A Comparison of Object-Oriented Programming in Four Modern Languages

Robert Henderson Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

CU-CS-641-93 Revised July 1993

University of Colorado at Boulder

Technical Report CU-CS-641-93
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Robert Henderson Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430

A Comparison of Object-Oriented Programming in Four Modern Languages*

Robert Henderson Benjamin Zorn
Department of Computer Science
Campus Box #430
University of Colorado, Boulder 80309–0430
(303) 492-4398

Revised July 1993

Abstract

Object-oriented programming has become a widely-used, important programming paradigm that is supported in many different languages. The paper evaluates Oberon-2, Modula-3, Sather, and Self in the context of object-oriented programming. While each of these programming languages provide support for inheritance, dynamic dispatch, code reuse, and information hiding, they do so in very different ways and with varying levels of efficiency and simplicity. A single application was coded in each language and the experience gained forms the foundation on which the subjective critique is based. By comparing the actual run-times of the language implementations, we also present an objective analysis of the efficiency of current implementations of the languages. The application was also coded in C++, thereby providing a well-known baseline against which the execution times could be compared.

1 Introduction

Object-oriented programming has become a widely-used, important programming paradigm that is supported in many different languages. In this paper, we evaluate how several modern programming languages support object-oriented programming and specifically examine features that provide inheritance, dynamic dispatch, code reuse, and information hiding. We also compare the run-time efficiency and compilation time of widely-available implementations of the languages. The programming languages considered are Oberon-2, Modula-3, Sather, and Self. C++ is also included as it provides a well-known baseline for comparison. To facilitate a fair comparison between the languages, a single application was selected and coded in each language. The experience gained coding this application provided the foundation on which the comparison was made. This single

*This material is based upon work supported by the National Science Foundation under Grant No. CDA-8922510.

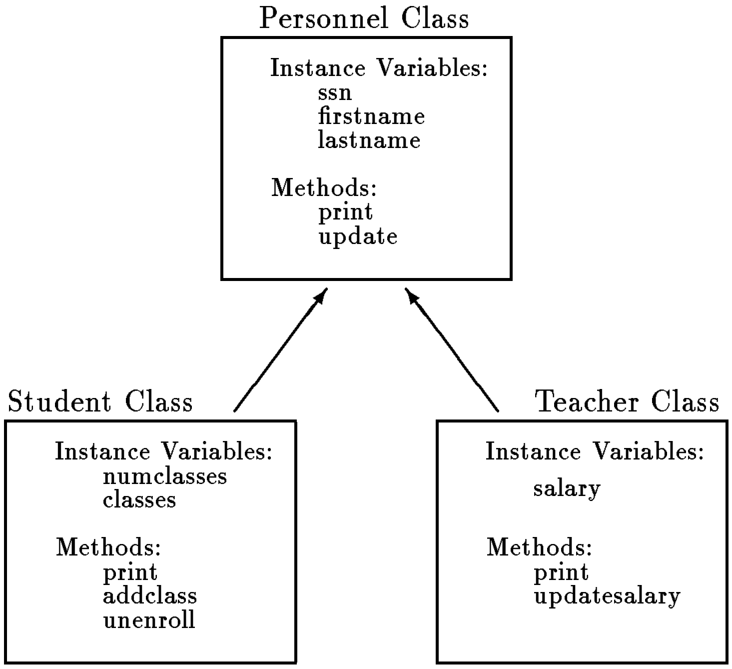

Figure 1: The Database Application Class Hierarchy

application also made it possible to objectively compare the run-time efficiency of the language implementations. In the remainder of this section we describe the application that we used to compare the languages and provide an overview of each of the languages selected.

1.1 The Application

The application selected was a database of university personnel files. The database includes files for students and teachers, both of which contain a social security number and name. Student files also contain information about courses in which the student is enrolled while teacher files include salary information. Classes with inheritance provide the ideal framework for the implementation of this hierarchy. Personnel files provide the parent class from which student and teacher files can be subclassed. The database is implemented as a linked-list of personnel files. The classes and the instance variables and methods they provide are shown in Figure 1.

We selected this application specifically for the purpose of evaluating and comparing the object-oriented programming features mentioned above. While the application program benefits from features that support object-oriented programming, it is not overly complex. We intentionally chose a simple application for two reasons. First, simplicity made it possible to write implementations in several different languages in a reasonable time. Second, the simplicity enhances the language comparison because the languages considered all provide support for the set of features needed by the application. If, for example, the application had required threads, it would have been more difficult to compare the implementations since support for threads is either vastly different between languages or non-existent. However, one drawback of using a simple application is that we cannot use it to compare the suitability of the languages for larger programming projects.

This application benefits from several features supported by the object-oriented paradigm. The first is simply the need for inheritance since the teacher and student classes both use instance variables and methods provided by the parent personnel class. The application also benefits from the ability to override the definition of the print method in the subclasses of the personnel class. To explore code reuse, we have the print methods for student and teacher classes call the print method defined in the personnel class (where the social security number is defined). Finally, the need for dynamic dispatch arises in the print method for the database class. Because the database is a linked-list of personnel files, when the print message is sent to the personnel file, we want the proper print method to be called depending on whether the personnel file is actually a student file or a teacher file.

This application actually resembles many others that are commonly-used when examples of object-oriented programming are presented. For instance, the structure of the application resembles that of an interactive drawing program where a number of geometric shapes and/or text are being manipulated. In that case, instead of a linked-list of personnel files, a list of different geometric shapes is manipulated. Likewise, in a text-formating program, such as the Interviews `doc` application, a box of text is represented as a sequence of possibly different glyphs. Finally, a simulation system that manipulates many different types of similar objects, such as an airport simulation, could have a structure very similar to that of our application.

The source code implementing the database application in each language is available on the Internet via anonymous ftp from ftp.cs.Colorado.EDU as /pub/cs/misc/comparison-code.tar.Z.

