# The When, Why and Why Not of the BETA Programming Language

Bent Bruun Kristensen
University of Southern Denmark
Campusvej 55
DK-5230 Odense M, Denmark
+45 65 50 35 39
bbk@mmmi.sdu.dk

Ole Lehrmann Madsen
University of Aarhus
Åbogade 34
DK-8200 Århus N, Denmark
+45 89 42 56 70
ole.l.madsen@daimi.au.dk

Birger Møller-Pedersen
University of Oslo
Gaustadalleen 23
NO-0316 Oslo, Norway
+47 22 85 24 37
birger@ifi.uio.no

## Abstract

This paper tells the story of the development of BETA: a programming language with just one abstraction mechanism, instead of one abstraction mechanism for each kind of program element (classes, types, procedures, functions, etc.). The paper explains how this single abstraction mechanism, the pattern, came about and how it was designed to be so powerful that it covered the other mechanisms.

In addition to describing the technical challenge of capturing all programming elements with just one abstraction mechanism, the paper also explains how the language was based upon a modeling approach, so that it could be used for analysis, design and implementation. It also illustrates how this modeling approach guided and settled the design of specific language concepts.

The paper compares the BETA programming language with other languages and explains how such a minimal language can still support modeling, even though it does not have some of the language mechanisms found in other object-oriented languages.

Finally, the paper tries to convey the organization, working conditions and social life around the BETA project, which turned out to be a lifelong activity for Kristen Nygaard, the authors of this paper, and many others.

**Categories and subject descriptor**s: D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications – BETA; D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features; K.2 [HISTORY OF COMPUTING] Software; D.1.5 [PROGRAMMING TECHNIQUES]: Object-oriented Programming; **General Terms**: Languages; **Keywords**: programming languages, object-oriented programming, object-oriented analysis, object-oriented design, object-oriented modeling, history of programming.

## 1. Introduction

This paper is a description of what BETA is, why it became what it is and why it lacks some of the language constructs found in other languages. In addition, it is a history of the design and implementation of BETA, its main uses and its main influences on later research and language efforts.

BETA is a programming language that has only one abstraction mechanism, the pattern, covering abstractions like record types, classes with methods, types with operations, methods, and functions. Specialization applies to patterns in general, thus providing a class/subclass mechanism for class patterns, a subtype mechanism for type patterns, and a specialization mechanism for methods and functions. The latter implies that inheritance is supported for methods – another novel characteristic of BETA. A pattern may be virtual, providing virtual methods as in other object-oriented languages. Since a pattern may be used as a class, virtuality also supports virtual classes (and types).

This paper is also a contribution to the story of the late Kristen Nygaard, one the pioneers of computer science, or informatics as he preferred to call it. Nygaard started the BETA project as a continuation of his work on SIMULA and system description. This was the start of a 25-year period of working with the authors, not only on the design of the BETA language, but also on many other aspects of informatics.

The BETA project was started in 1976 and was originally supposed to be completed in a year or two. For many reasons, it evolved into an almost lifelong activity involving Nygaard, the authors of this paper and many others. The BETA project became an endeavor for discussing issues related to programming languages, programming and informatics in general.

The BETA project covers many different kinds of activities from 1976 until today. We originally tried to write this paper in historic sequence, and so that it can be read with little or no prior knowledge of BETA. We have not, however, organized the paper according to time periods, since the result included a messy mix of distinct types of

events and aspects, too much overlap, and too little focus on important aspects. The resulting paper is organized as follows:

- Section 2 describes the background of the project.
- Section 3 describes the course of the BETA project, including people, initial research ideas, project organization and the process as well as personal interactions.
- Section 4 describes the motivation and development of the modeling aspects and the conceptual framework of BETA.
- Section 5 describes parts of the rationale for the BETA language, the development of the language, and essential elements of BETA.
- Section 6 describes the implementation of BETA.
- Section 7 describes the impact and further development of BETA.

Sections 3-6 form the actual story of BETA enclosed by background (section 2) and impact (section 7). Sections 3-6 describe distinct aspects of BETA. The story of the overall BETA project in section 3 forms the foundation/context for the following aspects. This is how it all happened. Modeling is essential for the design of BETA. This perspective on design of and programming in object-oriented languages is presented in section 4. Throughout the presentation of the various language elements in the following section the choices are discussed and motivated by the conceptual framework in section 4. Section 5 presents the major elements of BETA. Because BETA may be less known, a more comprehensive presentation is necessary in order to describe its characteristics. However, the presentation is still, and should be, far from a complete definition of the language. Finally, the implementation of BETA, historically mainly following after the language design, is outlined in section 6.

In order to give a sequential ordering of the various events and activities, a timeline for the whole project is shown in the appendix. In the text events shown in the timeline are printed in Tunga font. For example, text like: "… BETA Project start …" means that this is an event shown in the time line.

## 2. Background

This section describes the background and setting for the early history of BETA. It includes personal backgrounds and a description of the important projects leading to the BETA project.

### 2.1 People

The BETA project was started in 1976 at the Computer Science Department, Aarhus University (DAIMI). Bent Bruun Kristensen and Ole Lehrmann Madsen had been students at DAIMI since 1969 – Birger Møller-Pedersen originally started at the University of Copenhagen, but moved to DAIMI in 1974. Nygaard was Research Director at the Norwegian Computing Centre (NCC), Oslo, where the SIMULA languages [32-34, 130] were developed in the sixties.

In the early seventies, the programming language scene was strongly influenced by Pascal [161] and structured programming. SIMULA was a respected language, but not in widespread use. Algol 60 [129] was used for teaching introductory programming at DAIMI. Kristensen and Madsen were supposed to be introduced to SIMULA as the second programming language in their studies. However, before that happened Pascal arrived on the scene in 1971, and most people were fascinated by its elegance and simplicity as compared to Algol. Pascal immediately replaced SIMULA as the second language and a few years later Pascal also replaced Algol as the introductory language for teaching at DAIMI. A few people, however, found SIMULA superior to Pascal: the Pascal record and variant record were poor substitutes for the SIMULA class and subclass.

Although SIMULA was not in widespread use, it had a strong influence on the notion of structured programming and abstract data types. The main features of SIMULA were described in the famous book by Dahl, Dijkstra and Hoare on structured programming [29]. Hoare's groundbreaking paper *Proof of Correctness of Data Representation* [56] introduced the idea of defining abstract data types using the SIMULA class construct and the notion of class invariant.

Kristen Nygaard visiting professor at DAIMI**.** Nygaard became a guest lecturer at DAIMI in 1973; in 1974/75 he was a full-time visiting professor, and after that he continued as a guest lecturer for several years. Among other things, Nygaard worked with trade unions in Norway to build up expertise in informatics. At that time there was a strong interest among many students at DAIMI and other places in the social impact of computers. Nygaard's work with trade unions was very inspiring for these students. During the '70s and '80s a number of similar projects were carried out in Scandinavia that eventually led to the formation of the research discipline of *system development with users*, later called *participatory design*. The current research groups at DAIMI in object-oriented software systems and human computer interaction are a direct result of the cooperation with Nygaard. This is, however, another story that will not be told here. The design of BETA has been heavily influenced by Nygaard's overall perspective on informatics including social impact, system description with users, philosophy, and programming languages. For this reason the story of BETA cannot be told without relating it to Nygaard's other activities.

Morten Kyng was one of the students at DAIMI who was interested in social aspects of computing. In 1973 he listened to a talk by Nygaard at the Institute of Psychology at Aarhus University. After the talk he told Nygaard that he was at the wrong place and invited him to repeat his talk at DAIMI. Kyng suggested to DAIMI that Nygaard be invited as a guest lecturer. The board of DAIMI decided to do so, since he was considered a good supplement to the many theoretical disciplines in the curriculum at DAIMI at that time. Madsen was a student representative on the board; he was mainly interested in compilers and was thrilled about Nygaard being a guest lecturer. He thought that DAIMI would then get a person that knew about the SIMULA compiler. This turned out not to be the case: compiler technology was not his field. This was our first indication that Nygaard had a quite different approach to informatics and language design from most other researchers.

## 2.2 The SIMULA languages

Since SIMULA had a major influence on BETA we briefly mention some of the highlights of SIMULA. A comprehensive history of the SIMULA languages may be found in the HOPL-I proceedings [35] and in [107]. SIMULA and object-oriented programming were developed by Ole-Johan Dahl and Nygaard. Nygaard's original field was operations research and he realized early on that computer simulations would be a useful tool in this field. He then made an alliance with Dahl, who – as Nygaard writes in an obituary for Dahl [132] – had an exceptional talent for programming. This unique collaboration led to the first SIMULA language, SIMULA I, which was a simulation language. Dahl and Nygaard quickly realized that the concepts in SIMULA I could be applied to programming in general and as a result they designed SIMULA 67 – later on just called SIMULA. SIMULA is a general-purpose programming language that contains Algol as a subset.

Users of today's object-oriented programming languages are often surprised that SIMULA contains many of the concepts that are now available in mainstream object-oriented languages:

- Class and object: A class defines a template for creating objects.
- Subclass: Classes may be organized in a classification hierarchy by means of subclasses.
- Virtual methods: A class may define virtual methods that can be redefined (sometimes called overridden) in subclasses.
- Active objects: An object in SIMULA is a coroutine and corresponds to a thread.
- Action combination: SIMULA has an "inner" construct for combining the statement-parts of a class and a subclass.
- Processes and schedulers: It is straightforward in SIMULA to write new concurrency abstractions including schedulers.
- Frameworks: SIMULA provided the first object-oriented framework in form of class Simulation, which provided SIMULA I's simulation features.
- Automatic memory management, including garbage collection.

Most of the above concepts are now available in object-oriented languages such as C++ [148], Eiffel [125], Java [46], and C# [51]. An exception is the SIMULA notion of an active object with its own action sequence, which strangely enough has not been adopted by many other languages (one exception is UML). For Dahl and Nygaard it was essential to be able to model concurrent processes from the real world.

The ideas of SIMULA have been adopted over a long period. Before object orientation caught on, SIMULA was very influential on the development of abstract data types. Conversely, ideas from abstract data types later led to an extension of SIMULA with constructs like **public**, **private** and **protected** – originally proposed by Jakob Palme [137].

## 2.3 The DELTA system description language

When Nygaard came to DAIMI, he was working on system description and the design of a new language for system description based on experience from SIMULA. It turned out that many users of SIMULA seemed to get more understanding of their problem domain by having to develop a model using SIMULA than from the actual simulation results. Nygaard together with Erik Holbæk-Hanssen and Petter Håndlykken had thus started a project on developing a successor to SIMULA with main focus on system description, rather than programming. This led to a language called DELTA [60].

DELTA means 'participate' in command form in Norwegian. The name indicates another main goal of the DELTA language. As mentioned, Nygaard had started to include users in the design of systems and DELTA was meant as a language that could also be used to communicate with users – DELTA (participate!) was meant as an encouragement for users to participate in the design process.

The goal of DELTA was to improve the SIMULA mechanisms for describing real-world systems. In the real world, activities take place concurrently, but real concurrency is not supported by SIMULA. To model concurrency SIMULA had support for so-called quasi-parallel systems. A simulation program is a so-called discrete event system where a simulation is driven by discrete events generated by the objects of the simulation. All state changes had to be described in a standard imperative way by remote procedure calls (message calls),

assignments and control structures. DELTA supports the description of true concurrent objects and uses predicates to express state changes and continuous changes over time. The use of predicates and continuous state changes implied that DELTA could not be executed, but as mentioned the emphasis was on system description.

DELTA may be characterized as a specification language, but the emphasis was quite different from most other specification languages at that time such as algebraic data types, VDL, etc. These other approaches had a mathematical focus in contrast to the system description (modeling) focus of DELTA.

DELTA had a goal similar to that of the object-oriented analysis and design (OOA/OOD) methodologies (like that of Coad and Yourdon [25]) that appeared subsequently in the mid-'80s. The intention was to develop languages and methodologies for modeling real-world phenomena and concepts based on object-oriented concepts. Since SIMULA, modeling has always been an inherent part of language design in the Scandinavian school of object orientation. The work on DELTA may be seen as an attempt to further develop the modeling capabilities of object-orientation.

The report describing DELTA is a comprehensive description of the language and issues related to system description. DELTA has been used in a few projects, but it is no longer being used or developed.

The system concept developed as part of the DELTA project had major influence on the modeling perspective of BETA – in Section 4.1 we describe the DELTA system concept as interpreted for BETA.

## 2.4 The Joint Language Project

BETA project start. The BETA project was started in 1976 as part of what was then called the Joint Language Project (JLP). The JLP was a joint project between researchers at DAIMI, The Regional Computing Center at the University of Aarhus (RECAU), the NCC and the University of Aalborg.

Joint Language Project start. The initiative for the JLP was taken in the autumn of 1975 by the late Bjarner Svejgaard, director of RECAU. Svejgaard suggested to Nygaard that it would be a good idea to define a new programming language based on the best ideas from SIMULA and Pascal. Nygaard immediately liked the idea, but he was more interested in a successor to SIMULA based on the ideas from DELTA. In the BETA Language Development report from November 1976 [89] the initial purpose of the JLP was formulated as twofold:

1. To develop and implement a high-level programming language as a projection of the DELTA system description language into the environment of computing equipment.
2. To provide a common central activity to which a number of research efforts in various fields of informatics and at various institutions could be related.

The name GAMMA was used for this programming language.

JLP was a strange project: on the one hand there were many interesting discussions of language issues and problems, while on the other hand there was no direct outcome. At times we students on the project found it quite frustrating that there was no apparent progress. We imagine that this may have been frustrating for the other members of the project as well. In hindsight we believe that the reason for this may have been a combination of the very different backgrounds and interests of people in the team combined with Nygaard's lack of interest in project management. Nygaard's strengths were his ability to formulate and pursue ambitious research goals, and his approach to language design with emphasis on modeling was unique.

Many issues were discussed within the JLP, mainly related to language implementation and some unresolved questions about the DELTA language. As a result six subprojects were defined:

- **Distribution and maintenance**. This project was to discuss issues regarding software being deployed to a large number of computer installations of many different types. This included questions such as distribution formats, standardized updating procedures, documentation, interfaces to operating systems, etc.
- **Value types**. The distinction between object and value was important in SIMULA and remained important in DELTA and BETA. For Nygaard classes were for defining objects and types for defining values. He found the use of the class concept for defining abstract data types a 'doubtful approach, easily leading to conceptual confusion' with regard to objects and values. In this paper we use the term value type[1] when we refer to types defining values. The purpose of this subproject was to discuss the definition of value types. We return to value types in Sections 5.1.1 and 5.8.2.
- **Control structures** within objects. The purpose of this subproject was to develop the control structures for GAMMA.
- **Contexts**. The term "system classes" was used in SIMULA to denote classes defining a set of predefined concepts (classes) for a program. The classes SIMSET and SIMULATION are examples of such system classes.

[1] In other contexts, we use the term *type,* as is common within programming languages.

Møller-Pedersen later revised and extended the notion of system classes and proposed the term "context". In today's terminology, class SIMSET was an example of a class library providing linked lists and class SIMULATION was an example of a class framework (or application framework).

- **Representative states**. A major problem with concurrent programs was (and still is) to ensure that interaction between components results only in meaningful states of variables – denoted representative states in JLP. At that time, there was much research in concurrent programming including synchronization, critical regions, monitors, and communication. This subproject was to develop a conceptual approach to this problem, based upon the concepts of DELTA and work by Lars Mathiassen and Morten Kyng.
- **Implementation language**. In the early seventies, it was common to distinguish between general programming languages and implementation languages. An implementation language was often defined as an extended subset of the corresponding programming language. The subset was supposed to contain the parts that could be efficiently implemented – an implementation language should be as efficient as possible to support the general language. The extended part contained low level features to access parts of the hardware that could not be programmed with the general programming language. It was decided to define an implementation language called BETA as the implementation language for GAMMA. The original team consisted of Nygaard, Kristensen and Madsen – Møller-Pedersen joined later in 1976.

As described in Section 3.1 below, the BETA project was based on an initial research idea. This implied that there was much more focus on the BETA project than on the other activities in JLP. For the GAMMA language there were no initial ideas except designing a new language as a successor of SIMULA based on experience with DELTA and some of the best ideas of Pascal. In retrospect, language projects, like most other projects, should be based on one or more good ideas – otherwise they easily end up as nothing more than discussion forums. JLP was a useful forum for discussion of language ideas, but only the BETA project survived.

## 2.5 The BETA name and language levels

The name BETA was derived from a classification of language levels introduced by Nygaard, introducing a number of levels among existing and new programming languages. The classification by such levels would support the understanding of the nature and purpose of individual languages. The classification also motivated the existence of important language levels.

- The δ-level contains languages for system description and has the DELTA language as an example. A main characteristic of this level is that languages are non-executable.
- The γ-level contains general-purpose programming languages. SIMULA, Algol, Pascal, etc. are all examples of such languages. The JLP project was supposed to develop a new language to be called GAMMA. Languages at this level are by nature executable.
- The β-level contains implementation languages – and BETA was supposed to be a language at this level.
- The α-level contains assembly languages – it is seen as the basic "machine" level at which the actual translation takes place and at which the systems are run.

The level sequence defines the name of the BETA language, although the letter β was replaced by a spelling of the Greek letter β. Other names were proposed and discussed from time to time during the development of BETA. At some point the notion of beta-software became a standard term and this created a lot of confusion and jokes about the BETA language and motivated another name. For many years the name SCALA was a candidate for a new name – SCALA could mean SCAndinavian Language, and in Latin it means ladder and could be interpreted as meaning something 'going up'. The name of a language is important in order to spread the news appropriately, but names somehow also appear out of the blue and tend to have lives of their own. BETA was furthermore well known at that time and it was decided that it did not make sense to reintroduce BETA under a new name.

# 3. The BETA project

The original idea for BETA was that it should be an implementation language for a family of application languages at the GAMMA level. Quite early[2] during the development of BETA, however, it became apparent that there was no reason to consider BETA 'just' an implementation language. After the point when BETA was considered a general programming language, we considered it (instead of GAMMA) to be the successor of SIMULA. There were good reasons to consider a successor to SIMULA; SIMULA contains Algol as a subset, and there was a need to simplify parts of SIMULA in much the same way as Pascal is a simplification of Algol. In addition we thought that the new ideas arriving with BETA would justify a new language in the SIMULA style.

## 3.1 Research approach

The approach to language design used for BETA was naturally highly influenced by the SIMULA tradition. The

[2] In late 1978 and early 1979.

SIMULA I language report of 1965 opens with these sentences:

"The two main objectives of the SIMULA language are:

- To provide a language for a precise and standardised description of a wide class of phenomena, belonging to what we may call "discrete event systems".
- To provide a programming language for an easy generation of simulation programs for "discrete event systems"."

Thus, SIMULA I was considered as a language for system description as well as for programming. It was therefore obvious from the beginning that BETA should be used for system description as well as for programming.

In the '70s the SIMULA/BETA communities used the term system description to correspond to the term model (analysis and design models) used in most methodologies. We have always found it difficult to distinguish analysis, design and implementation. This was because we saw programming as modeling and program executions as models of relevant parts of the application domain. We considered analysis, design and implementation as programming at different abstraction levels.

The original goal for JLP and the GAMMA subproject was to develop a general purpose programming language as a successor to SIMULA. From the point in time where BETA was no longer just considered to be an implementation language, the research goals for BETA were supplemented by those for GAMMA. All together, the research approach was based on the following assumptions and ideas:

- BETA should be a modeling language.
- BETA should be a programming language. The most important initial idea was to design a language based on one abstraction mechanism. In addition BETA should support concurrent programming based on the coroutine mechanisms of SIMULA.
- BETA should have an efficient implementation.

### *3.1.1 Modeling and conceptual framework*

Creating a model of part of an application domain is always based on certain conceptual means used by the modeler. In this way modeling defines the perspective of the programmer in the programming process. Object-oriented programming is seen as one perspective on programming identifying the underlying model of the language and executions of corresponding programs.

Although it was realized from the beginning of the SIMULA era (including the time when concepts for record handling were developed by Hoare [52-54]) that the class/subclass mechanism was useful for representing concepts including generalizations and specializations, there was no explicit formulation of a conceptual framework for object-oriented programming. The term object-oriented programming was not in use at that time and neither were terms such as generalization and specialization. SIMULA was a programming language like Algol, Pascal and FORTRAN – it was considered superior in many aspects, but there was no formulation of an object-oriented perspective distinguishing SIMULA from procedural languages.

In the early seventies, the notion of functional programming arrived, motivated by the many problems with software development in traditional procedural languages. One of the strengths of functional programming was that it was based on a sound mathematical foundation (perspective). Later Prolog and other logic programming languages arrived, also based on a mathematical framework.

We did not see functional or logic programming as the solution: the whole idea of eliminating state from the program execution was contrary to our experience of the benefits of objects. We saw functional/logic programming and the development of object-oriented programming as two different attempts to remedy the problems with variables in traditional programming. In functional/logic programming mutable variables are eliminated – in object-oriented programming they are generalized into objects. We return to this issue in Section 4.2.

For object-oriented programming the problem was that there was no underlying sound perspective. It became a goal of the BETA project to formulate such a conceptual framework for object-oriented programming.

The modeling approach to designing a programming language provides overall criteria for the elements of the language. Often a programming language is designed as a layer on top of the computer; this implies that language mechanisms often are designed from technical criteria. BETA was to fulfill both kinds of criteria.

### *3.1.2 One abstraction mechanism*

The original design idea for BETA was to develop a language with only one abstraction mechanism: the pattern. The idea was that patterns should unify abstraction mechanisms such as class, procedure, function, type, and record. Our ambition was to develop the ultimate abstraction mechanism that subsumed all other abstraction mechanisms. In the DELTA report, the term pattern is used as a common term for class, procedure, etc. According to Nygaard the term pattern was also used in the final stages of the SIMULA project. For SIMULA and DELTA there was, however, no attempt to define a language mechanism for pattern.

The reason for using the term pattern was the observation that e.g. class and procedure have some common aspects: they are templates that may be used to create instances. The

instances of a class are objects and the instances of procedures are activation records.

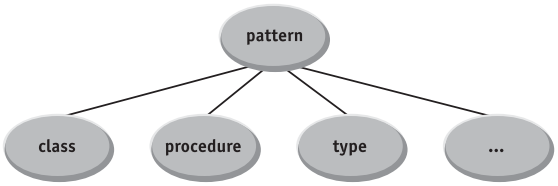


**Figure 1 Classification of patterns**

In the beginning it was assumed that BETA would provide other abstraction mechanisms as specializations (subpatterns) of the general pattern concept illustrated in Figure 1. In other words, BETA was initially envisaged as containing specialized patterns like `class`, `procedure`, `type`, etc. A subpattern of `class` as in

```
MyClass: class (# ... #)
```

would then correspond to a class definition in SIMULA. In a similar way a subpattern of `procedure` would then correspond to a procedure declaration. It should be possible to use a general pattern as a class, procedure, etc. As mentioned in Section 5.8.5, such specialized patterns were never introduced.

Given abstraction mechanisms like class, procedure, function, type and process type, the brute-force approach to unification would be to merge the elements of the syntax for all of these into a syntax describing a pattern. The danger with this approach might be that when a pattern is used e.g. as a class, only some parts of the syntactic elements might be meaningful. In addition, if the unification is no more than the union of class, procedure, etc., then very little has been gained.

The challenges of defining a general pattern mechanism may then be stated as follows:

- The pattern mechanism should be the ultimate abstraction mechanism, subsuming all other known abstraction mechanisms.
- The unification should be more than just the union of existing mechanisms.
- All parts of a pattern should be meaningful, no matter how the pattern is applied.

The design of the pattern mechanism thus implied a heavy focus on abstraction mechanisms, unification, and orthogonality. Orthogonality and unification are closely associated, and sometimes they may be hard to distinguish.

### *3.1.3 Concurrency*

It was from the beginning decided that BETA should be a concurrent programming language. As mentioned, SIMULA supported the notion of quasi-parallel system, which essentially defines a process concept and a cooperative scheduling mechanism. A SIMULA object is a coroutine and a quasi-parallel process is defined as an abstraction (in the form of a class) on top of coroutines.

The support for implementing hierarchical schedulers was one of the strengths of SIMULA; this was heavily used when writing simulation packages. Full concurrency was added to SIMULA in 1995 by the group at Lund University headed by Boris Magnusson [158].

Conceptually, the SIMULA coroutine mechanism appears simple and elegant, but certain technical details are quite complicated. For BETA, the SIMULA coroutine mechanism was an obvious platform to build upon. The ambition was to simplify the technical details of coroutines and add support for full concurrency including synchronization and communication. In addition it should be possible to write cooperative as well as pre-emptive schedulers.

### *3.1.4 Efficiency*

Although SIMULA was used by many communities in research institutions and private businesses, it had a relatively small user community. However, it was big enough for a yearly conference for SIMULA users to take place.

One of the problems with making SIMULA more widely used was that it was considered very inefficient. This was mainly due to automatic memory management and garbage collection. Computers at that time were quite slow and had very little memory compared to computers of today. The DEC 10 at DAIMI had 128Kbyte of memory. This made efficient memory management quite challenging.

One implication of this was that object orientation was considered to be quite inefficient by nature. It was therefore an important issue for BETA to design a language that could be efficiently implemented. In fact, it was a goal that it should be possible to write BETA programs with a completely static memory layout.

Another requirement was that BETA should be usable for implementing embedded systems. Embedded systems experts found it provoking that Nygaard would engage in developing languages for embedded systems – they did not think he had the qualifications for this. He may not have had much experience in embedded systems, but he surely had something to contribute. This is an example of the controversies that often appeared around Nygaard.

As time has passed, static memory requirements have become less important. However, this issue may become important again, for example in pervasive computing based on small devices.

## 3.2 Project organization

The process, intention, and organization of the BETA project appeared to be different from those of many projects today. The project existed through an informal

cooperation between Nygaard and the authors. During the project we had obligations as students, professors or consultants. This implied that the time to be used on the project had to be found in between other activities.

As mentioned, Kristensen, Møller-Pedersen and Madsen were students at DAIMI, Århus. In 1974 Kristensen completed his Masters Thesis on error recovery for LR-parsers. He was employed as assistant professor at the University of Ålborg in 1976. Madsen graduated in 1975, having written a Master's thesis on compiler-writing systems, and continued at DAIMI as assistant professor and later as a PhD student. Møller-Pedersen graduated in 1976 with a Master's thesis on the notion of context, with Nygaard as a supervisor. He was then employed by the NCC, Oslo, in 1976 and joined the BETA project at the same time. Nygaard was a visiting professor at DAIMI in 1974-75 – after that he returned to the NCC and continued at DAIMI as a guest lecturer.

Most meetings took place in either Århus or Oslo, and therefore required a lot of traveling. At that time there was a ferry between Århus and Oslo. It sailed during the night and took 16 hours – we remember many pleasant trips on that ferry – and these trips were a great opportunity to discuss language issues without being disturbed. Later when the ferry was closed we had to use other kinds of transportation that were not as enjoyable.

Funding for traveling and meetings was limited. Research funding was often applied for, but with little success. Despite his great contributions to informatics through the development of the SIMULA languages, Nygaard always had difficulties in getting funding in Norway. The Mjølner project described in Section 3.4 is an exception, by providing major funding for BETA development – however, Nygaard was not directly involved in applying for this funding.

The project involved a mixture of heated discussions about the design of the BETA language and a relaxed, inspiring social life. It seemed that for Nygaard there was very little difference between professional work and leisure.

**Meetings**. The project consisted of a series of more or less regular meetings with the purpose of discussing language constructs and modeling concepts. Meetings were planned in an ad hoc manner. The number of meetings varied over the years and very little was written or prepared in advance.

Meetings officially took place at NCC or at our universities, but our private homes, trams/buses, restaurants, ferries, and taxies were also seen as natural environments in which the work and discussions could continue – in public places people had to listen to loud, hectic discussions about something that must have appeared as complete nonsense to them. But people were tolerant and seemed to accept this weird group.

In October 1977, Madsen and family decided to stay a month in Oslo to complete the project – Madsen's wife, Marianne was on maternity leave – and they stayed with Møller-Pedersen and his family. This was an enjoyable stay, but very little progress was made with respect to completing BETA – in fact we saw very little of Nygaard during that month.

**Discussions**. A meeting would typically take place without a predefined agenda and without any common view on what should or could be accomplished throughout the meeting. The meetings were a mixture of serious concentrated discussions of ideas, proposals, previous understanding and existing design and general stuff from the life of the participants.

