Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P. Klint

Modularization and reusability in current programming languages

Computer Science/Department of Software Technology Report CS-R8635 November

Bibliotheek
Centrum voor Wiskunde en Informatica
Amsterdam

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Copyright © Stichting Mathematisch Centrum, Amsterdam

Modularization and Reusability in current Programming Languages

Paul Klint

Department of Software Technology, Centre for Mathematics and Computer Science
Department of Computer Science, University of Amsterdam

How well do modularization constructs in current programming languages allow the construction of truly reusable modules? This question is answered by examining the implementation of the datatype queues in Pascal, Modula-2, Ada and Smalltalk. The merits of object-oriented languages versus algorithmic languages are discussed from the perspective of reusability.

1986 CR Categories: D.2.2 [**Software Engineering**]: Tools and Techniques - *Modules and interfaces, Software libraries;* D.3.3 [**Programming Languages**]: Language Constructs - *Modules, packages.*
1980 Mathematics Subject Classification: 68B10 [**Software**]: Analysis of Programs - *Semantics.* 69D40
Key Words and Phrases: software engineering, reusable software, modules, abstract data types, object-oriented programming.
Note: This paper will be published in the proceedings of the 1986 CERN Summer School on Computing, held in Renesse, The Netherlands, august 31 - september 13, 1986.

1. INTRODUCTION

Over the last two decades it has become widely recognized that *abstraction* is the key technique that allows software developers to control the ever increasing complexity of the software systems they are building. The primary concerns are to control the *quality* (conformance to specifications, reliability, maintainability) and *costs* of the resulting product. The quality of a software product is determined by the quality of all steps in its production process, ranging from requirements analysis, problem specification and initial program design to coding, testing and maintenance. The costs of a software product are also determined by the costs of all steps in its production process. More often than not, the objectives of maximizing quality and minimizing costs cannot be reconciled.

A common abstraction technique is to decompose a program into a number of *modules* and to distinguish between the behaviour of the module as it can be perceived from the outside, and its internals, i.e. the details of the implementation that realizes its external behaviour. The advantages of this form of information hiding are manifold. The interfaces between modules can be minimized (thus controlling the number of inter-module dependencies) and the implementation of modules can be changed or optimized without affecting the users of these modules.

A common technique for reducing the costs of software products is to *reuse* parts of previously written and tested programs. This technique is by no means new. In the mid-fifties, one noticed already that many programs require common operations such as computing the *cosine* function, inverting a matrix, sorting a file or computing a standard deviation. The programmer of each of these operations needs to have expertise in a particular area. The user of those operations might not have this expertise. This observation has led to the construction of many subroutine libraries in specialized fields of application. In order to accommodate the needs of as many future users as possible, the operations provided by these libraries tend to be very general and highly parameterized.

Report CS-R8635
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Clearly, the modules of a modularized program also lend themselves to reuse. But which properties should the module constructor in a programming language have in order to maximize the possibilities of reuse? As a general rule, which will be detailed in the following sections, the possibilities of reuse of a module are maximized if the assumptions about its use are minimized. This can be achieved by delaying, as much as possible, the moment that a module is bound to its "users" (i.e. other modules). Contrasting this with the desire to bind such information also as early as possible--if one wants to perform static (compile-time) typechecking in order to achieve run-time efficiency--we have concisely formulated one of the many dilemmas a language designer has to face.

I will discuss modularity and reuse issues in two families of current programming languages: the "algorithmic languages" and the "object-oriented languages". The major distinction being that the algorithmic languages stay within the framework of statically typed languages and have limited capabilities of reuse, while object-oriented languages use dynamic typing and lend themselves to more general reuse.

This paper is organized as follows. In section 2 a common example is introduced that will be used throughout. In section 3 the algorithmic languages Pascal, Modula-2 and Ada are discussed. Section 4 discusses object-oriented languages in general, and Smalltalk-80 in particular. Section 5 contains some concluding remarks.

