The Birth of Object Orientation: the Simula Languages[⋆]

Ole-Johan Dahl

Dept. of Informatics, University of Oslo, Norway

Abstract. The development of the programming languages Simula I and Simula 67 is briefly described. An attempt is made also to explain the cultural impact of the languages, in particular the object oriented aspects.

1 Introduction

In 1962 Kristen Nygaard, KN, initiated a project for the development of a discrete event simulation language, to be called Simula. At the time KN was the director of research at the Norwegian Computing Center, NCC, (a semi-governmental institution). KN also served as the administrative leader for the duration of the project. This required much creative manipulation in an environment that outside the NCC was largely hostile. The language development proper was a result of a close cooperation between KN and the author, OJD, whereas implementation considerations were mainly the responsibility of the latter.

We were both fostered at the Norwegian Defence Research Establishment in the pioneering group headed by Jan V. Garwick, the father of Computer Science in Norway. But our backgrounds were nevertheless quite different. KN had done Monte Carlo computations calibrating uranium rods for a nuclear reactor and later operations research on military systems. OJD had developed basic software together with Garwick and designed and implemented a high level programming language. Our difference in background probably accounts for some of the success of the Simula project.

The present paper mainly deals with language issues, including some thoughts on their possible cultural impact, especially on later programming languages. For other aspects of the project the reader is referred to [30].

Two language versions were defined and implemented. The first one, later called Simula I, was developed under a contract by UNIVAC. (UNIVAC wanted us to provide also a Fortran-based version, but that was abandoned because the block structure turned out to be essential to our approach.) It was up and running by the end of 1964. The second version, Simula 67, was sponsored by the NCC itself. It is a generalization and refinement of the former, fairly ambitious, intended mainly as a general purpose programming language, but with simulation capabilities.

[⋆] An almost identical version of this paper has been published in *Software pioneers*, Springer, 2002.

2 Simula I

It was decided at an early stage that our language should be based on a well known one. Algol 60 was chosen for the following main reasons:

– the block structure,
– good programming security,
– European patriotism.

We realised that in order to profit from block structure in simulation models it would be necessary to break out of the strict LIFO regime of block instances in Algol. Thus, a new storage management system was developed based on a list structured free store, [3]. Then a useful simulation language could be defined by adding a few special purpose mechanisms to Algol 60:

– A procedure-like **activity** declaration giving rise to so called "processes". Processes could range from record-like data structures to block structured programs executing in a coroutine-like fashion, [35], [9], over a simulated system time.
– Explicit process pointers for dynamic naming and referencing. (The pointers were indirect through list forming "element" records.)
– A mechanism for accessing, from the outside, quantities local to the outermost block level of processes, designed so that the access security inherent in Algol would be maintained (the **inspect** construct).
– A few run time mechanisms for the scheduling and sequencing of processes in system time, such as *hold*(...), suspending the calling process for a specified amount of system time.

The following skeleton example could be a small fragment of a road traffic simulation. It is taken from the Simula I manual, [4], but slightly extended. It may serve to indicate the flavour of the language.

```
SIMULA begin activity Car;
             begin real X0, T0, V;
                   real procedure X; X := X0+V*(time−T0);
                   procedure newV(Vnew); real Vnew;
                     begin X0 := X; T0 := time; V := Vnew end;
                   Car behaviour: ......; hold(<travel time>); ......
             end of Car;
             activityPolice;
             begin ......; inspect <process> when Car do
                   if X <is within city> and V > 50 then
                     begin newV(50); <give fine> end; ......
             end of Police;
             main program: <initialise>; hold(<simulation period>)
end of simulation model;
```

The example shows that the idea of data objects with associated operators was under way already in 1965. According to a comment in [4] it was a pity that the variable attributes of a *Car* process could not be hidden away in a subblock. It would have required the accessing procedures to be hidden similarly.

New processes would be generated explicitly. For programming security reasons, however, process deletions had to be implicit, in our implementation through reference counts and a last resort garbage collector. The bulk of the implementation effort therefore consisted in writing a new run time system for the Algol system provided by UNIVAC; the compiler extensions, on the other hand, were minimal. The "block prefix" **SIMULA** served to introduce the Simula I additions to Algol. Consequently any Algol program not containing that keyword would execute normally on our compiler. That was an important consideration in those days.