1.2 The Languages

This compares four modern languages that support object-oriented programming. In this section, we explain why we chose these languages and describe each briefly. First, all the languages chosen were created through an evolution of existing successful languages. Oberon and Modula-3 both evolved (in very different directions) from Modula-2, Sather evolved from Eiffel, Self evolved from Smalltalk. Thus, all these languages represent attempts to improve existing languages with new and very different ideas. Second, implementations of all the languages are publicly available making these languages easy to obtain and compare. Finally, these languages represent a broad spectrum of programming language design principles, ranging from emphasis on minimality and simplicity to emphasis on support for rapid prototyping. Of course, many other languages, such as CLOS and Objective-C, could also have been included, however we felt that investigating more than four languages was beyond the scope of a single paper. We include C++ in the descriptions below because it is used as a baseline for performance comparisons later in the paper.

C++ C++ [13] is a widely-used object-oriented programming language and provides a well-known baseline for comparison. C++ supports object-oriented programming by extending C with classes that group functions and data, provide information hiding, and support inheritance. C++ is statically-typed and attempts to significantly reduce the run-time overhead of object-oriented programming. Programmers who do not use the object-oriented features of C++ get the same performance they would using C. If programmers need dynamic dispatch in C++, they explicitly use "virtual functions," otherwise functions are statically bound.

Oberon Oberon [18] was designed by Niklaus Wirth as a successor to Modula-2 [17]. Oberon is largely the result of removing features from Modula-2, however a few features were added. The most important addition is *type extensions* [19], which provide basic support for inheritance. Oberon has evolved into Oberon-2, the most recent of the Oberon languages, and is described by Mossenbock and Wirth [9]. Oberon-2 was influenced by Object-Oberon [8] and incorporates *type-bound procedures*, which are equivalent to methods. Type-extensions and type-bound procedures provide the support for object-oriented programming generally provided by explicit classes in object-oriented programming languages. As with each of the other languages except C++, garbage collection is provided to free the programmer from the onerous task of memory management.

Modula-3 While Oberon represents a minimal evolution of Modula-2 to support object-oriented programming, Modula-3 [2, 4] is a larger language providing support for large, multi-person programming projects with separate module interface specification and exception handling as well as concurrency and garbage collection. Modula-3 is a strongly-typed language with an emphasis on safety. However, the safety features of the language can be explicitly circumvented when increased efficiency or functionality are necessary. Modula-3 supports object-oriented programming by providing *object types*, which are incorporated into the basic module structure of the language.

Sather Sather [11, 10] was derived from Eiffel with an emphasis on simplicity and efficiency. While Oberon-2 and Modula-3 are procedural languages that provide support for both modules and objects, Sather places a central emphasis on objects and classes. Sather provides only classes as a way of grouping related data (no modules) and all functions must be methods in some class. While in many ways the philosophy of Sather is closer to that of Smalltalk and other purely object-oriented languages, Sather is interesting because it also emphasizes static-typing and performance.

Self Self [15] is a dynamically typed, object-oriented programming language with message passing that was designed to support exploratory programming. Self represents a deviation from organization by classes supported by most object-oriented programming languages. Instead of classes with instance variables and methods, Self incorporates *prototypes* with *slots*. Prototypes provide templates from which objects are cloned and slots combine instance variables and methods into a single construct. The Self implementation we used is also very interesting because there is a great deal of emphasis on using compiler optimization to reduce the overhead of the dynamic features. Our results indicate the success that the Self implementors have had in achieving this goal.

2 Related Work

A number of papers comparing the features of different object-oriented languages have been published recently. Wolf's comparison of object-oriented programming in C++ and Flavors is based on building systems for designing electronic hardware [20]. He qualitatively compares the languages' features, including support of objects, typing, memory management, etc. Because the languages compared are so different, only a minimal quantitative comparison of performance is included. In addition to having more quantitative performance comparisons, our paper also considers a greater number of approaches; we examine a broad spectrum of experimental approaches to object-oriented programming.

Blaschek *et al* compare the languages C++, Eiffel, Oberon, and Smalltalk-80 [1]. Their comparison includes a general description of each language and a point-by-point comparison of a number of specific features including inheritance, efficiency, complexity, and reliability. Schmidt and Omohundro compare the features for object-oriented programming in CLOS, Eiffel and Sather [12]. Their comparison includes language features, program performance, and available programming environments. While both of these comparisons are detailed and complete, they are not conducted in the context of a single specific example. Furthermore, our approach allows us to carefully investigate the cost of dynamic dispatch in each of the languages implementations evaluated, while theirs does not.

Other comparisons of object-oriented techniques have also appeared. Wileden, Clark, and Wolf evaluate object definition techniques in prototyping systems [16]. Their investigation focuses only on object definition and the utility of these techniques for building large prototype systems.

This paper differs from others like it because it compares a wide variety of experimental approaches to object-oriented programming in the context of a simple example. The simplicity of the example serves to highlight some fundamental differences in object-oriented programming supported by these languages. Furthermore this paper compares not only the language designs but also extensively compares the relative performance of existing implementations of features that support object-oriented programming in these languages.

3 Support for Object-Oriented Programming

In this section we discuss how each language supports inheritance, dynamic dispatch, code reuse, and information hiding. In doing so, we present simplified code examples from each language illustrating how a simple inheritance relation from our database application is implemented.

3.1 Inheritance

Inheritance is one of the key features of object-oriented programming and all of the languages studied provide support for it. Recall that Figure 1 illustrates the relation between the Student and Personnel classes in our database application. In the next four figures, we show how this inheritance relation is expressed in each of the languages we studied. We will return to these code examples as we discuss how the languages support different features in turn.

