division of Computer Science and Mathematics
Marist College, Poughkeepsie, NY

One of the present authors has been developing an object oriented extension to Modula-2 since the first of this year, and the other has a strong attachment to standard Modula-2 This paper arose from answering the question: "How close is Modula-2 to being object oriented?" As it turns out the answer is: "Quite close indeed "

Object oriented programming is a discipline of software development which embraces strong encapsulation of the data and procedural elements, the notion of inheritance from classes (types) and/or objects (instances of types), and the idea of overloading of method (procedure) names Additionally, a paradigm of message passing may be adopted Some languages provide more support, and some less support, for these ideas We will consider each of these in turn

I Strong Encapsulation

Strong encapsulation within *objects* provides a method by which the program text and the run-time system may enforce the strong linkages between semantic entities and the operations which manipulate them To be truly object oriented, a language must also provide syntactic support for the notion of *classes* of objects Classes are similar to types, primarily defining some notion of structure The objects within a single class have a structure and a behavior determined by that class In Smalltalk [1] this encapsulation is provided by a mechanism whereby the implementation of an entity such as a queue and the operations which operate on the queue are defined within the same language construct (Ada [2] provides a similar facility with the package construct, and Modula-2 [3] with modules)

In an object oriented programming language (OOPL) the notion of an Abstract Data Type, represented as a collection of well-defined actions on a (perhaps hidden) collection of data structures, must be supported by language constructs which clearly "enclose" these elements and separate them from other elements Pascal [4] had no such facility (although Object Pascal [5], an extension of UCSD Pascal, does support encapsulation) A queue is defined in Pascal by giving several distinct declarations, each of which defines some aspect of the queue The data structure declaration is separate from the declaration of insert, which is also separate from the declaration of each other procedure

Two additional attributes which have come to be closely tied to object oriented programming are *inheritance*, (subtyping), and overloading of the *method* (procedure) names

II Inheritance

Inheritance may be defined in various ways The two main notions are "inheritance from classes" and "inheritance from objects" In inheritance from

classes, objects are created within a class and derive their properties (structural and procedural) from the class They do not however derive specific values (states) from the class While classes may themselves be objects, not all objects are classes Classes may derive some of their properties from other classes forming a hierarchy (e g Smalltalk) or directed network (e g Trellis/Owl [6]) of related classes

The other major definition of inheritance, "inheritance from objects", does not distinguish between objects and classes, and any object may inherit properties (structural, procedural, as well as state values) from any other This notion is called *delegation* and one object is said to delegate responsibility for a property to another object An object may decide to always be the same "color" as another object, but to have other distinct properties such as "position" or "spin" The first object is then an *exemplar* for the second This notion is supported in several languages (e g ThingLab [7]) but will not be further discussed here This is not a judgement about the usefulness of the idea (quite the contrary) but a desire to focus on the notion of "inheritance from classes" as is done in Smalltalk-like languages

III Overloading of Names

The idea of overloading method (procedure) names comes from the strong desire in OOPL to make the syntax of a specific program match the semantics to the greatest degree possible If we have a program that must use several different structures of the same general form (e g Queue), but each structure has different particulars, (e g array representation, list representation, queue of integer, queue of instantiation record), it is semantically useful to be able to refer to the operations on them by the same names (e g insert, remove) and not have to rely on cleverly constructed variations on these names Such naming is typically redundant in its use

For example, when we say in Pascal, PushReal(myStack,someItem), redundancy results from the fact that the system knows that "PushReal" works for "stack of reals" only, that "myStack" is indeed a "stack of reals" and that "someItem" is indeed a "real" This redundancy is useful in strongly typed languages In fact it is the main reason for using strong compile time typing, so that the compiler may check that we "got it right" However some of the inherent redundancy may also be used to reduce the syntactic overload required In Pascal, if we must also have "stack of integer", "stack of ", we will then be saying "PushInteger" or whatever, although the notion of Push is independent of most of that sort of detail Overloading of procedure names, using one name, "Push", for a whole collection of procedures, is a means of factoring out common properties (a semantic notion) and making the syntax match the meaning The system itself may be made to remember that Push(myStack,someItem) must refer to the procedure which operates on stacks of reals because myStack and someItem are consistent only with that procedure