A paper on Simula I was published in CACM 1966, [5]. It was also the main topic of lectures given by OJD at the NATO Summer School at Vilard-de-Lans the same year. The lectures were written up and published as a chapter of [6].

The language was used for simulation purposes as well as for teaching at several locations at home and abroad, also within the UNIVAC organization. A modified version was used for Burroughs computers. This was through the advocacy of Don Knuth and J. McNeley, the authors of SOL, another Algol-like simulation language.

3 Simula 67

In spite of the success of Simula I as a practical tool it became increasingly clear that the activity/process concepts, if stripped from all references to simulated time, would be useful for programming and system design in general. If possible the special purpose simulation facilities should be definable within the new language. Also the list processing facilities of Simula I would be useful, although we felt that the referencing mechanism should be simplified.

At the Vilard-de-Lans Summer School Tony Hoare had put forth a proposal for "record handling" with record classes and subclasses, as well as record references restricted to, or "qualified by", a given class or subclass by declaration. Attribute accessing was by dot notation, see [19], as well as [17] and [18].

We chose the terms "class" and "objects" of classes for our new Simula. The notion of subclass was especially appealing to us, since we had seen many cases of objects belonging to different classes having common properties. It would be useful to collect the common properties in a separate class, say C to be specialised differently for different purposes, possibly in different programs. The solution came with the idea of class prefixing: using C as a prefix to another class, the latter would be taken to be a subclass of C inheriting all properties of C.

Technically the subclass would be seen as "concatenated" class in which the parameter parts, the block heads, and the block tails of the two classes were juxtaposed (The block tail of the prefixing class could be separated into initial actions and final actions, that of the prefixed class sandwiched between them.) The attributes of a compound object would be accessible by dot notation down to the prefix level of the qualifying class of the reference used. Access to deeper levels could be achieved by class discrimination as in Simula I.

The breakthrough happened in January of 1967. An IFIP sponsored working conference on simulation languages had been approved to take place in Oslo in May. There followed some hectic winter months during which our new concepts were explored and tested. A paper was ready just in time for advance distribution to the invitees, [7]. The new language was to be called Simula 67, [8]. The paper occurring in the proceedings was mended by the addition of "virtual" specifications, see below.

One way of using a class, which appeared important to us, was to collect concepts in the form of classes, procedures, etc under a single hat. The resulting construct could be understood as a kind of "application language" defined on top of the basic one. It would typically be used as a prefix to an in-line block making up the application program.

We illustrate this idea by showing a simplified version of a *SIMULATION* class defining the simulation oriented mechanisms used in our Simula I example.

```
class SIMULATION;
begin class process;
        begin real EventTime, NextEvent; ...... end;
        ref(process) current;
        comment current points to the currently operating process.
                    It is the head of the "time list" of scheduled ones,
                    sorted with respect to nondecreasing EventTimes;
        real procedure time; time := current.EventTime;
        procedure hold(deltaT); real deltaT;
        begin current.EventTime := time+deltaT;
                if time ≥ current.NextEvent.EventTime then
                begin ref(process)P; P :- current; current :- P.NextEvent;
                        ⊲move P to the right position in the time list⊳;
                resume(current) end end of hold;
        ......
end of SIMULATION;

SIMULATION begin
  process class Car;
  begin real X0, T0, V;
          real procedure X; X := X0+V*(time−T0);
          procedure newV(Vnew); real Vnew;
             begin X0 := X; T0 := time; V := Vnew end;
          Car behaviour: ......; hold(⊲travel time⊳); ......
  end of Car;
  process Police;
  begin ......; inspect ⊲process⊳ when Car do
```

```
            if $X$ ⊲is within city⊳ and $V > 50$ then
               begin $newV(50)$; ⊲give fine⊳ end; ......
    end of Police;
    main program: ⊲initialise⊳; hold(⊲simulation period⊳)
end of simulation model;
```

Thus, the "block prefix" of Simula I is now an ordinary class declared within the new language, and the special purpose **activity** declarator is replaced by *process* **class**.