State-of-the-art relevant research topics were rarely subjects for discussion. Nygaard did not consider this important – at least not for the ongoing discussions and elaboration of ideas. Such knowledge could be relevant later, in relation to publication, but was usually not taken seriously. In some sense Nygaard assumed that we would take care of this. Also established understanding, for example, on the background, motivations and the actual detailed contents of languages like Algol, SIMULA or DELTA was not considered important. It appeared to be much better to develop ideas and justify them without historical knowledge or relationships. The freedom was overwhelming and the possibilities were exhausting.

The real strengths of Nygaard were his ability to discuss language issues at a conceptual level and focus on means for describing real-world systems. Most language designers come from a computer science background and their language design is heavily based on what a computer can do: A programming language is designed as a layer on top of the computer making it easier to program. Nygaard's approach was more of a modeling approach and he was basically interested in means for describing systems. This was evident in the design of SIMULA, which was designed as a simulation language and therefore well suited for modeling real systems.

**Plans**. There was no clear management of the project, and plans and explicit decisions did not really influence the project. We used a lot of time on planning, but most plans were never carried out. Deadlines were typically controlled by the evolution of the project itself and not by a carefully worked out project plan.

Preparation and writing were in most cases the result of an individual initiative and responsibility. Nygaard was active in writing only in the initial phase of the project. Later on Nygaard's research portfolio mainly took the form of of huge stacks of related plastic slides, but typically their relation became clear only at Nygaard's presentations. The

progress and revisions of his understanding of the research were simply captured on excellent slides.

At the beginning of the project it was decided that Kristensen and Madsen should do PhDs based on the project. As a consequence of the lack of project organization, it quickly became clear that this would not work.

The original plan for the BETA project was that '*a firm and complete language definition*' should be ready at the end of 1977 [89]. An important deadline was February 1977 – at that time a first draft of a language definition should be available. In 1977 we were far from a complete language definition and Nygaard did not seem in a hurry to start writing a language definition report. However, two working notes were completed in 1976/77. In Section 3.3 and in Section 5.10, we describe the content of these working notes and other publications and actual events in the project.

**Social life**. Meetings typically lasted whole days including evenings and nights. In connection with meetings the group often met in our private homes and had dinner together, with nice food and wine. The atmosphere was always very enjoyable but demanding, due to an early-morning start with meetings and late-evening end. Dinner conversation was often mixed with debate about current issues of language design. Our families found the experience interesting and inspiring, but also often weird. Nygaard often invited various guests from his network, typically without our knowing and often announced only in passing. Guests included Carl Hewitt, Bruce Moon, Larry Tesler, Jean Vaucher, Stein Krogdahl, Peter Jensen, and many more. They were all inspiring and the visits were learning experiences. In addition there were many enjoyable incidents as when passengers on a bus to the suburb where Madsen lived watched with surprise and a little fear as Nygaard (tall and insistent) and Hewitt (all dressed in red velour and just as insistent) loudly discussed not commonly understandable concepts on the rear platform of the bus.

Nygaard was an excellent wine connoisseur and arranged wine-tasting parties on several occasions. We were "encouraged" to spend our precious travel money on various selected types of wines, and it was beyond doubt worth it. Often other guests were invited and had similar instructions about which wine to bring. At such parties we would be around 10 people in Nygaard's flat, sitting around their big dinner table and talking about life in general. The wines would be studied in advance in Hugh Johnson's "World Atlas of Wine" and some additional descriptions would be shared. The process was controlled and conducted by Nygaard at the head of the table.

**Crises**. The project meetings could be very frustrating since Nygaard rarely delivered as agreed upon at previous meetings. This often led to very heated discussions. This seemed to be something that we inherited from the SIMULA project. In one incident Kristensen and Madsen arrived in Oslo at the NCC and during the initial discussions became quite upset with Nygaard and decided to leave the project. They took a taxi to the harbor in order to enter the ferry to Århus. Nygaard, however, followed in another taxi and convinced them to join him for a beer in nearby bar – and he succeeded in convincing them to come back with him.

Crises and jokes were essential elements of meetings and social gatherings. Crises were often due to different expectations to the progress of the language development, unexpected people suddenly brought into the project meetings, and problems with planning of the meeting days. Crises were solved, but typically not with the result that the next similar situation would be tackled differently by Nygaard. Serious arguments about status and plans were often solved by a positive view on the situation together with promises for the future. Jokes formed an essential means of taking 'revenge' and thereby to overcome crises. Jokes were on Nygaard in order to expose his less appealing habits, as mentioned above, and were often simple and stupid, probably due to our irritation and desperation. Nygaard was an easy target for practical jokes, because he was always very serious about work, which was not something you joked about.[3] On one occasion in Nygaard's office at Department of Informatics at University of Oslo, the telephone calls that Nygaard had to answer seemed to never end, even if we complained strongly about the situation. One time when Nygaard left the office, we taped the telephone receiver to the base by means of some transparent tape. When Nygaard returned, we arranged for a secretary to call him. As usual Nygaard quickly grabbed the telephone receiver, and he got completely furious because the whole telephone device was in his hand. He tried to wrench the telephone receiver off the base unit, but without success. Nygaard blamed us for the lost call (which could be very important as were the approximately 20 calls earlier this morning) and left the office running to the secretary in order to find out who had called. He returned disappointed and angry, but possibly also a bit more understanding of our complaints. At social events he was on the other hand very entertaining and had a large repertoire of jokes – however, practical jokes were not his forte.

## 3.3 Project events

In this section we mention important events related to the project process as a supplement to the description in the previous sections. Events related to the development of the

[3] We never found out whether or not this was a characteristic of Nygaard or of Norwegians in general☺.

conceptual framework, the language and its implementation are described in the following sections.

Due to the lack of structure in the project organization, it is difficult to point to specific decisions during the project that influenced the design of BETA. The ambition for the project was to strive for the perfect language and it turned out that this was difficult to achieve through a strict working plan. Sometimes the design of a new programming language consists of selecting a set of known language constructs and the necessary glue for binding them together. For BETA the goal was to go beyond that. This implied that no matter what was decided on deadlines, no decisions were made as long as a satisfactory solution had not been found. In some situations we clearly were hit by the well known saying, 'The best is the enemy of the good'.

The start of the JLP and the start of the BETA project were clearly important events. There was no explicit decision to terminate the JLP – it just terminated.

As mentioned, two working notes were completed in 1976/1977. The first one was by Peter Jensen and Nygaard [66] and was mainly an argument why the NCC should establish cooperation with other partners in order to implement the BETA system programming language on microcomputers.

First language draft. The second working note was the first publication describing the initial ideas of BETA, called *BETA Language Development – Survey Report, 1, November 1976* [89]. A revised version was published in September 1977.

Draft Proposal of BETA. In 1978 a more complete language description was presented in *DRAFT PROPOSAL for Introduction to the BETA Programming Language as of 1st August 1978* [90] and a set of examples [91]. A grammar was included. Here BETA was still mainly considered an implementation language. The following is stated in the report: "*According to the conventional classification of programming languages BETA is meant to be a system programming language. Its intended use is for programming of operating systems, data base systems, communication systems and for implementing new and existing programming languages. … The reason for not calling it a system programming language is that it is intended to be more general than often associated with system programming languages. By general is here meant that it will contain as few as possible concepts underlying most programming concepts, but powerful enough to build up these. The BETA language will thus be a kernel of concepts upon which more application oriented languages may be implemented and we do not imagine the language as presented here used for anything but implementation of more suitable languages. This will, however, be straight forward to do by use of a compiler-generator. Using this, sets of concepts may be defined in terms of BETA and imbedded in a language. Together with the BETA language it is the intention to propose and provide a 'standard super BETA'.*"

As mentioned, BETA developed without any explicit decision into a full-fledged general programming language. In this process it was realized that GAMMA and special-purpose languages could be implemented as class frameworks in BETA. With regard to class frameworks, SIMULA again provided the inspiration. SIMULA provided class `Simulation` – a class framework for writing simulation programs. Class `Simulation` was considered a definition of a special-purpose language for simulation – SIMULA actually has special syntax only meaningful when class `Simulation` is in use. For BETA it provided the inspiration for work on special-purpose languages. The idea was that a special-purpose language could be defined by means of a syntax definition (in BNF), a semantic definition in terms of a class framework, and a syntax-directed transformation from the syntax to a BETA program using the class framework. This is reflected in the 1978 working note.

First complete language definition. In February 1979 – and revised in April 1979 – the report *BETA Language Proposal* [92] was published. It contained the first attempt at a complete language definition. Here BETA was no longer considered just an implementation language: "*BETA is a general block-structured language in the style of Algol, Simula and Pascal. … Most of the possibilities of Algol-like sequential languages are present*". BETA was, however, still considered for use in defining application-oriented languages – corresponding to what are often called domain-specific languages today.

The fact that BETA was considered a successor to SIMULA created some problems at the NCC and the University of Oslo. The SIMULA communities considered SIMULA to be THE language, and with good reason. There were no languages at that time with the qualities of SIMULA and as of today, the SIMULA concepts are still in the core of mainstream languages such as C++, Java and C#.

Many people became angry with Nygaard that he seemed willing to give up on SIMULA. He did not look at it that way – he saw his mission as developing new languages and exploring new ideas. However, it did create difficulties in our relationship with the SIMULA community. SIMULA was at that time a commercial product of the NCC. When it became known that Nygaard was working on a successor for SIMULA, the NCC had to send out a message to its customers saying that the NCC had no intentions of stopping the support of SIMULA.

Around 1980 there was in fact an initiative by the NCC to launch BETA as a language project based on the model used for SIMULA. This included planning a call for a standardization meeting, although no such meeting ever took place. The plan was that BETA should be frozen by the end of 1980 and an implementation project should then be started by the NCC. However, none of this did happen.

A survey of the BETA Programming Language. In 1981 the report 'A Survey of the BETA Programming Language' [93] formed the basis for the first implementation and the first published paper on BETA two years later [95]. As mentioned in Section 6.1, the first implementation was made in 1983.

Several working papers about defining special-purpose languages were written (e.g. [98]), but no real system was ever implemented. A related subject was that the grammar of BETA should be an integrated part of the language. This led to work on program algebras [96] and metaprogramming [120] that made it possible to manipulate BETA programs as data. Some of the inspiration for this work came during a one-year sabbatical that Madsen spent at The Center for Study of Languages and Information at Stanford University in 1984, working with Terry Winograd, Danny Bobrow and José Meseguer.

POPL paper: Abstraction Mechanisms in the BETA Programming Language**.** An important milestone for BETA was the acceptance of a paper on BETA for POPL in 1983 [95]. We were absolutely thrilled and convinced that BETA would conquer the world. This did not really happen – we were quite disappointed with the relatively little interest the POPL paper created. At the same conference, Peter Wegner presented a paper called *On the Unification of Data and Program Abstractions in Ada* [159]. Wegner's main message was that Ada contained a proliferation of abstraction mechanisms and there was no uniform treatment of abstraction mechanisms in Ada. Naturally we found Wegner's paper to be quite in line with the intentions of BETA and this was the start of a long cooperation with Peter Wegner, who helped in promoting BETA.

Hawthorne Workshop. Peter Wegner and Bruce Shriver (who happened to be a visiting professor at DAIMI at the same time as Nygaard) invited us to the Hawthorne workshop on object-oriented programming in 1986. This was one of the first occasions where researchers in OOP had the opportunity to meet and it was quite useful for us. It resulted in the book on Research Directions in Object-Oriented Programming [145] with two papers on BETA [100, 111]. Peter Wegner and Bruce Shriver invited us to publish papers on BETA at the Hawaii International Conference on System Sciences in 1988.

Sequential parts stable**.** In late 1986/early 1987 the sequential parts of the language were stable, and only minor changes have been made since then.

Multisequential parts stable. A final version of the multisequential parts (coroutines and concurrency) was made in late 1990, early 1991.

BETA Book**.** Peter Wegner also urged us to write a book on BETA and he was the editor of the BETA book published by Addison Wesley/ACM Press in 1993 [119].

For a number of years we gave BETA tutorials at OOPSLA, starting with OOPSLA'89 in New Orleans. Dave Thomas and others were quite helpful in getting this arranged – especially at OOPSLA'90/ECOOP'90 in Ottawa, he provided excellent support.

At OOPSLA'89 we met with Dave Unger and the Self group; although Self [156] is a prototype-based language and BETA is a class-based language, we have benefited from cooperation with the Self group since then. We believe that Self and BETA are both examples of languages that attempt to be based on simple ideas and principles.

The Mjølner (Section 3.4) project (1986-1991) and the founding of Mjølner Informatics Ltd. (1988) were clearly important for the development of BETA.

Apple and Apollo contracts**.** During the Mjølner project we got a contract with Apple Computer Europe, Paris, to implement BETA for the Macintosh – Larry Taylor was very helpful in getting this contract. A similar contract was made with Apollo Computer, coordinated by Søren Bry.

In 1994, BETA was selected to be taught at the University of Dortmund. Wilfried Ruplin was the key person in making this happens. A German introduction to programming using BETA was written by Ernst-Erich Doberkat and Stefan Diβmann [38]. This was of great use for the further promotion of BETA as a teaching language.

Dahl & Nygaard receive ACM Turing Award**.** In 2001 Dahl and Nygaard received the ACM Turing Award ("for their role in the invention of object-oriented programming, the most widely used programming model today").

Dahl & Nygaard receive the IEEE von Neumann Medal. In 2002 they received the IEEE John von Neumann Medal ("for the introduction of the concepts underlying object-oriented programming through the design and implementation of SIMULA 67"). Dahl was seriously ill at that time so he was not able to attend formal presentations of these awards, including giving the usual Turing Award lecture. Dahl died on June 29, 2002. Nygaard was supposed to give his Turing Award lecture at OOPSLA 2002 in Vancouver, October 2002, but unfortunately he died on August 10, just a few weeks after Dahl. Life is full of strange coincidences. Madsen was invited to give a lecture

at OOPSLA 2002 instead of Nygaard. The overall theme for that talk was '*To program is to understand*', which in many ways summarizes Nygaard's approach to programming. One of Nygaard's latest public appearances was his after-dinner talk at ECOOP 2002 in Malaga, where he gave one of his usual entertaining talks that even the spouses enjoyed.

## 3.4 The Mjølner Project

Mjølner Project start. The Mjølner[4] Project (1986-1991) [76] was an important step in the development of BETA. The objective of the Mjølner project was to increase the productivity of high-quality software in industrial settings by designing and implementing object-oriented software development environments supporting specification, implementation and maintenance of large production programs. The project was carried out in cooperation between Nordic universities and industrial companies with participants from Denmark, Sweden, Norway and Finland. In the project three software development environments were developed:

- **Object-oriented SDL and tools**: The development of Object-oriented SDL is described in Section 7.3.
- **The Mjølner Orm System**: a grammar-based interactive, integrated, incremental environment for object-oriented languages. The main use of Orm was to develop an environment for SIMULA.
- **The Mjølner BETA System**: a programming environment for BETA.

Mjølner Book**.** The approach to programming environments developed within the Mjølner Project is documented in the book *Object-Oriented Environments – the Mjølner Approach* [76], covering all these three developments.

The development of the Mjølner BETA System was in a Scandinavian context a large project. The project was a major reason for the success of BETA. During this project the language developed in the sense that many details were clarified. For example, the Ada-like rendezvous for communication and synchronization was abandoned in favor of semaphores, pattern variables were introduced, etc. It was also during the Mjølner project that the fragment system found its current form – cf. Section 5.8.4.

Most of the implementation techniques for BETA were developed during the Mjølner project, together with native compilers for Sun, Macintosh, etc. Section 6 contains a description of the implementation.

A complete programming environment for BETA was developed. In addition to compilers there was a large collection of libraries and application frameworks including a meta-programming system called Yggdrasil[5], a persistent object store with an object browser, and application frameworks for GUI programs built on top of Athena, Motif, Macintosh and Windows. A platform independent GUI framework with the look and feel of the actual platform was developed for Macintosh, Windows and UNIX/Motif.

The environment also included the MjølnerTool, which was an integration of the following tools: a source code browser called Ymer, an integrated text- and syntax-directed editor called Sif, a debugger called Valhalla, an interface builder called Frigg, and a CASE tool called Freja supporting a graphical syntax for BETA – see also Section 4.5.

**Mjølner Informatics.** The Mjølner BETA System led in 1988 to the founding of the company Mjølner Informatics Ltd., which for many years developed and marketed the Mjølner BETA System as a commercial product. Sales of the system never generated a high profit, but it gave Mjølner Informatics a good image as a business, and this attracted a lot of other customers. Today the Mjølner BETA System is no longer a commercial product, but free versions may be obtained from DAIMI.

It may seem strange from the outside that three environments were developed in the Mjølner Project. And it is indeed strange. SDL was, however, heavily used by the telecommunication industry, and there was no way to replace it by say BETA – the only way to introduce object orientation in that industry seemed to be by adding object orientation to SDL. Although SDL had a graphical syntax, it also had a textual syntax, and it had a well-defined execution semantics, so it was more or less a domain-specific programming language (the domain being telecommunications) and not a modeling language. Code generators were available for different (and at that time specialized) platforms. SDL is still used when code generation is needed, but for modeling purposes UML has taken over. UML2.0 includes most of the modeling mechanisms of SDL, but not the execution semantics.

The BETA team did propose to the people in charge of the Mjølner Orm development – which focused on SIMULA – that they join the BETA team, and that we concentrate on developing an environment for BETA. BETA was designed as a successor of SIMULA, and we found that it would be better to just focus on BETA. However, the SIMULA people were not convinced; it is often said that SIMULA, like Algol 60, is one of the few languages that is better than

[4] In the Nordic myths Mjølner is the name of Thor's hammer; Thor is the Nordic god of thunder. Mjølner is the perfect tool: it grows with the task, always hits the target, and always returns safely to Thor's hand.

[5] Most tools in the Mjølner System had names from Nordic mythology.

most of its successors – we are not the ones to judge about this with respect to BETA. The lesson here is perhaps that in a project like Mjølner more long-term goals would have been beneficial. If the project had decided to develop one language including a graphical notation that could replace SDL, then this language might have had a better chance to influence the industry than each of OSDL, SIMULA and BETA.

The motivation for modeling languages like SDL (and later UML) was that industries wanted to be independent of (changing) programming languages and run-time environments. A single language like BETA that claims to be both a programming language and a modeling language was therefore not understood. Even Java has not managed to get such a position. It is also interesting to note that while the Object Management Group advocates a single modeling language, covering many programming languages and platforms, Microsoft advocates a single programming language (or rather a common language run-time, CLR) on top of which they want to put whatever domain-specific modeling language the users in a specific domain require.

## 4. Modeling and conceptual framework

We believe that the success of object-oriented programming can be traced back to its roots in simulation. SIMULA I was designed to describe (model) real-world systems and simulate these systems on a computer. This eventually led to the design of SIMULA 67 as a general programming language. Objects and classes are well suited for representing phenomena and concepts from the real world and for programming in general. Smalltalk further refined the object model and Alan Kay described object-oriented programming as a view on *computation as simulation* [68] (see also the formulation by Tim Budd [22]). An important aspect of program development is to understand, describe and communicate about the application domain and BETA should be well suited for this. In the BETA book [119] (page 3) this is said in the following way:

> **To program is to understand**: The development of an information system is not just a matter of writing a program that does the job. It is of utmost importance that development of this program has revealed an in-depth understanding of the application domain; otherwise, the information system will probably not fit into the organization. During the development of such systems it is important that descriptions of the application domain are communicated between system specialists and the organization.

The term "*To program is to understand*" has been a leading guideline for the BETA project. This implied that an essential part of the BETA project was the development of a conceptual framework for understanding and organizing knowledge about the real world. The conceptual framework should define the object-oriented perspective on programming and provide a semantic foundation for BETA. Over the years perhaps more time was spent on discussing the conceptual framework than the actual language. Issues of this kind are highly philosophical and, not being philosophers, we could spend a large amount of time on this without progress.

Since BETA was intended for modeling as well as programming there was a rule that applied when discussing candidates for language constructs in BETA: a given language construct should be motivated from both the modeling and the programming point of view. We realized that many programmers did not care about modeling but were only interested in technical aspects of a given language – i.e. what you can actually express in the language. We thus determined that BETA should be usable as a 'normal' programming language without its capabilities as a modeling language. We were often in the situation that something seemed useful from a modeling point of view, but did not benefit the programming part of the language and vice versa.

We find the conceptual framework for BETA just as important as the language – in this paper we will not go into details, but instead refer to chapters 2 and 18 in the book on BETA [119]. Below we will describe part of the rationale and elements of the history of the conceptual framework. In Section 5, where the rationale for the BETA language is described, we will attempt to describe how the emphasis on modeling influenced the language.

### 4.1 Programming as modeling

As mentioned, DELTA was one of the starting points for the BETA project. For a detailed description of DELTA the reader is referred to the DELTA report [60]. Here we briefly summarize the concepts that turned out to be most important for BETA.

The system to be described was called the *referent system*. A referent system exists in what today's methodologies call application domain. A description of a system – the *system description* – is a text, a set of diagrams or a combination describing the aspects of the system to be considered. Given a system description, a *system generator* may generate a *model system* that simulates the considered aspects of the referent system. These concepts were derived from the experience of people writing simulation programs (system descriptions) in SIMULA and running these simulations (model systems).

Programming was considered a special case of system description – a program was considered a system description and a program execution was considered a model system. Figure 2 illustrates the relationship between the referent system and the model system.

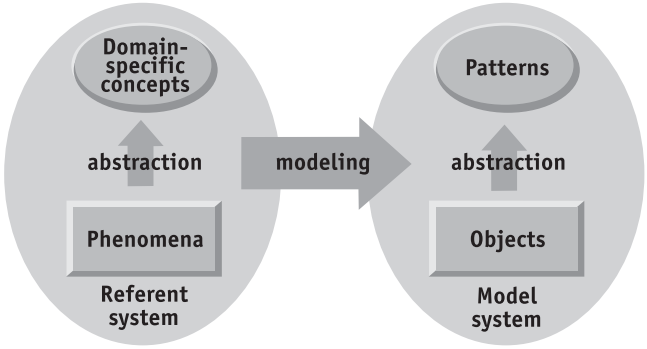


**Figure 2 Modeling**

As illustrated in Figure 2, phenomena and (domain-specific) concepts from the referent system are identified and represented as (realized) objects and concepts (in the form of patterns) in the model system (the program execution). The modeling activity of Figure 2 includes the making of a system description and having a system generator generate the model system according to this description.

## 4.2 Object-orientation as physical modeling

For BETA it has been essential to make a clear distinction between the program and the program execution (the model system). A program is a description in the form of a text, diagrams or a combination – the program execution is the dynamic process generated by the computer when executing the program. At the time when BETA was developed, many researchers in programming and programming languages were focusing on the program text. They worked on the assumption that properties of a program execution could (and should) be derived from analysis of the program text, including the use of assertions and invariants, formal proofs and formal semantics. Focusing on the (static) program text often made it difficult to explain the dynamics of a program execution. Especially for object-oriented programming, grasping the dynamic structure of objects is helped by considering the program execution. But considering the program execution is also important in order to understand mechanisms such as recursion and block structure.

The discussion of possible elements in the dynamic structure of a BETA program execution was central during the design of BETA. This included the structure of coroutines (as described in Section 5.7 below), stacks of activation records, nested objects, references between objects, etc. Many people often felt that we were discussing implementation, but for us it was the semantics of BETA. It did cover aspects that normally belonged to implementation, but the general approach was to identify elements of the program execution that could explain to the programmer how a BETA program was executing.

At that time, formal semantics of programming languages was an important issue and we were often confronted with the statement that we should concentrate on defining a formal semantics for BETA. Our answer to that was vague in the sense that we were perhaps uncertain whether or not they were right, but on the other hand we had no idea how to approach a formal semantics for a language we were currently designing. It seemed to us that the current semantic models just covered well known language constructs and we were attempting to identify new constructs. Also, our mathematical abilities were perhaps not adequate to mastering the mathematical models used at that time for defining formal semantics.

Many years later we realized that our approach to identifying elements of the program execution might be seen as an attempt to define the semantics of BETA – not in a formal way, but in a precise and conceptual way.

The focus on the program execution as a model eventually led to a definition of object-oriented programming based on the notion of physical model – first published at ECOOP in ’88 [116]:

> **Object-oriented programming**. A program execution is regarded as a physical model simulating the behavior of either a real or imaginary part of the world.

The notion of physical is essential here. We considered (and still do) objects as physical material used to construct models of the relevant part of the application domain. The analogy is the use of physical material to construct models made of cardboard, wood, plastic, wire, plaster, LEGO bricks or other substances. Work on object-oriented programming and computerized shared material by Pål Sørgaard [149] was an essential contribution here.

Webster defines a model in the following way: “In general a model refers to a small, abstract or actual representation of a planned or existing entity or system from a particular viewpoint” [1]. Mathematical models are examples of abstract representations whereas models of buildings and bridges made of physical material such as wood, plastic, and cartoon are examples of actual representations. Models may be made of existing (real) systems as in physics, chemistry and biology, or of planned (imaginary) systems like buildings, and bridges.

We consider object-oriented models[6] to be actual (physical) representations made from objects. An object-oriented model may be of an existing or planned system, or a

[6] The term modeling is perhaps somehow misleading since the model eventually becomes the real thing – in contrast to models in science, engineering and architecture. We originally used the term *description,* in SIMULA and DELTA terminology, but changed to modeling when OOA/OOD and UML became popular.

combination. It may be a reimplementation of a manual system on a computer. An example may be a manual library system that is transferred to computers. In most cases, however, a new (planned) system is developed. In any case, the objects and patterns of the system (model) represent phenomena and concepts from the application domain. An object-oriented model furthermore has the property that it may be executed and simulate the behavior of the system in accordance with the computation-is-simulation view mentioned above.

The application domain relates to the real world in various ways. Most people would agree that a library system deals with real world concepts and phenomena such as books and loans. Even more technical domains like a system controlling audio/video units and media servers deal with real-world concepts and phenomena. Some people might find that a network communication protocol implementing TCP/IP is not part of the real world, but it definitely becomes the real world for network professionals, just as an electronic patient record is the real world for healthcare professionals. Put in other words: Even though the real world contains real trees and not so many binary search trees or other kinds of data structures, the modeling approach is just as valuable for such classical elements of (in this case) the implementation domain.

In any case the modeling approach should be the same for all kinds of application domains – this is also the case for the conceptual means used to understand and organize knowledge about the application domain, be it the real world or a technical domain. In the approach taken by OO and BETA we apply conceptual means used for organizing knowledge about the real world, as we think this is useful for more technical and implementation-oriented domains as well. In Chapter 5 we describe how the modeling approach has influenced the design of the BETA language.

From the above definition it should be evident that phenomena that have the property of being physical material should be represented as objects. There are, however, other kinds of phenomena in the real world. This led to a characterization of the essential qualities of phenomena in the real world systems of interest for object-oriented models:

- Substance – the physical material transformed by the process.
- Measurable properties of the substance.
- Transformations of the substance.

People, vehicles, and medical records are examples of phenomena with substance, and they may be represented by objects in a program execution. The age, weight or blood pressure of a person are examples of measurable properties of a person and may be represented by values (defined by value types) and/or functions. Transformations of the substance may be represented by the concurrent processes and procedures being executed as part of the program execution. The understanding of the above qualities had a profound influence on the semantics of BETA.

## 4.3 Relation to other perspectives

In order to arrive at a conceptual understanding of object orientation, we found it important to understand the differences between object-orientation and other perspectives such as procedural, functional, and constraint programming. We thus contrasted our definition of object orientation (see e.g. our ECOOP'88 paper [116] and Chapter 2 in the BETA book [119]) to similar definitions for other perspectives. In our understanding, the essential differences between procedural and functional programming related to the use of mutable variables. In procedural programming a program manipulates a set of mutable variables. In pure functional programming there is no notion of mutable variable. A function computes its result solely based on its arguments. This also makes it easy to formulate a sound mathematical foundation for functional programming. We are aware that our conception of functional programming may not correspond to other people's understanding. In most functional programming languages you may have mutable variables and by means of closures you may even define object-oriented programming language constructs as in CommonLisp. However, if you make use of mutable variables it is hard to distinguish functional programming from procedural programming. Another common characteristic of functional languages is the strong support for higher functions and types. However, higher-order functions (and procedures) and types may be used in procedural as well as object-oriented programming. Algol and Pascal support a limited form of higher-order functions and procedures, and generic types are known from several procedural languages. Eiffel and BETA are examples of languages supporting generic classes (corresponding to higher-order types), and for BETA it was a goal to support higher-order functions and procedures. When we discuss functional programming in this paper, it should be understood in its pure form where a function computes its result solely based on its arguments. This includes languages using non-mutable variables as in `let x=e1 in e2.`

For BETA it was not a goal to define a pure object-oriented language as it may have been for Smalltalk. On the contrary, we were interested in integrating the best from all perspectives into BETA. We thus worked on developing an understanding of a unified approach that integrated object-oriented programming with functional, logic and procedural programming [116]. BETA supports procedural programming and to some extent functional programming. We also had discussions with Alan Borning and Bjorn Freeman-Benson on integrating constraint-oriented programming into BETA. The idea of using equations

(constraints) to describe the state of objects was very appealing, but we never managed to identify primitive[7] language constructs that could support constraints. However, a number of frameworks supporting constraints were developed by students in Aarhus.