2. A COMMON EXAMPLE

As a common example I will use the queue data type. A queue is a first-in-first-out list of elements on which the following operations are defined:

empty	true if there are no elements in the queue, false otherwise.
full	true if no more elements can be added to the queue, false otherwise.
insert element	insert an element at the rear of the queue.
delete element	delete an element from the front of the queue; return that element.

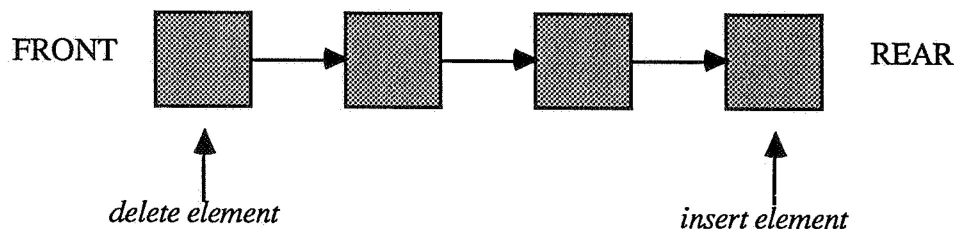

Figure 1. The queue data type.

Figure 1 illustrates these operations. There are two obvious implementations of queues: by means of a linked list and by means of a circular buffer. The former allows the implementation of unbounded queues, the latter provides a more compact representation since no explicit links between the elements in the queue have to be stored. See [Knuth68] for a discussion of these alternatives. The entities needed for the circular buffer implementation (also see figure 2) are:

`MaxQsize`	maximal number of elements in a queue.
`elems`	array containing the queue elements.
`first`	front of the queue.
`last`	the rear of the queue.
`size`	the number of elements in the queue.

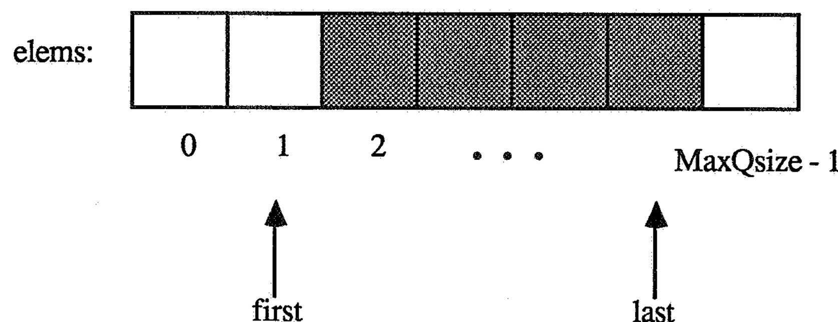

Figure 2. Queue implemented by circular buffer.

In this paper we are interested in the following three questions:

(1) Is it possible to define the data type queue in such a way that the implementation can be changed (e.g. switching from a circular buffer to linked lists) without affecting the users of the data type?
(2) Is it possible to define the data type queue once, such that queues with elements of a different type can be defined using that definition?
(3) Is it possible to define queues in which the elements are heterogeneous, i.e. a queue may contain elements of different types?

3. ALGORITHMIC LANGUAGES

3.1. Pascal

Pascal [Wirth74] is a widely used general purpose programming language. It is a descendant of Algol60 [Naur63]. Most innovative were, at the time of its design, the facilities for defining datastructures (records) and the static (compile-time) typechecking of programs using them.

The Pascal implementation of queues is shown in figure 3. Note how the wrap-around of the indices `first` and `last` is achieved using the **`mod`** function. The consequence of this choice is that indices of queue elements range from `0` to `MaxQsize-1`, and the last element of `elems` remains unused (it is not possible in Pascal to declare arrays with constant expressions as bounds such as, e.g., `MaxQsize-1`). The procedures `underflow` and `overflow` are left unspecified.