We chose to introduce a special set of operators for references, in order to make it clear that the item in question is a reference, not (the contents of) the referenced object. The *resume* operator is a basic coroutine call, defined for the whole language.

Notice that the class *SIMULATION* is completely self-contained. If some necessary initializing operations were added, it could be separately compiled and then used repeatedly in later programs. In actual fact a somewhat more elaborate class is predefined in Simula 67, providing an application language for simulation modelling. That class is itself prefixed by one containing mechanisms for the management of circular lists.

It is fair to say that Simula 67 invites to bottom up program design, especially through the possibility of separate compilation of classes. As a last minute extension, however, we introduced a top down oriented mechanism through a concept of "virtual procedures".

In general attribute identifiers may be redeclared in subclasses, as is the case of inner blocks. The identity of an attribute is determined by the prefix level of the accessing occurrence, or, if the access is remote, by the class qualifying the object reference in question. In this way any ambiguity of identifier binding is resolved textually, i.e at compile time; we call it static binding.

On the other hand, if a procedure P is specified as **virtual** in a class C the binding scheme is semi-dynamic. Any call for P occurring in C or in any subclass of C will bind to that declaration of P which occurs at the innermost prefix level of the actual object containing such a declaration (and similarly for remote accesses). Thus, the body of the procedure P may, at the prefix level of C, be postponed to occur in any subclass of C. It may even be replaced by more appropriate ones in further subclasses.

This binding scheme is dynamic in the sense that it depends on the class membership of the actual object. But there is nevertheless a degree of compiler control; the access can be implemented as indirect through a table produced by the compiler for C and for each of its subclasses.

As a concrete example the "fine giving" operation of the above example could be formalised as a virtual procedure, as follows: Redefine the head of the prefixed block as a subclass *RoadTraffic* of *SIMULATION*. In addition to the classes *Car* and *Police* declarations introduce the following specification:

virtual procedure *Fine*(*cr*); **ref**(*Car*)*cr*;

If appropriate the *RoadTraffic* class may be separately compiled. Using that class as a block prefix at some later time, a suitable fining procedure can be defined in that block head.

There is an alternative more implementation oriented view of virtual procedures. As mentioned in connection with Simula I, deletion of objects would have to be implicit (in Simula 67 by garbage collector alone). But then there is a danger of flooding the memory with useless data, especially if there are implicit pointers between block instances. In Algol 60 there must be a pointer from a procedure activation back to its caller in order to implement procedure parameters and parameters "called by name". Such pointers from objects back to their generating block instance would have been destructive. So, it was decided that parameters to objects must be called by "value" (including object references). The absence of procedure parameters, however, was felt to be a nuisance. Fortunately the virtual procedure mechanism provided a solution to the dilemma: a virtual procedure can be seen as a parameter, where the actual parameter is a procedure residing safely within the object itself, at an appropriate prefix level. There is the additional advantage that the procedure has direct access to attributes of the object containing it.

Similar considerations led to forbidding class prefixing across block levels. Fortunately this would not prevent the use of separately compiled, "external" classes. Since there is no reference to nonlocal quantities in such a class, it can be called in as an external one at any block level of a user program.

4 Language Finalisation and Distribution

A couple of weeks after the IFIP Conference a private "Simula Common Base Conference" was organised, attended by several interested persons. The objective was to agree on the definition of a common core language. We made a proposal to the CBC to extend the language by "class-like" types giving rise to permanently named objects, directly accessed, thus extending the Algol variable concept. The proposal was prudently voted down, as not sufficiently worked through. However, a Pascal-like **while** statement was added, and the virtual mechanism was slightly revised.

A "Simula Standards Group", SSG, was established, to consist of representatives from the NCC and various implementation groups. 5 compilers were implemented initially. It was decided that the NCC would propose mechanisms for text handling, I/O, and file handling. Our good colleague Bjørn Myhrhaug of the NCC gave three alternatives for text handling and I/O. The ones chosen by the SSG would have required class-like types in order to be definable within the Common Base.