## 4.4 Concepts and abstraction

It was of course evident from the beginning that the class/subclass constructs of SIMULA were well suited to representing traditional Aristotelian concepts (for a description of Aristotelian concepts, see the BETA book) including hierarchical concepts. The first example of a subclass hierarchy was a classification of vehicles as shown in Figure 3.

Abstraction is perhaps the most powerful tool available to the human intellect for understanding complex phenomena. An abstraction corresponds to a concept. In the Scandinavian object-oriented community it was realized in the late seventies by a number of people, including the authors and collaborators, that in order to be able to create models of parts of the real world, it was necessary to develop an explicit understanding of how concepts and phenomena relate to object-oriented programming.

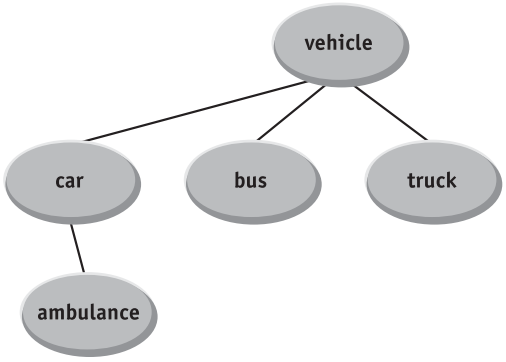


**Figure 3 Example of a subclass hierarchy**

In the late seventies and early eighties the contours of an explicit conceptual framework started to emerge – there was an increasing need to be explicit about the conceptual basis of BETA and object orientation in general. The ongoing discussions on issues such as multiple inheritance clearly meant that there was a need for making the conceptual framework explicit. These discussions eventually led to an explicit formulation of a conceptual framework by means of Aristotelian concepts in terms of intension, extension and designation to be used in object-oriented modeling. An important milestone in this work was the Master's thesis of Jørgen Lindskov Knudsen [71] and part of the PhD Thesis of Jørgen Lindskov Knudsen and Kristine Thomsen [78].

[7] We do not consider equations to be programming-language primitives.

Knudsen supplemented the conceptual framework with the so-called *prototypical concepts* inspired by Danish philosopher Sten Folke Larsen [42], who argued that most everyday concepts are not Aristotelian but fuzzy (prototypical). An Aristotelian concept is characterized by a set of defining properties (the intension) that are possessed by all phenomena covered by the concept (the extension). For a prototypical concept the intension consists of examples of properties that the phenomena may have, together with a collection of typical phenomena covered by the concept, called *prototypes*. An Aristotelian concept structure has well defined boundaries between the extensions of the concepts, whereas this is not the case for a prototypical concepts structure. In the latter the boundaries are blurred/fuzzy. A class is well suited to representing Aristotelian concepts, but since most everyday concepts are prototypical, a methodology should allow for prototypical concepts to be used during analysis. Prototypical concepts should not be confused with prototype-based languages. A prototypical concept is still a concept – prototypical objects are not based on any notion of concept. Prototypical concepts are described in the BETA book [119], and the relationship between prototypical concepts and prototype-based languages is discussed by Madsen [113].

In the seventies there was similar work on modeling going on in the database and AI communities and some of this work influenced on the BETA project. This included papers such as the one by Smith & Smith [147] on database abstraction.

The realization that everyday concepts were rarely Aristotelian made it clear that it was necessary to develop a conceptual framework that was richer than the current programming language in use. In the early days, there might have been a tendency to believe that SIMULA and other object-oriented languages had all mechanisms that were needed to model the real world – this was of course naive, since all languages put limitations on the aspects of the real world that can be naturally modeled. Programmers have a tendency to develop an understanding of the application domain in terms of elements of their favorite programming language. A Pascal programmer models the real world in terms of Pascal concepts like records and procedures. We believed that the SIMULA concepts (including class, and subclass) were superior to other programming languages with respect to modeling.

The conceptual framework associated with BETA is deliberately developed to be richer than the language. In addition to introducing prototypical concepts, the BETA book discusses different types of classification structures that may be applied to a given domain, including some that cannot be directly represented in mainstream programming languages. The rationale for the richer conceptual framework is that programmers should understand the

application domain by developing concepts without being constrained by the programming language. During implementation it may of course be necessary to map certain concepts into the programming language. It is, however, important to be explicit about this. This aspect was emphasized in a paper on teaching object-oriented programming [77].

## 4.5 Graphical syntax for modeling

When object-oriented programming started to become mainstream in the early eighties, code reuse by means of inheritance was often seen as the primary advantage of object-oriented programming. The modeling capabilities were rarely mentioned. The interest in using object-oriented concepts for analysis and design that started in the mid-eighties was a positive change since the modeling capabilities came more in focus.

One of the disadvantages of OOA/OOD was that many people apparently associated analysis and design with the use of graphical languages. There is no doubt that diagrams with boxes and arrows are useful when designing systems. In the SIMULA and BETA community, diagrams had of course also been used heavily, but when a design/model becomes stable, a textual representation in the form of an abstract program is often a more compact and comprehensive representation.

The mainstream modeling methodologies all proposed graphical languages for OOA/OOD, which led to the UML effort on designing a standardized graphical language for OOA/OOD. We felt that this was a major step backwards – one of the advantages of object-orientation is that the same languages and concepts can be applied in all phases of the development process, from analysis through design to implementation. By introducing a new graphical language, one reintroduced the problem of different representations of the model and the code. It seems to be common sense that most software development is incremental and iterative, which means that the developer will iterate over analysis, design and implementation several times. It is also generally accepted that design will change during implementation. With different representations of the model and the code it is time consuming to keep both diagrams and code in a consistent state.

In the early phases of Mjølner Project it was decided to introduce a graphical syntax for the abstraction mechanisms of BETA as an alternative to the textual syntax. The Freja CASE tool [141, 142] was developed using this syntax. In addition, Freja was integrated with the text and structure editor in such a way that the programmer could easily alternate between a textual and graphical representation of the code.

When UML became accepted as a common standard notation, the developers at Mjølner Informatics decided to replace the graphical syntax defined for BETA by a subset of UML. Although major parts of BETA had a one-to-one correspondence with this UML subset, some of the problems of different representations were reintroduced.

It is often said that a picture says more than a thousand words. This is true. Nygaard in his presentations often used a transparency with this statement (and a picture of Madonna). This was always followed by one saying that a word often says more than a thousand pictures, illustrated by a number of drawings of vehicles and the word ‘vehicle’. The point is that we use words to capture essential concepts and phenomena – as soon as we have identified a concept and found a word for it, this word is an efficient means for communication among people. The same is true in software design. In the initial phase it is useful to use diagrams to illustrate the design. When the design stabilizes it is often more efficient to use a textual representation for communication between the developers. The graphical representation may still be useful when introducing new people to the design.

## 4.6 Additional notes

It was often difficult to convey to other researchers what we understood by system description and why we considered it important. As mentioned, there was an important workshop at the IBM Hawthorne Research Center in New York in 1986, organized by Peter Wegner and Bruce Shriver, in which Dahl, Nygaard and Madsen participated. Here we had long and heated debates with many researchers – it was difficult to agree on many issues, most notably the concept of multiple inheritance. We later realized that for most people at that time the advantage of object-orientation was from a reuse point of view – a purely technical argument. For us, coming from the SIMULA tradition, the modeling aspect was at least as important, but the difference in perspective was not explicit. Later Steve Cook [26] made the difference explicit by introducing the ideas of the ‘Scandinavian School’ and the ‘U.S. School’ of object-orientation.

At that time the dominant methodology was based on structured analysis and design followed by implementation – SA/SD [162]. SIMULA users rarely used SA/SD, but formulated their designs directly in SIMULA. The work on DELTA and system description was an attempt to formulate concepts and languages for analysis and design – Peter Wegner later said that SIMULA was a language with a built-in methodology. We did find the method developed by Michael Jackson [63] more interesting than SA/SD. In SA/SD there is focus on identifying functionality. In Jackson’s method a model of the application domain is first constructed and functionality is then added to this model. The focus on modeling was in much more agreement with our understanding of object-orientation.

In the mid-eighties, Yourdon and others converted to object orientation and published books on object-oriented analysis and design, e.g. [25]. This was in many ways a good turning point for object orientation, because many more people now started to understand and appreciate its modeling advantages.

In 1989 Madsen was asked to give a three-day course on OOD for software developers from Danish industry. He designed a series of lectures based on the abstraction mechanisms of BETA – including the conceptual framework. At the end of the first day, most of the attendees complained that this was not a design course, but a programming course. The attendees were used to SA/SD and had difficulties in accepting the smooth transition from design to implementation in object-oriented languages – it should be said that Madsen was not trying to be very explicit about this. There was no tradition for this in the SIMULA/BETA community – design was programming at a higher level of abstraction.

It actually helped that, after some heated discussions with some of the attendees, a person stood up in the back of the room presenting himself and two others as being from DSB (the Danish railroad company) – he said that his group was using SIMULA for software development and they have been doing design for more than 10 years in the way it had been presented. He said that it was very difficult for them to survive in a world of SA/SD where SIMULA was quite like a stepchild – the only available SIMULA compiler was for a DEC 10/20 which was no longer in production, and they therefore had to use the clone produced by a third party. However, together with the course organizer, Andreas Munk Madsen, Madsen redesigned the next two days' presentations overnight to make more explicit why this was a course on design.

The huge interest in modeling based on object orientation in the late eighties was of course positive. The disadvantage was that now everybody seemed to advocate object orientation just because it had become mainstream. There were supporters (or followers) of object-orientation who started to claim that the world *is* object-oriented. This is of course wrong – object orientation is a *perspective* that one may use when modeling the world. There are many other perspectives that may be used to understand phenomena and concepts of the real world.

# 5. The Language

In this section we describe the rationale for the most important parts of BETA. We have attempted to make this section readable without a prior knowledge of BETA, although some knowledge of BETA will be an advantage. The reader may consult the BETA book [119] for an introduction to BETA.

The BETA language has evolved over many years and many changes to the semantics and syntax have appeared in this period. It would be too comprehensive to describe all of the major versions of BETA in detail. We will thus describe BETA as of today, with emphasis on the rationale and discussions leading to the current design and to intermediate designs. In Section 5.10, we will briefly describe the various stages in the history of the language.

As mentioned in Section 3.1, most language mechanisms in BETA are justified from a technical as well as a modeling point of view. In the following we will attempt to state the technical as well as the modeling arguments for the language mechanisms being presented.

## 5.1 One abstraction mechanism

From the beginning the challenge was to design a programming language mechanism called a *pattern* that would subsume well-known abstraction mechanisms. The common characteristic of abstraction mechanisms is that they are *templates* for generating *instances* of some kind. In the mid seventies when the BETA project started, designers and programmers were not always explicit about whether or not a given construct defined a template or an instance and when a given instance was generated. In this section we describe the background, rationale and final design of the pattern.

### *5.1.1 Examples of abstraction mechanisms*

When the BETA project was started, research in programming languages was concerned with a number of abstraction mechanisms. Below we describe some of the abstraction mechanisms that were discussed in the beginning of the BETA project. We will explicitly use a terminology that distinguishes templates from instances.

**Record type**. A record type as known from Pascal defines a list of fields of possibly different types. The following is an example of a Pascal record type, which is a template for records:

```
type Person =
  record name: String; age: integer end;
```

Instances of `Person` may be defined as follows:

```
var P:Person;
```

Fields of the record `P` may be read or assigned as follows:

```
n:= P.name; P.age:= 16
```

**Value type**. Value types representing numbers, Boolean values, etc. have always been important in programming languages. New abstraction mechanisms for other kinds of value types were proposed by many people. This included compound value types like complex number, enumeration types such as color known from Pascal and numbers with a unit such as speed. The main characteristic of a value type is that it defines a set of values that are assignable and

comparable. Value types may to some extent be defined by means of records and classes, but as mentioned in Section 2.4, we did not think that this was a satisfactory solution. We return to this in Section 5.8.2.

**Procedure/function**. A procedure/function may be viewed as a template for activation records. It is defined by a name, input arguments, a possible return type, and a sequence of statements that can be executed. A typical procedure in a Pascal-like language may look like

```
integer distance(var p1,p2: Point)
   var dist: real
   begin ...; return dist; end
```

A procedure call of the form `d := distance(x,y)` generates an instance in the form of an activation record for `distance`, transmits `x` and `y` to the activation record, executes the statement part and returns a value to be assigned to `d`.

The notion of pure function (cf. Section 4.2) was also considered an abstraction mechanism that should be covered by the pattern.

**Class**. A (simple) class in the style of SIMULA has a name, input arguments, a possible superclass, a set of data fields, and a set of operations. Operations are procedures or functions in the Algol style. In today's object-orientation terminology the operations are called methods. One of the uses of class was as a mechanism for defining abstract data types.

**Module**. The module concept was among others proposed as an alternative to the class as a means for defining abstract data types. One of the problems with module – as we saw it – was that is was often not explicit from the language definition whether a module was a template or an instance. As described in Section 5.8.8, we considered a module to be an instance rather than a template.

**Control abstraction**. A control abstraction defines a control structure. Over the years a large variety of control structures have been proposed. For BETA it was a goal to be able to define control abstractions. Control abstractions were mainly found in languages like CLU that allowed iterators to be defined on sets of objects.

**Process type**. A process type defines a template for either a coroutine or a concurrent process. In some languages, however, a process declaration defined an instance and not a template. In SIMULA, any object is in fact a coroutine and a SIMULA class defines a sequence of statements much like a procedure. A SIMULA class may in this sense be considered a (pseudo) process type. For a description of the SIMULA coroutine mechanism see e.g. Dahl and Hoare [30]. In Concurrent Pascal the SIMULA class concept was generalized into a true concurrent process type [17].

The relationship between template and instance for the above abstraction mechanisms is summarized in the table below:

| Abstraction/template | Instance |
|---|---|
| record type | record |
| value type | value |
| procedure/function | activation record |
| class | object |
| control abstraction | control activation |
| module? | module |
| process type | process object |

The following observations may further explain the view on abstraction mechanisms and instances in the early part of the BETA project:

- Some of terms in the above table were rarely considered by others at the programming level, but were considered implementation details. This is the case for activation record, control activation and process object. As we discuss elsewhere, we put much focus on the program execution – the dynamic evolution of objects and actions being executed – for understanding the meaning of a program. This was in contrast to most programming-language schools where the focus was on the program text.
- The notion of value type might seem trivial and just a special case of record type. However, as mentioned in Section 2.4, Nygaard found it doubtful to use the class concept to define value types – we return to this subject in Section 5.8.2.
- A record type is obviously a special case of a class in the sense that a class may 'just' define a list of data fields. The only reason to mention record type as a case here is that the borderline between record type, value type and class was not clear to us.
- If one follows Hoare, an object or abstract data type could only be accessed via its operations. We found it very heavyweight to insist that classes defining simple record types should also define accessor functions for its fields. This issue is further discussed in Section 5.5.

### *5.1.2 Expected benefits from the unification*

As mentioned previously, the pattern should be more than just the union of the above abstraction mechanisms. Below we list some language features and associated issues that should be considered.

- **Pattern**. The immediate benefit of unifying class, procedure, etc. is that this ensures a uniform treatment of all abstraction mechanisms. At the conceptual level,

programmers have a general concept covering all abstraction mechanisms. This emphasizes the similarities among class, procedure, etc., with respect to being abstractions defining templates for instances. From a technical point of view, it ensures orthogonality among class, procedure, etc.

- **Subpattern**. It should be possible to define a pattern as a subpattern of another pattern. This is needed to support the notion of subclass. From the point of view of orthogonality, this means that the notion of subpattern must also be meaningful for the other abstraction mechanisms. For example, since a procedure is a kind of pattern, inheritance for procedures must be defined – and in a way that makes it useful.
- **Virtual pattern**. To support virtual procedures, it must be possible to specify virtual patterns. Again, the concept of virtual pattern must be meaningful for the other abstraction mechanisms as well. As a virtual pattern can be used as a class, the concept of virtual class must be given a useful meaning. It turned out that the notion of virtual class (or virtual type) was perhaps one of the most useful contributions of BETA.
- **Nested pattern**. Since Algol, SIMULA, and DELTA are block-structured languages that support nesting of procedures and classes, it was obvious that BETA should also be a block-structured language. I.e., it should be possible to nest patterns arbitrarily.
- **Pattern variable**. Languages like C contain pointers to procedures. For BETA, procedure typed variables were not considered initially, but later suggested by Ole Agesen, Svend Frølund and Michael H. Olsen in their Master's thesis on persistent objects [4]. The uniformity of BETA implied that we then had classes, procedures, etc. as first-order values.

In addition to being able to unify the various abstraction mechanisms, it was also a goal to be able to describe objects directly without having to define a pattern and generate an instance. This lead to the notion of singular objects:

- **Singular objects**. In Algol and SIMULA it is possible to have inner blocks. In Pascal it is possible to define a record variable without defining a record type. For BETA it was a design goal that singular objects (called *anonymous classes* in Java and Scala [133-135]) should apply for all uses of a pattern. That is, it should be possible to write a complete BETA program in the form of singular objects – without defining any patterns at all.

### *5.1.3 Similarities between object and activation record*

As mentioned in Section 3.1.2, the observation about the similarities between objects and activation records was one of the main motivations for unifying e.g. class and procedure. From the beginning the following similarities between objects and activation records were observed:

- An *object* is generated as an instance of a *class*. An *activation record* is generated as part of a *procedure* invocation. In both cases *input parameters* may be transferred to the object/activation record.
- An object consists of *parameters*, and *data items* (fields). An activation record also consists of *parameters* and *data items* in the form of local variables.
- An object may contain *local procedures* (methods). In a block-structured language an activation record may have *local (nested) procedures*.
- In a block-structured language, an activation record may have a pointer to the statically enclosing activation record (often called the *static link* or *origin*). In SIMULA, classes may be nested, so a SIMULA object may also have an origin.
- An activation record may have a reference pointing to the activation record of the calling procedure (often called the *dynamic link* or *caller*). In most object-oriented languages there is no counterpart to a dynamic link in an object. In SIMULA this is different since a SIMULA object is potentially a coroutine.

Figure 4 shows an example of a SIMULA program except that we use syntax in the style of C++, Java and C#. This example contains the following elements:

```
class Main:
{
  class Person:
    { name: text; age: integer; };
  class Set(size: integer):
    { rep: array(size);
      proc insert(e: object): { do ... };
      virtual proc display(): { do ... };
    };
  proc main():
    { Person Joe = new Person();
      Set S
    do S = new Set(99);
       S.insert(Joe);
       S.display()
    };
}
```

**Figure 4 SIMULA-like program**

- The class `Main` with local (nested) classes `Person` and `Set` and a procedure `main`.
- The class `Person` with instance variables `name` and `age`.
- The class `Set` with a parameter `size`, an instance variable (array) `rep` for representing the set, a non-virtual procedure (method) `insert`, and a virtual procedure `display`.
- The procedure `main` with reference variables `Joe` and `S`.

The example has the following characteristics:

- `Person`, `Set` and `main` are nested within class `Main`.
- Class instances (objects) are created by **`new Set()`.**
- Procedure instances are created by `S.insert(Joe)` and `S.display()`.

Figure 5 shows a snapshot of the execution of the program in Figure 4 at the point where `S.insert(Joe)` is executed at the end of `main`.

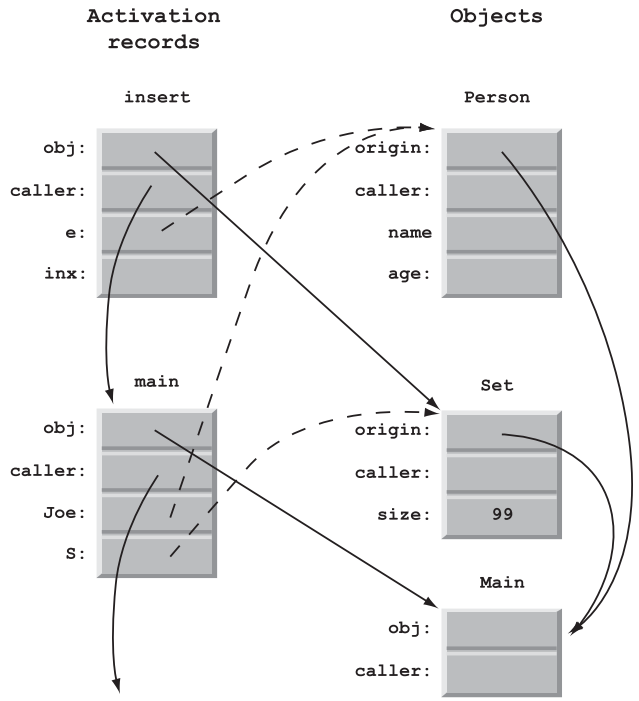


**Figure 5 Objects and activation records**

- The box named `main` is the activation record for `main`. It has a `caller` reference, which in this case is `null` since `main` is the first activation of this program. There is also an object reference (`obj`) to the enclosing `Main` object. In addition it has two data items `Joe` and `S` referring to instances of class `Set` and class `Person`
- The boxes named `Set` and `Person` are `Set` and `Person` objects respectively. Since the example is SIMULA-like, both objects have an `origin` for representing block structure and a `caller` representing the coroutine structure. For both objects `origin` refer to the enclosing `Main` object. The `caller` is `null` since `Set` and `Person` have no statement part.
- The box named `insert` is the activation record for the call of `S.insert(Joe)`. `Caller` refers to `main`. It has an object reference (`obj`) to the `Set` object on which the method is activated. In addition it has two instance variables `e` and `inx`. The variable `e` refers to the same object as `Joe`.
- The box named `Main` represents the `Main` object enclosing the `Set` and `Person` objects and the `main` activation record.

From the above presentation it should be clear that there is a strong structural similarity between an object and an activation record. The similarity is stronger for SIMULA than for languages like C++, Java and C#, since SIMULA has block structure and objects are coroutines. Technically one may think of a SIMULA class as a procedure where it is possible to obtain a reference to the activation record – the activation record is then an instance of the class.

### *5.1.4 The pattern*

From the discussion of the similarities between class and procedure it follows that the following elements are candidates for a unified pattern:

- The name of the pattern
- The input parameters
- A possible superpattern
- Local data items
- Local procedures (methods) – virtual as well as non-virtual
- Local classes – possible nested classes
- A statement part – in the following called a do-part

One difference between a class and procedure is that a procedure may return a value, which is not the case for a class. To justify the unification we then had a minor challenge in defining the meaning of a return value for a pattern used as a class. We decided that we did not need an input parameter part for patterns. The rationale for this decision and the handling of return values are discussed in Section 5.8.1.

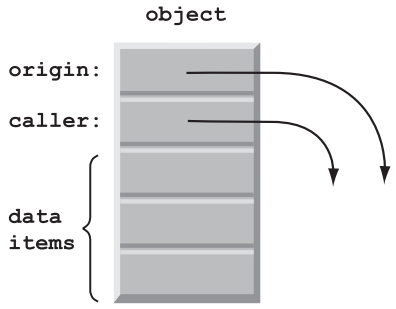


**Figure 6 Object layout**

In conclusion, we decided that a BETA object should have the layout shown in Figure 6. `Origin` represents the static link for nested procedures and objects and the object reference for method activations. Note that since a method is actually nested inside a class, there is no difference between the `origin` of a method activation and its object reference (`obj` in the above example). For patterns that are

not nested within other patterns, `origin` may be eliminated. `Caller` represents the dynamic link for activation records and coroutine objects. For patterns without a do-part, `caller` may be eliminated.

Figure 7 shows how the example from Figure 4 may be expressed in BETA. All of the classes and procedures have been expressed as patterns.

```
Main:
  (# Person: (# ... #);
     Set:
       (# insert:
           (# e: ^object
           enter e[] do ... #);
          display:< (# ... #);
          ...
       #);
     main:
       (# Joe: ^Person;
          S: ^Set
       do &Person[] -> Joe[];
          &Set[] -> S[];
          Joe[] -> S.insert;
          S.display
       #);
  #)
```

**Figure 7 Pattern example**

Basically it is a simple syntactic transformation:

- The keywords `class` and `proc` are removed.
- The brackets `{` and `}` are replaced by `(#` and `#).`
- The `insert` input parameter part `(e: Object)` is replaced by a declaration of an object reference variable (`e: ^object`) and a specification that `e` is the input parameter (`enter e[]`).
- The symbol `^` in a declaration of a variable specifies that the variable is a (dynamic) reference – we show below that variables can also be defined as static.
- The symbol '`<`' specifies that `display` is a virtual pattern.
- The keyword '**`do`**' separates the declaration part and the do-part.
- The symbol `&` corresponds to `new` – i.e. `&Set` generates an instance of pattern `Set`.
- The symbol `[]` in an application of a name, as in `Joe[],` signals that the value of `Joe[]` is a *reference* to the object `Joe`. This is in contrast to `Joe,` which has the *object* `Joe` as its value.
- Assignment has the form `exp -> V`, where the value of `exp` is assigned to `V`.
- The parentheses `()` are removed from procedure declarations and activations.
- The arrow `->` is also used for procedure arguments, which are treated as assignments. This is the case with `Joe[] -> S.insert`, where `Joe[]` is assigned to the input parameter `e[]`.
- Instances of a pattern may be created in two ways:
  - The constructs `&Set[]`. The value of `&Set[]` is a reference to the new `Set` instance and corresponds to the `new` operator in most object-oriented languages.
  - `S.display`. Here a new `display` instance is generated and executed (by executing its `do`-part). This corresponds to a procedure instance as shown in Figure 4.

The general form for a pattern is

```
P: superP                  // super pattern
   (# A1; A2;...;An        // attribute-part
   enter (V1, V2,...,Vs) // enter-part
   do I1; I2;...;Im        // statements
   exit (R1, R2,...,Rt)  // exit-part
   #)
```

- The super-part describes a possible super pattern.
- The attribute-part describes declaration of attributes including variables and local patterns
- The enter-part describes an optional list of input parameters.
- The do-part describes an optional list of executable statements.
- The exit-part describes an optional list of output values.

As is readily seen from this general form for a pattern, a pattern may define a simple record type (defining only attributes), it may define a class with methods (in which case the local patterns are methods), or it may define procedures/functions, in which case the enter/exit lists work as input/output parameters.

```
ia: S.insert;
ia = new S.insert();
ia.e = Joe;
ia.execute();
```

**Figure 8 Decomposition of `S.insert(Joe)`**

As mentioned above, a procedure call may be described as a generation of an activation record, transfer of arguments, and execution of the code of the procedure. In Figure 8, such a decomposition of the call `S.insert(Joe)` from Figure 4 is shown:

- A variable `ia` of type `S.insert` is declared.
- An instance of `S.insert()` is assigned to `ia`.
- The argument `e` is assigned the value of `Joe`.
- The statement part of `ia` is executed.

In BETA it is possible to write this code directly. This also implies that the activation record (object) `ia` may be executed several times by reapplying `ia.execute()`. Such procedure objects were referred to as static procedure instances and considered similar to FORTRAN subroutines.

Although in a pure version of BETA one could imagine that all procedure calls would be written as in Figure 8, this would obviously be too clumsy. From the beginning an abbreviation corresponding to a normal syntax for procedure call was introduced.