The problems with this Pascal implementation are three-fold:

P1: *Implementation hiding:* the representation of queues by means of an array is visible and can not be changed transparently to another representation using, for instance, linked lists. It is even impossible to guarantee the consistency of this implementation, since users of the queue may freely change elements of record `Queue`.
P2: *Generic types:* the type of the queue elements is fixed and the code as shown in figure 3 (which implements queues of integers) has to be repeated for each desired element type. The only true modification required in each case is the substitution of the desired type for the type `ElemType` .

```
const MaxQsize   = 100;
type ElemType = integer;
     Queue    = record
                    first, last, size : integer;
                    elems : array[0..MaxQsize] of ElemType
                end;

procedure initialize(var Q : Queue);
begin
   Q.first := 0; Q.last := 0; Q.size := 0
end; {initialize}

function empty(Q : Queue) : boolean;
begin
   empty := Q.size = 0
end; {empty}

function full (Q : Queue) : boolean;
begin
   full := Q.size = MaxQsize
end; {full}

procedure insert(var Q : Queue; E : ElemType);
begin
    if full(Q) then overflow
    else
      begin Q.size := Q.size + 1;
            Q.last := (Q.last + 1) mod MaxQsize;
            Q.elems[Q.last] := E
      end
end; {insert}

procedure delete (var Q : Queue; var E : ElemType);
begin
    if empty(Q) then underflow
    else
      begin Q.size := Q.size - 1;
            Q.first := (Q.first + 1) mod MaxQsize;
            E := Q.elems[Q.first]
      end
end; {delete}
```

Figure 3. Queues in Pascal.

```
types ElemTypes = (booltype, inttype, realtype);
      ElemType  = record case kind : ElemTypes of
                         booltype : (boolvalue : boolean);
                         inttype  : (intvalue  : integer);
                         realtype : (realvalue : real)
                  end;
```

Figure 4. Heterogeneous queue implemented with variant records.

P3: *Heterogeneous types:* it is impossible to implement heterogeneous queues. A possible implementation using variant records for a queue with elements of type `boolean`, `integer` and `real` is shown in figure 4. This approach has the disadvantage that the type definition of the variant record that lists all allowed types has to be edited each time a new type is added. This becomes very bothersome when generic operations on queues are defined such as, for instance, a print procedure (see section 4.3 for a further discussion of this topic).

3.2. Modula-2

Modula-2 [Wirth85] is a descendant of Pascal and is intended for systems programming in general, and multiprogramming in particular. The language extends Pascal in several ways. Here we are only interested in the *module* concept. Modules are defined in two separate parts: a *specification part* defining the external interface of a module and an *implementation part* defining its implementation.This distinction provides not only proper implementation hiding of modules, but also forms the basis for separate compilation in the Modula-2 implementation.

The implementation of queues in Modula-2 is given in figure 5. Note that only the *name* of the type `Queue` and the name and type of the five operations defined on queues are specified in the definition module. All information concerning the representation of queues and the algorithms used for implementing the five queue operations is hidden in the implementation module. The type `Queue` is defined as a pointer to a `QueueRecord` (which contains the actual information related to the queue). This construction is necessary to circumvent the constraints imposed on the "opaque" export of a type, which only exports the name of a type but not its representation.

The advantage of the Modula-2 implementation is that a true division has been achieved between the specification of a module and its implementation, thus eliminating problem P1 (implementation hiding). However, the Modula-2 implementation shares problems P2 (generic types) and P3 (heterogeneous types) with the Pascal implementation. Modula-2 also permits the (unsatisfactory) solution of P3 based on variant records.

3.3. Ada

Ada is based on a language designed by J. Ichbiah (Cii Honeywell Bull) in 1977. After several revisions and international reviews, the reference manual of the resulting language "Ada" was published in 1980 and the definition was finilized and accepted by ANSI in 1983 [DoD83].

Ada supports the concept of separating the definition and the implementation of a module in a comparable but more general way than Modula-2. Modules are called *packages* in Ada terminology and definition part and implementation part are called *package specification* and *package body*, respectively. However, Ada extends the notion of modules in a major way, by allowing *generic packages*. A generic package is a template for a package, but it is not a package in itself. Generic packages can best be compared with compile-time macros which exist in many languages: a macro definition consists of a macro-name, some formal parameters and a replacement text possibly containing these parameters. Every occurrence of the macro in the text of a program is then replaced --at compile-time--by the replacement text of the macro definition after proper substitution of the parameters. In a similar way, generic packages are instantiated by properly substituting the parameters of the generic package. Generic packages form a mechanism for delayed binding, i.e. the "tailoring" of the generic package is performed when the package is instantiated.