The class concept as it was formulated originally, was too permissive for the purpose of developing large systems. There was no means of enforcing a programming discipline protecting local class invariants (such as those expressed verbally for the *Simulation* class example). This was pointed out by Jacob Palme of the Swedish defence research institute. He proposed hiding mechanisms for protecting variable attributes from unauthorised updating. The proposal was approved by the SSG as the last addition ever to the language. The authors toyed with the idea of class-like types for some time, but it was never implemented.

The first compilers were operative already in 1969, three for Control Data computers. Then came implementations for UNIVAC and IBM machines. The general distribution of the compilers was, however, greatly hampered by the high prices asked for the compilers by the NCC, very unwisely enforced by the NTNF (Norwegian Council for Scientific and Technical Research) stating that programming languages only had a 3-5 years lifetime and thus had to provide profits within this time span. However, a nice compiler for the DEC 10 system, implemented by a Swedish team in the early 1970's, contributed considerably to the spreading of the language. Lectures by OJD at NATO Summer Schools, as well as a chapter in [9] must have made the new concepts better known in academic circles.

The most important new concept of Simula 67 is surely the idea of data structures with associated operators (and with or without own actions), called *objects*. There is an important difference, except in trivial cases, between

– *the inside view* of an object, understood in terms of local variables, possibly initialising operations establishing an invariant, and implemented procedures operating on the variables maintaining the invariant, and
– *the outside view*, as presented by the remotely accessible procedures, including some generating mechanism, dealing with more "abstract" entities.

This difference, as indicated by the *Car* example in Simula I, and the associated comments, underlies much of our program designs from an early time on, although not usually conscious and certainly not explicitly formulated. (There is e.g an intended invariant of any *Car* object vaguely stating that its current position X is the right one in view of the past history of the object.)

It was Tony Hoare who finally expressed mathematically the relationship of the two views in terms of an "abstraction function", see [20]. He also expressed requirements for the concrete operations to correctly represent the corresponding abstract ones. Clearly, in order to enforce the use of abstract object views, read access to variable attributes would also have to be prevented.

5 Cultural Impact

The main impact of Simula 67 has turned out to be the very wide acceptance of many of its basic concepts: objects, but usually without own actions, classes, inheritance, and virtuals, often the default or only way of binding "methods", (as well as pointers and dynamic object generation).

There is universal use of the term "object orientation", OO. Although no standard definition exists, some or all of the above ideas enter into the OO paradigm of system development. There is a large flora of OO languages around for programming and system specification. Conferences on the theory and practice of OO are held regularly. The importance of the OO paradigm today is such that one must assume something similar would have come about also without the Simula effort. The fact remains, however, that the OO principle was introduced in the mid 60's through these languages.

Simula 67 had an immediate success as a simulation language, and was, for instance extensively used in the design of VLSI chips, e.g. at INTEL. As a general programming language, its impact was enhanced by lectures at NATO Summer Schools given by OJD, materialized as a chapter in a book on structured programming, [9]. The latter has influenced research on the use of abstract data types, e.g., the CLU language, [29], as well as research on monitors and operating system design, [21].

A major new impact area opened with the introduction of workstations and personal computers. Alan Kay and his team at Xerox PARC developed Smalltalk, [15], an interactive language building upon Simula's objects, classes and inheritance. It is oriented towards organising the cooperation between a user and her/his personal computer.

An important step was the integration of OO with a graphical user interfaces, leading the way to the Macintosh Operating System, and then to Windows.

In the larger workstation field, Lisp was (and in some places still is) an important language, spawning dialects such as MACLISP, [16], at MIT, and InterLisp at Xerox PARC. Both got OO facilities, MACLISP through ZetaLisp introducing also multiple inheritance, [2], and InterLisp through LOOPS (Lisp Object-Oriented Programming System). The object-oriented component of the merged effort, CommonLisp, is called CLOS (Common Lisp Object System), [24].