### *5.1.5 Subpatterns*

With respect to subpatterns a number of issues were discussed over the years. As with a SIMULA subclass, a subpattern inherits all attributes of its superpattern. We did discuss the possibility of cancellation of attributes as in Eiffel, but found this to be incompatible with a modeling approach where a subpattern should represent a specialization of its superpattern. We also had to consider how to combine the `enter`-, `do`- and `exit`-parts of a pattern and its superpattern. For the `enter`- and `exit`-parts we decided on simple concatenation as in SIMULA, although alternatives were discussed, such as allowing an inner statement (cf. Section 5.6) inside an `enter`/`exit` part to specify where to put the `enter`/`exit` part of a given subpattern. The combination of do-parts is discussed in Section 5.6. The following is an example of a subpattern:

```
Student: Person (# ... #)
```

The pattern `Student` is defined as a subpattern of `Person`. The usual rules regarding name-based subtype substitutability applies for variables in BETA. As in most class-based languages, an instance of `Student` has all the properties defined in pattern `Person`.

In SIMULA a subclass can only be defined at the same block level as that in which its superclass is defined. The following example where `TT` is not defined at the same block level as its superclass `T` is therefore illegal:

```
class A: {                  // block-level 0
   class T: { ... }         // block-level 1
   class X: {               // block-level 1
      class TT: T { ... } // block-level 2
   }
}
```

BETA does not have this restriction. The restriction in SIMULA was because of implementation problems. We return to this question in Section 6.4.

Multiple inheritance has been an issue since the days of SIMULA – we return to this issue in Section 5.5.1 and 5.8.12.

### *5.1.6 Modeling rationale*

The rationale for one pattern as described in Section 3.1.2 and above is mainly technical. For a modeling language it is essential to be able to represent concepts and phenomena of the application domain. Abstraction mechanisms like class, procedure and type may represent specialized concepts from the application domain. It seemed natural to be able to represent a concept in general. The idea of having one pattern mechanism generalizing all other abstraction mechanisms was then considered well motivated from this point of view also.

Since the primary purpose of patterns was to represent concepts, it has always been obvious that a subpattern should represent a specialized concept and thereby be a specialization of the superpattern. This implies that all properties of the superpattern are inherited by the subpattern. The ideal would be to ensure that a subpattern is always a behavioral specialization of the superpattern, but for good reasons it is not possible to ensure this by programming language mechanisms alone. The language rules were, however, designed to support behavioral specialization as much as possible.

The notions of type and class are closely associated. Programming language people with focus on the technical aspects of a language often use the term ‘type’, and the purpose of types is to improve the readability of a program, to make (static) type checking possible and to be able to generate efficient code. A type may, however, also represent a concept, and for BETA this was considered the main purpose of a type. Many researchers (like Pierre America [8] and William Cook [27]) think that classes and types should be distinguished – classes should only be used to construct objects and types should be used to define the interfaces of objects. We have always found that it was an unnecessary complication to distinguish between class and type.

There was also the issue of name or structural equivalence of types/classes. From a modeling point of view it is rarely questioned that name equivalence is the right choice. People in favor of structural equivalence seem to have a type-checking background. The names and types of attributes of different classes may coincidentally be the same, but the intention of the two classes might be quite different. Boris Magnusson [121] has given as example class `Cowboy` and class `Figure` that both may have a `draw` method. The meaning of `draw` for `Cowboy` is quite different from the meaning of `draw` for `Figure`. The name equivalence view is also consistent with the general view of a concept being defined by its name, intension (attributes) and extension (its instances).

Another issue that is constantly being discussed is whether or not a language should be statically or dynamically typed. From a modeling point of view there was never any doubt that BETA should be statically typed since the type (class) annotation of variables is an essential part of the description

of a model. BETA is, however, not completely statically typed – cf. Section 5.4.4 on co- and contravariance.

## 5.2 Singular objects

BETA supports singular objects directly and thereby avoids superfluous classes. The following declaration is an example of a singular object:

```
myKitchen: @ Room(# ... #)
```

The name of the object is `myKitchen`, and it has `Room` as a superpattern. The symbol `@` specifies that an object is declared.[8] The object is singular since the object-descriptor `Room(# … #)` is given directly instead of a pattern like `Kitchen`.

Technically it is convenient to be able to describe an object without having to first declare a class and then instantiate an object – it is simply more compact.

With respect to modeling the rationale was as follows:

- When describing (modeling) real-life systems there are many examples of one-of-a-kind phenomena. The description of an apartment may contain various kinds of rooms, and since there are many instances of rooms it is quite natural to represent rooms as patterns and subpatterns (or classes and subclasses). An apartment usually also has a kitchen, and since most apartments have only one kitchen, the kitchen may be most naturally described as a singular object. It should be mentioned that any description (program) is made from a given perspective for a given purpose. In a description of one apartment it may be natural to describe the kitchen as a singular object, but in a more general description that involves apartments that may have more than one kitchen it may be more natural to include a kitchen pattern (or class).
- Development of system descriptions and programs is often evolutionary in the sense that the description evolves along with our understanding of the problem domain. During development it may be convenient to describe phenomena as singular objects; later in the process when more understanding is obtained the description is often refactored into patterns and objects. Technically it is easy to change the description of a singular object to a pattern and an instance – in the same way as a description of a singular phenomenon is easily generalized to be a description of a concept. For an elaboration of this see Chapter 18 in the BETA book [119].

Exploratory programming emphasizes the view of using objects in the exploratory phase, and it is the whole basis for prototype-based object-oriented programming as e.g. in Self. However, as discussed by Madsen [113], prototype-based languages lack the possibility of restructuring objects into classes and objects when more knowledge of the domain has been obtained.

## 5.3 Block structure

Algol allowed nesting of blocks and procedures. SIMULA in addition allowed general nesting of classes, although there were some restrictions on the use of nested classes. For BETA it was quite natural that patterns and singular objects could be arbitrarily nested. Nesting of patterns comes almost by itself when there is no distinction between class and procedure. A pattern corresponding to a class with methods is defined as a class pattern containing procedure patterns, and the procedure patterns are nested inside the class pattern. The pattern `Set` in Figure 7 is an example – the patterns `display` and `insert` are nested inside the pattern `Set`. It is thus quite natural that patterns may be nested to an arbitrary level – just as procedures may be nested in Algol, Pascal and SIMULA, and classes may be nested in SIMULA. With nesting of patterns, nesting of singular objects comes naturally. A singular object may contain inner patterns, just as a pattern may contain inner singular objects.

A major distinction between Smalltalk and the SIMULA/BETA style of object-oriented programming is the lack of block-structure in Smalltalk. Since the mid-eighties, we have been trying to convince the object-oriented community of the advantages of block-structure, but with little success. In 1986 a paper with a number of examples of using block structure was presented at the Hawthorne Workshop (see Section 3.3) and also submitted to the first OOPSLA conference, but not accepted. It was later included in the book published as the result of the Hawthorne Workshop [111]. C++ (and later C#) does allow textual nesting of classes, but only to limit the scope of a given class. In C++ and C# a nested class cannot refer to variables and methods in the enclosing object. Block structure was added to Java in one of the first revisions, but there are a number of restrictions on the use of nested classes, which means that some of the generality is lost. As an example, it is possible to have classes nested within methods (local nested classes), but instances of these classes may not access nonfinal variables local to the method.

In Algol and SIMULA, the rationale for block structure was purely technical in the sense that it was very convenient to be able to nest procedures, classes and blocks. Block structure could be used to restrict the scope and lifetime of a given data item. For some time it was not at all obvious that block structure could be justified from a modeling point of view.

The first step towards a modeling justification for block structure was taken by Liskov and Zilles [109]. Here a

[8] This is in contrast to `^`, which specifies that a reference to an object is declared.

problem with defining classes representing grammars was presented. One of the elements of a grammar is its symbols. It is straightforward to define a class `Grammar` and a class `Symbol`. The problem pointed out by Liskov and Zilles was that the definition of class `Symbol` in their example was dependent on a given `Grammar`, i.e. symbols from an Algol grammar had different properties from symbols from a COBOL grammar. With a flat class structure it was complicated to define a class `Symbol` that was dependent on the actual grammar. With block structure it was straightforward to nest the `Symbol` class within the `Grammar` class.

Another example that helped clarify the modeling properties of block structure was the so-called *prototype abstraction relation problem* as formulated by Brian Smith [146]. Consider a model of a flight reservation system:

- Each entry in a flight schedule like SK471 describes a given flight by SAS from Copenhagen to Chicago leaving every day at 9:40 am with a scheduled flight time of 8 hours.
- A flight entry like SK471 might naturally be an instance of a class `FlightEntry.`
- Corresponding to a given flight entry there will be a number of actual flights taking place between Copenhagen and Chicago. One example is the flight on December 12, 2005 with an actual departure time of 9:45 and an actual flight time of 8 hours and 15 minutes. These actual flights might be modeled as instances of a class `SK471`.
- The dilemma is then that SK471 may be represented as an instance of class `FlightEntry` or as a class `SK471`.
- With nested classes it is straightforward to define a class `FlightEntry` with an inner class `ActualFlight`. SK471 may then be represented as an instance of `FlightEntry`. The SK471 object will then contain a class `ActualFlight` that represents actual instances of flight SK471.

The grammar example and the prototype abstraction relation problem are discussed in Madsen's paper on block structure [111] and in the BETA book [119].

Eventually block structure ended up being conceived as a means for describing concepts and objects that depend on and are restricted to the lifetime of an enclosing object. In the BETA book [119], the term *localization* is used for this conceptual means. The modeling view is in fact consistent with the more technical view of block structure as a construct for restricting the lifetime of a given data item.

## 5.4 Virtual patterns

One of the implications of having just one abstraction mechanism was that we would need a virtual pattern mechanism in order to support virtual procedures. Since a pattern may be used as e.g. a class, we needed to assure that it was meaningful to use a virtual pattern as a class. Algol, SIMULA and other languages had support for higher-order procedures and functions and proposals for higher-order types and classes had started to appear. Quite early in the BETA project it was noticed that there was a similarity between a procedure as a parameter and a virtual procedure. It was thus obvious to consider a unification of the two concepts. In the following we discuss virtual patterns used as virtual procedures and as virtual classes. Then we discuss parameterized classes and higher-order procedures and functions.

### *5.4.1 Virtual procedures*

Virtual patterns may be used as virtual procedures, as illustrated by the pattern `display` in Figure 7. The main difference from virtual procedures in SIMULA and other languages is that in BETA a virtual procedure is not redefined in a subclass, but extended. The reason for this was a consequence of generalizing virtual procedure to cover virtual class, as described in the next section. Consider the example:

```
Person:
  (# name: @text;
     display:<
       (# do name[]->out.puttext; inner #)
  #)
Employee: Person
  (# salary: @integer;
     display::<(# do salary->out.putint #)
  #)
```

The `display` procedure in `Employee` is combined using inner with the one in `Person` yielding the following pattern

```
display:
  (# do name[] -> out.puttext;
        salary -> out.putint
  #)
```

For further details, see the BETA book [119]; we return to this discussion in the sections on virtual class and specialization of actions.

Wegner and Zdonik [160] characterized the different notions of class/subclass relationships as name-, signature-, or behavior-compatible. SIMULA has signature equivalence since the signature of the method in the super- and subclass must be the same. This is not the case for Smalltalk since there are no types associated with the declaration of arguments. I.e. a method being redefined must have the same name and number of arguments as the one from the superclass, but the types may vary. For BETA it was obvious that at least the SIMULA rule should apply. The modeling emphasis of BETA implied that from a semantic point of view a subclass should be a specialization of the superclass – i.e. behaviorally compatible. This means that code executed by a redefined method should not break invariants established by the method in the superclass.

Behavioral equivalence cannot be guaranteed without a formal proof and therefore cannot be expressed as a language mechanism. For BETA we used the term structural compatibility as a stronger form than signature compatibility. In BETA it is not possible to eliminate code from the superclass. It is slightly closer to behavioral equivalence since it is guaranteed that a given sequence of code is always executed – but of course the effect of this can be undone in the subclass.

### *5.4.2 Virtual classes*

As mentioned above, it was necessary to consider the implications of using a virtual pattern as a class. At a first glance, it was not clear that this would work, as illustrated by the following example:

```
Set:
  (# ElmType:< (# key: @integer #);
     newElement:
       (# S: ^ElmType;
       do &ElmType [] -> S[];
          newKey -> S.key;
        #)
  #);

PersonSet:
  Set(# ElmType::< (# name: @Text #)#)

PS: @PersonSet;
```

The pattern `Set` has a virtual pattern attribute `ElmType`, which is analogous to the virtual pattern attribute `display` of `Person` above. In `Person`, the pattern `display` is used as a procedure, whereas `ElmType` in `Set` is used as a class. In `newElement`, an instance of `ElmType` is created using `&ElmType[]`. This instance is assigned to the reference `S` and the attribute `key` of `S` is assigned to in `newKey -> S.key`.

In SIMULA a virtual procedure may be redefined in a subclass. If redefinition is also the semantics for a pattern used as a class then the `ElmType` in instances of `PersonSet` will be `ElmType` as defined in `PersonSet`. This implies that an execution of `&ElmType[]` in `PS.newElement` will create an instance of `ElmType` defined as `(# name: @Text #)`, and with no `key` attribute. A subsequent execution of `newKey -> S.key` will then break the type checking. This was considered incompatible with the type rules of SIMULA where at compile time it is possible to check that a remote access like `newKey -> S.key` is always safe.

We quickly realized that if `PersonSet.ElmType` was a subclass of `Set.ElmType`, then the type checking would not break, i.e. the declaration of `PersonSet` should be interpreted as:

```
PersonSet: Set
  (# ElmType::<
       Set.ElmType(# name: @Text #)
  #)
```

That is, `ElmType` in `Set` is implicitly the superpattern of `ElmType` in `PersonSet`. We introduced the term *further binding* for this to distinguish the extension of a virtual from the traditional redefinition semantics of a virtual.

With redefinition of virtual patterns being replaced by the notion of extension, it was necessary to consider the implications of this for virtual patterns used as procedures. It did not take long to decide that extension was also useful for virtual patterns used as procedures. A very common style in object-oriented programming is that most methods being redefined start by executing the corresponding method in the superclass using super. With the extension semantics, one is always guaranteed that this is done. Furthermore, as discussed below in the section on specialization of actions, it is possible to execute code before and after code in the subclass.

As mentioned in Section 5.4.1, we assumed that signature compatibility from SIMULA should be carried over to BETA. The extension semantics includes this and in addition gives the stronger form of structural compatibility. Again, from a modeling point of view it was pretty obvious (to us) that this was the right choice.

The disadvantage of extension is less flexibility. With redefinition you may completely redefine the behavior of a class. One of the main differences between the U.S. school and Scandinavian school of object-orientation was that the U.S. school considered inheritance as a mechanism for incremental modification or reuse (sometimes called code sharing) [160]. It was considered important to construct a new class (a subclass) by inheriting as much as possible from one or more superclasses and just redefine properties that differ from those of the superclasses. Belonging to the Scandinavian school, we found it more important to support behavioral compatibility between subclasses than pure reuse.

The only situation we were not satisfied with was the case where a virtual procedure was defined as a default behavior and then later replaced by another procedure. This was quite common in SIMULA. We did consider introducing default bindings of virtuals, but if a default binding was specified, then it should not be possible to use information about the default binding. That is, if a virtual `V` is declared as `V:< A` and `AA` (a subpattern of `A`) is specified as the default binding, then `V` is only known to be an `A`. New attributes declared in `AA` cannot be accessed in instances of `V`. We did never implement this, but this form of default bindings was later included in SDL 92 (see Section 7.3).

### *5.4.3 Parameterized classes*

It was a goal that virtual patterns should subsume higher-order parameter mechanisms like name and procedure parameters and traditional virtual procedures. In addition it was natural to consider using virtual patterns for defining

parameterized classes. The use of (locally defined) virtual patterns as described above was a step in the right direction: the pattern `PersonSet` may be used to represent sets of persons by their `name`, and other attributes like `age` and `address` may also be added to `ElmType`. We would, however, like to be able to insert objects of a pattern `Person` into a `PersonSet`. In order to do this we may define a parameterized class `Set` in the following way:

```
Set:
  (# ElmType:< (# #);
     insert:<
       (# x: ^object; e: ^ElmType
       enter x[]
       do &ElmType[] -> e[];
          inner;
          e[] -> add
       #)
  #)
```

The virtual pattern `ElmType` constitutes the type parameter of `Set`. The pattern `add` is assumed to store `e[]` in the representation of `Set`. A subclass of `Set` that may contain `Person` objects may be defined in the following way:

```
PersonSet: Set
  (# ElmType::< (# P: ^Person #) #);
     insert::<(# do x[] -> e.P[] #)
  #)
```

The virtual pattern `ElmType` is extended to include a reference to a `Person`. The parameter `e[]` of insert is stored in `e.P[]`. This would work, but it is an indirect way to specify that `PersonSet` is a set of `Person` objects. Instead one would really like to write:

```
Set:
  (# ElmType:< object;
     insert:<
       (# x: ^ElmType
       enter x[]
       do (* add X[] to the rep. of Set *)
       #)
   #)

PersonSet: Set (# ElmType::< Person #)
```

Here `ElmType` is declared as a virtual pattern of type `object`. In `PersonSet`, `ElmType` is extended to `Person`, and in this way, the declaration of `PersonSet` now clearly states that it is a set of `Person` objects. It turned out that it was quite straightforward to allow this form of semantics where a virtual in general can be qualified by and bound to a nonlocal pattern – just as a combination of local and nonlocal patterns would work. The general rule is that if a virtual pattern is declared as `T:< D` then `T` may be extended by `T::< D1` if `D1` is a subpattern of `D`. `T` may be further extended using `T::< D2` if `D2` is a subpattern of `D1`. The `PersonSet` above may be extended to a set holding `Students`, as in

```
StudentSet:
  PersonSet(# ElmType::< Student #)
```

A *final binding* of the form `ElmType:: Student` may be used to specify that `ElmType` can no longer be extended.

Both forms (`V:< (# … #)` and `V:< A`) of using virtual patterns have turned out to useful in practice – examples may be found in the OOPLSA'89 paper [117], and the BETA book [119].

### *5.4.4 Co- and contravariance*

For parameterized classes, static typing, subclass substitutability and co- and contravariance have been central issues. Most researchers seem to give static typing the highest priority, leading to – in our mind – limited and complicated proposals for supporting parameterized classes. In our 1990 OOPSLA paper [115] the handling of these issues in BETA was discussed. Subclass substitutability is of course a must, and covariance was considered more useful and natural than say contravariance. This implies that a limited form of run-time type checking is necessary when using parameterized classes – which in BETA are supported by patterns with virtual class patterns.

SIMULA, BETA, and other object-oriented languages do contain run-time type checking for so-called reverse assignment where a less qualified variable is assigned to a more qualified variable – like

```
aVehicle -> aBus
```

The run-time type checking necessary to handle covariance is similar to that needed for checking reverse assignment.

With emphasis on modeling it was quite obvious that covariance was preferred to contravariance, and it was needed for describing real-life systems. The supporters of contravariance seem mainly to be people with a static type-checking approach to programming.

It is often claimed in the literature (see e.g. [20]) that BETA is not type safe. This is because BETA requires some form of run-time type checking due to covariance. The compiler, however, gives a warning at all places where a run-time type check is inserted. It has often been discussed whether we should insist on an explicit cast in the program at all places where this run-time check is inserted. In SIMULA a reverse assignment may be written as

```
aBus :- aVehicle
```

In this case it is not clear from the program that an implicit cast is inserted by the compiler. SIMULA, however, also has explicit syntax for specifying that a cast is needed for a reverse assignment. It is possible to write

```
aBus:- aVehicle qua Bus
```

Here it is explicit that a cast is inserted. Introducing such an explicit syntax in BETA for reverse assignment and covariant parameters has often been discussed. As an afterthought, some of us would have preferred doing this from the beginning, since this would have 'kept the *static*

*typeziers away'*☺. However, whenever we suggested this to our users, they strongly objected to having to write an explicit cast. With respect to type safety it does not make any difference since a type error may still occur at run-time. We do, however, think that from a language design point of view it would be the right choice to insist on an explicit syntax, since it makes it clear that a run-time check is carried out.

With respect to static typing, it is pointed out in our OOPSLA'90 paper [115] that although the general use of virtual class patterns will involve run-time type checking, it is possible to avoid this by using final bindings and/or part objects (cf. Section 5.5). This has turned out to be very common in practice.

### *5.4.5 Higher-order procedures and functions*

In many languages a procedure[9] may be parameterized by procedures. A procedure specified as a parameter is called a *formal procedure*. The procedure passed as a parameter is called the *actual procedure*. It was an issue from the beginning of the project that formal procedures should be covered by the pattern concept – and it was quickly realized that this could be done by unifying the notions of virtual procedure and formal procedure.

Consider a procedure `fsum` parameterized by a formal procedure `f`, as in:

```
real proc fsum(real proc f){ ... }
```

An invocation of `fsum` may pass an actual procedure `sine` as in:

```
fsum(sine)
```

In BETA a formal procedure may be specified using a virtual procedure pattern as in:

```
fsum:(# f:< realFunction; ... #)
```

An invocation then corresponds to specifying a singular subpattern of `fsum` and a binding of `f` to the `sine` pattern:

```
fsum(# f:: sine #)
```

SIMULA inherited call-by-name parameters from Algol. Value parameters are well suited to pass values around – this is the case for simple values as well as references. Call-by-name-parameters, like formal procedures, involve execution of code. For a call-by-name parameter the actual parameter is evaluated every time the formal parameter is executed in the procedure body – this implies that the context of the procedure invocation must be passed (implicitly) as an argument. It was a goal to eliminate the need for call-by-name parameters, and the effect of call by name can in fact be obtained using virtual patterns.

[9] In this section, *procedure* may be read as *procedure and/or function*.

### *5.4.6 Pattern variables*

Virtual patterns partially support higher-order procedures in the sense that a virtual pattern may be considered a parameter of a given pattern. Originally BETA had no means for a pattern to return a pattern as a value. In general, virtual patterns do not make patterns first class values in the sense that they may be passed as arguments to procedures (through the enter part), returned as values (through the exit part) and be assigned to variables. For some years we thought that using virtual patterns as arguments fulfilled most needs to support higher-order procedures, although it was not as elegant as in functional languages.

Indirectly, the work of Ole Agesen, Svend Frølund and Michael H. Olsen on persistent objects for BETA [4, 5] made it evident that a more dynamic pattern concept was needed. When a persistent object is loaded, its class (pattern) may not be part of the program loading the object. There was thus a need to be able to load its associated pattern and assign it to some form of pattern variable. Agesen, Frølund, and Olsen suggested the notion of a *pattern variable*, which forms the basis for supporting patterns as first-class values.

Consider a pattern `Person` and subpatterns `Student` and `Employee`. A pattern variable `P` qualified by `Person` may be declared in the following way:

```
P: ## Person
```

`P` denotes a pattern that is either `Person` or some subpattern of `Person`. This is quite similar to a reference `R: ^Person` where `R` may refer to an instance of `Person` or subpattern of `Person`. The difference is that `R` denotes an object, whereas `P` denotes a pattern. `P` may be assigned a value in the following way:

```
Student## -> P##
```

`P` now denotes the pattern `Student` and an instantiation of `P` will generate an instance of `Student`. `P` may be assigned a new value as in:

```
Employee## -> P##
```

`P` now denotes the pattern `Employee` and an instantiation of `P` will result in an `Employee` object.

Pattern variables give full support to higher-order procedures in the sense that patterns may be passed as arguments to procedures, returned as values and assigned to variables.

## 5.5 Part objects

From the very start we distinguished between variables as references to autonomous objects separate from the referencing objects, and variables as part objects being constituents of a larger object. We had many examples where this distinction was obvious from a modeling point

of view: car objects with part objects body and wheels and references to a separate owner object, patient objects with organ part objects and a reference to a physician object, book objects with part objects (of type `Text`) representing the title and a reference to an author object, etc. In most of these examples there is always a question about perspective: for the owner, the car is not a car without four wheel part objects, while a mechanic has no problem with cars in which the wheels are separate (i.e. not part) objects.

We were not alone in thinking that from a modeling point of view it is obvious that (physical) objects consist of parts. At the ECOOP'87 conference Blake and Cook presented a paper on introducing part objects on top of Smalltalk [12]. In [163] Kasper Østerbye described the proper (and combined) use of 'parts, wholes and subclasses'. The example we used in our paper on part objects [118] was inspired by an example from Booch [14]: the problem presented there was to represent (in Smalltalk) buildings (for the purpose of heating control) as objects consisting of objects representing the parts of the building. While the Booch method and notation had no problem in modeling this, it was not possible in Smalltalk, where only references were supported.

In our paper we used an apartment with kitchen, bath, etc. as example:

```
Apartment:
   (# theKitchen: @Kitchen;
      theBathroom: @Bathroom;
      theBedroom: @Bedroom;
      theFamilyRoom: @FamilyRoom;
      theOwner: ^Person;
      theAddress: @Address;
      ...
   #)
```

Note the difference between the rooms of the apartment modeled by part objects (using `@`) and the owner modeled by a *reference variable* (`theOwner`) to a separate object (using `^`).

Although BETA was designed from a modeling point of view, it was still a programming language, so we did not distinguish between parts objects modeling real parts (as the rooms above) and part objects implementing a property (`theAddress` property above) – in BETA terms they are all part objects.

Another problem with Smalltalk was that it allowed external access only to methods, while all instance variables were regarded as private. The example would in Smalltalk have to have access methods for all rooms, and in order to get to the properties of these rooms, one would have to do this via these access methods. In BETA we allowed access to variables (both part objects and references) directly, so with the example above it is possible to e.g. invoke the `paint` method in `theKitchen` as follows:

```
...; myApartment.theKitchen.paint; ...
```

Comparing BETA with Java, a reference to an object (like `theOwner` variable above) corresponds to a Java reference variable typed with `Person`, while a part object is a final reference variable.

A less important rationale for part objects was that part objects reflected the way ordinary variables of predefined value types like Integer, Real, Boolean, etc. were implemented, and we regarded e.g. Integer, Real and Boolean as (predefined) patterns.

### *5.5.1 Inheritance from part objects*

In the part object paper we wrote:

> 'In addition to the obvious purpose of modeling that wholes consist of parts, part objects may also be used to model that the containing object is characterized by various aspects,[10] where these aspects are defined by other classes.'

This reflects discussions we had, but they never led to additional language concepts. It does, however, illustrate the power of combining part objects, block structure and virtual patterns.

Multiple inheritance by part objects. We explored the possibility of using part objects to represent various aspects of a concept. This was partially done in order to provide alternatives to multiple inheritance (see also Section 5.8.12). In the following we give an example of using part objects to represent aspects.

`Persons` and `Companies` are examples of objects that may be characterized by aspects such as being addressable and taxable. The aspect of being addressable may be represented by the pattern:

```
Addressable:
  (# street: @StreetName;
     ...
     printLabel:< (# ... #);
     sendMail:< (# ... #)
  #)
```

Similarly, a taxable aspect may be represented by:

```
Taxable:
  (# income: @integer;
     ...
     makeTaxReturn: < (# ... #);
     pay:< (# do ... #)
  #)
```

A pattern `Person` characterized by being addressable and taxable may then be described as follows:

[10] Here *aspect* is used as a general term and does not refer to *aspect-oriented programming*.

```
Person:
  (# name: @PersonName;
     myAddr: @Addressable
       (# printLabel::<
            (# do ...;name.print;... #);
          sendMail::< (# ... #)
       #);
     myTaxable: Taxable
       (# makeTaxReturn::<(# ... #);
          pay::< (# ... #)
       #)
  #)
```

As the descriptor of the `myAddr` part object has `Addressable` as a superpattern, the `printLabel` and `sendMail` virtuals can be extended[11]. Since these extensions are nested within pattern `Person`, an attribute like `Name` is visible. This implies that it is possible to extend `printLabel` and `sendMail` to be specific for `Person`.

A pattern `Company` may be defined in a similar way:

```
Company:
  (# name: @CompanyName;
     logo: @Picture;
     myAddr:@Addressable
       (# printLabel::<
            (# ...;
               name.print;
               logo.print; ...
            #);
          sendMail::< (# ... #)
       #);
     myTaxable: Taxable(# ... #)
  #)
```

Again, notice that a virtual binding like `printLabel` may refer to attributes of the enclosing `Company` object.

In languages with multiple inheritance, `Person` may be defined as inheriting from `Addressable` and `Taxable`. From a modeling point of view we found it doubtful to define say `Person` as a subclass of `Addressable` and `Taxable`. From a technical point of view the binding of virtuals of `Addressable` and `Taxable` in `Person` will all appear at the same level when using multiple inheritance. Using part objects these will be grouped logically. A disadvantage is that these virtuals have to be denoted via the part object, as in

```
aPerson.myAddr.printLabel
aCompany.myTaxable.pay
```

The advantage is that the possibility of name conflicts does not arise.

