GOODS to Appear on the Stage

Invited Speech at ECOOP 97

Kristen Nygaard

Department of Informatics, University of Oslo
P.O.Box 1080 Blindern, N-0316 Oslo, Norway
e-mail: kristen@ifi.uio.no

Abstract. The lecture will trace the development of some important object-oriented concepts and point out the analogy between performances at the stage of a theatre and the operation of information systems (and program executions). This metaphor will be used in a description of the ideas pursued and developed in the GOODS Project (General Object-Oriented Distributed Systems), a three year project supported by The Norwegian Research Council, starting January 1997. GOODS aims at extending the framework of object-oriented programming to include a multi-layered approach to the organisation of the relationship between people, computer hardware, organisational rules and programs in general distributed systems. GOODS also aims at introducing general tools for specifying visibilities of objects (scopes) and the precise dealing with the identities of objects that exist in many versions in a distributed environment.

1 Introduction

1.1 Back to Research

When the ECOOP '97 organisers invited me to give this speech, they offered me the choice between giving my overview and evaluation of the first thirty years of object-oriented programming 1967-1997, or talking about my own current research and views about the future of object-oriented programming. Since I now have passed seventy years, the attraction of presenting nostalgic and grandiose reflections on the past has considerably diminished. Rather exasperated reactions: "What! Even more new crazy ideas!" than: "Impressive! The old man still has an audible voice and is capable of standing on his feet a full lecture."

My six-year political assignment as national leader of the campaign against Norwegian membership in the European Union was finished with our victory on the 28th November 1994. I was very much looking forward to returning to research. Research is more exciting than politics, but sometimes politics is more important. The first thing to do in 1995 was to get hands-on experience with the new generation of workstations and with multimedia hardware and software that had appeared since 1988. The second was to decide what kind of research I should engage in and then try to compose a new team.

I commented in a recent paper [Nygaard, K. 1996]: "Many people have observed that the research teams in which I have been active, usually have contained people

much younger than me. I have been asked if that is so because I prefer to work with young people. The answer is no. I prefer to work with people who believe that my newest ideas are worth while working on, and themselves have ideas that fit in. Older, more established people usually did not believe that, and don't. There are exceptions, like the people in the BETA team. In the EU battle I was working in many teams, most teams composed by people from a wide age bracket. In research it has been different, as I have told." [IRIS 96]

During my six "political years" (1988-1994) I still was not inactive in research. I could see that ideas about the "theatre metaphor" that I had worked on (with Professor Dag Belsnes at the Norwegian Computing Center and the BETA team) in the late 1970s and early 1980s, became more and more relevant, and I made some efforts to generalise these ideas and integrate them with other ideas.

In 1991 I was asked by Professor Brian Randell to present my subjective views on the "Past, Present and Future of Programming Languages" at The 25th Anniversary Newcastle Conference in 1992. I had, of course, to put quite much effort into the lectures, and as a result a program for future research started to emerge. In 1995 it turned out that audiences were particularly interested in just the main points of that program. Not surprising, because of their relevance to distributed systems. It was time to start assembling a team and to apply for funding.

1.2 Overview

The format of an invited speech may be rather different from that of an ordinary reviewed paper. Many styles are allowed, and a personal note is expected. Strong opinions and postulated facts may be stated in absolute earnestness or with a tongue-in-cheek. The interpretation may be left to the audience. In these senses this is a typical invited speech.

The lecture starts (Section 1.3) with some quotations from the application that gave us some funds for my current project, the GOODS project. You will probably observe that we have embarked on a very ambitious endeavour: To extend the conceptual framework of object-oriented programming as in the SIMULA, DELTA and BETA-tradition to include also a joint description of the *performances* (program executions) according to their given *scripts* (by programs, constraints and other kinds of rules) by *ensembles* (of human actors and information processing equipment linked by *connectors* (communication channels)).

In Section 2 the most relevant elements for the GOODS conceptual platform are selected from earlier SIMULA, DELTA and BETA papers from 1963 on. In an article for a journal this would have had to be cut down to a series of references, tacitly making the assumption that the reader had these references available and, even more unrealistically, would have the interest and energy to look them up. The presentation is brief, but should at least give some clues to the world view of our understanding of object-oriented programming.

The theatre metaphor is introduced (in Section 3) through the system generator (process generator, phenomenon generator) concept from DELTA together with our definition of the model concept. The extension and initial exploration of this metaphor

follows (Section 4), using "the play within the play"-example from Shakespeare's Hamlet, Act III, Scene II, introducing multi- layered actor/role and ensemble/performance situations.

We will (in Section 5) have a go at a general approach to scoping: The description/ prescription of visibility/accessibility of system properties, components and states from a given location in a performance (program execution).

The reference to the (Kabuki) theatre is made to point out that scoping problems also appear in the setting of the theatre and of other process generators. The analogy with the theatre is certainly exaggerated in the next section (Section 6), in the brief discussion of clones: many versions of "the same" object in a distributed environment:

In dealing with general object-oriented distributed systems we also need tools for identifying and distinguishing different versions of an object and of state descriptors providing partial descriptions of objects states at a given time.

The lecture ends with some points relating to scripts, (plays, programs, specifications) dealing with open and closed, persistent and transient scripts, and with analysis and design in system development (Section 7).

1.3 The GOODS Project

The Norwegian Research Council demands applications that are serious and worded in concise and preferably impressive terms. For these reasons we started our application for funds with these paragraphs [GOODS, 1996]:

"To master the complexities of interaction in information systems, the magnitude of these tasks and the needs for frequent restructuring, object-oriented techniques have become dominant.

The penetration of information technology into the co-operation patterns of modern organisations has made it an important and interesting task to extend the object-oriented paradigm to encompass and integrate not only the computer programs but also the hardware executing the programs, the human actors and the communication functions in information systems. This extension should be directly linked to implementable new basic language constructs, and it should address both the analysis and design, the implementation and use, and the maintenance of systems."

"The project will be linked to research efforts at the universities in Glasgow and Århus, and to a user organisation introducing very comprehensive information systems.The framework shall be closely related to object-oriented programming languages and lend itself to supporting staging, operation and maintenance in a variety of such languages."

"The problem area has been cultivated, particularly as a field in software engineering, and notions such as actors, roles and views have been explored, often in connection with standardisation efforts. *Our use of the theatre metaphor is oriented towards the introduction of basic language mechanisms for creating layered system organisations, and our approach is similar to that used in developing object-oriented programming.* (Italics added by KN.). It is a project within basic research, not a software engineering project, in a field that in our opinion is not yet sufficiently explored to be suitable for standardisation.

We will use the BETA programming language as our platform because of its conceptual simplicity and generality, based upon a modelling view of programming languages. We feel that BETA will be the best substrate for our conceptual approach, integrating "what is done"- and "how it is done"-descriptions.