IV Message Passing (and Information Hiding)

A complete notion of object oriented programming then must include these three properties: Strong methods of encapsulation, inheritance (here using inheritance from classes) and overloading or factoring of names To this we shall add the message passing paradigm In most object oriented programming systems the objects conceptually must be considered to be autonomous entities which have "knowledge" of their own representations and their own allowable actions (strong encapsulation) A program generates changes of state by *sending messages* (method names plus parameters) to objects The names are of course

overloaded, so that the object itself will "figure out" which actual procedure should be executed to respond to the message It is not clear that this is essential to the notion of object oriented programming, but this style of writing (and thinking) will be adopted here and incorporated as a fourth property of OOPL A push message can be sent to the stack myStack as: myStack push(someItem) myStack receives the message, decides that it knows how to "push" and then it "does the right thing" with someItem, modifying its own internal state In some systems all messages generate a reply In others they do not The reply is similar to a function result returned to the caller

A corollary of the message passing paradigm is that information hiding must be enforced on the state variables of the objects If an object is autonomous, and behaves by responding to messages then it is inconsistent to allow other parts of a program to manipulate the state variables directly This may be enforced by programmer discipline, but a better way is to get help from the language itself by means of appropriate scoping rules for names Block structured languages of the Pascal family are all deficient in this regard and artificial means must generally be used to achieve it Thus, in Modula-2 the stacks discussed above could be opaque types, preventing clients from manipulating the state variables, but then we would be restricted to the use of implementations that represented the stack as a pointer (though, admittedly, it could be a pointer to an array)

V How Close is Modula-2 to Being an OOPL?

This question will be approached in two ways First, we will present an example in standard Modula-2 which demonstrates what may be done simply by adopting an OOPL style When one adds ideas of record type extension as discussed by N Wirth in March 1987 at the ACM SIGCSE conference and his subsequent article [8] we will see that we are almost there To show this we will present a second example in a hypothetical language which adopts Wirth's record type extensions We then analyze the resulting language in terms of the adequacy of its support for OOPL concepts

We would first like to answer the question: "Why objects in Modula-2?" This question has two sides: Modula-2 is very good in its design domain, and several excellent OOPL's exist A number of reasons for wanting to extend Modula-2 could be given, based on various criteria One important reason is that the basic structure of the language is very sound and not so extensive that it may not be mastered readily The strong compile time typing of Modula-2 (also present in some but not all OOPLs) is another reason The general usefulness of Modula-2's overall support for software engineering (including separation of specification and implementation, separate compilation) is important in our view Another reason is that the vast number of programmers who grew up loving Pascal would be able to easily adapt to such a language A final reason is because we like the language conceptually for its beauty, orthogonality and relative lack of shortcomings

VI Programming in the OOPL Style Using Modula-2

Using DEFINITION MODULEs and record and procedure types we may achieve strong encapsulation and name overloading, and simulate a message passing style in a primitive way Inheritance and information hiding are the difficult items The examples below are not an attempt to define an object oriented style within Modula-2, but to show what the language lacks to support the concepts fully

As we can see in Listing 1, there is some awkwardness in trying to provide a subclass mechanism The standard syntax requires extra redundancy in trying

to refer to that part of the definition of a QUEUEwithLength which was inherited from QUEUE We must refer to the original insert operation by qwl^ inheritedqueue^ insert( ) to get "inside" the inherited part Some things could be done differently, using variant records, but to this we cry "foul" as it is not proper in true software engineering (where an important issue is reusability of functionality) to go back and fix the old stuff so that it is consistent with the new We would rather add the new stuff, keeping unchanged whatever we can of the old