### *5.5.2 References to part objects*

Subtype substitutability is a key property of object-oriented languages: if e.g. `Bus` is a subclass of `Vehicle` then a reference of type `Vehicle` may refer to instances of class `Bus`. For an aspect like `Addressable` there is not a class/subclass relationship with e.g. class `Person`. If multiple inheritance is used to make `Person` inherit from `Addressable` then a reference of type `Addressable` may refer to instances of class `Person`.

In BETA it is possible to obtain a reference to a part object. This means that a reference of type `Addressable` may refer to a part object of type `Addressable` embedded within a `Person` object. If `anAddr1` and `anAddr2` are of type `Addressable` then the statements below will imply that `anAddr1` and `anAddr2` will refer the `Addressable` part-object of `aPerson` and `aCompany` respectively:

```
aPerson.myAddr[] -> anAddr1[];
aCompany.myAddr[] -> anAddr2[];
```

The effect of this is that `anAddr1` and `anAddr2` refer indirectly to a `Person` and a `Company` object, respectively. This is analogous to a reference of type `Vehicle` may refer to an instance of class `Bus`. It is thus possible to have code that handles `Addressable` objects independently of whether the `Addressable` objects inherits from `Addressable` or have `Addressable` as a part object: Suppose that we have defined `Company` as a subpattern of `Addressable` and `Person` containing an `Addressable` part object as shown above. We may then assign to `anAddr1` and `anAddr2` as follows (assuming that `andAddr1` and `anAddr2` are of type `Addressable`):

```
aCompany[] -> anAddr1
aPerson.myAddr[] -> anAddr2[]
```

Figure 9a shows how `anAddr1` may refer to a `Company`-object as a subpattern of `Addressable`. Figure 9b shows how `anAddr2` may refer to an `Addressable` part object of a `Person` object.

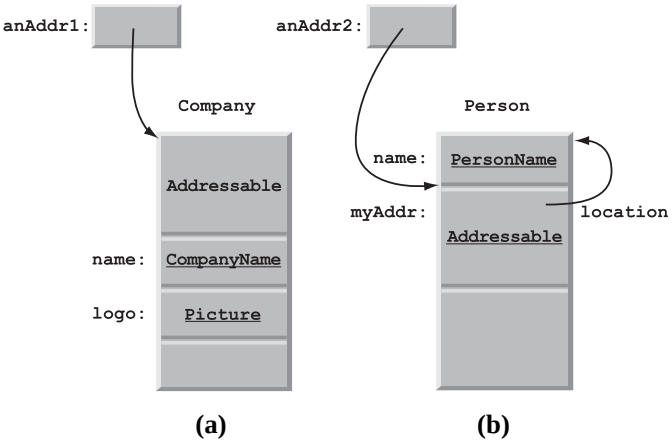


**Figure 9 Inheritance from `Addressable` as super and as part object**

A procedure handling `Addressable` objects – like calling `PrintLabel` – may then be called with `anAddr1` or `anAddr2` as its argument.

[11] It is not important that extension semantics be used – the same technique may be used with redefinition of virtuals.

In many object-oriented languages it is also possible to make a reverse assignment (sometimes called casting) like

```
aVehicle[] -> aBus[]
```

Since it cannot be statically determined if `aVehicle` actually refers an instance of class `Bus`, a run-time type check is needed.

In order to be able to do a similar thing for part objects, we proposed in our part-object paper that a part object be given an extra *location* field containing a reference to the containing object. That is the `myAddr` part of a `Person` object is referencing the containing `Person` object. It is then possible to make a reverse assignment of the form

```
anAddr1.location[] -> aPerson[]
```

As for the normal case of reverse assignment, a run-time check must be inserted in order to check that `andAddr1` is actually a part of a `Person`. After publication of the part object paper, we realized that it would be possible to use the syntax

```
anAddr1[] -> aPerson[]
```

and extend the runtime check to check whether the object referred by `anAddr1` is a subclass of `Person` or a part object of `Person.`

In Figure 9b, the `location` field of the `Addressable` part object is shown. The concept of location was experimentally implemented, but did not become part of the released implementations.

## 5.6 Specialization of actions

From the very beginning we had the approach that specialization should apply to all aspects of a pattern, i.e. it should also be possible to specialize the behavior part of a pattern, not only types of attributes and local patterns. The inspiration was the inner mechanism of SIMULA. A class in SIMULA has an action part, and the inner mechanism allows the combination of the action parts of superclass and subclass. However, we had to generalize the SIMULA inner. In SIMULA, inner was simply a means for syntactically splitting the class body in two. The body of a subclass was defined to be a (textual) concatenation of the pre-inner body part of the superclass, the body part of the subclass, and the post-inner body part of the superclass. In BETA we rather defined inner as a special imperative that – when executed by the superpattern code – implied an execution of the subpattern do-part. This implied that an inner may appear at any place where a statement may appear, be executed several times (e.g. in a loop) and that an action part may contain more than one inner.

### *5.6.1 Inner also for method patterns*

The fact that inner was defined for patterns in general and not only for classes as in SIMULA implied that it was useful also for patterns that defined methods. It was thereby possible to define the general behavior of e.g. the method pattern `Open` of class `File`, with bookkeeping behavior before and after inner, use this general `Open` as a superpattern for `OpenRead` and `OpenWrite`, adding behavior needed for these, and finally have user-defined method patterns specializing `OpenRead` and `OpenWrite` for specific types of files.

Independently of this, Jean Vaucher had developed the same idea but applied to procedures in SIMULA [157].

### *5.6.2 Control structures and iterators*

As mentioned in Section 5.1.1, it was a goal for BETA that it should be possible to define control structures by means of patterns. A simple example of the use of inner for defining control structures is the following pattern:

```
cycle: (# do inner; restart cycle #)
```

Given two file objects `F` and `G`, the following code simply copies `F` to `G`:

```
L: cycle(#
     do (if F.eos then (* end-of-stream *)
           leave L
        if);
        F.get -> G.put
     #);
```

This is done by giving the copying code as the main do-part of an object being a specialization of `cycle`. The copying code will be executed for each execution of `inner` in the superpattern `cycle`.

The perhaps most striking example of the use of inner for defining control structure abstractions is the ability to define iterators on collection objects. If `mySet` is an instance of a pattern `Set` then the elements of the `mySet` may be iterated over by

```
mySet.scan(# do current.display #)
```

The variable `current` is defined in `scan` and refers to the current element of the set. The superpattern `mySet.scan` is an example of a remote pattern used as a superpattern. This is an example of another generalization of SIMULA.

Someone[12] has suggested that a data abstraction should define its associated control structures. By using patterns and inner it is possible in BETA to define control structures associated with any given data structure. Other languages had this kind of mechanism as built-in mechanisms for built-in data structures, while we could define this for any kind of user-defined data structure. At this time in the development there were languages with all kinds of fancy control structures (variations over while and for statements). We refrained from doing this, as it was

[12] We think this was suggested by Hoare, but have been unable to find a reference.

possible to define these by a combination of inner and virtual patterns.

The basic built-in control structures are `leave` and `restart`, which are restricted forms of the goto statement, and the conditional if statement. We were much influenced by the strong focus on structured programming in the seventies. Dijkstra published his influential paper on 'goto considered harmful' [36]. `Leave` and `restart` have the property that they can only jump to labels that are visible in the enclosing scope, i.e. continue execution either at the end of the current block or at the beginning at an enclosing "block". In addition the corresponding control graphs have the property of being reducible. In another influential paper on guarded commands [37], Dijkstra suggested nondeterministic if and while statements. In addition, there was no else clause since Dijkstra argued that the programmer should explicitly list all possible cases. We found his argument quite convincing and as a consequence the BETA if-statement was originally nondeterministic and had no else clause. However, any reasonable implementation of an if statement would test the various cases in some fixed order and our experience is that the programmer quickly relies on this – this means that the program may be executed differently if a different compiler is used. It is of course important to distinguish the language definition from a concrete implementation, but in this case it just seems to add another source of errors. In addition it is quite inconvenient in practice not to have an else clause. We thus changed the if statement to be deterministic and added an else clause.

In principle we could have relied on `leave`/`restart` statements and `if` statements, but also a `for` statement was added. It is, however, quite simple to define a `for` statement as an abstraction using the existing basic control structures. However, the syntax for using control abstractions was not elegant – in fact Jørgen Lindskov Knudsen once said that it was clumsy. Today we may agree, and e.g. Smalltalk has a much more elegant syntax with respect to these matters. In the beginning of the BETA project we assumed that there would be (as an elegant and natural way to overcome these kinds of inconveniences) a distinction between basic BETA and standard BETA where the latter was an extension of basic BETA with special syntax for common abstractions. This distinction was inspired by SIMULA, which has special syntax for some abstractions defined in `class Simulation`.

Furthermore, we considered special syntax for `while` and `repeat` as in Pascal, but this was never included. The `for` statement may be seen as reminiscent of such special syntax.

### *5.6.3 Modeling*

The phenomena of a given application domain include physical material (represented by objects), measurable properties (represented by values of attributes) and transformations (represented by actions) of properties of the physical material – cf. Section 4.2. The traditional class/subclass mechanisms were useful for representing classification hierarchies on physical material. From a modeling point of view it was just as necessary to represent classification hierarchies of actions. This guided the design of using the inner mechanism to combine action parts and thereby be able to represent a classification hierarchy of methods and/or concurrent processes. The paper *Classification of Actions or Inheritance Also for Methods* [101] is an account of this.

## 5.7 Dynamic structure

From the beginning it was quite clear that BETA should be a concurrent object-oriented programming language. This was motivated from a technical as well as a modeling point of view. In the seventies there was lot of research activity in concurrent programming. Most of the literature on concurrency was quite technical and we spent a lot of time analyzing the different forms of concurrency in computer systems and languages. This led to the following classification of concurrency:

- **Hidden concurrency** is where concurrent execution of the code is an optimization made by a compiler – e.g. concurrent execution of independent parts of an expression.
- **Exploited concurrency** is where concurrency is used explicitly by a programmer to implement an efficient algorithm – concurrent sorting algorithms are examples of this.
- **Inherent concurrency** is where the program executes in an environment with concurrent nodes – typically a distributed system with several nodes.

We felt that it was necessary to clarify such conceptual issues in order to design programming language mechanisms. With the emphasis on modeling it was quite clear that *inherent concurrency* should be the primary target of concurrency in BETA.

The quasi-parallel system concept of SIMULA was the starting point for designing the dynamic structure of BETA systems. As mentioned in Section 3.1.3, quasi-parallel systems in SIMULA are based on coroutines, but the SIMULA coroutine mechanism did not support full concurrency and is furthermore quite complex. Conceptually, the SIMULA coroutine mechanism appears simple and elegant, but certain technical details are quite complicated. The coroutine system described by Dahl and Hoare in their famous book on structured programming [29] is a simplified version of the SIMULA coroutine system.

SIMULA did not have mechanisms for communication and synchronization, but several research results within

concurrent programming languages were published in the seventies. Concurrent Pascal [17] was a major milestone with regard to programming languages for concurrent programming. Concurrent Pascal was built upon the SIMULA class concept, but the class concept was specialized into three variants, class, process and monitor. This was sort of the opposite of the BETA goal of unification of concepts. In addition Concurrent Pascal did not have subclasses and virtual procedures. The monitor construct suggested by Brinch-Hansen and Hoare [57] has proved its usability in practice, and was an obvious candidate for inclusion in BETA. However, a number of problems with the monitor concept were recognized and several papers on alternative mechanisms were published by Brinch-Hansen and others [18].

Another important research milestone was CSP [58] where communication and synchronization was handled by input and output commands. Nondeterministic guarded commands were used for selecting input from other processes. We were very much influenced by CSP and later Ada [2] with respect to the design of communication and synchronization in BETA. In Ada communication and synchronization were based on the rendezvous mechanism, which is similar to input/output commands except that procedure calls are used instead of output commands.

### *5.7.1 The first version*

From the beginning, a BETA system was considered a collection of coroutines possibly executing in parallel. Each coroutine is organized as a stack of objects corresponding to the stack of activation records. A coroutine is thus a simple form of thread. In the nonconcurrent situation, at most one coroutine is executing at a given point in time. Since activation records in BETA are subsumed by objects, the activation records may be instances of patterns or singular objects.

The SIMULA coroutine mechanism was quite well understood and the main work of designing coroutines for BETA was to simplify the SIMULA mechanism. SIMULA has symmetric as well as asymmetric coroutines [30]. In BETA there are only asymmetric coroutines – a symmetric coroutine system can be defined as an abstraction. In BETA it is furthermore possible to transfer parameters when a coroutine is called.

Conceptually it was pretty straightforward to imagine BETA coroutines executing concurrently. It was much harder to design mechanisms for communication and synchronization and this part went through several iterations.

The first published approach to communication and synchronization in BETA was based on the CSP/Ada rendezvous mechanism, mainly in the Ada style since procedure calls were used for communication. From a modeling point of view this seemed a good choice since the rendezvous mechanism allowed direct communication between concurrent processes. With monitors all communication was indirect – of course, this may also be justified from a modeling point of view. However, since monitors could be simulated using processes and rendezvous we found that we had a solution that could support both forms of communication between processes.

In CSP and Ada input and output commands are not symmetric: input-commands (accept statements) may be used only in a guarded command. The possibility of allowing output commands as guards in CSP is mentioned by Hoare [58]. For BETA we considered it essential to allow a symmetric use of input and output commands in guarded commands. We also found guarded commands inexpedient for modeling a process engaged in (nondeterministic) communication with two or more processes. Below we give an example of this.

The following example is typical of the programming style used with guarded commands:

- Consider a process `Q` engaged in communication with two other processes `P1` and `P2`.
- `Q` is engaged in the following sequential process with `P1`

  ```
  Q1: cycle{P1.get(V1); S1; P1.put(e1); S2}
  ```

  `Q` gets a value from `P1`, does some processing, sends a value to `P1` and does some further processing. Note that rendezvous semantics is assumed for method invocations. This means that `Q1` may have to wait at e.g. `P1.get(V1)` until `P1` accepts the call.
- `Q` is also engaged in the following sequential process with `P2`:

  ```
  Q2:cycle{ P2.put(e2); S3; P2.get(V2); S4 }
  ```
- `Q1` and `Q2` may access variables in `Q`. A solution where `Q1` and `Q2` are executed concurrently as in:

  ```
  Q: { ... do (Q1 || Q2) }
  ```

  where `||` means concurrency will therefore not work unless access to variables in `Q` is synchronized. And this is not what we want – in general we want to support cooperative scheduling at the language level.
- The two sequential processes have to be interleaved in some way to guarantee mutual access to variables in `Q`. It is not acceptable to wait for `P1` if `P2` is ready to communicate or vice versa. For instance, when waiting for `P1.get` one will have to place a guard that in addition accepts `P2.put` or `P2.get`. It is, however, difficult to retain the sequentiality between `P1.get` and `P1.put` and between `P2.put` and `P2.get`. Robin Milner

proposed the following solution using Boolean variables[13]:

```
Q:
{... do
 if
  B1     and P1.get(V1) then S1;B1:= false
  not B1 and P1.put(e1) then S2;B1:= true
  B2     and P2.put(e2) then S3;B2:= false
  not B2 and P2.get(V2) then S4;B2:= true
 fi
}
```

From a programming as well as a modeling point of view, we found this programming style problematic since the two sequential processes `Q1` and `Q2` are implicit. We did think that this was a step backward since `Q1` and `Q2` were much better implemented as coroutines. One might consider executing `Q1` and `Q2` in parallel but this would imply that `Q1` and `Q2` must synchronize access to shared data. This would add overhead and extra code to this example.

Eventually we arrived at the notion of *alternation*, which allows an active object to execute two or more coroutines while at most one at a time is actually executing. The above example would then look like

```
Q:
{...
 Q1: alternatingTask
     {cycle{ P1.get(V1);S1;P1.put(e1);S2}
 Q2: alternatingTask
     {cycle{ P2.put(e2);S3;P2.get(V2);S4}
do (Q1 | Q2)
}
```

The statement `(Q1 | Q2)` implies that `Q1` and `Q2` are executed in alternation. Execution of `Q1` and `Q2` may interleave at the communication points. At a given point in time `Q1` may be waiting at say `P1.put(e1)` and `Q2` at `P2.get(V2)`. If `P1` is ready to communicate, then `Q1` may be resumed. If on the other hand `P2` is ready before `P1` then `Q2` may be resumed.

The statement `(Q1 | Q2)` is similar to `(Q1 || Q2)` – the former means alternation (interleaved execution at well defined points) and the latter means concurrent execution. Note, that the example is not expressed in BETA, whose syntax is slightly more complicated.

The version of BETA based on CSP/Ada-like rendezvous and with support for alternation is described in our paper entitled Multisequential Execution in the BETA Programming Language [97] (also published in [145]).

### *5.7.2 The final version*

We were happy with the generalized rendezvous mechanism – it seemed simple and general, But when we started using and implementing it, we discovered a number of problems:

- Although the rendezvous mechanism can be used to simulate monitors it turned out to be pretty awkward in practice. As mentioned above the monitor is one of the few concurrency abstractions that have proved to be useful in practice.
- It turned out to be inherently complicated to implement symmetric guarded commands – at least we were not able to come up with a satisfactory solution. In [70] an implementation was proposed, but it was quite complicated.

In addition we realized that the technique for defining a monitor abstraction as presented by Jean Vaucher [157] could also be used to define a rendezvous abstraction, alternation and several other types of concurrency abstractions including semi-coroutines in the style of SIMULA, and alternation. In late 1990 and early 1991, a major revision of the mechanisms for communication and synchronization was made. As of today, BETA has the following mechanisms:

- The basic primitive for synchronization in BETA is the *semaphore*.
- Higher-order concurrency abstractions such as monitor, and Ada-like rendezvous, and a number of other concurrency abstractions are defined by means of patterns in the Mjølner BETA libraries. The generality of the pattern concept, the inner mechanism and virtual patterns are essential for doing this. Wolfgang Kreutzer and Kasper Østerbye [80, 165] also defined their own concurrency abstractions.
- In BETA it is possible to define cooperative as well as preemptive (hierarchical) schedulers in the style of SIMULA. Although there were other languages that allowed implementation of schedulers, they were in our opinion pretty ad hoc and not as elegant and general as in SIMULA. At that time and even today, there does not seem to be just one way of scheduling processes.

For details about coroutines, concurrency, synchronization, and scheduling see the BETA book [119].

### *5.7.3 Modeling*

From a modeling perspective there was obviously a need for full concurrency. The real world consists of active agents carrying out actions concurrently.

In DELTA it is possible to specify concurrent objects, but since DELTA is for system description and not programming, the DELTA concepts were not transferable to a programming language. To understand concurrency from a technical as well as a modeling point of view, we engaged in a number of studies of models for concurrency especially based on Petri nets. One result of this was the

[13] The syntax is CSP/Ada-like,

language Epsilon [65], which was a subset of DELTA formalized by a Petri net model.

For coroutines it was not obvious that they could be justified from a modeling perspective. The notion of *alternation* was derived in order to have a conceptual understanding of coroutines from a modeling point of view. An agent in a travel agency may be engaged in several (alternating) activities like 'tour planning', 'customer service' and 'invoicing'. At a given point in time the agent will be carrying out at most one of these activities.

As for coroutines, the notion of scheduling was not immediately obvious from a modeling point of view. This, however, led to the notion of an ensemble as described in Section 5.8.7 below.

## 5.8 Other issues

Here we discuss some of the other language elements that were considered for BETA. This includes language constructs that were discussed but not included in BETA.

### *5.8.1 Parameters and return values*

In block-structured languages like Algol, the parameters of a procedure define an implicit block level:

```
foo(a,b,c: integer) { x,y,z: real do ... }
```

Here the parameters `a,b,c` corresponds to a block level and the local variables `x,y,z` are at an inner block level. For BETA the goal was that the implicit block level defined by the parameters should be explicit. A procedure pattern like `foo` should then be defined as follows:

```
foo:
  (# a,b,c: integer
  do (#  x,y,z: integer do ... #)
  #)
```

The parameters are defined as data items at the outermost level, and the local variables are defined in a singular object in the do part.

With respect to return values, the initial design was to follow the Algol style and define a return value for a procedure – which in fact is still the style used in most mainstream languages. In most languages a procedure may also return values using call-by-reference and/or call-by-name parameters. However, many researchers considered it bad style to write a procedure that returns values through both its parameters and its return value. This style was (and still is), however, often used if a procedure needs to return more than one value. For BETA (as mentioned elsewhere), call-by-name was not an issue since it was subsumed by virtual patterns. As mentioned below, we did find that call-by-reference parameters would blur the distinction between values and objects. There were language proposals suggesting call-by-return as an alternative to call-by-reference. The advantage of call-by-return was that the actual parameter did not change during the execution of the procedure, but was first changed when the procedure terminated. We did find a need to be able to return more than one value from a procedure and in some languages (like Ada) a variable could be marked as `in`, `out` or `inout` corresponding to call-by-value, -return or both. Finally, there was also a discussion on whether or not arguments should be passed by position or by the name of the parameter. In the first version of BETA all data items at the outermost level could be used as arguments and/or return values, and the name of a data item was used to pass arguments and return values. The pattern `foo` above might then be invoked as follows:

```
foo(put a:=e1, b:=e2) (get v:=b, w:=c)
```

We later found this too verbose, and position-based parameters were introduced in the form of enter/exit lists. The pattern `foo` would then be declared as follows:

```
foo: (# a,b,c: integer
     enter (a,b) do (# ... #)
     exit (b,c)
     #)
```

and invoked as follows:

```
(e1,e2) -> foo -> (v,w)
```

In this example, `enter` corresponds to defining `a,b` as `in` parameters and `exit` corresponds to defining `b,c` as `out` parameters, i.e. `b` was in fact an `inout` parameter.

There were a number of intermediate steps before the enter/exit parts were introduced in their present form. One step was replacing the traditional syntax for calling a procedure with the above (and current) postfix notation. In a traditional syntax the above call would look like:

```
(v,w) := foo(e1,e2)
```

If `e1` and `e2` also were calls to functions, a traditional call might look like:

```
(v,w):= foo(bar(f1,f2),fisk(g1,g2))
```

We did not find this to be the best syntax with respect to readability – in addition, we would like to write code as close as possible to the order of execution. This then led to the postfix notation where the above call will be written as

```
((f1,f2)->bar,(g1,g2)->fisk)->foo->(v,w)
```

We found this more readable, but others may of course disagree.

The enter/exit part may be used to define value types. In this case, the exit part defines the value of the object and the enter part defines assignment (or enforcement) of a new value on the object. The following example shows the definition of a complex number:

```
complex:
 (# x,y: @ real enter(x,y) exit(x,y)#)
```

Complex variables may be defined as follows:

```
C1,C2: @complex
```

They may be assigned and compared: In `C1 -> C2`, the `exit` part of `C1` is assigned to the `enter` part of `C2`. In `C1 = C2` the exit part of `C1` is compared to the `exit`part of `C2`.

As part of defining a value type we would also like code to be associated with the value and assignment. For this reason, the enter/exit-part is actually a list of evaluations that may contain code to be executed. For purely illustrative purposes the following definition of complex keeps track of the number of times of the value is read or assigned:

```
complex:
  (# x,y: @real; n,m: @integer
  enter (# enter(x,y) do n+1 -> n #)
  exit (# do m+1 -> m exit (m,y) #)
  #)
```

Complex may also have a do-part, which is executed whenever enter or exit is executed. If `C1` is a complex object with a do-part then

- In `C1 -> E`, the `do`, and `exit` part of `C1` is executed
- In `E -> C1`, the `enter`- and `do` part of `C1` is executed
- In `E -> C1 -> F`, the `enter, do` and `exit` parts of `C1` are executed.

The `do` part is thus executed whenever an object is accessed, the `enter` part when it is assigned and the `exit` part when the value is fetched.

One problem with the above definitions of `complex` is that the representation of the value is exposed. It is possible to assign simple values and decompose the exit part, as in

```
(3.14,1.11) -> C1 -> (q,w)
```

To prevent this, it was once part of the language that one could restrict the type of values that could be assigned/read:

```
complex:
  (# x,y: @real
  from complex enter(x,y)
  to complex exit(x,y)
  #)
```

In general any pattern could be written after from/to, but there was never any use of this generality and since we never became really happy with using enter/exit to define value types, the from/to-parts were abandoned.

The SIMULA assignment operators `:=` and `:-` were taken over for BETA. In the beginning `=>` was used for assignment of values and `@>` for assignment of references. However, since enter/exit-lists and lists in general may contain a mixture of values and references, we either had to introduce a third assignment operator to be used for such a mixture, or use one operator. Eventually `->` was selected. The distinction between value and object is thus no longer explicit in the assignment operator. Instead, this is expressed by means of `[]`. An expression `X[]` denotes the reference to the object referred by `X`. An expression `X` denotes the value of the object.

### *5.8.2 Value concept*

The distinction between object and value has been important for the design of BETA. This is yet another example of the influence of SIMULA as exemplified through the operators `:=` and `:-`. In the previous section, we have described how enter/exit may be used to define value types. In this section we discuss some of the design considerations regarding the value concept.

As mentioned, the SIMULA class construct was a major inspiration for the notion of abstract data types developed in the seventies. For Nygaard a data type was an abstraction for defining values, and he found that the use of the class concept for this purpose might create conceptual confusion. In SIMULA, Dahl and Nygaard tried to introduce a concept of value types at a very late stage, but some of the main partners developing the SIMULA compilers refused to accept a major change at that late point of the project. The notion of value type was further discussed in the DELTA project and, as mentioned in Section 2.4, was one of the subprojects defined in JLP. Naturally the concept of value types was carried over to the BETA project.

One may ask why it should be necessary to distinguish value types from classes – why are values not just instances of classes? The distinction between object and value is not explicit in mainstream object-oriented languages. In Smalltalk values are immutable objects. In C++, Java and C# values are not objects, but there does not seem to be a conceptual distinction between object and value – the distinction seems mainly to be motivated by efficiency considerations.

From a modeling point of view, it is quite important to be able to distinguish between values and objects. As mentioned in Section 4, values represent measurable properties of objects. In 1982 MacLennan [110] formulated the distinction in the following way:

> … values are abstractions, and hence atemporal, unchangeable, and non-instantiated. We have shown that objects correspond to real world entities, and hence exist in time, are changeable, have state, and are instantiated, and can be created, destroyed and shared. These concepts are implicit in most programming languages, but are not well delimited.

One implication of the distinction between value and object was that support for references to values as known from Algol 68 and C was ruled out from the beginning. A variable in BETA either holds a value or a reference to an object.

Another implication was that a value conceptually cannot be an instance of a type. Consider an enumeration type:

```
color = (red, green, blue)
```

`Color` is the type and `red`, `green`, and `blue` are its values. Most people would think of `red`, `green` and `blue` as instances of `color`. For BETA we ended up concluding that it is more natural to consider `red`, `green` and `blue` as subpatterns of `color`. The instances of say `green` are then all *green* objects. In Smalltalk `True` and `False` are subclasses of `Boolean`, but they are also objects. Numbers in Smalltalk are, however, considered instances of the respective number classes. For BETA we considered numbers to be subpatterns and not instances. Here we are in agreement with Hoare [55] that a value, like `four`, is an abstraction over all collections of *four* objects.

Language support for a value concept was a constant obstacle in the design of BETA. The enter/exit-part of a pattern, the unification of assignment and method invocation to some extent support the representation of a value concept. For Nygaard this was not enough and he constantly returned to the subject. Value type became an example of a concept that is well motivated from a modeling perspective, but it turned out to difficult to invent language mechanisms that added something new from a technical point of view.

### *5.8.3 Protection of attributes*

There has been a lot of discussion of mechanisms for protecting the representation of objects. As mentioned, the introduction of abstract data types (where a data type was defined by means of its operations) and Hoare's paper on using the SIMULA class construct led to the introduction of **private** and **protected** constructs in SIMULA. Variants of **private** and **protected** are still the most common mechanism used in mainstream object-oriented languages like C++, Java and C#. In Smalltalk the rule is that all variables are private and all methods are public.

We found the private/protected constructs too ad hoc and the Smalltalk approach too restricted. Several proposals for BETA were discussed at that time, but none was found to be adequate.