To the project members' knowledge, there exists no unified high level description and programming language for this kind of systems. There are protocols and various standards (e.g. CORBA), but they do not describe systems at the (user) application level. There is also much work done on general referent models and associated sets of languages (RM-ODP, Referent Model - Open Distributed Processing). We intend to address very wide classes of systems in terms of concepts implementable in a unified, programming language that includes the structured introduction of open segments that will be closed by human actors."

I want you to observe that we point out that the results of the project should be applicable also in the contexts of other reasonably well-structured object-oriented programming languages.

The Norwegian Research Council decided to support the project with some resources during a three-year period, starting in January 1997.

The GOODS team (as of March 1997) consists of:

- Associate Professor Dag Sjøberg, working in the field of persistent object-oriented distributed databases.
- Ph.D.-student Ole Smørdal, working on combining activity theory and the theatre metaphor in object-oriented analysis and design.
- Ph.D.-student Haakon Bryhni, working on high speed communication protocols in distributed systems.
- Another Ph.D.-student, starting later this spring.
- The author.

The GOODS reference group consists of researchers participating in seminars and discussions of ideas:

- Professor Dag Belsnes
- Gisle Hannemyhr
- Øystein Myhre
- Associate Professor Birger Møller-Pedersen.

The GOODS team and reference group have contributed to this lecture with both ideas and useful comments.

2 The Conceptual Platform of Object-Oriented Programming

2.1 SIMULA I and SIMULA 67: System Description and Programming

The foundation of the GOODS project should be established by tracing and bringing together some of the important lines of thought from the SIMULA languages, the DELTA language and the BETA language, as well as from system development research in which I have participated. (The SIMULA languages were developed by Ole-Johan Dahl and me, and with Bjørn Myhrhaug as a particularly important member of the very competent teams that participated in our work.)

The SIMULA I language report from 1965 [SIMULA I, 1965, p.2] opens with these sentences:

"The two main objects 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"."

Since SIMULA 67 came later and has got a much wider use than SIMULA I, some seem incorrectly to believe that the former is "more object-oriented" than the latter. The central concept is in place in SIMULA I, but regarded as and named "process" and not "object" (which emphasises the substance aspect of the process). Classes are named "activities", and qualified references with inheritance did not appear until SIMULA 67:

" ... to achieve greater flexibility and unity, SIMULA has integrated the two kind of entities (*comment 1997*: passive "data carriers" and acting "event routines") into one. The basic concept in SIMULA is the *process*, being characterised by a *data structure* and an *operation rule*.

The individual members of the data structure of a process will be called attributes".

"Thus SIMULA may be used to describe systems which satisfy the following requirement:

The system is such that it is possible to regard its operation as consisting of a sequence of instantaneous events, each event being an active phase of a process.

The number of processes may be constant or variable and they all belong to one or more classes called activities.

Since the set of system times at which events occur forms a discrete point set on the system time axis, and since every action in the system is a part of an event, we will name these systems *discrete event systems*." [SIMULA I, 1965, p. 7-9].

SIMULA I was a simulation programming language that turned out to become a powerful general programming language. SIMULA 67 is a general programming language that also is a powerful platform for other, specialised programming languages, as e.g. simulation languages.

SIMULA 67 was triggered of by the invention of inheritance: "Usually a new idea was subjected to rather violent attacks in order to test its strength. The prefix idea was the only exception. We immediately realised that we now had the necessary foundation

for a completely new language approach, and in the days which followed the discovery we decided that:

We would design a new general programming language, in terms of which an improved SIMULA I could be expressed.

The basic concept should be *classes of objects.*

The prefix feature, and thus the subclass concept, should be a part of the language.

Direct, qualified references should be introduced."

In SIMULA 67 the *object* is the fundamental concept, and categories of objects are called *classes*. Direct and *qualified references* are introduced, as opposed to the indirect, unqualified element references of SIMULA I. The need to combine the safety of qualified referencing (from Tony Hoare in 1965, see [Hoare, C.A.R., 1968]) with flexibility led to the *subclass* construct. That again to the powerful notion of *inheritance* which introduced the expressiveness of *generalisation-specialisation* into programming languages. As a corollary, the notion of *virtual quantities* and late binding followed.

Does inheritance belong to object-oriented programming as an essential aspect? Some people think so, I don't. The language mechanism of generalisation-specialisation is useful in a wider context. On the other hand, since the programming language constructs for inheritance were not available in 1967, we had to invent them because we needed them.

Apart from these important extensions, the basic perspective on computing from SIMULA I was carried over to SIMULA 67: The *program execution* was the fundamental phenomenon to be structured, generated, operated and observed, providing us with information. The program execution was a *model system* of the *referent system* whose *structure* was described by the program. The referent could be *manifest*, in the outside world, as e.g. a warehouse or a harbour, or *mental*, existing in peoples' minds, as e.g. an envisioned new VLSI chip or information system, or a combination of manifest and mental components. This approach is commonly called the modelling view, or the Scandinavian view on object-oriented programming.

If we compare with current object-oriented programming languages, we find that most of the basic language constructs were introduced first in the SIMULA languages of the 1960s. If Dahl and I had not invented SIMULA in the first half of the 1960s, what would now have been the state of object-oriented programming? Approximately the same as it is today.

Why? Because object-orientation is one of the basic ways of structuring an information process. If we had not invented it, someone else certainly would have done so, probably in good time before 1980. One may ask: Was object-oriented programming invented or discovered? Well, since it did not exist before SIMULA except as exemplified in a number of specialised programs, not bringing out the clean general and basic concepts, it was obviously not discovered. The correct answer is that SIMULA was derived. It was derived as the answer to the task to which Ole-Johan Dahl and I had dedicated ourselves: To create a language suited to the description of a very large class of systems - the class of discrete-event systems which Operational Research workers would want to understand, analyse, design and simulate on a computer. We succeeded, and it turned out that this class was so rich that it also encompassed most of the organ-

isationally complex information systems we now want to create programs for. Also, object-oriented analysis and design are becoming key technologies in system development in general.

2.2 From DELTA to BETA

Already in 1963, extension of SIMULA into the world of real-time computing was considered [Nygaard, K., 1963], and in the SIMULA I report of 1965 it was stated [SIMULA I, 1965, p. 9] that: "By introducing suitable processes SIMULA also may be used to describe with the desired degree of accuracy continuously changing systems, as well as systems in which some processes have continuous changes, other processes discrete changes."The task of generalising SIMULA to cope with continuously changing states was put aside for the more imminently important development of SIMULA 67.