With the general acceptance of object-orientation, object-oriented databases started to appear in the 1980's. The demand for software reuse also pushed OO tools, and in the 1990's OO tools for system design and development became dominant in that field. UML (Unified Modeling Language), [1], is very much used, and CORBA (Common Object Request Broker Architecture), is a widely accepted tool for interfacing OO systems. The Microsoft Component Object Model, COM, [27], is an important common basis for programming languages such as C♯, as well as other tools.

A large number of OO programming languages have appeared. We list below some of the more interesting or better known languages, in addition to those mentioned above.

- BETA is a compilable language built around a single abstraction mechanism, that of patterns, which can be specialised to classes, singular objects, types, as well as procedures. It was developed from the later 1970's by KN and colleagues in Denmark, [25], [26].
- Bjarne Stroustrup extended the Unix-related language C with several Simula-inspired mechanisms. The language, called C++, has been much used and has contributed importantly to the dissemination of the OO ideas, [33]. Since C is fairly close to machine code, security aspects are not the best. As a result, complex systems may be difficult to implement correctly. C++ has been revised and extended, e.g. by multiple inheritance.

– Eiffel, [28], is an OO programming language designed by Bertrand Meyer in the 1980's, well known and quite widely used. It has pre- and post-conditions and invariants.

– SELF, [34], is an OO language exploring and using object cloning instead of object generation from a class declaration.

– JAVA, [22], is a recent Simula-, Beta-, and C++-inspired language, owing much of its popularity to its integration with the Internet. Its syntax is unfortunately rather close to that of C++ and thus C (but with secure pointers). It contains Beta-like singular objects and nested classes, but not general block structure. Parallel, "multi-threaded", execution is introduced, but outside compiler control. As a result, much of the programming security otherwise inherent in the language is lost. The synchronisation mechanisms invite to inefficient programming and do not facilitate good control of process sequencing, see [14].

The authors believed that the use of class declarations for the definition of "application languages" as natural extensions of a basic one would be of special importance in practice. However, although various kinds of packages or modules are defined for many languages, they are not consequences of a general class declaration as in Simula 67.

The coroutine-like sequencing of Simula has not caught on as a general purpose programming tool. A natural development, however, would have been objects as concurrent processes, e.g. as in COM.

One may fairly ask how it could happen that a team of two working in the periphery of Europe could hit on programming principles of lasting importance. No doubt a bit of good luck was involved. We were designing a language for simulation modelling, and such models are most easily conceived of in terms of cooperating objects. Our approach, however, was general enough to be applicable to many aspects of system development.

KN oriented his activities for some years to trade union work, as well as system development and description, see [23]. In 1976 he turned back to programming language design, see BETA above. In [32] he introduced new constructs for OO layered distributed systems.

OJD has been professor of Informatics at Oslo University for the period 1968–1999, developing curricula including OO programming. He has explored the concept of time sequences to reason about concurrent systems, [10], [11]. In [13] he applies techniques, such as Hoare logic and Guttag-Horning axiomatisation of types and subtypes, [31], to the specification and proof of programs, including OO ones. See also [12].

Of the Simula authors especially KN has been consistently promoting the OO paradigm for system development.

Acknowledgments

The author is greatly indebted to Kristen Nygaard for helping to explain the impact of object orientation in various areas of programming and system work. Also Olaf Owe has contributed.