Figure 2 shows the Oberon-2 module declarations for the Personnel and Student modules. In Oberon-2, names are not exported explicitly in an interface section but instead exported names are indicated with a asterisk that appears after them. For example, the Print procedure is exported from the Personnel module. This form of specification allows a module interface to be automatically constructed.

```
------------------------- Personnel.Mod -----------------------------------

MODULE Personnel;
IMPORT Out, Texts;
TYPE
        File* = POINTER TO FileData;
        FileData* = RECORD
                ssn: INTEGER;
                firstname, lastname: ARRAY 32 OF CHAR;
        END;

(* Print method *)
PROCEDURE (pf: File) Print* ();
BEGIN
        (* Body of method omitted *)
END Print;

END Personnel.

------------------------- Student.Mod -------------------------------------

MODULE Student;
IMPORT  Out, Personnel;
TYPE
        File* = POINTER TO FileData;
        FileData = RECORD (Personnel.FileData)
                nclasses: INTEGER;
                class: ARRAY 10 OF INTEGER;
        END;

(* Override the parent Print method *)
PROCEDURE (sf: File) Print*;
BEGIN
        sf.Print^();           (* Invoke overridden print method *)
        sf.PrintStudentInfo(); (* Print student specific information *)
END Print;

END Student.

---------------------------------------------------------------------------
```

Figure 2: Personnel and Student Modules in Oberon-2.

Figure 2 also illustrates the use of type-extension in Oberon-2. Inheritance is supported by type-extensions that allow record data types to declared as *extensions* of previously declared types. Objects of the extended type include fields from both the base and extended types. In the example, the type Student.FileData extends the imported type Personnel.Filedata. Oberon-2's type-bound procedures make it possible to bind procedures to types, providing the mechanism for linking a type with its behavior. Since a type can be the extension of only a single other type, multiple inheritance is not possible.

Figure 3 shows the Modula-3 module implementation and interface declarations for the Personnel and Student modules. Unlike Oberon-2, module interfaces and interfaces are specified separately in Modula-3. As is immediately clear from the example, even relatively simple inheritance relationships are quite complex to express in Modula-3. Reasons for this complexity include the separation between module interfaces and implementations, mechanisms for information hiding, and the interaction between Modula-3 modules and objects. While describing this example entirely is beyond the scope of this paper, we will mention the object-oriented Modula-3 features used in this example both here and later in the paper.

Modula-3 provides *Object* types, which embody both a data record and a method suite. When an object type is declared (for example, in the interface of the Student module), another object type can be specified as the supertype, thus providing inheritance. An object type declaration can include only a single supertype so multiple inheritance is not possible.

Figure 4 shows the class declarations for the Personnel and Student classes in Sather. This example illustrates how the complexity of specifying an inheritance relationship is reduced if only classes and not modules with objects (or type-extensions) are supported in a language. In Sather, all code is organized into *classes* and a class definition must be contained within a single file. A class definition includes *attribute* specifications (instance variables) and *routine* specifications (methods). Inheritance is achieved by a conceptually simple model of textual inclusion. For example, notice that in the definition of the student class, the personnel class is named. This reference to PERSONNEL has the same semantics as if the text defining the personnel class were textually included in the definition of the student class. Since multiple classes can be named in a class definition, multiple inheritance is possible. By using the textual inclusion model, name conflicts between mul-

```
------------------------- Personnel.i3 ------------------------------------
INTERFACE Personnel;
    TYPE
        File <: Public_File;
        Public_File = OBJECT
            METHODS
                Print();
        END;
END Personnel.
------------------------- Personnel.m3 ------------------------------------
MODULE Personnel;
    IMPORT Stdio, Text, Wr;
    FROM Stdio IMPORT stdout;
    REVEAL File = Public_File BRANDED OBJECT
        ssn: INTEGER;
        firstname,lastname: Text.T;
        OVERRIDES
            Print := PrintFile;
        END;

    (* Print method *)
    PROCEDURE PrintFile(pf: File) =
    BEGIN
        (* Body of print method omitted *)
    END PrintFile;

BEGIN
END Personnel.

------------------------- Student.i3 --------------------------------------
INTERFACE Student;
    IMPORT Text, Personnel;
    TYPE
        File <: Public_File;
        Public_File = Personnel.File OBJECT
        END;
END Student.
------------------------- Student.m3 --------------------------------------
MODULE Student;
    IMPORT Stdio, Text, Wr, Personnel;
    FROM Stdio IMPORT stdout;
    REVEAL File = Public_File BRANDED OBJECT
        nclasses: INTEGER;
        class: ARRAY [1..10] OF INTEGER;
        OVERRIDES
            Print := PrintFile;
        END;

    (* Print method *)
    PROCEDURE PrintFile(sf: File) =
    BEGIN
        Personnel.File.Print(sf);  (* Invoke overridden print method *)
        sf.PrintStudentInfo();     (* Print student specific information *)
    END PrintFile;

BEGIN
END Student.
---------------------------------------------------------------------------
```

Figure 3: Personnel and Student Module Interfaces and Implementations in Modula-3.

```
------------------------- personnel.sa -----------------------------------

class PERSONNEL is
   private ssn:INT;
   private firstname, lastname: STR;

   -- Print method
   print is
      -- body of print method omitted
   end; -- print

end; -- class PERSONNEL

------------------------- student.sa -------------------------------------

class STUDENT is
   PERSONNEL;                  -- Inherit from PERSONNEL
   private nclasses:INT;       -- Add number of classes
   private classes:ARRAY{INT}; --     and the array of classes

   alias personnel_print print;

   -- Print method
   print is
      self.personnel_print;    -- Invoke overridden print method
      self.print_student_info; -- Print student specific information
   end; -- print

end; -- class STUDENT

--------------------------------------------------------------------------
```

Figure 4: Personnel and Student Classes in Sather.

```
------------------------ personnel.self ---------------------------------

traits dbbench personnelTraits _Define: (|
  _ parent* = traits clonable.
  ^ print = ( "body of print method omitted" ).
|)

prototypes dbbench personnelProto _Define: (|
  _ traitsparent* = traits dbbench personnelTraits.
  ^ ssn.
  ^ firstname.
  ^ lastname.
|)

------------------------ student.self -----------------------------------

traits dbbench studentTraits _Define: (|
  _ traitsparent* = traits dbbench personnelTraits.
  ^ clone = (_Clone dataParent: personnelProto clone).
  ^ print = ( resend.print.     "Invoke overridden print method"
              printStudentInfo. "Print student specific information" ).
|)

prototypes dbbench studentProto _Define: (|
  _ traitsparent* = traits dbbench studentTraits.
  _ dataparent**.
  ^ nclasses.
  ^ classes.
|)

--------------------------------------------------------------------------
```

Figure 5: Personnel and Student Traits and Prototypes in Self.

tiple parents are resolved in a last-defined fashion; the redefinition of a name hides the previous declaration.