After the SIMULA efforts I moved into cooperation with the trade unions to evaluate the workplace impacts of information technology, build their competence in the field and start the study of participatory design. This was later followed up by research in system development in parallel with my programming language research.

In a paper from 1986 [Nygaard, K., 1986: "Program Development as a Social Activity"] I make the remark: "I have been criticized for not using more time in the 1970s to promote the SIMULA language. Many other people have done a much larger job than I. It was a conscious choice. Should a single idea or project use up your whole life as a researcher? SIMULA (and object oriented programming) is like a child: You have helped create it, you are responsible for its young years, you must see to that it gets a chance to succeed. Then your responsibility ends. You may be proud of it, wish it well, but realize that it will develop on its own and is no longer your property. Your duty is now to care for the new baby and then for any future children."

The task of generalising SIMULA was, however, addressed later, in the years 1973-75, by the DELTA team consisting of Erik Holbæk-Hanssen, Petter Håndlykken and myself.

DELTA's objectives are stated on p. 5 in the DELTA language report [DELTA, 1975]:"The purpose of this report is to develop a conceptual framework for conceiving systems, and an associated system description language. Our starting point has been a language which is designed as a tool for both system description and the writing of computer programs: the SIMULA language. We have, however, freed ourselves from the restrictions imposed by the computer, described above. We hope to have provided a tool which makes it possible to conceive systems in ways natural to human beings, using and extending the properties of programming languages, making it possible to combine algorithmic, mathematical and natural language modes of expression."

In the SIMULA development, the notions of process, object, class, system etc. were precisely introduced in programming language terms, but a careful examination of the concepts lacked. This analysis, redefinition and introduction of precise concepts was given in Chapter 3 of the DELTA Report (pp. 14-221), and that chapter became later the conceptual platform and reference for the corresponding platform for the

BETA language project. (In DELTA the basic concept is called a component. It is an object with a action substructure more complex than in SIMULA and BETA.)

The early work on BETA was started in 1975 as the Joint Language Project (JLP) and is described in [Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1977]. On p.2 the initial purpose is stated as: "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."

In BETA a continuing discussion, development and critical evaluation of the concepts was an essential part of the project. (The BETA team consisted during the initial language development stage of Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and me. Later many others have contributed, in Oslo and particularly in Mjølner Informatics, Århus, and the University of Århus.)

The confusion surrounding the system concept is discussed on pp. 14-15: "The underlying misunderstanding is that these questions and remarks imply that the quality of being a system is innate to certain collections of objects (living or inanimate).

In our opinion, any system definition must point out that a part of the world may be called a system when, and only when we *choose* to regard it in a certain way."

This points to the introduction in BETA of *perspective* as a fundamental aspect of system description and programming:

"A *perspective* is a part of a person's cognitive universe that may structure her or his cognitive process when relating to situations within some domain

- by *selecting* those properties of the situation that are being considered (and, by implication, those that are ignored), and
- by providing concepts and other cognitions that are being used in the *interpretation* of the selected properties."

(A discussion of the perspective concept is given in [Nygaard, K. and Sørgaard, P., 1987].) To regard a phenomenon as a system then becomes a choice of perspective:

A *system* is a part of the world that a person (or group of persons) during some time interval chooses to regard as

- a whole consisting of components,
- each component characterized by properties that are selected as being relevant and
- by state transitions relating to these properties and to other components and their properties.

In BETA once more the process is the basic concept:

A *process* is a phenomenon regarded as the development of a part of the world through transformations during a time interval. *Structure* of a process is limitations of its set of possible states and of transitions between these states.

Examples of structures are:

- Written and unwritten rules being obeyed
- The effect of programs
- Perspectives

The process perspective may be further specialised to that of *information process*:
A process is regarded as an information process when the qualities considered are:

- its *substance*, the physical matter that it transforms,
- its *state*, represented by *values* obtained by mapping of measurements of *attributes*, attributes being selected properties of its substance,
- its *transitions*, the transformations of its substance and thus its measurable properties.

A computer program execution is an information process. Its substance consists of what is materialised upon the substrate of its storage media, its state consists of the values of variables (which we observe or "read" as "2.35", "HEATHROW", "false" etc.), evaluated functions and references, its transitions of the sequences of imperatives being executed. Another common example is the operation of an information system with both computers and people carrying out actions, and with program executions and paper documents being operated upon by these actors.

Now we may give a definition of the science of informatics (computer science in US and UK):

Informatics is the science that has as its domain information processes and related phenomena in artifacts, society and nature.

And by applying the system perspective upon an information process we get:

An *information system* is a part of the world that a person (or group of persons) during some time interval chooses to regard as an information process with

- its *substance* consisting of components,
- its *state* being the union of states of each of the components
- its *transitions* being carried out by the components and affecting their properties and the properties of other components.

We are now able to give a precise general definition of object-oriented programming:

In *object-oriented programming* an information process is regarded as an information system developing through transformations of its state:

- The substance of the system is organised as objects, building the system's components.
- Any measurable property of the substance is a property of an object.
- Any transformation of state is the result of actions of objects.

In an information process organised by object-oriented programming there are no loose crumbs of substance lying around, every piece of substance is a part of an object. There are no aspects of state not related to an object, and no Acts of God, only acts of objects.

The three basic aspects of information process are, not surprisingly, represented by abstractions in programming languages: The *class declaration* abstracts and catego-

rises *substance*, the *type declaration* abstracts and categorises value and thus *state*, the *procedure declaration* abstracts and categorises state *transitions*.

In my opinion *system-oriented programming* would have been a better term than object-oriented programming, but the term object from SIMULA stuck and there is no point in any attempt to argue for a renaming. Object-oriented programming may also be characterised as *substance-oriented programming*, and instead of talking about logic or constraint-oriented programming and functional programming, we could have referred to state and transition:

State-oriented programming: The computing process is viewed as a deduction process, developing from an initial state, the process being restricted by sets of constraints and by inputs from the environment, the information about the set of possible states being deduced by an inferencing algorithm.

Transition-oriented programming: The computing process is viewed as a sequence of transitions between representative (meaningful) states, in which transformations of inputs produces outputs that in their turn are inputs to new transformations.

The capabilities for *hierarchisation* are among the important characteristics of programming languages. A language should offer:

- *Substance hierarchies*, as exemplified by nesting in block-structured languages.
- *State hierarchies*, that is values associated with sets of states, and subvalues associated with subsets of these sets. This has been contemplated for BETA, and it is on the list of constructs that may be introduced in GOODS.
- *Transition hierarchies*, that is actions decomposed into subactions and subaction's subactions, as offered by the action stack mechanism.
- *Category hierarchies*, as offered by inheritance: as the class/subclass construct in SIMULA 67 (pattern/subpattern in BETA).