The Ada implementation of queues is given in figure 6. This implementation is inspired by an example given in [Hibbard, Hisgen, Rosenberg, Shaw & Sherman 81]. In this case all initializations of the queue are performed in the declaration of the type `Queue`.The type `Queue` is declared **`limited private`** ; this forbids assignment operations and equality tests for queues. An instance of the generic package `Queue` can, for example, be obtained by the following declaration:

```
definition module Queue;
type ElemType = integer;
     Queue;
procedure initialize(var Q : Queue);
procedure empty(Q : Queue) : boolean;
procedure full(Q : Queue) : boolean;
procedure insert(var Q : Queue; E : ElemType);
procedure delete(var Q : Queue; var E : ElemType);
end Queue.

implementation module Queue;
   from storage import allocate;
   from system import tsize;
const MaxQsize = 100;
type QueueRecord = record
                     first, last, size : integer;
                     elems : array [0 .. MaxQsize-1] of ElemType;
                    end;
     Queue = pointer to QueueRecord;
procedure initialize(var Q : Queue);
begin
   allocate(Q, tsize(QueueRecord));
   Q^.first := 0; Q^.last := 0; Q^.size := 0
end initialize;
procedure empty(Q : Queue) : boolean;
begin
   return Q^.size = 0
end empty;
procedure full (Q : Queue) : boolean;
begin
   return Q^.size = MaxQsize
end full;
procedure insert(var Q : Queue; E : ElemType);
begin
    if full(Q) then overflow
    else
       Q^.size := Q^.size + 1;
       Q^.last := (Q^.last + 1) mod MaxQsize;
       Q^.elems[Q^.last] := E
    end
end insert;
procedure delete (var Q : Queue; var E : ElemType);
begin
    if empty(Q) then underflow
    else
       Q^.size := Q^.size - 1;
       Q^.first := (Q^.first + 1) mod MaxQsize;
       E := Q^.elems[Q^.first]
    end
end delete;
end Queue.
```

Figure 5. Queues in Modula-2.

```
package intqueue is new queue(ElemType => integer, MaxQsize => 100);
```

The result is a package declaration, defining queues of at most 100 integers. Ada allows us to hide the implementation of queues and also to define generic types. However, Ada does not solve P3 (heterogeneous types), in another way than using variant records as can be done in Pascal and Modula-2.

```
generic
   type ElemType is private;
package Queue is
   type Queue(MaxQsize : Natural) is limited private;
   function empty(Q : in Queue) return Boolean;
   function full(Q :  in Queue) return Boolean;
   procedure insert(E : in ElemType; Q : in out Queue);
   procedure delete(Q : in out Queue; E : out ElemType);
end Queue;

package body Queue is
   type Queue(MaxQsize : Natural) is
      record
         first, last, size : Natural := 0;
         elems : array(0..MaxQsize) of ElemType;
      end record;

   function empty(Q : in Queue) return Boolean is
   begin
      return Q.size = 0;
   end empty;

   function full(Q : in Queue) return Boolean is
   begin
      return Q.size = Q.MaxQsize;
   end full;

   procedure insert(E : in ElemType; Q : in out Queue) is
   begin
      if full(Q) then overflow
      else
         Q.size := Q.size + 1;
         Q.last := (Q.last + 1) mod Q.MaxQsize;
         Q.elems(Q.last) := E;
      end if
   end insert;

   procedure delete(Q : in out Queue; E : out ElemType) is
   begin
      if empty(Q) then underflow
      else
         Q.size := Q.size - 1;
         Q.first := (Q.first + 1) mod Q.MaxQsize;
         E := Q.elems(Q.first);
      end if
   end delete;

end Queue;
```

Figure 6. Queues in Ada.