Finally, Figure 5 shows the Personnel and Student traits and prototypes objects definitions in Self. Self makes use of prototypes with cloning rather than classes with instantiation. However, the behavior of classes with inheritance can be achieved in Self as described by Ungar *et al* [14]. To explain how this is done, we will use our database application as an example. Consider the relationship between student and personnel files. Student files inherit data fields and behavior (or *traits*) from personnel files.

The data and traits are divided into separate objects as illustrated in Figure 6. When a student file is created, the prototypical student object is *cloned*, thereby creating local copies of the student data fields, or *slots*. In addition, the prototypical personnel object is also cloned, thus creating local copies of the slots inherited from personnel files, as seen in the clone method for StudentTraits in Figure 5. These objects share behavior through the parent pointers to the traits objects. However, the creation of a new student file does not require that the traits objects be cloned since all student files can share a single copy of the methods.

Since multiple parent pointers can be specified, multiple inheritance is possible. In fact, in this simple example, multiple inheritance is necessary in studentProto even though there is no multiple inheritance inherent in the problem being solved. Furthermore, a precedence for the inheritance must be specified since two distinct print methods are reachable via the two parent pointers. In this example, dataparent has lower precedence, as indicated by the two asterisks following the name, so that the print method from studentTraits is used for student files.

An alternate approach eliminates data parents from the model [5]. Using this approach, the student prototype is created by cloning the personnel prototype and adding additional slots, thereby eliminating the dynamic indirection associated with accessing data slots of the parent. The parent pointer is also updated to point to the student traits object, eliminating the problem of multiple inheritance.

3.2 Dynamic Dispatch

Each of the languages provide a mechanism for the specification of dynamic dispatch. With the exception of Self, each also provides a way to specify static binding so the overhead associated with dynamic dispatch can be avoided. With Oberon-2, all type-bound procedures are dispatched dynamically and standard procedures can be used to achieve static binding.

In Modula-3, methods may be declared using one of two mechanisms: METHODS and OVERRIDES. If a method is declared in a parent class, then subclasses can either define a new method with the same name (METHODS) or override the previous definition (OVERRIDES). Both forms of declaration result in dynamically dispatched functions. The difference between the two forms of declaration is subtle and related to name scopes. In the case of the OVERRIDES definition, the new method definition completely replaces the definition it overrides; the overridden definition is

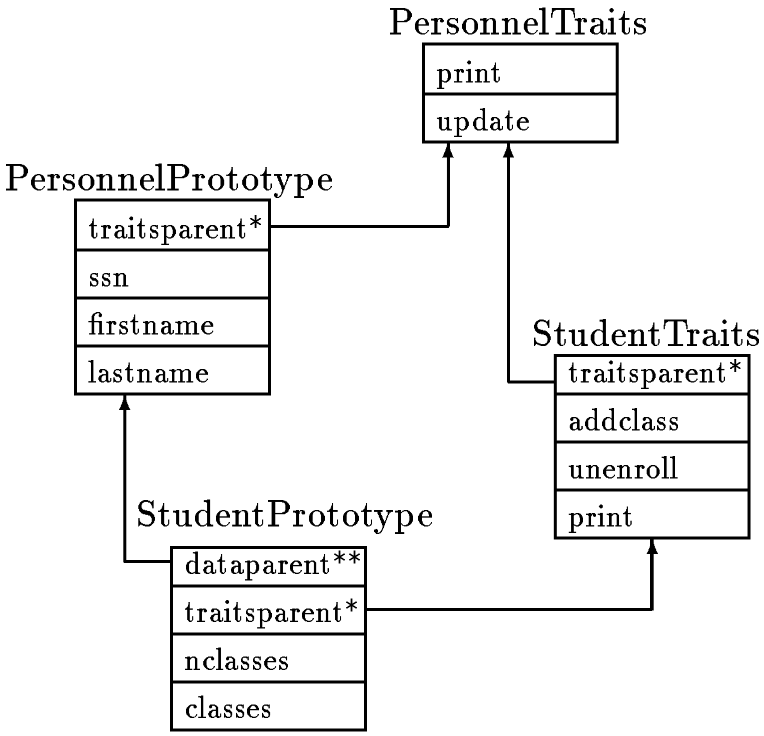

Figure 6: Personnel/Student Object Hierarchy Using Self Prototypes

not accessible even if the object's type is narrowed[1] to the type of the parent. In the case of the METHODS definition, both the old and the new definition remain available, although normally the new definition shadows the original definition. In this case, narrowing an object to its parent type will cause the original definition to be used.