### 5.8.4 *Modularization*

The concept of interface modules and implementation modules as found in Modula was considered a candidate for modularization in BETA. From a modeling point of view we needed a mechanism that would make it possible to separate the representative parts of a program – i.e. the part that represented phenomena and concepts from the application domain – from the pure implementation details. Interface modules and implementation modules were steps in the right direction.

However, we found that we needed more than just procedure signatures in interface modules, and we also found the concept of interface and implementation modules in conflict with the 'one-pattern concept'. In our view, modules were a mechanism that was used for two purposes: modularizing the program text and as objects encapsulating declarations of types, variables and procedures. In Section 5.8.8 we describe how the object aspect of a module may be interpreted as a BETA object.

For modularization of the program text we designed a mechanism based on the BETA grammar. In principle any sentential form – a correct sequence of terminal and nonterminal symbols from the BETA grammar – can be a module. This led to the definition of the fragment system, which is used for modularization of BETA programs. This includes separation of interface and implementation parts and separation of machine-dependent and independent parts. For details of the fragment system, see the BETA book [119].

### *5.8.5 Local language restriction*

From the beginning of the project it was assumed that a pattern should be able to define a so-called local language restriction part. The idea was that it should be possible to restrict the use of a pattern and/or restrict the constructs that might be used in subpatterns of the pattern. This should be used when defining special purpose patterns for supporting class, procedure, function, type, etc. For subpatterns of e.g. a function pattern the use of global mutable data items and assignment should be excluded. Local language restriction was, however, never implemented as part of BETA, but remained a constant issue for discussion.

A number of special-purpose patterns were, however, introduced for defining external interfaces. These patterns are defined in an ad hoc manner, which may indicate that the idea of local language restriction should perhaps have been given higher priority.

### 5.8.6 *Exception handling*

Exception handling was not an issue when the BETA project started, but later it was an issue we had to consider. We did not like the dynamic approach to exception handling pioneered by Goodenough [45] and also criticized by Hoare [59]. As an alternative we adapted the notion of *static exception handling* as developed by Jørgen Lindskov Knudsen [72]. Knudsen has showed how virtual patterns may be used to support many aspects of exception handling and this style is being used in the Mjølner libraries and frameworks. The BETA static approach proved effective for exception handling in almost all cases, including large frameworks, runtime faults, etc. However, Knudsen [75] later concluded that there are cases (mostly related to third-party software) where static exception handling is not sufficient. In these cases there is a need either to have the compiler check the exception handling rules (as in e.g. CLU) or to introduce a dynamic exception handling concept in addition to the static one. In his paper he describes such a design and illustrates the strengths of combining both static and dynamic exception handling.

### 5.8.7 Ensemble

The relationship between the execution platform (hardware, and operating system) and user programs has been a major issue during the BETA project. As BETA was intended for systems programming, it was essential to be able to control the resources of the underlying platform such as processors, memory and external devices. An important issue was to be able to write schedulers.

The concept of ensemble was discussed for several years, and Dag Belsnes was an essential member of the team during that period. Various aspects of the work on ensembles have been described by the BETA team [93], Dag Belsnes [11], the BETA team [99], and Nygaard [131]. The first account of BETA's ensemble concept is in the thesis of Øystein Haugen [49].

A metaphor in the form of a theatre ensemble was developed to provide a conceptual/modeling understanding of an execution platform. A platform is viewed as an *ensemble* that is able to *perform* (execute) a *play* (program) giving rise to a *performance* (program execution). The ensemble has a set of *requisites* (resources) available in order to perform the play. Among the resources are a set of *actors* (processors). An actor is able to perform one or more *roles* (execute one or more objects) in the play. The casting of roles between actors (scheduling) is handled by the ensemble.

The interface to a given execution platform is described in terms of a BETA program including objects representing the resources of the platform. If a given platform has say four processors, the corresponding BETA program has four active objects representing the processors.

In addition to developing a conceptual understanding of an execution platform, the intention was to develop new language constructs. We think that we succeeded with the notion of ensemble as a concept. With respect to language constructs many proposals were made, but none of these turned out to be useful by adding new technical possibilities to the language. It turned out that the notions of active object and coroutine were sufficient to support the interface to processors and scheduling.

The ensemble concept did have some influence on the language. The Mjølner System includes an ensemble framework defining the interface to the execution platform. For most BETA implementations, one active object is representing the processor. A framework defines a basic scheduler, but users may easily define their own schedulers. An experimental implementation was made for a SPARC multiprocessor – here an active object was associated with each processor and a joint scheduler using these processors was defined as a framework.

Dynamic exchange of BETA systems**.** It was also a goal to be able to write a BETA program that could load and execute other BETA programs. In an unpublished working note [99], we described a mechanism for '*Dynamic exchange of BETA systems'*, which in some way corresponds to class loading in Java. Bjorn Freeman-Benson, Ole Agesen and Svend Frølund later implemented dynamic loaders for BETA.

Memory management was another issue we would have liked to support at the BETA level, but we did not manage to come up with a satisfactory solution.

### 5.8.8 Modules as objects

In the seventies the use of the class construct as a basis for defining abstract data types was often criticized since it implied an asymmetry between arguments of certain operations on a data type. Consider the following definition of a complex number:

```
class Complex:
  { real x,y;
    complex add(complex C) { ... }
    ...
  }

Complex A,B,C;
A:= B.add(C);
```

The asymmetry in the call `B.add(C)` between the arguments `B` and `C` was considered by many a disadvantage of using classes to define abstract data types. As an alternative a module-like concept was proposed by Koster [79]:

```
module ComplexDef: {
  type Complex = record real x,y end
  Complex add(Complex C1,C2) {... }
  ...
}

Complex A,B,C;
A := add(B,C);
```

As can be seen, this allows symmetric treatment of the arguments of `add`.

Depending on the language it was sometimes necessary to qualify the types and operation with the name of the module as in

```
ComplexDef.Complex A,B,C;
A := ComplexDef.add(A,B);
```

Languages like Ada, CLU and Modula are examples of languages that used a module concept for defining abstract data types.

For BETA, a module was subsumed by the notion of *singular* object. The reason for this was that a module cannot be instantiated – there is only one instance of a module and its local types and operations can be instantiated. A complex module may be defined in BETA as follows:

```
ComplexDef: @
 (# Complex:
      (# X,Y: @real enter(X,Y) exit
(X,Y)#)
    add: (# ... #);
    ...
 #)

A,B,C: @ComplexDef.Complex;
(A,B) -> ComplexDef.add -> (B,C)
```

For CLU it was not clear to us whether the cluster concept was an abstraction or an object.

### *5.8.9 Constructors*

The concept of constructors was often discussed in the project, but unfortunately a constructor mechanism was never included in BETA.

The idea of constructors for data types in general was introduced by Hoare (the idea was mentioned on page 55 top in [52] and the word constructor appears in [55]) and was obviously a good idea since it assured proper initialization of the objects. In SIMULA initialization of objects was handled by the do part of the object. As mentioned, all SIMULA objects are coroutines – when an object is generated it is immediately attached to the generating object and will thus start to execute its do part until it suspends execution. The convention in SIMULA was that initialization was provided by the code up to the first detach.

The SIMULA mechanism was not considered usable in BETA. In BETA an object is not necessarily a coroutine as in SIMULA. For BETA we wanted to support the notion of static procedure instance. This is illustrated by the example in Figure 8. The instance `ia` may be considered a static procedure instance and executed several times. We thought that it would not be meaningful to execute `ia` when it is generated. The do part of `insert` describes whatever `insert` should do and not its initialization.

We did consider having constructors in the style of C++, but we did not really like the idea of defining the constructor at the same level as the instance attributes (data items and procedures). We found constructors to be of the same kind as static procedures and static data items. As discussed elsewhere, we found static attributes superfluous in a block-structured language.

We liked the Smalltalk idea of a class object defining attributes like `new` to be global for a given class. Again, as described elsewhere, this should be expressed by means of block structure.

Unfortunately, the issue of constructors ended up as an example in which the search for a perfect solution ended up blocking a good solution – like C++ constructors.

### *5.8.10 Static (class) variables and methods*

Static variables and methods were never an issue. In a block-structured language variables and methods global to a class naturally belong to an enclosing object. Static variables and methods play the roles of class variables and class methods in Smalltalk, and Madsen's paper on block structure [111] discusses how to model metaclasses and thereby class variables and class methods by means of block structure.

A further benefit of block structure is that one may have as many objects of the enclosing class as required (representing different sets of objects of the nested class), while static variables give rise to only one variable for all objects.

### *5.8.11 Abstract classes and interfaces*

One implication of the one-pattern idea was that it was never an issue whether or not to have explicit support for abstract classes (or interfaces as found in Java). An abstract class was considered an abstraction mechanism on the line with class, procedure, type, etc.

If abstract class was to be included in BETA it would be similar to a possible support for class and procedure defined as patterns. I.e. one might imagine that BETA could have support for defining a pattern `AbstractClass` (or `Interface`).

For class and procedure we never really felt a need for defining special patterns. Since a pattern with only local patterns containing just their signature may be considered an abstract pattern, there was never a motivation to have explicit syntactic support for abstract patterns.

### *5.8.12 Multiple inheritance*

BETA does not have multiple inheritance. In fact we did not like to use the term 'inheritance', but rather used 'specialization'. This was deliberate: specialization is a relationship between a general pattern (representing a general concept) and patterns representing more special concepts, and with our conceptual framework as background this was most appealing. The specialized patterns should then have all properties of the more general pattern, and virtual properties could only be extended, not redefined. Inheritance should rather be a relationship between objects, as in everyday language. Specialized real-world phenomena cannot of course in general be substituted in the sense that they behave identically. But specialization implies substitutability in the following sense: a description (including program pieces) assuming certain properties of a general class of phenomena (like vehicles) should be valid no matter what kind of specialization (like car, bus or truck) of vehicle is substituted in the description (program piece). The description (program code) is safe in the sense that all properties are available but typically differ.

During the BETA project there was an ongoing discussion on multiple inheritance within object-oriented programming. Although one may easily recognize the need for multiple classifications of phenomena, the multiple inheritance mechanisms of existing languages were often justified from a pure code-reuse point of view: it was possible to inherit some of the properties of the superclasses but not all. Often there was no conceptual relation between a class and its multiple superclasses.

Language mechanisms for handling name conflicts between properties inherited from multiple superpatterns were a subject that created much interest. In one kind of approach they were handled by letting the order of superclasses define the visibility of conflicting names. From a modeling point of view it does not make sense for the order of the superclasses to be significant. In other approaches name conflicts should be resolved in the program text by qualifying an ambiguous name by the name of the superclass where it was declared. Name conflicts in general and as related to BETA were discussed by Jørgen Lindskov Knudsen [73]. He showed that no unifying name resolution rule can be devised since name conflicts can originate from different conceptual structures. The paper shows that there are essentially three necessary name-resolution rules and that these can coexist in one language, giving rise to great expressive power.

For BETA we would in addition have the complexity implied by inner, e.g. in which sequence should the superpattern actions be executed? There was a proposal that the order of execution should be nondeterministic. Kristine Thomsen [150] elaborated and generalized these ideas.

The heavy use of multiple inheritance for code sharing, and the lack of a need for multiple inheritance in real-world examples implied that we did not think that there was a strong requirement for supporting multiple inheritance. This was perhaps too extreme, but in order to include multiple inheritance the technical as well as modeling problems should be solved in a satisfactory way. We did not feel that we were able to do that.

In practice, many of the examples of multiple inheritance may be implemented using the technique with part objects as described in Section 5.5.1.

#### *5.8.13 Mixins and method combination*

In the beginning mixins, as known from Flavors [23], were never really considered for inclusion in BETA – i.e. covered by the pattern concept. The reason was that we considered mixins to be associated with multiple inheritance, and the concept of mixins seemed to be even further promoting multiple inheritance as a mechanism for code sharing. The semantics of multiple inheritance in Flavors, Loops [13] and Common Lisp [69] where the order of the superclasses was essential did not seem to fit well with a language intended for modeling. Perhaps the emphasis on code sharing in these Lisp-based languages did not make us realize that a mixin can be used to define an aspect of a concept, as discussed in Section 5.5.1.

We found the support for method combination in these languages interesting. Before and after methods are an alternative – and perhaps more general – to the inner mechanism. Method combination is an interesting and important issue. Thomsen proposed a generalization of inner for combination of concurrent actions [151]. Bracha and Cook proposed a mixin concept supporting super as well as inner [15].

## 5.9 Syntax

It is often claimed that BETA syntax is awkward. It is noteworthy that these claims most often come from people not using BETA. Students attending BETA courses and programmers using BETA readily got used to it and appreciated its consistency. We could of course say that 'syntax was not a big issue for us' and 'the semantics is the important issue', but the fact is we had many discussions on syntax and that there is a reason why the syntax became the way it is.

First of all, we had the idea of starting with the desired properties of program executions and then making syntax that managed to express these properties. The terms object and object descriptor are simple examples of this.

*Assignment*: As we generalized assignment and procedure call into execution of objects, and as it was desired to have sequences of object executions, there was obviously a need to have a syntax that reflected what really was going on. The general form therefore became

```
ex1 -> ex2 -> ... -> exn
```

where each of the $ex_i$ is an object execution. Execution involved assignment to the enter part, execution of the do part and assignment from the exit part.

Because references to objects could either be assigned to other references or be used in order to have the denoted object executed, we made the distinction syntactically:

```
ref1[] -> ref2[] ->...-> refn[]
 (* reference assignment *)

ref1 -> ref2 ->...-> refn
 (* object executions *)
```

The two forms could of course be mixed, so if the enter part of the object denoted by $ref_2$ required a reference as input, then that would be expressed by

```
ref1[] -> ref2
```

*Naming*: We devised the following consistent syntax for naming things and telling what they were:

```
<name> ':' <kind> <object descriptor>
```

By <kind> is meant *pattern*, *part object*, *reference variable*, etc. For part object the symbol @ is used to specify the kind and for reference variable ^ is used. For a pattern it was decided to use no symbol. So

```
P: super(# ... #)
```

simply was the syntax for a pattern. A possible super pattern was indicated by a name preceding the main part of the object descriptor and not (as in SIMULA) preceding the pattern name. Objects had two kinds and therefore different syntax: @ for a part object and ^ for a reference (to a separate object):

```
P:(# anA: @A;
     aRefToA: ^A;
     ...
  #)
```

Whenever a descriptor was needed (in order to tell e.g. what the properties of part object are) we allowed either an identifier (of a pattern) or a whole object descriptor:

```
P:(#
  anA1: @A;
    (* pattern-defined part object *)

  anA2: @(# ...#);
    (* singular part object *)

  aSpecialA: @A(# ...#);
  ...
#)
```

The last part object above has an object descriptor that specializes A. This was made possible by the above syntax where a super pattern is indicated by a name preceding the main part of the object descriptor.

*Parentheses*: There are two reasons for using (# ... #) instead of {...}. The first was that we imagined that there would be more than object descriptors that needed parentheses. At one point in time there were discussions about (@ ... @) meaning description of part objects and (= ... =) meaning description of values. This was never introduced, but later we introduced ( ) to mean begin end of more than just object descriptors, e.g.

```
(for ... repeat ... for)
(if ... if)
```

We felt that his was obviously nicer than e.g.

```
for ... repeat ... endfor
if ... endif
```

found in other languages at that time. Although we did not introduce e.g. (@ ... @), we still reserved the # to mean descriptor, so that (# ... #) could be read 'begin descriptor ... descriptor end'. The syntax for pattern variables uses # in order to announce that these are variables denoting descriptors.

## 5.10 Language evolution

In this section we briefly comment on how the language has evolved since 1976. Some of the events discussed below are also mentioned in Section 3.3.

The first account of BETA was the 1976 working note (First language draft [89]). At this stage the BETA project had mainly been concerned with discussing general concepts and sketching language ideas. A large part of the working note was devoted to a discussion of the DELTA concepts and their relation to BETA. The language itself was quite immature, but a first proposal for a pattern mechanism was presented. The report did not contain any complete program examples – an indication of the very early stage of the language.

The report includes a long analytical discussion of issues related to concurrency – this includes representative states and an interrupt concept. We had very little experience in issues related to concurrent programming. Various generalizations of the SIMULA coroutine mechanism were discussed. A lot of stacks were drawn and there were primitives like ATTACH X TO Y that could be used to combine arbitrary stacks. A few other language constructs were sketched, but not in an operational form – they were abandoned in future versions of BETA.

The syntax was quite verbose due to a heavy use of keywords. Parameters were passed by name and not by position. Objects had general enter/exit lists. The parameter mechanism made it possible to pass parameters and get return values to/from coroutines – something that is not possible in SIMULA.

The unification of name and procedure parameters and virtual procedures was mentioned but not described in the 1976 report. Virtual patterns were mentioned, and it was said that they would be as in SIMULA/DELTA).

The 1978 working note (Draft Proposal of BETA [90]) included a complete syntax, and the contour of the language started to emerge. Virtual patterns were used for method patterns, for typing functions and for typing elements of local arrays, that is virtual classes were in fact there. The syntax was very verbose with keywords, and very different from the final syntax, and the examples were sketchy.

The 1979 working note (First complete language definition [92]) included a complete definition of the language based on attribute grammars. In addition there were several examples.

With respect to language concepts, the 1981 working note (A survey of the BETA Programming Language) was quite similar to the 1979 working note, but there were major changes to the syntax. Most keywords were changed to special symbols: begin and end were replaced by (#, and

`#)`; `virtual` and `bind` were replaced by `:<` and `::<`; `if` and `endif` were replaced by `(if` and `if)`; etc.

As mentioned previously, the POPL'83 paper on BETA (POPL: Abstraction Mechanisms [95]) was an important milestone. The POPL paper described the abstraction mechanisms of BETA. All the basic elements of BETA were in place including pattern, subpattern, block structure, virtual patterns and enter/exit. The syntax was almost as in the final version. The main difference was the use of a **`pattern`** keyword and different assignment operators like `=>` and `@>` corresponding to `:=` and `:-` in SIMULA. It was stated that the application area of BETA was embedded and distributed systems. The distinction between *basic* BETA and *standard* BETA with an extension of basic BETA with special syntax for a number of commonly used patterns was also stated. The POPL'83 paper contains a proposal for a generalization of the virtual pattern concept. The idea was that any syntactic category of the BETA grammar could be used as a virtual definition. The idea of generalized virtuals was, however, never further explored.

The POPL paper was accompanied with a paper describing the dynamic parts – coroutines, concurrency and communication. Communication and synchronization was based on CSP- and Ada-like rendezvous. We never managed to get the concurrency paper accepted at an international conference although we made several attempts – eventually the paper was published in Sigplan Notices [97] in 1985.

A combined version of the POPL paper and the concurrency paper was later (1987) included in the book that was published as a result of the Hawthorne workshop in 1986 [145]. However, the syntax was revised to that used in the final version of BETA.

Syntax Directed Program Modularization**.** A paper on syntax-directed program modularization was published at a conference in 1983 in Stresa [94] describing a proposal to program modularization based on the grammar (cf. Section 5.8.4). These principles for program modularization were further developed in the Mjølner project.

In the March 1986 revision of Dynamic Exchange of BETA Systems, the syntax was still not the final one although it differs from that of the POPL 83 paper.

From late 1986/early 1987, the sequential parts of BETA were stable in the sense that only a few changes were made. Pattern variables were added, the if statement was made deterministic, an else clause was added, and a few other minor details were changed.

During the Mjølner project, the rendezvous mechanism was replaced by the semaphore as a basic primitive for synchronization. In 1975 Jean Vaucher [157] had already shown how inner combined with prefixed procedures can be used to define a monitor abstraction. This was immediately possible in BETA too. It also turned out that the pattern is well suited to build other higher-level concurrency abstractions, including Ada-like rendezvous and futures.

Many of the later papers on BETA were elaborations of the implications of the one-pattern approach. The simplicity of the pattern mechanism makes BETA simple to present, but the implications turned out to be difficult to convey. In many of the papers we therefore decided to use a keyword-based syntax and not the compact BETA syntax. Often redundant keywords like `class` and `proc` were introduced to distinguish between patterns used as classes and procedures. Some of the most important papers are the following:

- Classification of Actions – or Inheritance Also for Methods, presented at ECOOP'87 [101] and described how to use patterns and inner to define a hierarchy of methods and processes.
- What Object-Oriented Programming May Be and What It Does Not Have to Be, presented at ECOOP'88 [116]. Here we for the first time gave our definition of object-oriented programming and compared it with other perspectives on programming.
- Virtual Classes – a Powerful mechanism in Object-oriented Programming, which was presented at OOPSLA'89 [117]. The idea of virtual patterns was presented in the POPL'83 paper [95], but here the implications were presented in greater detail.
- Strong Typing of Object-Oriented Programming Revisited, presented at OOPSLA 90. The goal of this paper was to argue for our choice of covariance at the expense of run-time type checks.

The 1993 book on BETA [119] is the most comprehensive description of the language and the associated conceptual framework.

## 6. Implementations of BETA

During the first period of the BETA project, no attempts were made to implement a compiler. The reasons for this were mainly lack of resources: The implementation of SIMULA had been a major effort requiring a lot of resources. A number of large companies were involved in funding the SIMULA implementations, and we had nothing like this.

The SIMULA compilers were implemented in low-level languages – one of the compilers was even written in machine code. Implementation of the garbage collector, especially, had been a major task. In the beginning of the BETA project, we assumed that we would have to find funding for implementing BETA. We were thus working

from the assumption that we would have to establish a consortium of interested organizations.

There were other reasons than lack of funding. Nygaard was not a compiler person, Møller-Pedersen was employed by the NCC and could only use a limited amount of his time on BETA, and Kristensen and Madsen had to qualify for tenure.

In the early eighties an attempt was made to implement BETA by transforming a BETA program into a SIMULA program. The rationale for this was that we could then use the SIMULA compilers and run-time system for BETA. This project never succeeded – 90% of BETA was easy to map into SIMULA, but certain parts turned out to be too complicated.

During the BETA project, however, Kristensen and Madsen did substantial research on compiler technology and after some years realized that we had perhaps overestimated the job of implementing BETA.

## 6.1 The first implementation

The first implementation. In 1983 Madsen implemented the first BETA compiler in SIMULA. The first version generated code to an interpreter written in Pascal. The second version generated machine code for a DEC-10.

SUN Compiler**.** In 1985 this compiler was ported to a SUN workstation based on the Motorola 68020 microprocessor. This was an interesting exercise in bootstrapping. The SUN compiler was implemented using the DEC compiler, i.e. machine code was generated on the DEC-10 and transferred to the SUN for debugging. Since the turnaround time for each iteration was long, we manually corrected errors directly in the machine code on the SUN in order to catch as many errors as possible in each iteration. Afterwards such errors were fixed in the compiler. It was quite time-consuming and complicated to debug such machine code on the SUN.

The final step was to bootstrap the compiler itself. The DEC-10 was a slow machine and the BETA compiler was not very efficient. Using the compiler to compile itself was therefore a slow process. In addition, the DEC-10 was becoming more and more unstable – and it was decided that it should be closed down. The DEC-10 would not stay running for a whole compilation of the compiler. This meant that it was necessary to dump the state of the compiler at various stages and be able to restart it from such a dump in order to complete a full compilation of the compiler.

This was a complicated and time-consuming process and at one point Madsen did not believe that he would succeed. However, after three attempts, the bootstrapping succeeded and from then on the compiler was running on the SUN. This was a great experience.

The compiler implemented most parts of BETA – however, a garbage collector was not included. At that time we did not think that we had the qualifications to implement a garbage collector – we really needed some of the experienced people from the NCC.

## 6.2 The Mjølner implementations

The Mjølner Project provided the necessary time and resources to implement efficient working compilers for BETA. In the project it was decided to use the existing BETA compiler to implement the new compilers. Without a garbage collector this was not easy – a simple memory management scheme was added such that it was possible to mark the heap and subsequently release the heap until that mark. This was of course pretty unsafe, but we managed to implement the new compilers and the first versions of the MjølnerTool.

Knut Barra from the NCC wrote the first garbage collector [10] for BETA (as part of the SCALA project) and in 1987 a full workable implementation of BETA was available.

Macintosh Compiler**.** In the beginning of the Mjølner Project the SUN compiler was ported to a Macintosh. The Macintosh compiler was a special event. When we started working with Nygaard he did not use computers and he had not done any programming since the early sixties. When the Macintosh arrived we talked him getting a Mac and he quickly became a super user. It was therefore a great pleasure for us to be able to deliver the first Mac compiler to him on his 60th birthday in 1986.

Later in the Mjølner Project the compiler was ported to a number of machines, including Apollo and HP workstations. Nokia was a partner in the Mjølner Project. We ported BETA to an Intel-based telephone switch and implemented a remote debugger for BETA programs running on the switch on an Apollo workstation. This was a major improvement compared to the very long development cycles that were used by NOKIA for developing software for the switch.

The NOKIA compiler was later used as a basis for porting BETA to Intel-based computers running Windows or Linux. It took some time before these compilers were available since for many years the memory management on the Intel processors was based on segments, which were difficult to handle for a language with general references.

As of today there are or have been native BETA compilers for SUN, Apollo, HP, SGI, Windows, Linux and Macintosh.

## 6.3 The JVM, CLR, and Smalltalk VM compilers

In 2003 Peter Andersen and Madsen engaged in a project on language interoperability inspired by Microsoft .NET/CLR, which was announced as a platform supporting

language interoperability – in contrast to the Java/JVM platform. The goal of the project was to pursue to what extent CLR supported language interoperability. Another goal was to investigate to what extent this was supported by the JVM [9].

We managed to implement BETA on both JVM and CLR – i.e. full BETA compilers are running on top of JVM and CLR. The main difficulty was to make all the necessary type information from BETA available in the byte codes. Since BETA in many ways is more general than Java and C#, there are elements of BETA that do not map efficiently to these platforms. The most notorious example of this is coroutines, which are implemented on top of the thread mechanisms.

We are currently engaged in implementing BETA on a VM based on Smalltalk and intended for supporting pervasive computing – this VM is based on the Esmertec OSVM system and is being further developed in the PalCom project. One of the interesting features of this VM is that it has direct support for BETA-style coroutines.

## 6.4 Implementation aspects

We will touch only briefly on implementation aspects of BETA, since a complete description would take up a lot of space. The implementation is inspired by the SIMULA implementations [31, 122] and described by Madsen in the book about the Mjølner Project [112]. The generality of BETA implied that many people thought that it would be quite complicated (if not impossible) to make a reasonably efficient BETA implementation. Here are some of the major issues:

- **Virtual patterns**. The most difficult part of BETA to implement was virtual patterns. There are two aspects of virtual patterns: semantic analysis and run-time organization. The run-time organization was quite straightforward using a dispatch table. Semantic analysis appeared quite complicated – the problem was given the use of a virtual pattern to find the binding of the virtual that was visible at the point of use. The first attempt to write a semantic analyzer was made in a student project that failed, and for some time we were a bit pessimistic about whether or not we would succeed. It was not the virtual pattern concept by itself that was the real problem, but the combination with block structure. However, a (simple) solution was found and later documented by Madsen in a paper at OOPSLA'99: Semantic Analysis of Virtual Patterns [114].
- **Pattern**. The generality of the pattern concept imposed some immediate challenges for an efficient implementation. For a pattern (or singular object) used as a class, there should be code segments (routines) corresponding to generation (allocation and initialization of data items), enter, do and exit. For a pattern used as a procedure there should just be one code segment. We originally assumed that the compiler could detect the use of a given pattern and generate code corresponding to the use. However, with separate compilation of pattern libraries, this is not possible. We ended up with a reasonable approach, but the code is not as efficient as it can be with separate constructs for class and procedure. In practice this has not been considered a problem. With modern just-in-time and adaptive compilation techniques, it should be straightforward to generate code for a pattern depending on its use.
- **Block-structure and subpatterns**. The relaxation of the SIMULA restriction that a subclass may be defined only at the same block level as its superclass gave rise to some discussion of whether or not this would have negative implications for an efficient implementation of block structure as described by Stein Krogdahl [106]. Since Algol, a variable in a block-structured language has been addressed by a pair, [block-level, offset]. By allowing subpatterns at arbitrary block levels, a variable is no longer identified by a unique block level: let `X` be declared in pattern `P`, let `A` and `B` be different subpatterns of `P`, and let `A` and `B` be at different block levels; then `X` in `A` is not at the same block level as `X` in `B`. We instead adapted the approach proposed by Wirth for Pascal to address a data-item by following the static link (`origin`) from the use of a data item to its declaration. This implied that an object has an `origin`-reference for each subclass that is not defined on the same block level as its superclass. For details see Madsen's implementation article [112].
- **The dynamic structure**. The implementation of the dynamic structure has been a subject for much discussion. Due to coroutines, SIMULA objects and activation-records are allocated on the heap. A similar scheme was adapted in the first BETA implementations, i.e. the machine stack was not used. Many people found this too inefficient and the implementation was later changed to use the machine stack. We do not know whether this makes a significant difference or not, since no systematic comparison of the two different techniques has been made. We do know that the heap-based implementation is significantly simpler than that using the machine stack. Whenever BETA has been ported to a new platform, stack handling has been the most time-consuming part to port. The generalization of inner implied that an object will need a `caller`-reference corresponding to each subclass with a non-empty do-part. For the heap-base implementations, these `caller`-references are stored in the object. For the stack-based implementations, the caller references are stored on the machine stack and thus not explicitly in the objects. We have also considered using the native stacks on modern operating systems, but these are too heavyweight for

coroutines – a program may allocate thousands of coroutines, which is beyond the capacity of these systems.