In the SIMULA languages, only unisequential and deterministic alternation (quasi-parallel) sequencing were available. In BETA the full range of sequencing categories, including concurrency, are available.

3 *The Theatre:* A Simple But Typical Process Generator

3.1 The DELTA System Generator

The DELTA Report [DELTA 1975, pp. 23-25] discusses the situation in which system descriptions are communicated between *communicators* (people, or e.g. from people to computing equipment):

"The considered system, being described, will be called the *referent system*. The communicator making the description will be called the *system reporter* (or just *reporter*).

The communicator using the description to make a model will be called the *system generator*. The communication process may now be illustrated by Fig. 1. This conceptual framework covers a wider set of communications situations, however.

The system reporter or the system generator, or both, may consist of a group of communicators. The model system may be either mental or manifest. If the language is a computer programming language, the system description is a computer program and the system generator may be a computer generating a manifest model system in the form of a program execution.

Another example is offered by the following chain:

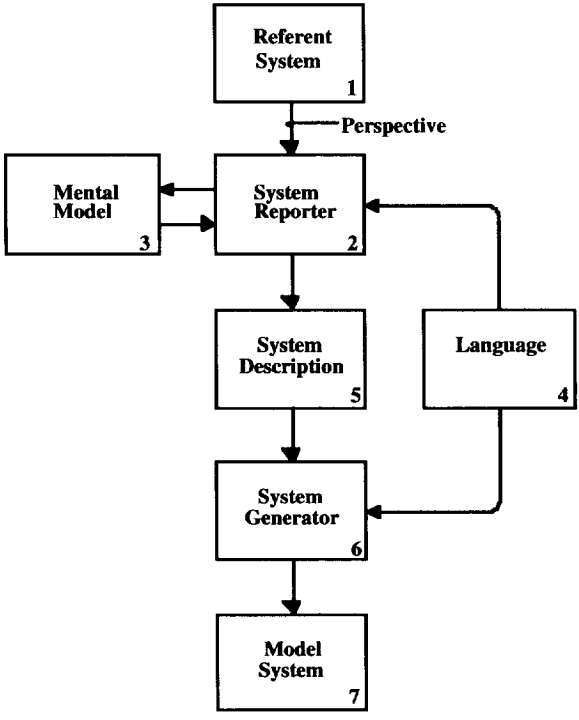
The referent system is mental, consisting of an island, a group of shipwrecked people, a magician named Prospero with his lovely daughter Miranda etc.

The reporter is William Shakespeare.

The language is a version of English, in which the description is organised into acts, scenes and lines, the lines containing ordinary English.

The system description is a play: "The Tempest".

Figure 1. The DELTA System Generator



In many situations, the real purpose of the process is to generate a manifest model system which fulfils some purpose. Examples are: The performance of a play portraying the author's mental referent system. The operation of an information processing system, achieved by the execution of a set of programs (a system description) which describes the programmers' and system designers' mental referent system of what the (model) information processing system should do.

In relation to computer programming, we will understand the communication process in the following manner:

The programmer (reporter) will have a mental referent system of what the computer should do (the program execution), and his program is a description which make it possible for the computer to portray this mental referent system".

The notion of system generator was further discussed and decomposed in [Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1977] on pp. 7-11: "In system programming it is, and to a much greater extent will become, necessary to control the components which are entering the program execution process: central processing units, storage media, data channels, peripheral equipment etc. This necessitates in our opinion a general conceptual approach to these apparently very different components. It will appear that a large proportion of the software complex now usually referred to as "basic software" and "operating system" will be regarded in our framework as being an integral part of the organisation of what we may call "logical components", as opposed to hardware components." "The "computer" concept is no longer useful for a precise discussion of the structure of the complex networks of interrelated computing equipment which we have to deal with in the computing systems of today and in the future. We shall regard such networks as system generators in the DELTA sense and introduce concepts in terms of which we may understand and describe such a network as a system generator."

"A system generator (computing equipment network) component upon which such systems may exist will be called a *substrate*. Disks, tapes, core storage, data screens are examples of substrates.

A system generator component which is able to change the state within a system (in the above sense of the term) will be called a *processor*. A central processing unit and a disk drive unit are examples of processors.

A system generator component which provides a connecting link between two components of theses categories (substrates, processors) will be called a *connector*. (It should be pointed out that the more complex "data channels" usually will have to be regarded as aggregates of substrates, processors and connectors at the basic "hardware level". They may be given a simpler structure at the "logical level".)

A collection of interacting substrates, processors and connectors will thus be called a system generator. We shall, in fact, understand any complex of computing equipment in these terms, any piece of equipment being regarded as belonging to one of the above component categories or as a subsystem consisting of such components. (A system generator may, of course, itself be regarded as a system.)"

The concepts of substrate, processor and connector were not further developed in the BETA language efforts. In the GOODS Project we will reexamine these concepts.

3.2 Models

When we relate to a situation, we cannot take into account at the same time all available information, we must always filter, select. This does of course also apply to modelling, and the perspective concept should to be introduced in any precise definition of

the term *model*, as exemplified by Lindsjørn and Sjøberg in their ECOOP '88 paper [Lindsjørn, Y. and Sjøberg, D., 1988]:

"A phenomenon M is a model of a phenomenon R, according to some perspective P, if a person regards M and R as similar according to P.

We will call R the referent phenomenon (or simply referent) and M the model phenomenon (or simply model).

We will call P the perspective of the model M."

3.3 Performances and Ensembles

The theatre metaphor is a very useful one, since very many terms relevant to that world also have their counterparts in the world of information systems, and that analogy has been explored by many. In my own work we started using the metaphor in the DELTA project (as shown in the quotes above). Then Dag Belsnes and I explored the metaphor further around 1980. I have used it in my research, teaching and lecturing on programming languages in general and object-oriented programming in special, since then. Belsnes used our ideas in his 1982 paper "Distribution and Execution of Distributed Systems", [Belsnes, D., 1982], and they were used in Master Theses by Øystein Haugen [Haugen, Ø., 1980] and Hans Petter Dahle [Dahle, H. P., 1981]. In the BETA team, the metaphor was used in work on the safe dynamic exchange of BETA components during program execution (see [Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B and Nygaard, K., 1986]) and in work by students of Ole Lehrmann Madsen and Bent Bruun Kristensen. Lindsjørn and Sjøberg gives a good example of the use of the metaphor in a multi-layered setting (see Section 4) in their ECOOP '88 paper.

(At this stage I want to remind you once more that I am not talking about what we have done, but about ideas for future work, to a varying degree based upon earlier work.)

By a performance we will understand a program execution in a computer, the operation of an information system containing people and computing artifacts, a performance of a play on a stage, as well as other processes created and stage-managed by a system generator in the sense we consider in this lecture.