Sather is similar to C++ in that methods are statically bound unless explicitly declared otherwise. However, the mechanism for the specification of dynamic binding in Sather is quite different than it is in C++. With C++, a method can be declared a *virtual function* and calls to that method will be dynamically dispatched. In Sather, the declared type of a variable determines whether method calls will be dispatched dynamically. This is done by declaring a variable to be of a dispatched type by simply preceding the type specification with a dollar sign. So, if *file* is declared of type *$PERSONNEL*, then the dotted access *file.print* will result in the desired dynamic dispatch. However, this approach can result in unnecessary overhead if the programmer is not cautious. Consider the two Sather code fragments in Figure 7. In the print method, we step through the database printing each personnel file. Since we want the appropriate print routine to be called for student and teacher files, we want the call to the print method, *tmp.file.print*, to be dynamically dispatched. We can achieve this by declaring the *file* field of the database class to be a dispatched type, namely *$PERSONNEL*. However, consider the locate method that scans the database looking for a file with the given social security number. Since the *file* field of the database class was declared as a dispatched type, the access of the social security number, *tmp.file.ssn*, is also incurring the overhead of dynamic dispatch even though it may be an instance variable access instead of a method invocation.

3.3 Code Reuse

One of the advantages of object-oriented programming is the ability to reuse code. If a function is implemented by a class, it can be made available to all subclasses. In addition, a method can be reimplemented in the subclass and the new method has the option of calling the method in the parent class that it is overriding. This is useful if the subclass wants to extend the behavior of a function, while still reusing the code defined in the parent class. Each of the languages provide

[1]The NARROW operation in Modula-3 makes it possible to view an object as being of the parent type and vice-versa.

```
-- Print the database
print is
   tmp: DATABASE;
   tmp := self;
   until tmp.next=void loop
      tmp.file.print;
      tmp := tmp.next;
   end;
end; -- print
```

```
-- Locate a file in the database
locate(ssn:STR): PERSONNEL is
   tmp: DATABASE;
   tmp := self;
   until tmp.next=void loop
      if tmp.file.ssn = ssn then
         res := tmp.file;
         return;
      end;
      tmp := tmp.next;
     end;
     res:=void;
end; -- locate
```

Figure 7: Sather Code Fragments from the Database Application

support for explicit calling of the parent method from within the overriding method, however there are some interesting differences.

In Oberon-2, the mechanism is quite straightforward; you simply append the method name with a caret(^). For example, in the Print procedure for student files, you can call the print procedure for personnel files by calling Print^ (Figure 2).

In Modula-3, to invoke the overridden method it is necessary to name the parent type and method to be called and explicitly pass the receiver of the message as an argument. For example, to call the print method in the personnel class, you call *Personnel.File.Print(file)* where *file* is the receiver of the message (Figure 3). As mentioned in the previous section, the NARROW mechanism does not give access to the overridden method.

The textual inclusion model of inheritance supported by Sather introduces a small problem since the redefinition of the print method hides the original definition. Sather includes an aliasing mechanism that addresses this problem. For example, after the inclusion of the personnel class in the definition of the student class, we can alias print to another name, for example *personnel_print* (Figure 4). We can then call this method using the aliased name. With multiple inheritance, it is up to the programmer to include the proper alias statements so that the correct methods are visible.

Self includes the concept of message resends, whereby a message can be resent to the parent class from within the overriding method definition. There is no restriction that the resent message must be the same as the current method. For example, from within the print method for student files, we can invoke the print method for personnel files with *resend.print* (Figure 5) or we can invoke some other method, foo, with *resend.foo*. It is also possible to perform a directed resend where the resend is sent to a specified parent. This makes it possible to override the specified multiple inheritance precedence.

3.4 Information Hiding

Oberon-2 provides a familiar import/export mechanism for information hiding like that found in the earlier Modula languages. Names are private to the module in which they are declared unless explicitly exported. Similarly, names exported by a module can only be used by a module that explicitly imports that module. Since modules are the grouping mechanism used by the language, extended types appearing in a module other than that in which the parent type is declared have no special access privileges to the fields of the type being extended. This is in contrast to many object-oriented programming languages where subclasses have unrestricted access to the fields of the parent class.

Modula-3 is also built around modules, however the incorporation of object types complicates the information hiding mechanism. In order to expose certain features of the class (ie., access methods) while hiding others (ie., the actual data fields) it becomes necessary to declare two actual classes for each conceptual class. A public class is declared in the interface that contains the features of the class that are to be exported. Then, the actual class is defined as a subclass of this public class. For example, in Figure 3 Public_File exports the method Print, while File, a subclass of Public_File, defines the hidden instance variables of the class and provides an implementation of the Print method. As with Oberon-2, subclasses have no special access privileges to their parent classes.

Sather takes a more direct approach to information hiding. Names within a class can be declared private, thus prohibiting direct access by instantiations of the class. However, subtypes have unrestricted access to the fields of the parent class. This is not surprising considering the textual inclusion model of inheritance supported by Sather.

In Self, slots can be declared public, private, read-only, or write-only. A method defined for an object has access to its private slots as well as the private slots of its ancestors. However, the dynamic nature of Self adds additional flexibility, and with it complexity, to the information

		Internet Location	
Language	Implementation	Machine	Directory
C++	GNU 2.2.2	gatekeeper.dec.com	pub/GNU
Modula-3	SRC 2.07	gatekeeper.dec.com	pub/DEC/Modula-3
Sather	Rel0.2i	icsi.berkeley.edu	pub/sather
Oberon-2	SPARC-2.5	neptune.ethz.ch	Oberon/SPARC
Self	2.0.1	self.stanford.edu	pub/Self-2.0.1

Table 1: Publicly available implementations of the five languages that we measured.

hiding mechanism. Dynamic name binding only requires that the receiver of a message has access to slots referenced in the corresponding method. There is no requirement that slots referenced in a method be defined by that object. For example, a method defined in the personnel class could access instance variables defined only in the student class. As long as the receiver of this message is actually a student file, it can use the method safely. Self performs run-time checking to assure that the receiver of a message provides the appropriate method and has access to any slots referenced within that method.

4 Performance Evaluation

In this section we evaluate the performance of publicly available implementations of the languages considered. Table 1 describes the implementations we measured as well as how they can be obtained on the Internet via anonymous FTP. It is important to note that the results in this section reflect the performance of a particular implementation of these languages and do not necessarily reflect performance limitations that exist in the languages independent of these implementations. In particular, the maturity of an implementation can have a profound effect on its performance. For example, the GNU C++ compiler, **g++**, is much more mature than the Sather compiler that we measured.