- **External interfaces**. No matter how nice, simple and safe a language you design, you will have to be able to interface to software implemented in other (unsafe) languages. For BETA a large number of external interfaces were made including C, COM, database systems, Java, and C#. This introduced major complications in the compiler since it was most often done on a by-need basis – often with little time for a proper design. In order to support various data types and parameters, BETA was polluted with patterns supporting e.g. pointers to simple data types like integers and C-structs. The handling of external calls further complicated the dynamic implementation since a coroutine stack may contain activations from external calls. If a callback is made from the external code, BETA activations may appear on top of the external stack. Perhaps the worst implication of this is that all BETA applications suffer from libraries and frameworks calling external code. The GUI-frameworks are examples of this: if they were used wrongly by the BETA programmer, the code was very difficult to debug. The lesson here is that external interfaces should be carefully designed and the implementation should encapsulate all external code in such a way that it cannot harm the BETA code – even though this may harm efficiency.
- **Garbage collection**. Over the years the Mjølner team became more and more experienced in writing garbage collectors and a number of different garbage collectors have been implemented varying from mark-sweep to generation-based scavenging. The first implementation of the Train algorithm was implemented for BETA by Jacob Seligmann and Steffen Grarup [144].

# 7. Impact

## 7.1 Teaching

BETA has been used for teaching object-oriented programming at a number of universities. The most important places we are aware of are as follows:

- BETA courses in Aarhus**.**

  At DAIMI, BETA was an integral part of the curriculum at both the undergraduate and graduate level.

  The Institute of Information Studies, Aarhus University is an interesting case, since this is a department in the Faculty of Humanities. Students within humanities traditionally have difficulties in learning programming. BETA was used for more than a decade and selected because of its clean and simple concepts, its modeling capabilities and its associated conceptual framework.

  First draft of BETA book. A first draft of the BETA book [102] was made available (in the late eighties) to these students, and several versions of the BETA book [119] were tested here before the final version was printed. Originally all examples in the book were typical computer science examples such as stack, queue, etc. Such examples are not motivating for students within the humanities, and all the examples were changed to be about real world phenomena such as bank accounts, flight schedules, etc. Kim Halskov Madsen was very helpful in this process. Preprints of the BETA book were for many years distributed at OOPSLA and ECOOP by Mjølner Informatics and for many people these red books were their first encounter with BETA.

- BETA courses in Oslo**.** At the University of Oslo there were courses on specification of systems by means of SDL and BETA in 1988 and 1993 (by Møller-Pedersen and Dag Belsnes) and on object-oriented programming in BETA in 1994 and 1995 (by Møller-Pedersen, Nygaard and Ole Smørdal).
- BETA courses in Aalborg**.** At Department of Computer Science, University of Aalborg courses on object-oriented programming in BETA were given by Kristensen in 1995 and 1996.
- BETA courses in Dortmund**.** As mentioned, BETA was used for introductory programming at the University of Dortmund, Germany. Here the lecturers wrote a book in German on programming in BETA [38].

We believe that teaching of programming should be based on a programming language that reflects the current state of the art and is simple and general. Many schools use mainstream programming languages used in industry. Our experience is that it is easier to teach a state-of-the-art language than a standard industrial language. Students familiar with the state of the art can easily learn whatever industrial language they need to use in practice. The other way around is much more difficult. For BETA it was for many years necessary to argue that it was well suited for teaching. With the arrival of Java this changed, and Java took over at all places where BETA was used.

## 7.2 Research

In general BETA is well cited in the research literature. Perhaps the most influential part of BETA with respect to research is the concept of virtual class based on the use of virtual patterns as classes: Thorup [152], Bruce [19], Thorup [153], Mezini [126], and Odersky [134]. Other aspects of BETA such as inner, singular objects, block structure, and the pattern mechanism, have also been cited by many authors, e.g. Goldberg [44], and Igarashi and Pierce [62]. In 1994, Bill Joy designed a language without subclasses based on the ideas of inheritance from part objects as described in Section 5.5.1 [67]. Also in 1994, Bill Joy gave a talk in a SUN world-wide video conference

where he mentioned BETA as the most likely alternative to C++.

When we designed BETA we did not have deep enough knowledge of formal type theory to be able to establish the precise relations. In 1988/89 Madsen and others at DAIMI started discussions with Jens Palsberg and Michael Schwartzbach on applying type theory to object-oriented languages. Initially the hypothesis of Palsberg and Schwartzbach was that standard type theory could be applied, but they also realized that subtype substitutability and covariance were nontrivial challenges. This led to a series of papers on type theory and object-oriented languages [138] and a well-known book [140]. The main impact for BETA was that we learned that concepts like co- and contravariance were useful for characterizing virtual patterns in BETA. We had a hard time – and still have – relating to concepts such as universal and existential qualifiers, but more recent work has shed some light on this issue. Researchers with interests in such matters might think that virtual patterns are essentially existential types, but this view is too simplistic. One crucial difference, pointed out by Erik Ernst and explored in his work on family polymorphism [40], is that virtual classes rely on a simple kind of dependent types to allow more flexible usage: The unknown type, when bound by an existential quantifier, must be prevented from leaking out, whereas virtual classes can be used in a much larger scope, because the enclosing object can be used as a first-class package.

Schwartzbach and Madsen discussed making a complete formal specification of BETA's type system. Schwartzbach concluded at that time that the combination of block structure and virtual patterns made it very hard and we never succeeded. Igarashi and Pierce [61] and the authors mentioned below have over the years provided elements of formalization, including virtual classes and block structure.

Palsberg and Schwartzbach also did a lot of interesting work on type inference [139]. Two students of Schwartzbach implemented a system that could eliminate most (all) of the run-time checks in BETA and also detect the use of a given pattern and thereby optimize the code generation. The technique assumed a closed world, which made it less usable in a situation with precompiled libraries and frameworks. The work on type inference was later refined by Ole Agesen for Self [6, 7].

In 1997, Kresten Krab Throrup [152] published a paper on how to integrate virtual classes with Java. This was the starting point for a number of papers on virtual classes. In addition to Thorup, the work of Erik Ernst [39, 40], and Mads Torgersen [154] has been very decisive for interest in virtual classes. Several other researchers have elaborated on or been inspired by the virtual class concept, including work by Bruce, Odersky and Wadler [19] and Igarashi and Pierce [61]. Ernst has pointed out that some authors use the term virtual type whereas he prefers (and we agree) the term virtual class. A virtual type may (only) be used to annotate variables whereas a virtual class may be used to create instances.

Erik Ernst has developed the language **gbeta,** which is a further generalization of the abstraction mechanisms of BETA. **gbeta** among others includes a type-safe dynamic inheritance mechanism [39]. **gbeta** also supports the use of virtual patterns as superpatterns. BETA did have a semantics for virtual patterns as superpatterns, and virtual super patterns were implemented in the first BETA compiler. They were, however, abandoned in later versions, since we never found a satisfactory efficient implementation. In **gbeta** the restrictions on virtual superpatterns are removed. In BETA it is possible to express a simple kind of dependent types by means of block-structure and virtual classes. This was identified and generalized by Ernst as the concept of family polymorphism [40]. The connection to existential types mentioned above builds on this notion of dependent types.

In order to have full static type checking, Torgersen has suggested forbidding invocation of methods with parameters that have a non-final virtual type [154]. For classes with such methods, a concrete subclass with all virtual types declared final must then be defined in order to invoke these methods.

The Scala language has *abstract type members,* which are closely related to virtual classes. Finally, the language Caesar [126] supports the notion of gbeta virtual classes in a Java context with some simplifications and restrictions.

At POPL'2006 [41], Erik Ernst, Klaus Ostermann, and William R. Cook presented a virtual class calculus that captures the essence of virtual classes. We think this is an important milestone because it is the first formal calculus with a type system and a soundness proof which directly and faithfully models virtual classes.

Ellen Agerbo and Aino Cornils [3] used virtual classes and part objects to describe some of the design patterns in The Gang of Four book [43].

In 1996, Søren Brandt and Jørgen Lindskov Knudsen made a proposal for generalizing the BETA type system [16]. The proposal generalizes the type system in two directions: first, by allowing type expressions that do not uniquely denote a class, but instead denote a closely related set of classes, and second, by allowing types that cannot be interpreted as predicates on classes, but must be more generally interpreted as predicates on objects. The resulting increase in expressive power serves to further narrow the gap between statically and dynamically typed languages, adding among other things more general generics, immutable references, and attributes with types not known until runtime.

Knudsen has made use of the BETA fragment system to support aspect-oriented programming [74].

Goldberg, Findler and Flatt [44] developed a language with both super and inner, arguing that programmers need both kinds of method combination. They also present a formal semantics for the new language, and they describe an implementation for MzScheme.

**GOODS.** Nygaard was the leader of General Object-Oriented Distributed Systems (GOODS), a three-year Norwegian Research Council-supported project starting in 1997. The aim of the project was to enrich object-oriented languages and system development methods by new basic concepts that make it possible to describe the relation between layered and/or distributed programs and the machine executing these programs. BETA was used as the foundation for the project and language mechanisms in BETA were studied, especially supporting the theatre ensemble metaphor. The GOODS team also included Haakon Bryhni, Dag Solberg and Ole Smørdal.

**STAGE**. The GOODS project continued in the STAGE Project (STAGing Environments) project at the NCC, aiming at establishing a commercial implementation of the GOODS idea. The STAGE team also included Dag Belsnes, Jon Skretting, and Kasper Østerbye. The project pursued the idea of the theater metaphor – cf. Section 5.8.7.

**The Devise project**. In 1990 three research groups at DAIMI decided to work together on research in tools, techniques, methods and theories for experimental system development. The groups were Coloured Petri Nets (headed by Kurt Jensen), systems work (HCI) (headed by Morten Kyng) and object-oriented programming (headed by Madsen). The rationale was that in order to make progress in system development, supplementary competences were needed. The implications for BETA were:

BETA was used as a common language for development of tools. One major example is the CPN Tool [28] for editing, simulating and analyzing Coloured Petri Nets. A unique characteristic of CPN Tools is that they were one of the first tools to use so-called post-WIMP interaction techniques, including tool glasses, marking menus, and bimanual interaction (using two mice). CPN Tools is in widespread use. Another major tool was a Dexter-based hypermedia [48], [47], [143]. A unique characteristic of this tool was the use of anchors that makes it possible to link between positions in different pages without modifying the pages. The hypermedia tool was the basis for a start-up company, Hypergenic Ltd.

BETA has played an important role in work on a multidisciplinary approach to experimental system development. Over the years the group developed techniques for people within programming, system development, participatory design, HCI and ethnography to work together on software development projects, often using BETA and the Mjølner System. The object-oriented conceptual framework turned out to be a common framework and the graphical syntax of BETA supported by the Mjølner Tool turned out to be a useful means for communication between system developers and (expert) users [24].

From the beginning it was a goal to integrate Petri nets and object-oriented programming languages. The motivation was that in the early days of the BETA project Petri nets had a major influence on our conception of concurrency. Jensen, Kyng and Madsen started working together in formalizing DELTA using Petri nets. Jensen continued working with Petri nets and the group at DAIMI is well known internationally. Numerous suggestions for integrating object-orientation and Petri nets were investigated, but no real breakthrough was obtained. There are many suggestions in the literature for integrating Petri nets and object orientation, [108, 123].

The Devise group has continued to work together and now forms the basis of the Center for Pervasive Computing in Aarhus.

**Conceptual Modeling and Programming**. Design of programming languages could be based on human conceptualization in a more general sense. The approach was to include alternative kinds of concepts and selected ingredients of these concepts into programming languages in order to support modeling. The approach is described in [64, 104] and explored further in [124]. Object orientation could be seen as a specialized use of this approach, where the focus mainly is on "things" and their modeling in terms of classes and objects. The intention was that certain additional kinds of general (but not application area specific) concepts would enrich programming languages. The purpose was to limit the gap between understanding, designing and programming also in order to reduce the amount of software. The advantage of the approach is that because humans already use various alternative kinds of concepts, the modeling process is efficient and the model becomes understandable. The challenge was that any given potential kind of concept had to be understood and interpreted, and did not immediately comply with the typical understanding of programming languages. Each candidate concept should therefore be adjusted to fit with and slightly modify the expectations and possibilities at the programming level including implementation techniques. Candidate concepts include:

- Activities [81, 82, 103] are abstractions over collaborations of objects.
- Complex associations [83] are abstractions over complex relationships between structured objects.

- Roles [84, 105] are abstractions over the use of roles for objects as special relationships between objects.
- Relations [164, 166] are abstractions over relationships between objects.
- Subjective behavior [85] means abstraction over different views on objects from external and internal perspectives.
- Associations [86-88] are abstractions over collaboration, and include both structural and interaction aspects by integrating activity and role concepts.

## 7.3 Impact on language development

**Object-oriented SDL**. In 1986 Elektrisk Bureau (later ABB) asked Dag Belsnes and Møller-Pedersen to develop an object-oriented specification language. At the start the idea was to make this from scratch, but the project soon turned into an extension ([127], [128]) of the specification language SDL standardized by ITU – the International Telecommunication Union. BETA had an impact in the sense that concurrent processes of SDL became the candidate objects, in addition to the data objects that were also part of SDL. Users of SDL were primarily using processes, and as BETA had concurrent objects (and thereby patterns/subpatterns of these), it was obvious to do the same with SDL. The underlying model of SDL is that of a SDL system consisting of sets of nonsynchronized communicating processes, where the behavior of each process is described by a state machine. Introducing object orientation to this model implied the introduction of process types (in addition to sets of processes) and process subtypes defining specialization of state machine behavior. The inner concept was generalized to virtual transitions, i.e. transitions of a process type that may be redefined in process subtypes. In addition, the notion of virtual procedures was introduced, enabling parts of transitions to be redefined. In addition to constraints on virtual procedures, SDL also introduced default bindings. Virtual types (corresponding to virtual inner classes) were introduced, with constraints, both in terms of a supertype (as in BETA) and by means of a signature. In [21] it is demonstrated how this may be used to define frameworks; the same idea is pursued in [167]. Finally, types were extended with context parameters, a kind of generalized generic parameters, where also the constraints on the type parameters followed the BETA style of constraining. All of these extensions were standardized in the 1992 version of SDL [136].

**Java**. We do not claim that BETA had a major impact on Java, but as a curiosum we could mention that the two first commercial licenses of the Mjølner BETA System were acquired by James Gosling and Bill Joy.

Madsen was a visiting scientist a SUN Labs in 1994-95 when Java appeared on the scene – he was involved in discussions on whether or not virtual types could be added to Java. However, this was never done.

Java includes final bindings and singular objects – called anonymous classes. Nested classes were later added to Java and called inner classes. As we understand, final bindings, anonymous classes and nested classes were inspired by BETA.

The recently added Wildcard mechanism [155] was developed by a research group at DAIMI based on research by Mads Torgersen, Kresten Krab Thorup, Erik Ernst and others and may be traced back to virtual patterns.

**UML2.0**. Shortly after Møller-Pedersen joined Ericsson in 1998, a number of UML users (including Ericsson) asked for a new and improved version of UML. On behalf of Ericsson Møller-Pedersen joined this work within OMG. The influence on UML2.0 was indirectly via SDL, i.e. the same kinds of concepts as in SDL were introduced in UML2.0 [50]. As an interesting observation, UML1.x had already classes with their own behavior, like in SIMULA and BETA, while (as mentioned above) most object programming languages do not have this. UML1.x also had nested classes, so the only new thing in UML2.0 is that they can be redefinable (i.e. virtual classes).

# 8. Conclusion

The BETA project has been an almost lifelong enterprise involving the authors, the late Kristen Nygaard and many other people. The approach to language design and informatics has been unusual compared to most other language projects we are aware of. The main reason is perhaps the emphasis on modeling, the working style, and the unusual organization of the project.

The project was supposed to be organized in a well-defined manner based on partners, contracts/grants and a firm working plan with milestones including a language specification in 1997. Since we did not succeeded in obtaining this, the project continued for many years as an informal collaboration among the team members. If we had delivered a language specification in 1997 it would have been quite different from what BETA is today and probably less interesting. A project with firm deadlines and a firm budget might not have achieved the same result. Instead we were able to continue to invent, discuss, and reject ideas over many iterations. We could keep parts open where we did not have satisfactory solutions. It was never too late to come up with a complete new idea. We could continue to strive for the perfect language.

From 1986 when the Mjølner projects started, there was an organization around BETA – although Mjølner was not supposed to develop the BETA language. We had to finalize the language and make decisions for the parts that

were not complete and even make decisions we were not happy about.

The "one abstraction mechanism" idea was an important driving factor, but it may not have been unusual to base a language project on one or more initial ideas. In fact, one should never engage in language design without overall major ideas. Languages based on the current state of art may be well engineered but will not add to the state of the art. Such languages may be highly influential on praxis and we have seen many examples of that.

As time has passed, many new ideas for improving BETA have been proposed and new challenges have appeared. But for many years we found that most of the proposals would not make a real difference for the users of BETA. The work on updating the language, the documentation and software was simply not worth the effort. The time has, however, arrived for a new language in the SIMULA/BETA style, but the one or two real breaking ideas perhaps remain to be seen.

Nygaard's system description (modeling) approach was an unusual approach to language design. Designing a programming language from a system description perspective is certainly different from basing it on whatever a computer can do or on a mathematical foundation.

Another unusual characteristic of the project was that we did not follow mainstream research in programming languages. As mentioned, Nygaard was not interested in the state of the art but left it to us. The advantage of this approach was that we were free to formulate new visions and not just focus on the next publication. Today most researchers seem mainly to focus on publishing minor improvements and solutions to state-of-the-art ideas. This does not create new big inventions.

The BETA project heavily influenced the participants and their relationships. We established lifelong valuable and appreciated personal and professional relationships. Being young and inexperienced researchers learning from working together with such an experienced person as Nygaard, many of our research attitudes were established during the project. The most valuable has been not to take established solutions for given, but rather question them, try to go for more general solutions, and to have alternative, ambitious, and long-reaching objectives.

Below we comment on the original research goals of the project.

**One abstraction mechanism**. We succeeded in developing the pattern as an abstraction mechanism subsuming most other abstraction mechanisms. Originally this was a theoretical challenge and we think that the pattern mechanism has proved its relevance and importance from a research perspective. The pattern mechanism has also proved to be useful in teaching and practical programming. As a teaching tool it is beneficial to teach students the pattern mechanism as part of their first programming language, but probably only with success if the approach is supported strongly by the learning environment. Still, in order to appreciate the beauty of the pattern mechanism ,the student has also to be familiar with the culture of the programming-language world including notions such as record, procedure, etc. Such cultural variations need to be appreciated before the unified, more abstract notion is relevant and appealing. For the skilled programmer who has already used several different programming languages, the presentation of the pattern mechanism seems to be a very fruitful experience. Such programmers typically learn yet another abstract level of programming and this knowledge is valuable through the daily life with the usual ordinary programming languages. Programmers with the opportunity to use the pattern for a longer period for real system development appreciate the freedom and powerfulness it supports.

The idea of one pattern replacing all other abstraction mechanisms worked out well in practice. The unification clearly implied a simplification of the language, just as the extra benefits as mentioned in Section 5.1.2 clearly paid off. We occasionally hear people complain that they find it to be a disadvantage that they cannot see from a pattern declaration whether it is a class or method.

Virtual patterns turned out to be a major strength of BETA – the use of virtual patterns as virtual classes/types has in addition provided the basis for further research by many others.

Singular objects, block structure, etc. have also proved their value in practice and are heavily used by all BETA programmers. These mechanisms are also starting to arrive in other languages.

The enter-exit mechanism is of course used for defining parameters and return values for methods – in addition, it is used for defining value types. Many people make heavy use of enter/exit for overloading assignment and/or reading the value of an object. Although the enter/exit-mechanism has turned out to be quite useful in practice, it does have some drawbacks. The name of an argument has to be declared twice – once with a type and then in the enter/exit-part – this is similar to Algol and SIMULA but is, however inconvenient for simple parameters. In addition, the implementation of enter/exit in its full generality turned out to be quite complex.

A constructor mechanism is perhaps the most profound language element that is missing in BETA.

**Coroutines and concurrency**. We think that BETA has further demonstrated the usefulness of the SIMULA coroutine mechanism to support concurrency and

scheduling. The coroutine mechanism together with the semaphore turned out to fulfill the original goals. The implementation was simple and straightforward, and it has showed its usefulness in practice.

In addition, the abstraction mechanisms of BETA have proved their usefulness in defining higher-order abstraction mechanisms. The BETA libraries contain several examples of high-level concurrency abstractions. Few people in practice, however, define their own concurrency abstractions. Most concurrency abstractions have been defined by the authors and implementers of BETA.

In general concurrency and the ability to define concurrency abstractions are not as heavily used as we think they should be. This may it may be due to the fact that concurrency has not been an integrated part of most object-oriented languages. Java has concurrency but as a fixed synchronization mechanism in the form of monitor – there are no means for defining other concurrency abstractions including schedulers. We think that it should be an integrated part of the design of frameworks and components also to define the associated concurrency abstractions including schedulers.

We also think that SIMULA/BETA style coroutines are yet to be discovered by other language designers.

**Efficiency**. The original goal of proving that an object-oriented programming language could be efficiently implemented turned out to be less important. Several successors to SIMULA starting with C++ proved this. In addition, a number of efficient implementation techniques and more efficient microprocessors have implied that lack of efficiency is hardly an issue anymore.

**Modeling**. The modeling approach succeeded in the sense that a comprehensive conceptual framework has been developed. The conceptual framework consists of a collection of conceptual means for understanding and organizing knowledge about the real world. It is furthermore described how these means are related to programming language constructs. But just as important, it is emphasized that some conceptual means are not supported by BETA and other programming languages. As mentioned previously, we think that it is necessary for software developers to be aware of a richer conceptual framework than that supported by a given language. Otherwise the programming language easily limits the ability of the programmer to understand the application domain. A conceptual framework that is richer than current programming languages can be used to define requirements for new programming languages. This leads to the other point where we think that the modeling approach has succeeded.

We have demonstrated that language constructs and indeed a whole language can be based on a modeling approach. As we hope we have demonstrated in this paper, almost all constructs in BETA are motivated by their ability to model properties of the application domain. They also had to have properties from a technical point of view and to be sufficiently primitive in order to be efficiently implemented. The art of designing a programming language is to balance the support of conceptual means and selection of primitives that may be efficiently implemented. We did e.g. not include dynamic classification and equations since we did not find that we could implement such constructs efficiently.

The goal for BETA was to design a language that could be used for modeling as well as programming. For many years the programming language community was not interested in modeling, and when object-orientation started to become popular, the main focus was on extensibility and reuse of code. This changed when the methodology schools started to become interested in object-oriented analysis and design. The approach to modeling in these schools was, however, different from ours. Most work on modeling aimed at designing special modeling languages based on a graphical syntax. As mentioned in Section 4, this reintroduced some of the problems of code generation and reverse engineering known from SA/SD. For BETA it was important to stress that the same language can be used for modeling as well as for programming and that syntax is independent of this. This was stressed by the fact that we designed both a textual and a graphical syntax for BETA. The attempts in recent years to design executable modeling languages in our opinion emphasizes that it was not a good idea to have separate modeling and programming languages.

There is no doubt that object orientation has become the mainstream programming paradigm. There are hundreds (or thousands) of books introducing object-oriented programming and methodologies based on object orientation. The negative side of this is that the modeling aspect that originated with SIMULA seems to be disappearing. Very few schools and books are explicit about modeling. It is usually restricted to a few remarks in the introduction; the rest of the book is then concerned with technical aspects of a programming language or UML or traditional software methodology.

We think that some of the advantages of object orientation have disappeared in its success and that there might be a need for proper reintroduction of the original concepts. OOA and OOD are in most schools nothing more than just programming at a high level of abstraction corresponding to the application domain. In order to put more content into this, there is room for making more use of the parts of the conceptual framework of BETA that go beyond what is supported by current programming languages. This would improve the quality of the analysis and design phases. We also think that future languages should be designed for

modeling as well as programming. Turning a modeling language into a programming language (or vice versa) may not be the best approach.

## 9. Acknowledgments

The Joint Language Project included at least: Bjarner Svejgaard, Leif Nielsen, Erik Bugge, Morten Kyng, Benedict Løfstedt, Jens Ulrik Mouritsen, Peter Jensen, Nygaard, Kristensen, and Madsen.

In addition to Nygaard and the authors, a large number of people have been involved in the development of BETA and the Mjølner BETA System including colleagues and students at Aarhus University, The Norwegian Computing Center, Oslo University, Aalborg University, and Mjølner Informatics A/S. In particular, the contributions of Dag Belsnes and Jørgen Lindskov Knudsen are acknowledged.

Development of the Mjølner BETA System in the Nordic Mjølner Project has been supported by the Nordic Fund for Technology and Research. Participants in the Mjølner Project came from Lund University, Telesoft, EB Technology, The Norwegian Computing Center, Telenokia, Sysware ApS, Aalborg University and Aarhus University. The main implementers of the Mjølner BETA System were Peter Andersen, Lars Bak, Søren Brandt, Jørgen Lindskov Knudsen, Henry Michael Lassen, Ole Lehrmann Madsen, Kim Jensen Møller, Claus Nørgaard, and Elmer Sandvad. In addition, Peter von der Ahe, Knut Barra, Bjorn Freeman-Benson, Karen Borup, Michael Christensen, Steffen Grarup, Morten Grouleff, Mads Brøgger Enevoldsen, Søren Smidt Hansen, Morten Elmer Jørgensen, Kim Falck Jørgensen, Karsten Strandgaard Jørgensen, Stephan Korsholm, Manmathan Muthukumarapillai, Peter Ryberg Jensen, Jørgen Nørgaard, Claus H. Pedersen, Jacob Seligmann, Lennert Sloth, Tommy Thorn, Per Fack and Peter Ørbæk have participated in various parts of the implementation.

In addition to the people mentioned above, we have had the pleasure of being engaged in research in object-orientation including issues related to BETA with a large number of people including Boris Magnusson, Görel Hedin, Erik Ernst, Henrik Bærbak Christensen, Mads Torgersen, Kresten Krab Thorup, Kasper Østerbye, Aino Cornils, Kristine S. Thomsen, Christian Damm, Klaus Marius Hansen, Michael Thomsen, Jawahar Malhotra, Preben Mogensen, Kaj Grønbæk, Randy Trigg, Dave Ungar, Randy Smith, Urs Hölzle, Mario Wolczko, Ole Agesen, Svend Frølund, Michael H. Olsen, and Alexandre Valente Sousa.

The writing of this paper has happened during a period of nearly two years. We would like to thank the HOPL-III Program Committee for their strong support and numerous suggestions. Our shepherds Julia Lawall and Doug Lea have been of great help and commented on everything including content, structure and our English language style. The external reviewers, Andrew Black, Gilad Bracha, Ole Agesen, and Koen Classen, have also provided us with detailed and helpful comments. Finally, we have received suggestions and comments from Peter Andersen, Jørgen Lindskov Knudsen, Stein Krogdahl, and Kasper Østerbye. We are grateful to Susanne Brøndberg who helped correct our English and to Katrina Avery for the final copy editing.

The development of BETA has been supported by the Danish Research Council, The Royal Norwegian Council for Scientific and Industrial Research, The European Commission, The Nordic Fund for Technology and Research, Apple Computer, Apollo, Hewlett Packard, Sun Microsystems, Microsoft Denmark, and Microsoft Research Cambridge.