A performance will be regarded as generated by an ensemble carrying out a script. In The American Heritage Dictionary of the English Language, [Third Edition 1992, Houghton Mifflin], we find:

"script (skrîpt) noun ...

3.a. The text of a play, broadcast, or movie. b. A copy of a text used by a director or performer. ... ".

I will use the term in a wider context, and include programs for computing artifacts and the sets of rules and conventions, written and unwritten, that provide the structure of performances.

In a performance on a theatre we identify subprocesses as the performance of roles in a setting given by backdrops, properties and costumes. Properties may also, however, "perform", representing something they are not. Consider, as an example, the daggers used for all the gruesome murders we witness on the stage and screen. All

parts of the setting may impact upon the enacting of the roles (Noblemen are stabbed to death by daggers, Macbeth loses his head), they may themselves change state through actions in the performances of the roles: a door opened, a window pane splintered.

In SIMULA we united the "event routines" and the "passive data structures" into the powerful unifying concept of objects. I feel that the same conceptual unification must be made in our understanding of performances in general: Macbeth, the witches, the daggers, the tables, even Macbeth's decapitated head, all play roles in the performances.

Correspondingly, in a meeting the subprocesses of the meeting performance may be the roles of reviewers, secretaries, chairpersons, exam papers, ranking lists, database terminals, protocols with passed decisions. In informations systems all roles are present in the performance as objects, their substance and state being operated upon by the actions of the transitions, together constituting the object processes.

Roles are performed, enacted, embodied by actors -and in our terms - also by properties. Laurence Olivier performs Hamlet, a collapsible theatre foil performs in his hand in the fencing scenes. (Perhaps a stuntman had to take over the fencing for Olivier as he grew older.) An ensemble is a collection of the actors participating in a performance. The interaction between two role processes in a performance we will refer to as communication. The interaction between a role and the actor carrying out the script of the role we will refer to as interpretation, and we will say that the actor performs the role. The communication between two roles in a performance is implemented through communication between the two actors who interpret the roles.And the ensembles themselves are in an object-oriented perspective to be regarded as a a system of interacting object processes.

In the GOODS project we will have to consider the building up of simple ensembles from processors, substrates and connectors from very simple basic components. Then more complex ensembles may be constructed from these, incorporating also human actors.

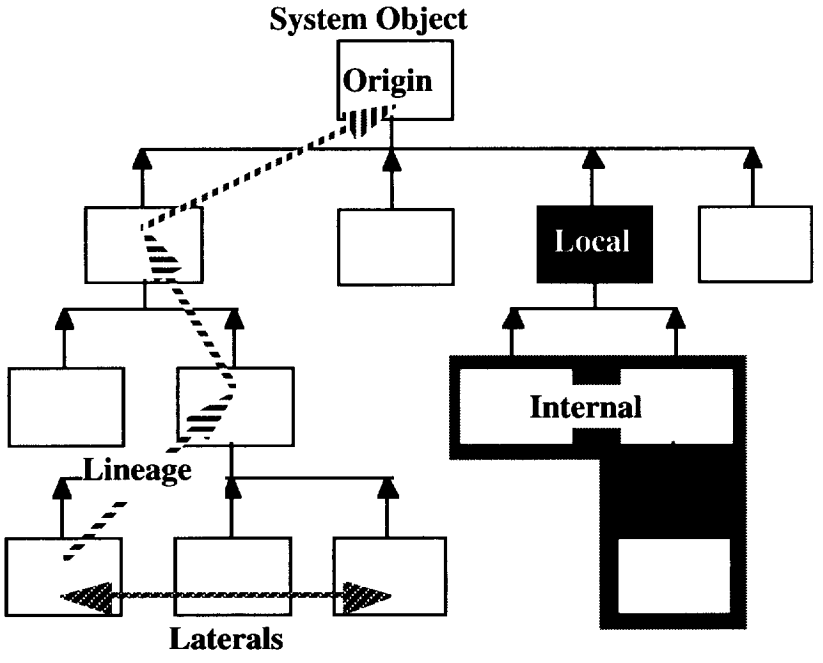
In both SIMULA, DELTA and BETA all systems were block-structured and had a common outer, enclosing object, named the system object. All other objects were enclosed by, nested inside the system object, their life spans delimited within the life spans of their enclosers. This is illustrated on Figure 2. We will call this a nested structure, or simply a nest.

In order to refer to objects in a nest, we must use the DELTA terminology for describing object relations in object-oriented nested structures. They are illustrated in Fig. 2.

In Figure 2, the root of the nested structure is at the top. We will also use figures with the root at the bottom, as in Figure 3.

If a performance P1 shall act as an ensemble and perform another performance P3, P1 must know about P3 and be able to refer to it. This is outside the scope of the references in the SIMULA/DELTA/BETA languages. We have to introduce *external references.*

Figure 2. Nested Object Structures



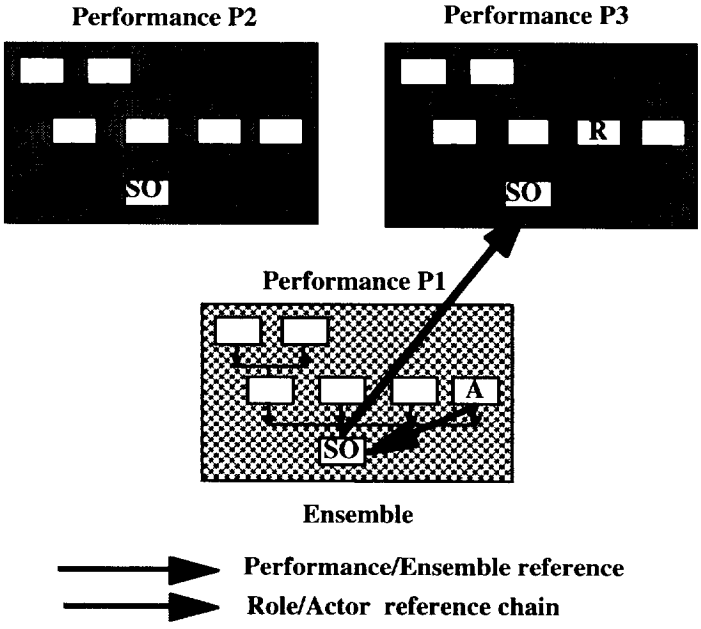
In distributed systems, the standard situation is that you are considering more than one nest, possibly widely separated, and any effort to make them have a common, enclosing system object will seem very artificial. This also necessitates external references if the nests shall communicate. External references then may be direct, or through a chain of nest-internal and nest-external references (Figure 3).

But the internal structure of P3 in Figure 3 may be unknown to P1. Should the external reference then be unqualified? I believe that instead the qualification of the external reference should be that of a pattern (class) containing the description of the characteristics of the *communication channel* and of the *communication protocol*. That is, the qualification of the *connector*.