In order to evaluate the performance of each implementation, a test program was written to exercise the database code. This test program first adds a large number of records to the database and then a series of file locates are performed. Next, the database is traversed a number of times with a set of operations performed on each record. Finally, all records are deleted from the database. This add/locate/traverse/delete process is repeated a small number of times, resulting in a reasonable amount of dynamic memory allocation and deallocation.

The run-time numbers presented were obtained using various techniques, depending upon the environment provided by the particular language implementation. The C++, Modula-3, and Sather compilers generate standalone `a.out` format binaries and the run-times were measured using the total elapsed time reported by the C-shell built-in `time` command. Since the Oberon-2 and Self systems do not generate standalone executables, other mechanisms were used. For Oberon-2, a call to *Oberon.Time()* is made immediatedly upon entering and prior to exiting the test program and the difference of the two reported times is taken. In Self, the built-in *time* mechanism is used. In all cases, a set of five runs were performed and the average of these runs is reported. The variance in execution time between runs in all cases was small. All measurements were made on an otherwise idle Sun SPARCstation-2 with 32 megabytes of memory.

4.1 Execution Time

The run-time results are shown in Table 2. Where the compiler provided optional optimization, run-time numbers are shown for both the optimized and non-optimized cases. As we have mentioned, the run-time numbers presented are as much, if not more, a measure of the ability of the particular compiler to generate efficient code as they are a reflection on the language itself. As can be seen by the large differences between the optimized and nonoptimized cases, the quality of the generated code has a large effect on run-time performance. However, some interesting observations can still be made related to the languages themselves.

Not surprisingly, Self is the slowest of the five languages, due largely to the dynamic nature of the language that requires a large amount of run-time checking not required by the other languages. However, the Self system does perform extensive innovative optimizations to reduce this run-time overhead [3, 6]. The Self database implementation used the optimization described in the Inheritance Section to avoid the dynamic indirection associated with data parents. Using the data parent model shown in Figure 6 the execution time was 29.2 seconds, so by avoiding this dynamic indirection the execution time was reduced by 40%.

In order to achieve the performance numbers shown in Table 2, the Oberon-2 and Modula-3 database implementations did not make exclusive use of methods. Exclusive use of type-bound

Language	Compiler	Optimization	Run-Time (s)
C++	GNU 2.2.2	No	8.4
C++	GNU 2.2.2	-O	3.7
C++	AT&T 2.1	No	9.3
C++	AT&T 2.1	-O	3.5
Modula-3	SRC 2.07	No	13.7
Modula-3	SRC 2.07	-O	6.0
Sather	Rel0.2i	No	13.3
Sather	Rel0.2i	-O	7.2
Oberon-2	SPARC-2.5	Default	4.8
Self	2.0.1	Default	17.4

Table 2: Execution Time on a Sun SparcStation-2 with 32 Mbytes of Memory

procedures in Oberon-2 and METHODS in Modula-3 results in all calls incurring the overhead of dynamic dispatch. In order to avoid this unnecessary overhead, standard procedure calls were used for those calls that were not dispatched dynamically. The application did require that 45% of all calls be dispatched so the associated cost is quite significant. The higher run-time of Oberon-2 as compared to C++ can be attributed largely to the compilers. The Oberon-2 compiler does not perform the level of optimization being performed by the C++ compilers. Another difference is that Oberon-2, as well as each of the other languages, provides garbage collection that is typically more expensive, although far less error-prone, than the explicit deallocation of memory provided by C++.

One reason for the higher run-time of Modula-3 as compared to C++ is that the instance variable and method offsets for subtypes of opaque types are not known at compile time and require an additional run-time indirection. This is a limitation of the current compiler implementation and not a penalty inherent in the language. By reorganizing the code, it is possible to make the offsets known to the compiler, at a loss of information hiding, and by doing so the run-time drops to 4.9 seconds. Furthermore, in Modula-3, it is possible to explicitly avoid the overhead of garbage collection by using untraced references. By doing so, the run-time is further reduced to 4.5 seconds. Clearly, as more mature Modula-3 compilers and run-time systems become available, the performance will surely improve.

Sather is similar to C++ in that method binding is static unless explicitly specified otherwise. However, Sather has a more general dispatch mechanism than C++ and, in the Rel0.2i implementation, there is a higher associated overhead. The Sather compiler uses a single word dispatch cache to minimize the cost of dispatch [7]. As a result, the cost of dispatch is highly dependent on the miss-rate of this cache. The effect of dispatch cache miss rate on run-time is discussed later in the paper.

Our initial version of the database application implemented social security numbers as strings. Because the test program searches the database using social security numbers, a large number of string comparisons were performed in this version. Therefore, the run-time numbers were significantly influenced by the efficiency of string comparisons. Even though the efficiency of string comparisons may be important in certain applications, the goal of this study was not to measure this effect. For this reason, the application was recoded in each language and social security numbers were implemented as integers. It is interesting to note that this conversion required no changes to the database implementation in Self. The test code that inserted records into the database was simply changed to use integers instead of strings and the rest of the code was unchanged. This is certainly a testament to the suitability of Self for exploratory programming. Even though the changes to the code for the other languages were not drastic, there was still a non-trivial coding/debugging process involved.