4. OBJECT-ORIENTED LANGUAGES

4.1. What are object-oriented languages?

Algorithmic languages are based on a computation model that distinguishes between *operands* (i.e. data values of a certain named type) and *operators* (i.e. functions or procedures operating on operands of certain types and producing a typed result). Operands are thus *passive* while operators

are *active:* operands represent the current global state of a computation while operators can transform this global state. It is essential that the context in which a certain computation is to be performed is responsible for the selection of a properly typed operator to carry out the desired computation. If, for instance, an addition operator exists for types integer, real and complex, one of these three operators has to be selected at each position where an addition operator is used.

The so-called object-oriented languages are based on a different computation model. The entities considered in this model are called *objects* which may have their own private storage (containing data values). The notion of computation is captured by the concept of *message passing:* each object is capable of answering certain messages. A message is a request to the object receiving it and the latter has the obligation to provide a *method* (cf. operator) for answering it. During the construction of the answer the private memory of the object may be changed and messages may be sent to other objects. It is essential in this model that the sender of a message does not know which method will provide the response to the message. In this way object-oriented languages provide data abstraction and--as we will see below--reusability.

Usually, many objects have equal properties, i.e. they respond to the same messages in the same way. It is common to organize those objects into *classes* . An element of a class is called an *instance* of that class. The advantage of this organization is that the methods needed for answering messages do not have to be specified separately for each object; they can be specified once for a whole class.

The notion of classes can be refined by relating classes in order of increasing specificity. A more specific class (e.g. automobiles) *inherits* properties (e.g. has a number of wheels) from a more general class (e.g. vehicles), but it can also add specific properties (e.g. consumes fuel). The more specific class is called a *subclass* of the more general one. In this way, a subclass can extend or modify an existing class by specifying the differences with it or by specifying new messages that are typical for the subclass.

A distinction should be made between an object-oriented *language* and object-oriented *programming*. The former is a programming language which is strictly based on the message/object computation model (see next section). The latter is a style of programming that simulates the message/object model in an existing programming language (see section 4.5). See [Cox86] for a general introduction to object-oriented programming.

4.2. Smalltalk

The language Simula [Dahl, Myhrhaug & Nygaard71] is a statically typed superset of Algol60. Simula introduced the notions of classes and of class inheritance (called class concatenation in Simula) in the context of solving simulation problems. These two concepts combined with the notion of dynamic binding stemming from Lisp form the basis for Smalltalk [Goldberg & Robson83], which is nowadays the prime example of an object-oriented programming language. Some aspects of Smalltalk are now discussed by commenting on the Smalltalk implementation of queues (see figure 7). For a complete description of the language the reader is referred to [Goldberg & Robson83].

Each class has a name (e.g. `Queue`) and may be a subclass of another class. All classes are ultimately a subclass of the most general class `Object`, which defines general operations such as testing the class of an instance, and comparing and copying instances. The "private memory" of each object consists of a number of *instance variables;* each instance of a class has a separate set of instance variables.

The messages accepted by a class have a name and (optional) parameters. Parameters may be separated by user-defined delimiters. Sending a message to some class instance is written as the juxtaposition of the instance and the message.

Some examples of messages that might be sent to `q` (an instance of class `Queue`) are:

`q empty`	(send message `empty` to `q`)
`q insert: 3`	(send message `insert:` with argument `3` to `q`)

Other examples are:

`2 + 3`	(send the message + with argument `3` to `2`; both `2` and `3` are instances of the class `SmallInteger`)
`a at: 3 put: 'abc'`	(send the message `at:put:` with arguments `3` and `'abc'` to `a`; if `a` is an instance of the class `Array`, this corresponds to the Pascal statement `a[3] := 'abc'`)
`a at: 3`	(fetch third elememt of `a`)

These examples show that some syntactic freedom can be achieved by using messages such as `at:` and `at:put:`. The parsing of these message patterns is done at compile-time.

The implementation of a message is called a *method* and is comparable to a procedure in a conventional language. Methods return a value to the sender of the message by means of the operator ↑. They may also refer to the receiving object itself (i.e. the class instance that is currently executing this method) by means of the pseudo-variable `self`. When a message is sent to an object, the selection of the method to be invoked is determined--at run-time--by the class of that object.