In the case of a qualified external reference from P1 to P3, some kind of matching procedure will be necessary to secure that P3 will respond correctly to communication from P1. Also, the pattern used in this procedure could exploit the scoping capabilities proposed in Section 5 to specify the desired security from unwanted aspects of the communication.

In more complex situations, as in a networked environment, the organisation of the external references probably would be implemented through specialised performances in the net, functioning as name-servers.

Figure 3. Nests and External References



How is a new script entered? In the SIMULA/DELTA/BETA languages there are no direct language facilities for the production (through import or on-line programming) of new program segments that then are becoming scripts for new performances. *For the time being, I believe that a safe and understandable structuring of performances is best achieved by insisting that program segments produced within a performance may only be executed as a part of another, external performance.*

Programs may provide persistent or transient structure. In a database environment, the structure of the database is persistent. The structures of, e.g., the SQL queries operating upon the database are transient and should be generated by the database user processes (actors), and then performed by query processes.

The theatre metaphor may be generalised to systems where a set of performances are distributed to a set of locations called *stages* where they are performed by ensembles. The process of setting up and controlling the interplay between script and the ensemble in producing the performance is called *staging* or stage-management. (The sub-process of assigning actors to roles is traditionally called *casting*.)

Hamlet, Act III, Scene II: Staging, Layered Communication and Interpretation

The need for more complex models of information systems and system development became evident during the 1970s, as well as the need for involving the users. These activities started in a cooperation between the Norwegian Computing Center and the Norwegian Trade Unions in the early 1970s and spread (as "Participatory

Design" or "The Scandinavian School") to other countries. An overview is given in [Nygaard, K., 1986]. In [Nygaard, K. and Håndlykken, P., 1980, p. 169] it is stated:

" ... , the two main tasks for which the system specialist has a particular responsibility are:

the organisation of the proper cooperation of a large and varied group of information processing and communication equipment, many operating in parallel.

the design of the modes of expression available in the involved peoples' interaction with the information system - from their pushing of buttons, through their use of a limited set of rigidly defined transactions to their use of a programming language. If the sum total of a particular person's modes of expression in relation to the system is called his language, then it is seen that design and implementation of languages are essential tasks.

The "operating system" now becomes an integral part of organisational structure, and should (according to e.g. agreements, laws) be designed, or at least to a very great extent be modifiable locally."

The first extension of the theatre metaphor was inspired by "the play within the play" in Shakespeare's Hamlet. In Act III, Scene II Hamlet is staging a play with a troupe of touring actors. Later it is performed before the court, with Hamlet, the Queen (Hamlet's mother), the current King (Hamlet's uncle, brother of the now deceased King, Hamlet's father) in the audience. Hamlet has staged a play portraying his suspicions: The previous King is murdered by his brother, who pours poison in the King's ear while he is asleep in his garden (considered a fail-safe procedure at the time).

This results in a two-layered performance (see Figure 3): The top-layer performance is staged on a stage-within-the-stage. The roles of the King and the assassin are performed by 1st Actor and 2nd Actor as the main actors in the ensemble. They are communicating (interacting), and the King dies.

The lower-layer performance is the events in the court: Hamlet observing his mother and uncle, the King deeply disturbed by seeing his own crime re-enacted, etc. The two actors, 1st and 2nd Actor, are at this level roles performed by actors in the underlying ensemble, consisting of human actors.

In this layered situation, the Assassin cannot communicate with Hamlet in the audience. If that was attempted the structure of the situation would break down, and we would see what may be called "absurd theatre". Also, the actor playing the King may not address Hamlet, even if the communication between the King and Hamlet is implemented through communication between the actor playing the King and Hamlet, respectively.

This organisation of performances in layers is often used in building complex operating systems, but has not been supported by specific language constructs that may, I believe, result in easier, safer and more comprehensible implementations. The availability of tools for this layering may also facilitate the creation of general classes of distributed systems.

The analogy with computer-based performances should be obvious and is illustrated in Figure 4, in which we even have a stunt processor stepping in whenever the stunt of rapid multiplications is to be performed.

Figure 4. Hamlet, Act III, Scene II

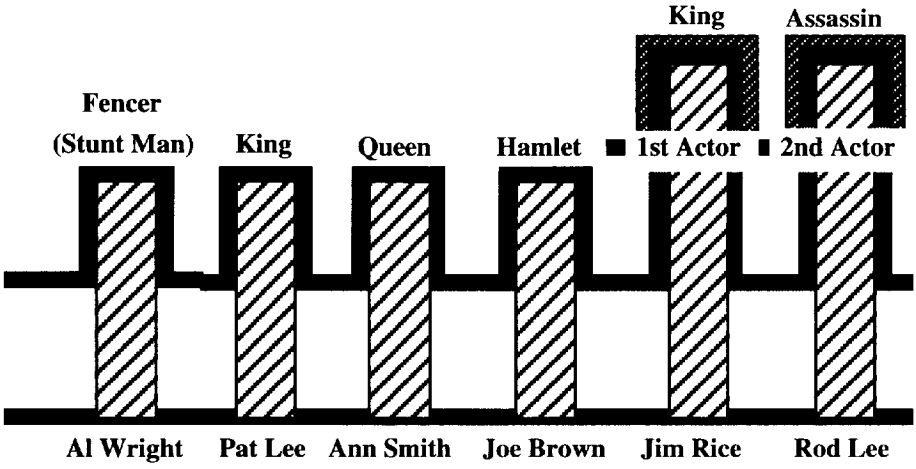
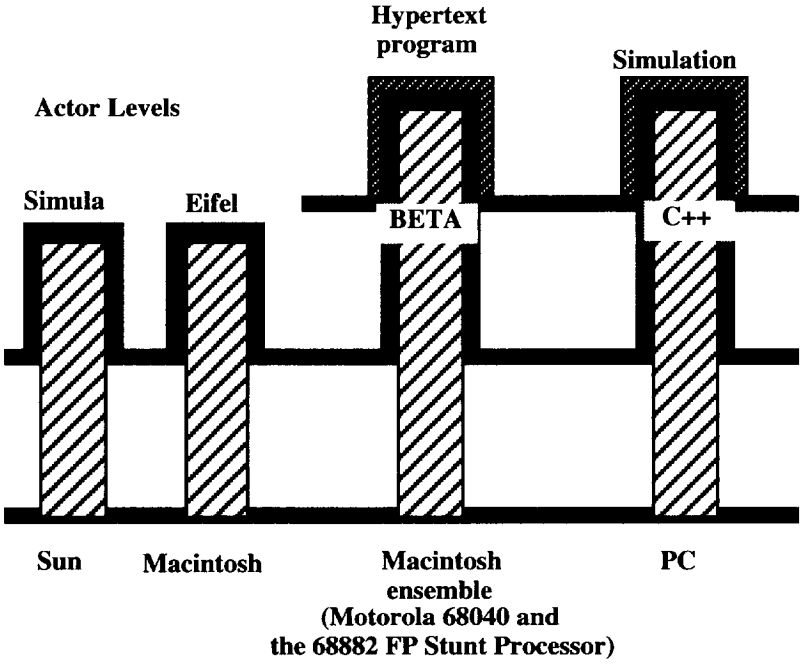