4.1.1 Static Method Binding Vs. Dynamic Dispatch

In each of the languages, with the exception of Self, it is possible to explicitly specify that a method invocation is bound statically. In Self, all method invocations are dispatched dynamically, however the compiler does perform optimizations to use static binding when possible. In Oberon-2 and Modula-3, standard procedures can be used to achieve static binding while type-bound procedures and methods are dispatched dynamically. In Sather and C++, methods are statically bound unless explicitly specified otherwise using dispatched types and virtual functions. Table 3 shows the results of varying the relative percentage of dispatched and static calls while keeping the total number of calls constant. With no dispatched calls, we see that the execution time for Sather actually drops below that of C++. However, as the percent dispatch increases we see the effect of the Sather implementation's increased dispatch cost. The irregularity in the Sather run-times as the percent

Language	Compiler	Percent Dynamic Dispatch 0%	23%	45%	68%	90%	100%	Avg. Cost/Dispatch (microseconds)
C++	GNU 2.2.2	3.3	3.5	3.7	4.0	4.3	4.4	0.20
C++	AT&T 2.1	3.0	3.2	3.5	3.8	4.1	4.3	0.24
Modula-3	SRC 2.07	4.4	5.1	6.0	6.7	7.5	7.6	0.58
Sather	Rel0.2i	2.8	6.8	7.2	7.5	7.9	9.5	1.22
Oberon-2	SPARC-2.5	4.1	4.6	4.8	5.2	5.4	5.5	0.25
Self	2.0.1	—	—	—	—	—	17.4	N/A

Table 3: Execution Time, In Seconds, For Various Dynamic Dispatch Percentages

dynamic dispatch increases is due to variability in the dispatch cache miss rate. For example, with 23% and 45% dynamic dispatch, the dispatch cache miss rate is 50% and 25%, respectively. The effect of the cache miss rate is discussed in more detail in the next section.

Since the total number of dispatched calls for the 100% dispatch case is known, it is possible to determine the average cost of dispatch. From Table 3, we see the higher cost of dispatch for the Sather implementation while the cost for SPARC Oberon-2 and GNU and AT&T C++ is relatively low. The run-time numbers for Self, listed in the 100% dispatch column, are slightly misleading. In addition to having all calls dispatched, in Self all slot accesses (or instance variable accesses) are also dispatched dynamically, which is not the case with the other languages. In order to have a completely fair comparison with C++, all instance variable accesses would have to be implemented using virtual functions in C++, which would certainly decrease the run-time gap between the languages.

4.1.2 Sather's Dispatch Cache

The SatherRel0.2i dispatch cache is a one-word cache that incurs a miss each time the type of a dispatched object changes, resulting in an 80–90 instruction penalty (on a Sun SparcStation) for general dispatch plus the cost of updating the cache. In the current application, it is possible to control the dispatch cache miss rate by varying the order in which records are added to the database. For example, consider the case where the database is a list alternating between teacher and student files. If the list is traversed and a single dispatched operation is performed on each object, then the object type changes with each call and each call incurs a cache miss. Similarly, if the list is a sequence of two teacher files followed by two student files, then every other call

Dispatch Cache	Percent Dynamic Dispatch				
Miss Rate	0%	23%	45%	68%	90%
0%	2.8	3.1	3.5	3.9	4.3
5%	2.8	3.5	4.3	5.0	5.7
10%	2.8	3.8	4.9	6.3	7.1
25%	2.8	4.9	7.2	8.7	11.5
50%	2.8	6.8	10.8	—	—
100%	2.8	10.4	—	—	—

Table 4: Sather Execution Time Vs. Dispatch Cache Miss Rate For Various Dispatch Percentages

would result in a miss. The run-time of the application for several combinations of cache miss rate and dispatch percentage is shown in Table 4. Even though the application does not permit measurement for each combination, the trends are quite apparent. With no cache misses and 90% dispatched calls, the average cost per dispatch drops to 0.30 microseconds, in contrast to the 1.22 microseconds shown in Table 3 when the cache miss rate was approximately 15%.

A detailed discussion of the Sather dispatch cache design and an evaluation of the dispatch efficiency was written by Lim and Stolcke [7]. They measure the dispatch cache miss rate of the Sather compiler (written in Sather) and find the average miss rate to be approximately 20%. From Table 4, we see that a miss rate on the order of 20% does not have too great an effect with relatively low dispatch percentages. However, high dispatch rates coupled with high miss rates are quite detrimental, although such an operating range is unlikely in a real application.

4.2 Code Size and Compile Time

Table 5 shows the compilation times, not including the link phase, and resulting code size for each of the language implementations. The compile times for the C++, Modula-3, and Sather implementations are all comparable at 19–30 seconds, with optimization adding a few additional seconds, while those for Self and Oberon-2 are significantly lower. It is important to realize that the AT&T C++, Sather, and Modula-3 compilers all generate intermediate C code that is then compiled using `cc`.

It is obvious from the compile time for Oberon-2 that Wirth has succeeded in designing a simple language for which an efficient compiler can be written. The SPARC Oberon 2.5 compiler is extremely efficient and generates object code directly rather than generating intermediate C

Language	Compiler	Optimized	Compile Time(sec)	Object Size(bytes)	Binary Size(bytes)	Stripped Binary(bytes)
C++	GNU 2.2.2	No	28	7004	311296	147456
C++	GNU 2.2.2	Yes	30	5748	311296	147456
C++	AT&T 2.1	No	30	12128	65536	49152
C++	AT&T 2.1	Yes	35	10464	65536	49152
Modula-3	SRC 2.07	No	24	24560	1015808	352256
Modula-3	SRC 2.07	Yes	30	21716	1015808	352256
Sather	Rel0.2i	No	19	5880	139264	122880
Sather	Rel0.2i	Yes	23	4660	139264	122880
Oberon-2	SPARC-2.5	Default	2	5587	—	—
Self	2.0.1	Default	9	—	426908	198364

Table 5: Program Code Size And Compile Time. A stripped binary has had the symbol table and relocation bits that are normally attached to a binary removed. The code size numbers for Self are only approximations based on code information provided by the _PrintMemory method. A standalone `a.out` format binary is not generated.

code. On the other hand, the Oberon-2 compiler performs far less optimization than is performed when optimization is specified with the C++ compilers. However, by comparing Oberon-2 with the unoptimized C++ implementations, we see that the compile times for Oberon-2 are much lower while the run-times, from Table 2, are also considerably lower.