We also see a bit of extra redundancy in the "message passing" method The message refers to the recipient twice, once as the first symbol in the expression and again as the first actual parameter in the call In fact the first symbol is actually an explicit procedure choosing mechanism, not a message recipient We declare that we wish to execute the procedure q^ insert, not that the object q is being sent the message "insert"

```
DEFINITION  MODULE  OOPQues;
  (*  EXPORT  QUALIFIED  QUEUE, INIT, nodeptr, itemtype;  *)
   TYPE  itemtype=INTEGER;
      nodeptr=  POINTER  TO  node;
      node  =  RECORD
                  info:itemtype;
                  next:nodeptr;
            END;
      QUEUE=POINTER  TO  queue;
      inserttype  =  PROCEDURE  (VAR  QUEUE, itemtype);
      removetype  =  PROCEDURE(VAR  QUEUE,  VAR  itemtype);
      emptytype  =  PROCEDURE(QUEUE):BOOLEAN;
      queue  =  RECORD
                   front, rear:nodeptr;
                   insert:inserttype;
                   remove:removetype;
                   empty:emptytype;
                END;
   PROCEDURE  INIT(VAR  q:  QUEUE  (*in  out*));
END  OOPQues

IMPLEMENTATION  MODULE  OOPQues;
   FROM  Storage
   FROM  InOut
    PROCEDURE  Empty(q:QUEUE(*in*)):BOOLEAN;
      BEGIN
         IF  q^ front = NIL THEN
        RETURN  TRUE;
       ELSE
        RETURN  FALSE;
       END;
      END Empty;
    PROCEDURE  Insert  (VAR  q:QUEUE  (*in  out*);  item:itemtype(*in*));
       (*  the queue is defined to have a FIFO discipline  *)
      VAR  p:nodeptr;
      BEGIN
       NEW(p);
        p^ info:=item;
        p^ next  :=  NIL;
         IF  Empty(q)  THEN
          q^ front:=p;
```

```
      ELSE
         q^ rear^ next:=p;
      END;
       q^ rear:=p;
    END Insert;
  PROCEDURE Remove (VAR q:QUEUE(*in out*); VAR item:itemtype(*out*));
        (* remove the item at the front if any such exists *)
    END Remove;
  PROCEDURE INIT(VAR q:QUEUE(*in out*));
   BEGIN
      q^ front:=NIL;  q^ rear:=NIL;
      q^ insert := Insert;          (* Note that different procedures could *)
      q^ remove:=Remove;            (* be inserted here, effectively providing *)
      q^ empty:=Empty;              (* name overloading of the names *)
   END INIT;                  (* insert,remove, and empty  *)
 END OOPQues

DEFINITION MODULE Inherit;
  FROM OOPQues IMPORT QUEUE;
  (* EXPORT QUALIFIED QUEUEwithLength,INITmodifiedqueue; *)
  TYPE   lengthtype=PROCEDURE(QUEUE):CARDINAL;
       QUEUEwithLength=POINTER TO queuewithlength;
       queuewithlength=RECORD
                        inheritedqueue:QUEUE; (* attempt at inheritance *)
                       length:lengthtype;       (* extend with a new method *)
                     END;
  PROCEDURE INITmodifiedqueue(VAR q:QUEUEwithLength(*in out*));
END Inherit

IMPLEMENTATION MODULE Inherit;
FROM OOPQues IMPORT QUEUE, INIT, nodeptr,itemtype;
FROM Storage
  PROCEDURE Length(q:QUEUE(*in*)):CARDINAL;
   VAR   p:nodeptr;
         c:CARDINAL;
    BEGIN
     p:=q^ front;
     c:=0;
     WHILE p#NIL DO
         p:=p^ next;
         INC(c);
     END;
     RETURN c ;
    END Length;
  PROCEDURE Insert(VAR q:QUEUE(*in out*);n:itemtype(*in*));
   (* redefines the discipline to LIFO *)
   VAR p:nodeptr;
   BEGIN
    NEW(p);
    p^ info:=n;
    IF q^ empty(q) THEN
        p^ next:=NIL;
        q^ rear :=p;
    ELSE
         p^ next:=q^ front;
    END;
    q^ front:=p;
```

```
      END Insert;
    PROCEDURE INITmodifiedqueue(VAR q:QUEUEwithLength(*in out*));
   BEGIN
      INIT(q^ inheritedqueue);
      q^ inheritedqueue^ insert:=Insert;  (* install the changed method *)
      q^ length:=Length;     (* install the new method *)
    END INITmodifiedqueue;
END Inherit

MODULE OOPQueueTest;
FROM OOPQues IMPORT QUEUE,INIT;
FROM Inherit IMPORT QUEUEwithLength,INITmodifiedqueue;
FROM Storage
FROM InOut
VAR q:QUEUE;
      qwl:QUEUEwithLength;
      n:INTEGER;
      i,len:CARDINAL;
BEGIN
    NEW(q); INIT(q);
     NEW(qwl); INITmodifiedqueue(qwl);
    FOR i:=1 TO 5 DO  (* insert items into each queue*)
      ReadInt(n);
      q^ insert(q,n);      (* send q and qwl the insert "messages" *)
      qwl^ inheritedqueue^ insert(qwl^ inheritedqueue,n);
      WriteLn;
   END;
    len := qwl^ length(qwl^ inheritedqueue); (* the length message to qwl *)
    WriteCard(len,0); WriteLn;
    WHILE NOT q^ empty(q) DO   (* delete items *)
      q^ remove(q,n);
      WriteInt(n,10);
   END;
     WHILE NOT qwl^ inheritedqueue^ empty(qwl^ inheritedqueue) DO
     qwl^ inheritedqueue^ remove(qwl^ inheritedqueue,n);
     WriteInt(n,10);
    END;    (* note the more complex syntax in this while to get the "remove"*)
END OOPQueueTest
```