Figure 5. Computer Ensemble



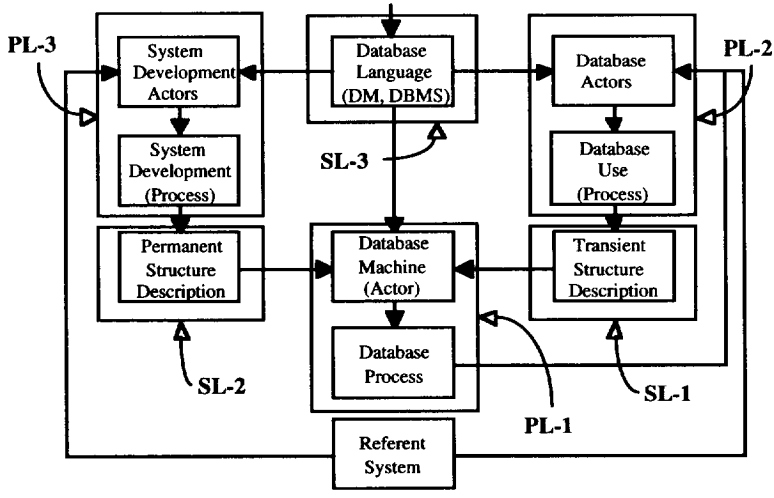
"When general object-oriented systems are considered, one needs also a generalisation of the theatre metaphor to multiple, communicating stages, as well as substages within a given stage. Our work till now shows that the generalisation to substages will not present large conceptual problems."

Some quotes from the GOODS project application:

"The actions are (will be) conceived as going on in layers, with hardware at the bottom, operating systems layers in between, and a final application task in the top layer. The structure of the application task, its program, is developed on the same equipment, and should get its proper place defined in a suitable layer." "A lower level object may at different stages at higher levels enact different views on this lower level object."

Figure 6 is from Lindsjørn and Sjøbergs ECOOP '88 paper and illustrates actors and performances at different layers in the construction and operation of a database.

Figure 6. Database Actors



We find a similar multi-layered structure when we analyse process/structure layers in system development [Nygaard, K., 1986]:

Process level 1: The information process (e.g. program executions, data processing by people and machinery in offices, etc.).

Structure level 1: The limitations imposed upon this process by computer programs, machine hardware properties, written and unwritten rules of human behaviour etc.

Process level 2: The system development process, including programming as a partial process, that has the structure of the information process (or the modification of its structure) as its product.

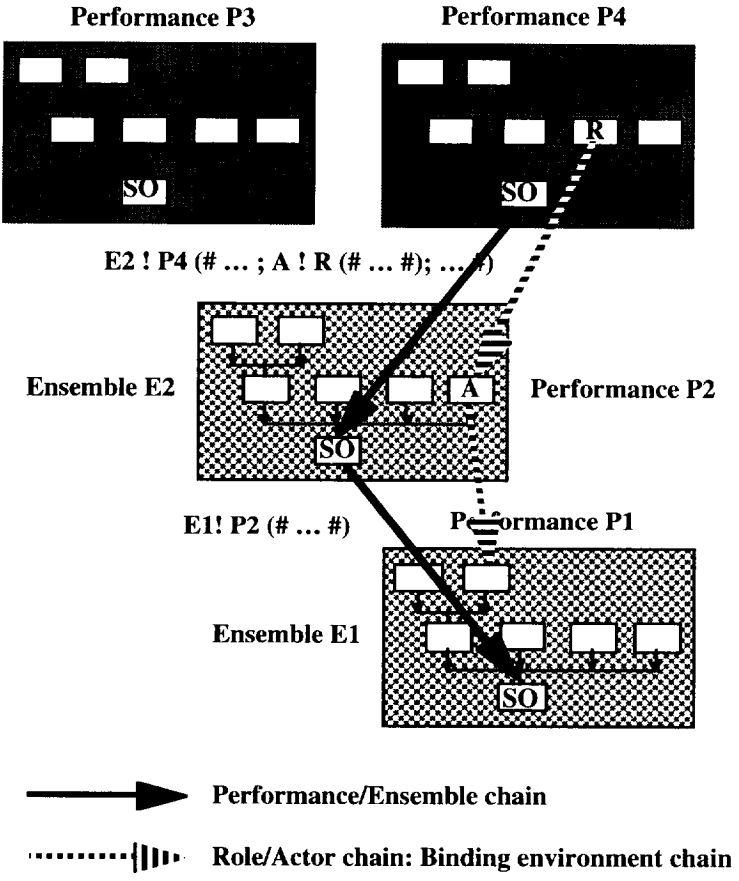
Structure level 2: The limitations imposed upon system development by organization, existing knowledge, available resources etc.

Process level 3: The process of learning within organizations, the research process, the adaptation of organizations to a changed environment.

I feel that this scheme should be reexamined in the context of system development and staging of distributed systems.

Our current ideas for language constructs are illustrated by Figure 7.

Figure 7. Performance Layers and Binding



The basic imperative: "Let actor A perform role R!", or "Let A perform R!". In programming language notation:

A!R

(As an alternative we could write R!A, read as "Let R be performed by A!". The final choice will depend upon what seems most logical when the elaboration of the grammar and semantics of the imperative have been better worked out.)

Since there may be further specification to be given for the duration of this actor/role linking, the imperative should be augmented by a additional part:

A!R(#..... #)

For the time being, we feel that external references out of nests should be between performances. (We may change our minds.) Probably the actor/role linking should then take place through a linking of an ensemble E to a performance P. P then is a an existing (persistent program execution) performance or a reference to a program (script) that will generate a new performance accordingly:

E!P (# ... ; A1!R1 (# ... #); A2!R2 (# ... #); ... #)

Binding rules must then be established. The obvious one is that x in R is initially bound in P, normally, in its own environment. If no binding is found (not even in the system object of P, the last location examined), x is bound in the environment of A, that is, within E (with A as the first object and the system object of E as the last object to be examined for a binding). Binding of x to a specified identifier y in A's environment may be made by:

E!P(# ... ; A!R(# ... ; x:y; ... #); ... #)

This possibility makes it possible to provide a tailor made interface of an actor A (e.g. a document) to be used in a given setting (e.g. a meeting performance), and thus multiple interfaces to any object. When this is combined with the scoping possibilities discussed in Section 5, a very rich set of capabilities will be available.