Control structures such as **if**- and **while**-statements are modelled by messages sent to the class of Boolean values. The **if**-statement, for instance, is provided by the message `ifTrue:ifFalse:`. The Smalltalk expression:

```
x > 0
   ifTrue: [ sign ← 1 ]
   ifFalse:[ sign ← -1 ]
```

corresponds to the following Pascal statement:

```
if  x > 0 then sign := 1 else sign := -1;
```

The semantics of the Smalltalk expression is as follows: the expression `x > 0` is evaluated and yields one of the two possible instances of the class `Boolean`: `true` or `false`. Next, the message `ifTrue:ifFalse:` is sent to this Boolean, which in turn responds by evaluating either the first or the second argument of that message.

The Smalltalk implementation solves all three problems we have posed in section 2. Problem P1 (implementation hiding) is solved by the class mechanism itself. Problems P2 (generic types) and P3 (heterogeneous types) are solved by run-time method selection which avoids nearly all static constraints on the types of the elements in the queue. I will further discuss this issue in the next section.

4.3. The effect of object-oriented programming on program organization

In languages based on the operator/operand model, programs are organized around the definitions of the operators required to solve a particular problem. In languages based on the message/object model, programs are organized around the definitions of classes; the definitions of operators are distributed over the class definitions. An example will illustrate this.

```
class name:             Queue
subclass of:            Object
instance variables:     first last size maxqsize elems

initialize: n
   first ← last ← size ← 0.
   maxqsize ← n.
   elems ← Array new: n

empty
   ↑ size = 0

full
   ↑ size = maxqsize.

insert: e
   self full
      ifTrue: [ ↑ 'overflow' ]
      ifFalse:[ size ← size + 1.
                last ← (last + 1) rem: maxqsize.
                elems at: (last + 1) put: e ]
delete
   self empty
      ifTrue: [ ↑ 'underflow' ]
      ifFalse:[ size ← size - 1.
                first ← (first + 1) rem: maxqsize.
                ↑ elems at: (first + 1) ]
```

Figure 7. Queues in Smalltalk.

Suppose we want to add a print operation to a heterogeneous queue data type. Assume that values of types $t_1, \ldots, t_n$ may occur as elements in the queue and that these values can be printed by the procedures $t_1$print, ..., $t_n$print. In a Pascal implementation using variant records (see figure 4), a print routine for queues will have the following general structure:

```
procedure print(Q : queue);
begin
   for each element E in Q do
      case E.kind of
         t1: t1print(E.t1value);
         ...
         tn: tnprint(E.tnvalue);
      end;
end print;
```

One can clearly see that this procedure has to be aware of the various types the queue elements may have. Whenever elements of another type are allowed to appear as elements in the queue, a new entry has to be added to the above case statement.

The same problem formulated in Smalltalk-style will have the following structure:

```
print                "print method of class Queue"
   self do: [ :e | e print ]
```

This can be paraphrased as:

```
print              "print method of class Queue"
   for each element E in Q do
       send print to E
```

In this case the selection of the precise code to be executed in response to the message `print` is not made in the print routine for queues, but is determined by the class of which each queue element is an instance. The code of the above print method thus becomes independent of the types of the elements in the queue.

This form of program organization is also known as *data-driven programming* and leads to easily extensible implementations. Another example may further illustrate this. In a conventional compiler for some language *L*, a source program is converted into an abstract syntax tree and this tree is processed in several phases. Typical operations to be performed are: *check* (check types), *dataflow* (perform dataflow analysis), *allocate* (allocate registers) and *generate* (generate code). Each of these operations will be implemented as a procedure operating on the complete syntax tree and knowledge about the various constructs that may occur in it has to be repeated in each of these four procedures. This same problem can be implemented in an object-oriented style by defining a class for each kind of statement in *L*. Each such class has to implement methods for the four desired operations. This leads to a totally different organization: all information about one language construct is concentrated in one class definition. This approach is superior over the traditional one, if language *L* is not fixed (i.e. *L* is still under design and language features may come and go), or if several dialects of *L* have to be implemented and one wants to maximize the sharing between these implementations. See [Veen86] for a description of this technique in the context of the SUMMER programming language [Klint85], and [Abelson, Sussman & Sussman85] for a description in the context of Lisp.