Listing 1

Two other difficulties are, first, the need for separate initialization routines (INIT and INITwithLength), with separate names, and second, the lack of any protection mechanism for the state variables of the queues defined More will be said about these problems below The reader should note however that (1) encapsulation is very strong, (2) information hiding is non-existent, (3) overloading is fine except for the name INIT, and (4) message passing and inheritance are here, but in a weak form

VII Incorporating Wirth's Record Type Extensions

Wirth, in proposing extensible types, however, opens the door a great bit further As an example of his record type extension, we may have a record type ALPHA with components alpha and tau Another type BETA might want to adopt all of the structure of ALPHA but supplement it with an element beta The syntax for doing this would be

```
TYPE
     ALPHA = RECORD
               alpha :    ;
               tau :    ;
             END;

     BETA = RECORD (ALPHA)
               beta:   ;
             END;
```

Records of type BETA will then have three components alpha, beta and tau Also, BETA will inherit structure from ALPHA BETA is called a *subtype* of ALPHA Furthermore any object of type BETA is considered to be an item of type ALPHA Thus the notion of type here is different from the strict notions in Modula-2 or Pascal This is the notion of type extension, or specialization Items of type BETA are assignment compatible with names associated with type ALPHA because they are in fact of that type

To have a *type extension*, in the strict semantic sense, however, the user of this construct must be careful In its strongest form subtyping or type inheritance has the following meaning: "BETA is a SUBTYPE of ALPHA provided that wherever a value of type ALPHA is required, a value of type BETA may be utilized " Thus we may (almost) say that LION is a subtype of ANIMAL because it is (almost) true that sentences that use the word ANIMAL (in its generic sense) are meaningful if the word is replaced with LION (Animals are motive, Animals reproduce but not, of course statements that refer to some but not all animals)

The import of this is that the extension construct described by Wirth must be used by the programmer in such a way that defining a subtype declares a "specialization" of a type and not a fundamental change in the meaning of the type Thus a subtype of QUEUE that declared fields and other elements in such a way that the behavior was that of something other than a queue would not be proper Note that this is exactly what has been done in Listing 1, as insert was changed to a LIFO discipline If we were interpreting "Queue" narrowly (i e FIFO) then we would have broken the subtype discipline Not all languages that permit inheritance of types enforce subtype discipline, except perhaps syntactically, and programmers usually have ways around the intentions of the language designer in any case (We note that it is sometimes useful to break the subtype discipline, when it is functionality that we wish to extend and not meaning)