References

1. G. Booch, J. Rumbaugh, I. Jacobson: *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
2. H. Cannon: *Flavors, A Non-Hierarchical Approach to Object-Oriented Programming.* Draft 1982.
3. O.-J. Dahl: *The Simula Storage Allocation Scheme.* NCC Doc. 162, 1963.
4. O.-J. Dahl, K. Nygaard: *SIMULA – A language for programming and description of discrete event systems. Introduction and user's manual.* NCC Publ. no. 11, 1965.
5. O.-J. Dahl, K. Nygaard: SIMULA – an ALGOL-based Simulation Language. *CACM 9(9),* 671–678, 1966.
6. O.-J. Dahl: Discrete Event Simulation Languages. In F. Genuys, ed.: *Programming Languages.* Academic Press, pp 349–394, 1968.
7. O.-J. Dahl, K. Nygaard: Class and Subclass Declarations. In J. Buxton, ed.: *Simulation Programming Languages.* Proceedings from the IFIP Working Conference in Oslo, May 1967. North Holland, 1968.
8. O.-J. Dahl, B. Myhrhaug, K. Nygaard: *SIMULA 67 Common Base Language.* Norwegian Computing Center 1968.
9. 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, pp. 175–220.
10. O.-J. Dahl: Can Program Proving be Made Practical? In M. Amirchahy, D. Neel: *Les Fondements de la Programmation.* IRIA 1977, pp. 57–114.
11. O.-J. Dahl: Time Sequences as a Tool for Describing Process Behaviour. In D. Bjørner, ed.: *Abstract Software Specifications, LNCS 86,* pp. 273-290.
12. O.-J. Dahl, O. Owe: Formal Development with ABEL. In *VDM91, LNCS 552,* pp. 320–363.
13. O.-J. Dahl: *Verifiable Programming*, Hoare Series, Prentice Hall 1992.
14. O.-J. Dahl: A Note on Monitor Versions. In *Proceedings of Symposium in the Honour of C.A.R. Hoare at his resignation from the University of Oxford.* Oxford University 1999. Also available at `www.ifi.uio.no/~olejohan`. (Department of Informatics, University of Oslo).
15. A. Goldberg, D. Robson: *Smalltalk-80: The Language and its Implementation.* Addison Wesley, 1984.
16. B.S. Greenberg: *The Multics MACLISP Compiler. The Basic Hackery – a Tutorial.* MIT Press 1977, 1988, 1996.
17. C.A.R. Hoare: Record Handling. In *ALGOL Bulletin no. 21.* 1965.
18. C.A.R. Hoare: Further Thoughts on Record Handling. In *ALGOL Bulletin no. 23.* 1966.
19. C.A.R. Hoare: Record Handling. In F. Genuys, ed.: *Programming Languages.* Academic Press, pp 291–346, 1968.

20. C.A.R. Hoare: Proof of the Correctness of Data Representations. *Acta Informatica*, vol. 1, 1972.
21. C.A.R. Hoare: Monitors: an Operating System Structuring Concept. *Comm. ACM* 17(10)(1974), pp. 549-557.
22. J. Gosling, Bill Joy, G. Steele: *The Java Language Specification.* Java(tm) Series, Addison-Wesley 1989.
23. P. Håndlykken, K. Nygaard: The DELTA System Description Language: Motivation, Main Concepts and Experience from use. In: Software Engineering Environments (ed. H. Hnke), GMD, North-Holland, 1981.
24. S.E. Keene: *Object-Oriented Programming in COMMON LISP-A Programmer's Guide to CLOS.* Addison-Wesley 1989.
25. B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Abstraction Mechanisms in the BETA Programming Language. *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages.* Austin, Texas, 1983.
26. O.L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object Oriented Programming in the BETA Programming Language.* Addison-Wesley/ACM Press 1993.
27. R.C. Martin: *Design Principles and Design Patterns.* Microsoft, `www.objectmentor.com`.
28. B. Meyer: *Eiffel: The Language.* Prentice Hall 1992.
29. B. Liskov, A. Snyder, R. Atkinson, C. Schaffert: Abstraction Mechanisms in CLU. *Comm. ACM 20:8* (1977), PP. 564-576.
30. K. Nygaard, O.-J. Dahl: SIMULA Session. In R. Wexelblatt, ed.: *History of Programming Languages.* ACM 1981.
31. O. Owe, O.-J. Dahl: Generator Induction in Order Sorted Algebras. *Formal Aspects of Computing* (1991), 3:2–20.
32. K. Nygaard: GOODS to Appear on the Stage. *Proceedings of the 11th European Conference on Object-Oriented Programming.* Springer 1997
33. B. Stroustrup: *The C++ Programming Language.* Addison-Wesley 1986.
34. D. Ungar, R.B. Smith: SELF: The Power of Simplicity. In *SIGPLAN Notices 22(12),* 1987.
35. A. Wang, O.-J. Dahl: Coroutine Sequencing in a Block Structured Environment. In *BIT 11* 425–449, 1971.