The compilation in the Self system is not a user-requested operation, but rather is performed as needed during program execution, a behavior that is particularly convenient during code development. The compile times listed represent the difference in run-time between the first run, when compilation is performed, and subsequent runs.

The code sizes also vary significantly between the language implementations. With the C++, Sather, and Modula-3 compilers, linking was performed using `ld` to generate a statically linked `a.out` format binary. While the GNU compiler generates more compact code than the AT&T compiler, the larger run-time library results in a larger executable. Modula-3's larger run-time system is also evident in the size of the binary. With the exception of C++, all languages provide garbage collection so the code for this task must be included in the binary.

5 Conclusions

In this paper we have evaluated the object-oriented programming features supported by Oberon-2, Modula-3, Sather, and Self. Specifically, we investigated features that support inheritance, dynamic dispatch, code reuse, and information hiding. We compared the languages by programming a small database application in each of them and in C++. The structure of the test application resembles that of a number of commonly-used examples of object-oriented applications. We evaluated both the usability of the object-oriented features we considered and also measured the performance of publicly-available implementations of the languages.

Each of the languages considered support object-oriented programming, however they do so in different ways. Oberon-2 provides the minimal extensions to a module-based language necessary to support classes with inheritance. Modules form the basic building blocks of the language and type extensions and type-bound procedures fit cleanly into this model. Oberon, due to its simplicity and clean design, appears to be a relatively easy language to implement efficiently. The SPARC-2.5 Oberon compiler that we measured is very fast and generates relatively efficient code without performing extensive optimizations.

Modula-3 also adds object types a module-based language; however Modula-3 is a larger and more complex language than Oberon-2. Modula-3 provides a number of features not found in Oberon-2, including threads for concurrency, exceptions, and separate module interface specifications. All of these features support the use of Modula-3 in programming large systems that may require many programmers. With both the Modula-3 and Oberon-2 implementations we measured, exclusive use of the object-oriented features in the language, namely methods in Modula-3 and type-bound procedures in Oberon-2, results in slight run-time penalties. While the SRC 2.07 Modula-3 that we measured was not as fast as the C++ implementations, the performance is due to a limitation in the compiler and not inherent in the language itself. As Modula-3 compilers mature, we would expect run-time performance similar to C++.

Sather abandons modules in favor of a more object-oriented approach with all code organized into classes. This approach has yielded a simple, easy-to-use object-oriented programming language. Instead of providing virtual functions as in C++, Sather includes dispatched types that provide a more general dispatch mechanism. Even though this dispatch mechanism is more expensive, a dispatch cache can be employed to reduce the associated overhead. One goal of Sather was to avoid features that mainly support programming-in-the-large and make programming-in-the-small more difficult. Based on programming our database application, we conclude that the Sather design is successful in achieving this goal.

Self's prototypes with cloning give the power needed to support object-oriented programming. In fact, the language provides flexibility in how classes are actually modeled. The dynamic nature of the language and short compile times support rapid prototyping at some cost in run-time efficiency. Aggressive, innovative optimization techniques have significantly reduced the run-time overhead of Self.

6 Acknowledgements

We would like to thank Urs Hölzle, Josef Templ, Eliot Moss, Dirk Grunwald, David Ungar, Stephen Omohundro, Dain Samples, Martin Trapp, and Ralph Johnson for their valuable help, comments, and criticisms. We would also like to thank all those involved with the implementation of the Oberon-2, Modula-3, Sather, Self, and GNU C++ compilers for making these tools available, without cost, to the programming community.

References

[1] Gunther Blashek, Gustav Pomberger, and Alois Stritzinger. A comparison of object-oriented programming languages. *Structured Programming*, 10(4):187–197, 1989.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Notices*, 27(8):15–43, August 1992.

[3] Craig Chambers and David Ungar. Custimization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 146–160, Portland, OR, June 1989.

[4] Sam Harbison. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[5] Urs Hölzle. Personal communications. January 1993.

[6] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1991.

[7] Chu-Cheow Lim and Andreas Stolcke. Sather language design and performance evaluation. Technical Report TR-91-034, International Computer Science Institute, Berkeley, CA, May 1991.

[8] Hanspeter Mossenbock and Josef Templ. Object Oberon — a modest object-oriented language. *Structured Programming*, 10(4):199–207, 1989.

[9] Hanspeter Mossenbock and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12:179–195, 1991.

[10] Stephen Omohundro and Chu-Cheow Lim. The Sather language and libraries. Technical Report TR-92-017, International Computer Science Institute, Berkeley, CA, February 1992.

[11] Stephen M. Omohundro. *The Sather Language*. International Computer Science Institute, Berkeley, CA, June 1991.

[12] Heinz W. Schmidt and Stephen Omohundro. CLOS, Eiffel, and Sather: a comparison. Technical Report TR-91-047, International Computer Science Institute, Berkeley, CA, September 1991.

[13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[14] David Ungar, Craig Chambers, Bar-Wei Chang, and Urs Hölzle. Organizing programs without classes. *LISP and Symbolic Computation: An International Journal*, 4(3):37–56, 1991.

[15] David Ungar and Randall B. Smith. Self: the power of simplicity. *LISP and Symbolic Computation: An International Journal*, 4(3):1–20, 1991.

[16] Jack C. Wileden, Lori A. Clarke, and Alexander L. Wolf. A comparative evaluation of object definition techniques for large prototyping systems. *ACM Transactions on Programming Languages and Systems*, 12(4):670–699, December 1990.

[17] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, New York, NY, 1982.

[18] Niklaus Wirth. From Modula to Oberon. *Software—Practice and Experience*, 18(7):661–670, July 1988.

[19] Niklaus Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[20] Wayne Wolf. A practical comparison of two object-oriented languages. *IEEE Software*, 6(5):61–68, September 1989.