However, if this record type extension were to be added to Modula-2, (perhaps replacing the variant record structure, which is a weaker notion) then the first example becomes much simpler and much closer to the object oriented style Listing 2 describes the first example in a hypothetical language E_Modula-2, which has E_MODULES, and, except for including record type extensions, is identical to Modula-2

As we see from Listing 2 we now have a much cleaner message passing syntax The extra field reference is no longer needed, as the fields of a queuewithlength are all at the same level, but we still need to refer to the recipient twice Encapsulation is still very good The inheritance structure is sound Name overloading is almost right

```
DEFINITION  E_MODULE  OOPQues;
   (*  EXPORT  QUALIFIED  QUEUE, queue, INIT, nodeptr, itemtype;  *)
```

```
    (* Same as the DEFINITION MODULE OOPQues of Listing 1  *)
END  OOPQues

IMPLEMENTATION  E_MODULE  OOPQues;
   FROM Storage
   FROM InOut
     ( Same as the IMPLEMENTATION MODULE OOPQues of Listing 1  *);
   END INIT;
END  OOPQues

DEFINITION  E_MODULE  Inherit;
   FROM  OOPQues IMPORT queue;
    (* EXPORT QUALIFIED QUEUEwithLength,INITmodifiedqueue;  *)
    TYPE    QUEUEwithLength=POINTER TO queuewithlength;
          lengthtype=PROCEDURE(QUEUEwithLength):CARDINAL;
         queuewithlength=RECORD(queue)     (* a subtype record, much nicer
inheritance*)
                          length:lengthtype;
                        END;
    PROCEDURE  INITmodifiedqueue(VAR  q:QUEUEwithLength(*in  out*));
END  Inherit

IMPLEMENTATION  E_MODULE  Inherit;
FROM  OOPQues  IMPORT    INIT,  nodeptr,itemtype;
FROM  Storage

    PROCEDURE  Length(q:QUEUEwithLength(*in*)):CARDINAL;
  VAR    p:nodeptr;
        c:CARDINAL;
  BEGIN
     p:=q^ front;
     c:=0;
     WHILE  p#NIL  DO
          p:=p^ next;
          INC(c);
     END;
     RETURN c ;
  END Length;

   PROCEDURE  Insert(VAR  q:QUEUEwithLength(*in  out*);n:itemtype(*in*));
     (* redefines the discipline to LIFO *)
  VAR p:nodeptr;
  BEGIN
     NEW(p);
     p^ info:=n;
     IF  q^ empty(q)  THEN
          p^ next:=NIL;
          q^ rear :=p;
     ELSE
          P^ next:=q^ front;
     END;
     q^ front:=p;
  END Insert;

   PROCEDURE  INITmodifiedqueue(VAR  q:QUEUEwithLength(*in  out*));
  BEGIN
     INIT(q);
```

```
      q^ insert:=Insert;  (* Simpler  syntax  here   *)
      q^ length:=Length;
   END INITmodifiedqueue;
END  Inherit

E_MODULE  OOPQueueTest;
FROM  OOPQues  IMPORT  QUEUE,INIT;
FROM  Inherit  IMPORT  QUEUEwithLength,INITmodifiedqueue;
FROM  Storage
FROM  InOut

VAR  q:QUEUE;
     qwl:QUEUEwithLength;
     n:INTEGER;
     i,len:CARDINAL;
BEGIN
    NEW(q); INIT(q);
    NEW(qwl); INITmodifiedqueue(qwl); (* It is no simpler here, though  *)
   FOR i:=1 TO 5 DO   (* insert items into each queue*)
     ReadInt(n);
     q^ insert(q,n);       (* send q and qwl the insert "messages" *)
     qwl^ insert(qwl,n);    (* much better here *)
     WriteLn;
   END;
    len  :=  qwl^ length(qwl); (* improved length message to qwl *)
    WriteCard(len,0); WriteLn;
    WHILE NOT q^ empty(q) DO   (* delete items *)
     q^ remove(q,n);
     WriteInt(n,10);
   END;
    WHILE NOT qwl^ empty(qwl) DO   (* nicer here *)
     qwl^ remove(qwl,n); (* much improved here *)
     WriteInt(n,10);
   END;
END  OOPQueueTest
```

Listing 2

VIII What is still lacking in Modula-2?