4.4. Disadvantages of object-oriented languages

The advantages of object-oriented languages for reusability and extensibility will by now be clear; what disadvantages do these languages have? First, the message/object model forces an asymmetric view on operations which are inherently symmetric such as, for instance, arithmetic operators. The problem then arizes how conversions can be inserted in mixed-mode expressions such as, e.g. an addition with operands of types integer and real. Straightforward application of the view that "`2 + 2.5` is evaluated by sending the message + with argument `2.5` to the integer instance `2`" automatically leads to selection of the addition operation on integers, which is clearly undesired. In this particular case, the problem can be solved by assigning a level of generality to all classes defining arithmetic operators and by defining coercion protocols between them: when a message is sent to a less general class with an argument of a more general one, the former should be coerced to the class of the latter and the message should be re-sent to the (coerced) class instance. No solutions exist for this phenomenon which are both elegant and general.

Secondly, not all problems can be modularized by means of strictly linear inheritance chains as provided by, for instance, Smalltalk. A typical example is the class `waterplane`, which could best be defined as a subclass of the classes `boat` and `plane`. Here one needs a form of *multiple inheritance*.

Finally, there is a certain overhead associated with run-time method selection. Measurements show that the operation of sending a message to an object is roughly twice as expensive as performing a procedure call [Cox86].

4.5. Object-oriented programming in existing languages

It was already mentioned in section 4.1 that object-oriented programming techniques can, in principle, also be used in conventional programming languages. In this section I will briefly review the language features that are required to support such an object-oriented style. These

features are: *information hiding, generic operations, dynamic binding, inheritance,* and *automatic storage management.*

Clearly, without an information hiding mechanism, it is impossible to implement a notion of "objects". Modules from Modula-2 and packages from Ada are acceptable from this point of view.

As we have seen in the preceding sections, it frequently occurs that messages with identical names are defined for several classes. Such messages are *generic* (or *polymorphic*), i.e. their behaviour depends on the type of object to which they are sent. Such operators are not expressible in Modula-2 (since overloading of names is forbidden) and only to a limited extend in Ada (since the language is defined in such a way that all overloaded and generic names can be resolved at compile-time).

The late moment of binding of names to their definition (also called *dynamic* binding) is one of the major reasons for the flexibility of object-oriented languages. This aspect of object-oriented languages is very hard to represent in statically typed languages, but is present in many not object-oriented languages such as, e.g., Lisp.

The notion of linear inheritance can be mimicked by the *import* and *use* constructs in, respectively, Modula-2 and Ada.

Strictly speaking, automatic storage management is a property of a programming language implementation, rather than a language feature in itself. Availability of automatic storage management, however, makes implementations of datatypes more reusable since details concerning allocation and deallocation can be omitted. If we consider, for instance, an implementation of queues based on linked lists, the question arizes whether a value just deleted from the queue should be deallocated or not. It is most likely that the implementor is forced to leave this decision to the user of the queue data type. But by doing so, he also compromises the implementation independence of his data type.

This discussion makes clear that the possibilities for object-oriented programming are absent in the case of Modula-2 and that they are limited in the case of Ada. What are the possibilities in other languages? I will briefly mention two cases: Lisp and C++.

Clearly, Lisp scores high when checking the features mentioned above. It is therefore not surprising that various Lisp extensions have been defined that provide primitives for object-oriented programming. The functionality of these systems is comparable to or even encompasses that of Smalltalk. In particular, improvements have been made in the area of introducing multiple inheritance and of user-defined inheritance schemes.

C++ [Stroustrup86] adds the notion of classes to the language C [Kernighan&Ritchie78], in a similar way as Simula added classes to Algol60. C++ is a superset of C and supports inheritance ("derived classes") and operator overloading, but--staying within the framework of compile-time typechecking--it does not support general polymorphic functions.