As said in Section 3, the BETA project have had a major influence on our personal life and our families have been deeply involved in the social interaction around the project. We thank from the Kristensen family: Lis, Unna, and Eline; from the Madsen family: Marianne, Christian, and Anne Sofie; from the Møller-Pedersen family: Kirsten, Kamilla, and Kristine; and Johanna Nygaard. When the project started Christian had just been born – by now we are all grandparents.

## 10. References

[1] *Webster's New World Compact School and Office Dictionary*: Prentice Hall Press, 1982.

[2] Ada: *Ada Reference Manual. Proposed Standard Document*: United States Department of Defense, 1980.

[3] Agerbo, E. and Cornils, A.: *How to Preserve the Benefits of Design Patterns*, OOPSLA'98 – ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, 1998, Sigplan Notices ACM Press.

[4] Agesen, O., Frølund, S., and Olsen, M. H.: Persistent and Shared Objects in BETA, Master thesis, Computer Science Department, Aarhus University, Aarhus 1989.

[5] Agesen, O., Frølund, S., and Olsen, M. H.: *Persistent Object Concepts*, in *Object-Oriented Environments—The Mjølner Approach*, Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B., Eds.: Prentice Hall, 1994.

[6] Agesen, O., Palsberg, J., and Schwartzbach, M. I.: *Type Inference of SELF*, ECOOP'93 – European Conference on Object-Oriented Programming, Kaiserslautern, 1993, Lecture Notes in Computer Science vol. 707, Springer.

[7] Agesen, O. and Ungar, D.: *Sifting Out the Gold*, OOPSLA'94 – Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1994, Sigplan Notices vol. 29, ACM Press.

[8] America, P.: *Inheritance and Subtyping in a Parallel Object-Oriented Language*, ECOOP'87 – European Conference on Object-Oriented Programming, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.

[9] Andersen, P. and Madsen, O. L.: *Implementing BETA on Java Virtual Machine and .NET—an Exercise in Language Interoperability*, unpublished manuscript, 2003.

[10] Barra, K.: *Mark/sweep Compaction for Substantially Nested Beta Objects*, Norwegian Computing Center, Oslo, DTEK/03/88, 1988.

[11] Belsnes, D.: *Description and Execution of Distributed Systems*, Norwegian Computing Center, Oslo, Report No 717, 1982.

[12] Blake, E. and Cook, S.: *On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*, ECOOP'87 – European Conference on Object-Oriented Programming, Paris, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.

[13] Bobrow, D. G. and Stefik, M.: *The LOOPS Manual*, Palo Alto AC: Xerox Corporation, 1986.

[14] Booch, G.: *Object-Oriented Analysis and Design with Applications*. Redwood City: Benjamin/Cummings, 1991.

[15] Bracha, G. and Cook, W.: *Mixin-based Inheritance*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming: Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, Sigplan Notices vol. 25, ACM Press

[16] Brandt, S. and Knudsen, J. L.: *Generalising the BETA Type System*, ECOOP'96 – Tenth European Conference on Object-Oriented Programming Linz , Austria, 1996, Springer Verlag.

[17] Brinch-Hansen, P.: *The Programming Language Concurrent Pascal*, IEEE Transactions on Software Engineering, vol. SE-1(2), 1975.

[18] Brinch-Hansen, P.: *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*: Springer, 2002.

[19] Bruce, K., Odersky, M., and Wadler, P.: *A Statically Safe Alternative to Virtual Types*, ECOOP'98 – European Conference on Object-Oriented Programming, Brussels, 1998, Lecture Notes in Computer Science vol. 1445, Springer Verlag.

[20] Bruce, K. and Vanderwaart, J. C.: *Semantics-Driven Language Design: Statically Type-safe Virtual Types in Object-Oriented Languages*, Fifteenth Conference on the Mathematical Foundations of Programming Semantics, 1998.

[21] Bræk, R. and Møller-Pedersen, B.: *Frameworks by Means of Virtual Types—Exemplified by SDL*, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII), 1998.

[22] Budd, T.: *An Introduction to Object-Oriented Programming, third edition*: Addison Wesley, 2002.

[23] Cannon, H.: *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming*, Symbolics Inc., 1982.

[24] Christensen, M., Crabtree, A., Damm, C. H., Hansen, K. M., Madsen, O. L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., and Thomsen, M.: *The M.A.D Experience: Multi-Perspective Application Development in Evolutionary Prototyping*, ECOOP'98 – European Conference on Object-Oriented Programming, Brussels, 1998, Lecture Notes in Computer Science vol. 1445, Springer Verlag.

[25] Coad, P. and Yourdon, E.: *Object-Oriented Analysis*. Englewood Cliffs, N.J: Prentice-Hall, 1991.

[26] Cook, S.: *Impressions of ECOOP'88*, Journal of Object-Oriented Programming, vol. 1, 1988.

[27] Cook, W.: *Peek Objects*, ECOOP'2006 – European Conference on Object-Oriented Programming, Nantes, France, 2006, Lecture Notes in Computer Science vol. 4067, Springer Verlag.

[28] CPNTOOLS: *Computer Tool for Coloured Petri Nets*.

[29] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R.: *Structured Programming*: Academic Press, 1972.

[30] Dahl, O.-J. and Hoare, C. A. R.: *Hierarchical Program Structures*, in *Structured Programming*: Academic Press, 1972.

[31] Dahl, O.-J. and Myhrhaug, B.: *SIMULA 67 Implementation Guide*, Norwegian Computing Center, Oslo, NCC Publ. No. S-9, 1969.

[32] Dahl, O.-J., Myhrhaug, B., and Nygaard, K.: *SIMULA 67 Common Base Language (Editions 1968, 1970, 1972, 1984)*, Norwegian Computing Center, Oslo, 1968.

[33] Dahl, O.-J. and Nygaard, K.: *SIMULA—a Language for Programming and Description of Discrete Event Systems*, Norwegian Computing Center, Oslo, 1965.

[34] Dahl, O.-J. and Nygaard, K.: *SIMULA: an ALGOL-based Simulation Language*, Communications of the ACM, vol. 9, pp. 671–678, 1966.

[35] Dahl, O.-J. and Nygaard, K.: *The Development of the SIMULA Languages*, ACM SIGPLAN History of Programming Languages Conference, 1978.

[36] Dijkstra, E. W.: *Go To Considered Harmful*, Letter to Communications of the ACM, vol. 11 pp. 147–148, 1968.

[37] Dijkstra, E. W.: *Guarded Commands, Nondeterminacy and the Formal Derivation of Programs*, Communications of the ACM, vol. 18, pp. 453–457, 1975.

[38] Doberkat, E.-E. and Diβmann, S.: *Einführung in die objektorienterte Programmierung mit BETA*: Addison Wesley Longman Verlag GmbH, 1996.

[39] Ernst, E.: *Dynamic Inheritance in a Statically Typed Language*, Nordic Journal of Computing, vol. 6, pp. 72–92, 1999.

[40] Ernst, E.: *Family Polymorphism*, ECOOP'01 – European Conference on Object-Oriented Programming, Budapest, Hungary, 2001, Lecture Notes in Computer Science vol. 2072, Springer Verlag.

[41] Ernst, E., Ostermann, K., and Cook, W. R.: *A Virtual Class Calculus*, The 33rd Annual ACM SIGPLAN – SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, 2006.

[42] Folke Larsen, S.: *Egocentrisk tale, begrebsudvikling og semantisk udvikling.*, Nordisk Psykologi, vol. 32(1), 1980.

[43] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[44] Goldberg, D. S., Findler, R. B., and Flatt, M.: *Super and Inner—Together at Last!*, OOPSLA'04 – Object-Oriented Programming, Systems Languages and Applications, Vancouver, British Columbia, Canada., 2004, SIGPLAN Notices vol. 39, ACM.

[45] Goodenough, J. B.: *Exception Handling: Issues and a Proposed Notation*, Communications of the ACM, vol. 18, pp. 683—696, 1975.

[46] Gosling, J., Joy, B., and Steele, G.: *The Java (TM) Language Specification*: Addison-Wesley, 1999.

[47] Grønbæk, K. and Trigg, R. H.: *Design Issues for a Dexter-based Hypermedia System*, Communications of the ACM, vol. 37, pp. 40–49, 1994.

[48] Halasz, F. and Schwartz, M.: *The Dexter Hypertext Reference Model*, Communications of the ACM, vol. 37, pp. 30–39, 1994.

[49] Haugen, Ø.: Hierarkibegreber i Programmering og Systembeskrivelse (Hierarchy Concepts in Programming and System Description), Master thesis, Department of Informatics, University of Oslo, Oslo 1980.

[50] Haugen, Ø., Møller-Pedersen, B., and Weigert, T.: *Structural Modeling with UML 2.0*, in *UML for Real*, Lavagno, L., Martin, G., and Selic, B., Eds.: Kluwer Academic Publishers, 2003.

[51] Hejlsberg, A., Wiltamuth, S., and Golde, P.: *The C# Programming Language*: Addison-Wesley, 2003.

[52] Hoare, C. A. R.: *Record Handling*, ALGOL Bulletin, 1965.

[53] Hoare, C. A. R.: *Further Thoughts on Record Handling*, ALGOL Bulletin, 1966.

[54] Hoare, C. A. R.: *Record Handling—Lecture Notes, NATO Summer School September 1966*, in *Programming Languages*, Genuys, Ed.: Academic Press 1968, 1966.

[55] Hoare, C. A. R.: *Notes on Data Structuring*, in *Structured Programming*. London: Academic Press, 1972.

[56] Hoare, C. A. R.: *Proof of Correctness of Data Representations*, Acta Informatica, vol. 1, 1972.

[57] Hoare, C. A. R.: *Monitors: An Operating Systems Structuring Concept*, Communications of the ACM, 1975.

[58] Hoare, C. A. R.: *Communicating Sequential Processes*, Communications of the ACM, vol. 21, 1978.

[59] Hoare, C. A. R.: *Turing Award 1980 Lecture: "The Emperor's Old Clothes"*, Communications of the ACM, vol. 24, pp. 75–83, 1981.

[60] Holbæk-Hanssen, E., Håndlykken, P., and Nygaard, K.: *System Description and the DELTA Language*, Norwegian Computing Center, Oslo, Report No 523, 1973.

[61] Igarashi, A. and Pierce, B. C.: *Foundations for Virtual Types*, The Sixth International Workshop on Foundations of Object-Oriented Languages – FOOL 6, San Antonio, Texas, 1999.

[62] Igarashi, A. and Pierce, B. C.: *On Inner classes*, Information and Computation, vol. 177, 2002. A special issue with papers from the 7th International Workshop on Foundations of Object-Oriented Languages (FOOL7). An earlier version appeared in Proceedings of ECOOP2000 – 14th European Conference on Object-Oriented Programming, Springer LNCS 1850, pages 129–153, June, 2000.

[63] Jackson, M.: *System Development*: Prentice-Hall, 1983.

[64] Jacobsen, E. E.: Concepts and Language Mechanisms in Software Modelling, PhD thesis, University of Southern Denmark, 2000.

[65] Jensen, K., Kyng, M., and Madsen, O. L.: *A Petri Net Definition of a System Description Language*, in *Semantics of Concurrent Computations, Evian*, Kahn, G., Ed.: Springer Verlag, 1979.

[66] Jensen, P. and Nygaard, K.: *The BETA Project*, Norwegian Computing Center, Oslo, Publication No 558, 1976.

[67] Joy, B.: *Personal Communication*. Sun Microsystems, 1994.

[68] Kay, A.: *Microelectronics and the Personal Computer*, Scientific America, vol. 237(3), pp. 230–244, 1977.

[69] Keene, S. E.: *Object-Oriented Programming in COMMON Lisp—a Programmer's Guide to CLOS*: Reading MA: Addison Wesley 1989.

[70] Knudsen, J. L.: *Implementing BETA Communication, a Proposal*, unpublished manuscript, 1981.

[71] Knudsen, J. L.: Aspects of Programming Languages: Concepts, Models and Design, thesis, Department of Computer Science, University of Aarhus, Aarhus 1982.

[72] Knudsen, J. L.: *Exception Handling—A Static Approach*, Software Practice and Experience, 1984.

[73] Knudsen, J. L.: *Name Collision in Multiple Classification Hierarchies*, ECOOP'88 – 2nd European Conference on Object-Oriented Programming, Oslo, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.

[74] Knudsen, J. L.: *Aspect-Oriented Programming in BETA Using the Fragment System*, Workshop on Object-Oriented Technology, 1999, Lecture Notes In Computer Science vol. 1743.

[75] Knudsen, J. L.: *Fault Tolerance and Exception Handling in BETA*, in *Advances in Exception Handling Techniques*, vol. 2022, *Lecture Notes in Computer Science*, Romanovsky, A., Dony, C., Knudsen, J. L., and Tripathi, A., Eds.: Springer Verlag, 2001.

[76] Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B.: *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.

[77] Knudsen, J. L. and Madsen, O. L.: *Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages*, ECOOP'88 – European Conference on Object Oriented Programming, Oslo, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.

[78] Knudsen, J. L. and Thomsen, K. S.: A Conceptual Framework for Programming Languages, Master thesis, Computer Science Department, Aarhus University, Aarhus 1985.

[79] Koster, C. H. A.: *Visibility and Types*, SIGPLAN 1976 Conference on Data, Salt Lake City, Utah, USA, 1976.

[80] Kreutzer, W. and Østerbye, K.: *BetaSIM—a Framework for Discrete Event Modelling and Simulation*, Simulation Practice and Theory, vol. 6, pp. 573–599, 1998.

[81] Kristensen, B. B.: *Transverse Activities: Abstractions in Object-Oriented Programming*, International Symposium on Object Technologies for Advanced Software (ISOTAS'93), Kanazawa, Japan, 1993.

[82] Kristensen, B. B.: *Transverse Classes & Objects in Object-Oriented Analysis, Design and Implementation*, Journal of Object-Oriented Programming, 1993.

[83] Kristensen, B. B.: *Complex Associations: Abstractions in Object-Oriented Modeling*, OOPSLA'94 – Object-Oriented Programming Systems, Languages and Applications, Portland, Oregon, 1994, Sigplan Notices vol. 29(10), ACM Press.

[84] Kristensen, B. B.: *Object-Oriented Modeling with Roles*, 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Dublin, Ireland, 1995.

[85] Kristensen, B. B.: *Subjective Behavior*, International Journal of Computer Systems Science and Engineering, vol. 16, pp. 13–24, 2001.

[86] Kristensen, B. B.: *Associative Modeling and Programming*, 8th International Conference on Object-Oriented Information Systems (OOIS'2002), Montpellier, France, 2002.

[87] Kristensen, B. B.: *Associations: Abstractions over Collaboration*, IEEE International Conference on Systems, Man & Cybernetics, Washington D. C., 2003.

[88] Kristensen, B. B.: *Associative Programming and Modeling: Abstractions over Collaboration*, 1st International Conference on Software and Data Technologies, Setúbal, Portugal, 2006.

[89] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Language Development, Survey Report, 1. November 1976*, Norwegian Computing Center, Oslo, Report No 559, 1977.

[90] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *DRAFT PROPSAL for Introduction to The BETA Programming Language as of 1st August 1978*, Norwegian Computing Center, Oslo, Report No 620, 1978.

[91] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Examples (Corresponding to the BETA Language Proposal as of August 1979)*, DAIMI IR-15, 1979.

[92] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *BETA Language Proposal as of April 1979*, Norwegian Computing Center, Oslo, NCC Report No 635, 1979.

[93] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *A Survey of the BETA Programming Language*, Norwegian Computing Center, Oslo, Report No 698, 1981.

[94] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Syntax Directed Program Modularization*, European Conference on Integrated Interactive Computing Systems, ECICS 82, Stresa, Italy, 1982, North-Holland.

[95] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Abstraction Mechanisms in the BETA Programming Language*, Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983.

[96] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *An Algebra for Program Fragments*, ACM SIGPLAN Symposium on Programming Languages and Programming Environments, Seattle, Washington, 1985.

[97] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Multisequential Execution in the BETA Programming Language*, Sigplan Notices, vol. 20, 1985.

[98] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Specification and Implementation of Specialized Languages*, unpublished manuscript, 1985.

[99] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Dynamic Exchange of BETA Systems*, unpublished manuscript, 1986.

[100] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *The BETA Programming Language*, in *Research Directions in Object Oriented Programming*, Shriver, B. D. and Wegner, P., Eds.: MIT Press, 1987.

[101] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Classification of Actions or Inheritance Also for Methods*, ECOOP'87 – European Conference on Object-Oriented Programming, Paris, 1987, Lecture Notes in Computer Science vol. 276, Springer Verlag.

[102] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language—Draft*, 1989.

[103] Kristensen, B. B. and May, D. C. M.: *Activities: Abstractions for Collective Behavior*, ECOOP'96 – European Conference on Object-Oriented Programming, Linz, 1996, Lecture Notes in Computer Science vol. 1098, Springer Verlag.

[104] Kristensen, B. B. and Østerbye, K.: *Conceptual Modeling and Programming Languages* SIGPLAN Notices, vol. 29, 1994.

[105] Kristensen, B. B. and Østerbye, K.: *Roles: Conceptual Abstraction Theory and Practical Language Issues*, Theory and Practice of Object Systems (TAPOS), 1996, vol. 2(3).

[106] Krogdahl, S.: *On the Implementation of BETA*, Norwegian Computing Center, Oslo, 1979.

[107] Krogdahl, S.: *The Birth of Simula*, IFIP WG9.7 First Working Conference on the History of Nordic Computing (HiNC1), Trondheim, 2003, Springer.

[108] Lakos, C.: *The Object Orientation of Object Petri Nets*, First international workshop on Object-Oriented Programming and Models of Concurrency—16th International Conference on Application and Theory of Petri Nets, 1995.

[109] Liskov, B. H. and Zilles, S. N.: *Programming with Abstract Data Types*, ACM Sigplan Notices, vol. 9, 1974.

[110] MacLennan, B. J.: *Values and Objects in Programming Languages*, SIGPLAN Notices, vol. 17, 1982.

[111] Madsen, O. L.: *Block Structure and Object Oriented Languages*, in *Research Directions in Object Oriented Programming*, Shriver, B. D. and Wegner, P., Eds.: MIT Press, 1987.

[112] Madsen, O. L.: *The implementation of BETA*, in *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.

[113] Madsen, O. L.: *Open Issues in Object-Oriented Programming*, Software Practice & Experience, vol. 25, 1995.

[114] Madsen, O. L.: *Semantic Analysis of Virtual Classes and Nested Classes*, OOPSLA'99 – Object-Oriented Programming, Systems Languages and Applications, Denver, Colorado, 1999, Sigplan Notices vol. 34, ACM Press.

[115] Madsen, O. L., Magnusson, B., and Møller-Pedersen, B.: *Strong Typing of Object-Oriented Languages Revisited*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming, Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, Sigplan Notices vol. 25, ACM Press

[116] Madsen, O. L. and Møller-Pedersen, B.: *What Object-Oriented Programming May Be—and What It Does Not Have to Be*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.

[117] Madsen, O. L. and Møller-Pedersen, B.: *Virtual Classes—A Powerful Mechanism in Object-Oriented Programming*, OOPSLA'89 – Object-Oriented Programming, Systems Languages and Applications, New Orleans, Louisiana, 1989, Sigplan Notices vol. 24, ACM Press.

[118] Madsen, O. L. and Møller-Pedersen, B.: *Part Objects and Their Location*, TOOLS'92: Technology of Object-Oriented Languages and Systems, Dortmund, 1992, Prentice Hall.

[119] Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Object-Oriented Programming in the BETA Programming Language*: Addison Wesley, 1993.

[120] Madsen, O. L. and Nørgaard, C.: *An Object-Oriented Metaprogramming System*, Hawaii International Conference on System Sciences, Hawaii, 1988, CRC Press.

[121] Magnusson, B.: *Code Reuse Considered Harmful*, JOOP – Journal of Object-Oriented Programming, vol. 4, 1991.

[122] Magnusson, B.: *Simula Runtime System Overview*, in *Object-Oriented Environments—The Mjølner Approach*: Prentice Hall, 1993.

[123] Maier, C. and Moldt, D.: *Object Coloured Petri Nets–A Formal Technique for Object Oriented Modelling*, in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, vol. 2001/2001, *Lecture Notes in Computer Science*: Springer, 2001.

[124] May, D. C.-M., Kristensen, B. B., and Nowack, P.: *Conceptual Abstraction in Modeling with Physical and Informational Material*, in *Applications of Virtual Inhabited 3D Worlds*, Qvortrup, L., Ed.: Springer Press, 2004.

[125] Meyer, B.: *Eiffel: The Language*: Prentice Hall, 1992.

[126] Mezini, M. and Ostermann, K.: *Conquering Aspects with Caesar*, AOSD'03, Boston, USA, 2003.

[127] Møller-Pedersen, B., Belsnes, D., and Dahle, H. P.: *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL*, SDL Forum'87 – State of the Art and future Trends, 1987, North-Holland.

[128] Møller-Pedersen, B., Belsnes, D., and Dahle, H. P.: *Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL*, Computer Networks, vol. 13, 1987.

[129] Naur, P.: *Revised Report on The Algorithmic Language ALGOL 60*, Communications of the ACM, vol. 6, 1963.

[130] Nygaard, K.: *System Description by SIMULA—an Introduction*, Norwegian Computing Center, Oslo, S-35, 1970.

[131] Nygaard, K.: *GOODS to Appear on the Stage*, ECOOP'97 – European Conference on Object-Oriented Programming, Jyväskylä, Finnland, 1997, Lecture Notes in Computer Science vol. 1241, Springer Verlag.

[132] Nygaard, K.: *Ole-Johan Dahl*, in *Journal of Object Technology*, vol. 1, 2002.

[133] Odersky, M.: *The Scala Experiment—Can We Provide Better Language Support for Component Systems?*, Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina, 2006.

[134] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M.: *An Overview of the Scala Programming Language*, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, SwitzerlandIC/2004/64, 2004.

[135] Odersky, M. and Zenger, M.: *Scalable Component Abstractions*, OOPSLA 2005 – Object-Oriented Programming, Systems Languages and Applications, San Diego, CA, USA, 2005, Sigplan Notices ACM Press.

[136] Olsen, A., Færgemand, O., Møller-Pedersen, B., Reed, R., and Smith, J. R. W.: *Systems Engineering Using SDL-92*: North-Holland, 1994.

[137] Palme, J.: *New Feature for Module Protection in SIMULA*, Sigplan Notices, vol. 11, 1976.

[138] Palsberg, J. and Schwartzbach, M. I.: *Type Substitution for Object-Oriented Programming*, Joint OOPSLA/ECOOP'90 – Conference on Object-Oriented Programming: Systems, Languages, and Applications & European Conference on Object-Oriented Programming, Ottawa, Canada, 1990, ACM SIGPLAN Notices vol. 25, ACM Press New York.

[139] Palsberg, J. and Schwartzbach, M. I.: *Object-Oriented Type Inference*, OOPSLA'91 – Conference on Object-Oriented Programming: Systems, Languages, and Applications, Phoenix, Arizona, 1991, SIGPLAN Notices vol. 26, ACM Press.

[140] Palsberg, J. and Schwartzbach, M. I.: *Object-Oriented Type-Systems*: John Wiley & Sons, 1994.

[141] Sandvad, E.: *Object-Oriented Development—Integrating Analysis, Design and Implementation*, Computer Science Department, University of Aarhus, Aarhus, DAIMI PB-302, 1990.

[142] Sandvad, E.: *An Object-Oriented CASE Tool*, in *Object-Oriented Environments—The Mjølner Approach*, Knudsen, J. L., Löfgren, M., Madsen, O. L., and Magnusson, B., Eds.: Prentice Hall, 1994.

[143] Sandvad, E., Grønbæk, K., Sloth, L., and Knudsen, J. L.: *A Metro Map Metaphor for Guided Tours on the Web: the Webvise Guided Tour System*, The Tenth International World Wide Web Conference, Hong Kong, 2001.

[144] Seligmann, J. and Grarup, S.: *Incremental Mature Garbage Collection Using the Train Algorithm*, ECOOP'95 – European Conference on Object-Oriented Programming, Aarhus, Denmark, 1995, Lecture Notes in Computer Science vol. 952, Springer Verlag.

[145] Shriver, B. D. and Wegner, P.: *Research Directions in Object Oriented Programming*: MIT Press, 1987.

[146] Smith, B.: *Personal Communication*, Madsen, O. L., Ed. Stanford, 1984.

[147] Smith, J. M. and Smith, D. C. P.: *Database Abstraction: Aggregation and Generalization*, ACM TODS, vol. 2(2), 1977.

[148] Stroustrup, B.: *The C++ Programming Language*, 2000.

[149] Sørgaard, P.: *Object-Oriented Programming and Computerised Shared Material*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.

[150] Thomsen, K. S.: Multiple Inheritance, a Structuring Mechanism for Data, Processes and Procedures, Master thesis, Department of Computer Science, Aarhus University, Aarhus 1986.

[151] Thomsen, K. S.: *Inheritance on Processes, Exemplified on Distributed Termination Detection*, International Journal of Parallel Programming, vol. 16(1), pp. 17–52, 1987.

[152] Thorup, K. K.: *Genericity in Java with Virtual Types*, ECOOP'97 – European Conference on Object-Oriented Programming, Jyväskylä, Finland, 1997, Lecture Notes in Computer Science vol. 1241, Springer Verlag.

[153] Thorup, K. K. and Torgersen, M.: *Unifying Genericity—Combining the Benefits of Virtual Types and Parameterized Classes*, ECOOP'99 – European Conference on Object-Oriented Programming, Lisbon, Portugal, 1999, Lecture Notes In Computer Science vol. 1628, Springer Verlag.

[154] Torgersen, M.: *Virtual Types are Statically Safe*, 5th Workshop on Foundations of Object-Oriented Languages, San Diego, CA, USA, 1998.

[155] Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G., and Gafter, N.: *Adding Wildcards to the Java Programming Language*, SAC, Nicosia, Cyprus, 2004, ACM Press.

[156] Ungar, D. and Smith, R. B.: *Self: The Power of Simplicity*, OOPSLA'87 – Object-Oriented Programming Systems, Languages and Applications, Orlando, Florida, USA, 1987, Sigplan Notices vol. 22, ACM Press.

[157] Vaucher, J.: *Prefixed Procedures: A Structuring Concept for Operations*, IN-FOR, vol. 13, 1975.

[158] Vaucher, J. and Magnusson, B.: *SIMULA Frameworks: the Early Years*, in *Object-Oriented Application Frameworks*, Fayad, M. and Johnsson, R., Eds.: Wiley, 1999.

[159] Wegner, P.: *On the Unification of Data and Program Abstraction in Ada*, Tenth ACM Symposium on Principles of Programming Languages, Austin, Texas, 1983, ACM Press.

[160] Wegner, P. and Zdonick, S.: *Inheritance as an Incremental Modification Mechanism or What Like is and Isn't Like*, ECOOP'88 – European Conference on Object-Oriented Programming, Oslo, Norway, 1988, Lecture Notes in Computer Science vol. 322, Springer Verlag.

[161] Wirth, N.: *The Programming Language Pascal*, Acta Informatica, vol. 1, 1971.

[162] Yourdon, E. and Constantine, L. L.: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*: Yourdon Press Computing Series, 1979.

[163] Østerbye, K.: *Parts, Wholes, and Subclasses*, 1990 European Simulation Multi-conference, 1990.

[164] Østerbye, K.: *Associations as a Language Construct*, TOOLS'99, Nancy, 1999.

[165] Østerbye, K. and Kreutzer, W.: *Synchronization Abstraction in the BETA Programming Language*, Computer Languages, vol. 25 pp. 165–187, 1999.

[166] Østerbye, K. and Olsson, J.: *Scattered Associations in Object-Oriented Modeling*, NWPER'98, Nordic Workshop on Programming Environment Research, Bergen, Norway, 1998, Informatics report 152, Department of Informatics, University of Bergen.

[167] Østerbye, K. and Quistgaard, T.: *Framework Design Using Inner Classes—Can Languages Cope?*, Library Centric Software Design Workshop '05 in connection with OOPSLA 2005, San Diego, 2005.

## Appendix: Time line

Below is a time line showing when events in the BETA project took place. After each event, the number of the section describing the event is shown. In some electronic versions of this paper there may be links to these sections as well as the part of the text describing the events.