What is still lacking for us to have a minimally useful OOPL? The last item mentioned in the section above is one clue Procedure name overloading is relatively straightforward for every method except for the INIT method which is separately needed for every class The fact is that Modula-2 suffers from one important deficiency as a language for software engineering in environments of many programmers in which implementers of functionality want to provide safe ways for clients to use that functionality The flaw is in separating the physical allocation mechanism for constructs (pointed to constructs especially) and the initialization mechanism, which makes them safe [9] We should not have to separately say

```
     NEW        (or ALLOCATE  )
     INIT
```

to create a new item For if a client forgets the protocol and does not initialize, we will have problems There ought to be a single mechanism, logical or semantic in nature, by which both of these functions may be achieved The implementer of the functionality will then be able to guarantee the safe use of the data types provided Thus, to avoid this problem in Modula-2 and to permit

full name overloading, a way must be found to initialize objects "automatically" so that the client need not be concerned The implementer must necessarily be able to specify the nature of the initialization and to guarantee that it will be done for each object created (Note that we could have overloaded the name INIT But to do so would require using another procedure whose name could not be overloaded The problem can be moved, but it stays around) Solving this problem would be very easy and straightforward

A harder problem is the double redundancy of the message recipient We have still used an explicit procedure selection mechanism, which appears statically within the program text It would be preferable to make this implicit, as the name of the recipient must appear somewhere, and it only knows one procedure by the given name So that either a syntax like insert(q,item) or q insert(item) would be preferable to q insert(q item)

Another problem arises if we are to consider this subtype mechanism as a means of defining *classes of objects* The scoping rules for visibility of names in Modula-2 is not consistent with the needs of strong encapsulation with information hiding It was, of course, designed for procedural abstraction and not explicitly for data abstraction In object oriented programming we are more interested in *entity abstraction* If the objects such as QUEUEs are to truly be encapsulated then they ought to be able to completely control all access to their internal state variables (of course the implementation should be irrelevant if the data type is truly "abstract") Thus the data components (not the procedural components) should be entirely invisible outside of the inheritance hierarchy We should not be able to refer to the "front" field of the QUEUEwithLength, except from within those procedures which implement the QUEUEwithLength To not insist on this is to build only half of a dike Careful programmers will avoid using implementation details as they do now in older languages, but clever programmers will find "tricky" ways to be "more efficient" and the encapsulation mechanism will fail Thus, an object oriented extension to Modula-2 must provide a way to hide the implementation details while revealing the "procedural specification" or interface One could change the scope rules to achieve this or one could find a way to hide some of the details in the IMPLEMENTATION MODULEs However, a solution which breaks the specification of a class of objects into two parts, with some in the DEFINITION MODULE and the rest in the IMPLEMENTATION MODULE is objectionable on conceptual grounds A language providing encapsulation should provide a way for the syntax to "surround" the specification of an item The lack of such syntax in many languages (e g Pascal) causes major difficulties in correctly and carefully implementing abstract data types as it requires several distinct specification statements to completely specify a single abstract data type

IX Conclusion

What would you have with a language which added such features to Modula-2? The list of benefits for maximally reusing old software, and for building big stuff in multi-person environments is quite long: Strong compile time checking, separation of specification from implementation, improved information hiding, sound inheritance aiding software reuse, separate compilation, and easily modularized design Modula-2 supports most of these things now Adding objects and classes in the correct way would improve on the language

Work is underway to develop a language along these lines which also exhibits more advanced features including generic classes, multiple inheritance, private classes, opaque classes, and, perhaps, moving of some of the type checking burden to the run time environment in certain controlled circumstances

Acknowledgment

The authors thank the Marist College Software Engineering Research Fund for providing support for this work