In object-oriented languages all operations are performed by means of message passing, independently of the amount of work required to compute them. For low-level operations such as, e.g., the addition of two numbers, the overhead of message passing will be relatively large. Object-oriented programming in a suitable conventional language has then distinct advantages since such low-level operations can be performed by direct procedure calls thus eliminating the message passing overhead.

5. CONCLUDING REMARKS

We have discussed two families of languages, each with different properties as far as reusability is concerned. The algorithmic languages place a strong emphasis on compile-time typechecking: types are identified by a name and all procedures and data structures in a program are explicitly typed. At compile-time, this type information can be used to check that all expressions and statements in a

program are properly typed. The advantages of this approach are reliability (all improperly typed expressions are detected at compile-time) and efficiency (there is no need to keep type information at run-time). The disadvantage is limited reusability due to the early binding of type information.

In object-oriented languages, types are not identified by name but by the set of operations that are defined on values of that type. This leads to very reusable code (each method only sends a limited set of different messages to its arguments and will therefore work for *all* types that implement these messages), but is less efficient (since run-time management of type information is required).

The distinction between compile-time and run-time typechecking is less absolute than is suggested here. By performing a sufficiently sophisticated analysis of programs one may *infer* type information that is only implicit in the text of the program. See, for instance, [Suzuki81] or [Boring & Ingalls 82] for an application of this idea to Smalltalk. In this way, one can eliminate most (but not all) run-time typechecks from a language that would otherwise require complete dynamic checking.

The best of two worlds can be obtained by using *elastic* typing schemes [Heering & Klint85], in which typechecking is performed as soon as type information becomes available (e.g., due to input operations) or can be inferred (e.g., by filling in details in an incomplete program).

ACKNOWLEDGEMENT

Ard Verhoog, Pum Walters and Freek Wiedijk made several comments on drafts of this paper.

LITERATURE

H. Abelson & G.J. Sussman with J. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, McGraw-Hill, 1985.

A.H. Boring & D.H.H. Ingalls, *A type declarartion and inference system for Smalltalk, Ninth ACM Symposium on Principles of Programming Languages*, 1982, pp. 133-141.

B.J. Cox, *Object Oriented Programming, An Evolutionary Approach* , Addison-Wesley, 1986.

O-J. Dahl, B. Myhrhaug & K. Nygaard, *SIMULA 67 Common Base Language,* Norwegian Computer Centre, Report S-22, 1971.

O-J. Dahl & C.A.R. Hoare, "Hierarchical Program Structures", in O-J. Dahl , E.W. Dijkstra & C.A.R. Hoare, *Structured Programming,* Academic Press, 1972.

A. Goldberg & D. Robson, *SMALLTALK-80, The Language and its Implementation,* Addison-Wesley, 1983.

J. Heering & P. Klint, " Towards monolingual programming environments", ACM Transactions on Programming Languages and Systems, 7 (1985) 2, pp. 183-213.

P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw & M. Sherman, *Studies in Ada Style,* Springer-Verlag, 1981.

K. Jensen & N. Wirth, *Pascal User Manual and Report,* Springer-Verlag 1974.

B.W. Kernigan & D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

P. Klint, *A Study in String Processing Languages,* Lecture Notes in Computer Science **205**, Springer-Verlag, 1985.

D.E. Knuth, *The Art of Computer Programming*, Volume 1: *Fundamental Algorithms*, Addison-Wesley, 1968.

P. Naur (ed.), "Revised report on the algorithmic language Algol 60", Communications of the ACM **6** (1963), pp. 1-17.

B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

N. Suzuki, "Inferring types in Smalltalk", *Eighth ACM Symposium on Principles of Programming Languages*, 1981, pp 187-199.

U.S. Department of Defense, *Reference Manual for the Ada Programming Language,* Mil-Std 1815a (1983).

A.H. Veen, *The Misconstrued Semicolon: Reconciling Imperative Languages and Dataflow Machines,* CWI Tract 26, Centre for Mathematics and Computer Science, Amsterdam, 1986.

N. Wirth, *Programming in Modula-2,* Third Edition, Springer-Verlag, 1985.