References

[1] Adele Goldberg, David Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley, 1983
[2] American National Standards Institute (1983) Military Standard Ada Programming Language ANSI/MIS-STD-1815A-1983
[3] Niklaus Wirth, *Programming in Modula-2*, Third, Corrected Edition, Springer-Verlag, 1985
[4] Kathleen Jensen and Niklaus Wirth, "Pascal User Manual and Report", Second Edition, Springer-Verlag, 1974
[5] Larry Tesler, "Object Pascal Report", Apple Computer, 1985
[6] Craig Schaffert, Topher Cooper et al , "An Introduction to Trellis/Owl" OOPSLA/86 Proceedings ACM September, 1986
[7] Alan Borning, "Classes Versus Prototypes in Object-Oriented Languages" ACM/IEEE Fall Joint Computer Conference, Nov 1986 (ThingLab)
[8] Niklaus Wirth, "Extensions of Record Types", SIGCSE Bulletin V19,N2, June 1987
[9] Stuart Greenfield and Roger Norton, "Detecting Uninitialized Modula-2 Abstract Objects" SIGPLAN Notices, 22(6) June 1987

This paper previously appeared in SIGPLAN Notices and in the Journal of Object Oriented Programming

PACE UNIVERSITY
NEW YORK • WESTCHESTER

OFFICE OF THE DEAN
SCHOOL OF COMPUTER SCIENCE
AND INFORMATION SYSTEMS

THE EVELYN AND JOSEPH I. LUBIN
GRADUATE CENTER
1 MARTINE AVENUE
WHITE PLAINS, N.Y. 10606-1909
TELEPHONE: (914) 422-4375
FAX: (914) 422-4019

April 25, 1990

Dr. Allen Stix
Computer Science Dept.
Pace University
School of Computer Science
and Information Systems
1 Martine Avenue
White Plains, NY

To the Editor:

This is a congratulatory note on the rebirth of the Technical Reports. You have done a fine job!

Without reference to the content of the first paper (in which I have an interest), I note that the volume is very well done; there is a high quality about it including editorial information, previous publication information, attention to copying rights, page numbers and so forth. Your editor's note assists us in categorizing the kinds of scholarly activity that we can initiate or continue.

Thank you very much for your outstanding leadership. I look forward to a long and successful life for the series.

Sincerely,

Susan M. Merritt
Dean

/jm

Editor's Note

May, 1990

The manuscript you hold in your hands has a distinguished history. It was written as a report on technical research and, consistent with its scholarly identity, was sent to and printed by SIGPLAN Notices (March 1988; Volume 23, Number 3; pages 73-82). There it was seen by the editors of the Journal of Object Oriented Programming, a slick-looking though eminently serious magazine, who co-opted it for their own publication (March/April 1989; Volume 1, number 6; Pages 31-38).

I am particularly pleased to be re-printing it in Technical Reports because the teaching of object oriented ideas and research into objected oriented programming and systems design directly involves both computer science and information systems. Not only is the potential present for collaborations across disciplinary lines, but the frontiers of the object oriented territories are still close by and wide open. Seeing that there is developed expertise "already in the family" in the form of Joe means that, in case you did not know, you have an accessible sounding board for new thinking and a source of help. (This is not to neglect a great deal of expertise in the form of other family members as well. There was even an earlier Technical Report, from March 1988, entitled "Getting Started with Smalltalk in Smalltalk/V: A Tutorial" by your's truly.)

My hope is that this reaches you in time for the suggestion of writing a paper over the summer to alight on the fertile soil of late spring. Don't forget, Technical Reports welcomes working research reports, the sharing of assignments and/or instructional pedagogy that worked particularly well, syntheses of the literature, reviews of realms of software, and discussions of the new adoption of technology from one realm in another. Of course trail-blazing discussions of truly revolutionary significance, or of merely paradigm-shaking significance, are also warmly greeted and likely to be found acceptable. I'll be expectantly watching my mail....

Allen Stix
Department of Computer Science
301 Alumni House
The College of White Plains of Pace University
78 North Broadway
White Plains, NY 10603

Telephone: 914 422-4191