The imperatives of the form E!P etc. do not belong within P. Do they belong within E, the ensemble, regarded as a lower level performance? Do they belong in a third performance, that of a Master Puppeteer, overseeing all performances, staging the stage managers? There are many possibilities, and it will be fun to explore them.

Will all the constructs proposed in this lecture and later in the GOODS project be implemented? If so, will the implementation be efficient, considering the basic nature of the mechanisms involved?

We intend to implement. We believe that implementations may be efficient, based upon experience with corresponding well-structured implementation of basic mechanisms earlier, dating back to SIMULA. But we cannot be sure.

Gisle Hannemyr in the GOODS reference group has pointed out that it will be very useful to be able to create, with little or moderate effort, a high-level implementation of a complex distributed system, even if it is not very efficient. Many of the crucial aspects may by this approach be evaluated before a more basic-level implementation is embarked upon.

4 *Black Objects, Kabuki and Scopes:* Precise Prescriptions of Visibility

Continuing our references to the theatre, I was very impressed by birds, dragons etc. in the Japanese Kabuki performances, carrying out very intelligent and most incredible feats. How? Simple: They were handled by stage workers dressed in black. Didn't this destroy the illusion? Not at all. The audience had worked the scoping rule into their conception of what they saw. They *knew* that people dressed completely in black were to be considered invisible.

By "scoping" we understand the mechanisms that in a program execution (performance) determine which other objects and object attributes are accessible at a given

location and time. Those that are accessible, we call "visible". Certain aspects of scopes are determined by the language structure, by "structural visibility". Other aspects may, to have effect, be supplemented by "specified visibility", that is, by specific declarations.

DELTA is a block-structured language, describing nested object-oriented structures. In the DELTA report [DELTA, 75], Chapter 3, such structures are discussed in detail in order to describe precisely the interaction that may occur. Two levels of interaction (or, more correctly: *involvement*) are introduced (p. 112):

- "the "comprehensive" versions (*strong involvement*) assume that the acting component has *full information* about the other component's *qualification*, and that not only the *title* (class name) of the qualification is known.
- the "weak" versions (*weak involvement*) do not assume the full information about the other component's qualification, at most information about its (class, pattern) title."

Let C1 and C2 be objects within the same nested system (that is: with one system object enclosing all other objects), then [DELTA,75], p. 116:

"We will say that a component *C1 is perceived by another component C2* if and only if the state of the system is such that C1 is the value of some direct or remote, structural or specified reference of C2."

If C1 is perceived by C2, this implies that C1 may be weakly involved in C2's actions. If C1's attributes in addition are known to C2, then C1 may be strongly involved in C2's actions, and we will say that *C1 is recognised by C2*.

In [DELTA, 75] the conclusion of the discussion is (p. 125): "As we have demonstrated, a component may perceive any other component (object) with the preservation of security, if only the necessary precautions are taken. In fact, it also follows from the discussion that a component may also recognise any other component. It is thus possible to state the rule: Any component is structurally operable by any other component.

Should we state such a rule? If not, which rules should be imposed, and what should be their justification?"

In SIMULA the answer given is that lateral objects are structurally visible, but only accessible if they also are specified as visible by references. Enclosing objects are structurally visible and accessible without further specification. (See Figure 2 for explanation of the terms.) Scopes may in SIMULA be further detailed by specification of "hidden" and "protected" attributes.

Specification of scoping in BETA is discussed in [Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1977] pp. 41-42 under the heading of "Interface specification":

"The set of possibilities available in the <interface description part> has not yet been discussed in any detail. Among the possibilities which may be useful we may mention, e.g.,

- the exclusion of any use of attributes of or references to (object) entities (only the entity itself and its internal entities being available).
- the exclusion of a specified list of identifiers (names and titles).

- the restriction to the use of only a subset of the imperatives of the language."

In BETA a very different answer was given than that of SIMULA: All objects and attributes that are structurally visible should be accessible, without restrictions imposed by the language designers. The rationale was that the scoping imposed is a part of the specification of the system at hand. Powerful tools for specifying visibility were discussed, but not worked out in detail and implemented.

In GOODS our attitude is that of the BETA language, and language constructs will be proposed. As of today, we are considering two general constructs for the declaration part of objects:

EXCLUDING <list of scoping clauses> and EXCLUDED IN <list of scoping clauses>.

A number of different categories of scoping clauses will be introduced:

- The non-terminal symbols of the BETA grammar are considered parts of the BETA language and a list of these non-terminals will be allowed as scoping clauses.
- Terms referring to internal, external, lateral and lineage objects, as well as their nesting level, in nested object-oriented structures.
- Names of attributes.
- Restriction from recognition to perception of objects that may be referenced.

The EXCLUDING declaration specifies terms that may not be used *within the local object* (where the declaration appears), and which object categories in the nested structure that may not be referred to or only perceived but not recognised.The EXCLUDED IN declaration specifies the contexts *outside the object* in which the object's descriptor and its attributes may not appear. Perhaps it will be convenient to introduce also as alternative versions ALLOWING and ALLOWED IN.

I believe that we may augment the set of options by creating clauses that prohibit binding of specified terms in the local performance, so that the binding is certain to take effect in the performing ensemble.

5 *Bring in the Clones:* Persistence in General object-oriented Systems

When objects are persistent, that is, when they have extended life spans and participate in a series of program executions, and also are moving between stages (locations), then more detailed and precise concepts are needed for distinguishing between objects stemming from the same source. (In addition to the problems of different versions of objects.) We may see objects that are cloned (copied) from one stage to another, the original still existing in the old location, or "transported", meaning that the original is deleted. We may have "dead" copies describing the state of the object at a specified time and location (state descriptor, snapshot, map), and active objects reconstructed from such a state descriptor. New structural attributes must be introduced in order to

cope with these situations, and insight from recent work on persistent programming and object-oriented databases will be exploited in our work.

The notion of representative states were discussed in [Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1977], pp. 41-42: "By a representative state we shall thus understand a state of aL- system component which has the property that it may be given a meaningful interpretation in terms of the task it performs in relation to other L-system components.

If a means is introduced of indicating stages of execution at which the state of a L-system component is representative in the above sense, then it is reasonable to require that

- execution of interrupts are initiated within an L-subsystem LSS, whose actions are executed by an L-processor LP, only when LSS as a whole is in a representative state.
- only values obtained in such representative states are read or assigned by other L-system components.

Similar notions have been treated by others, e.g. [Dijkstra, E., 1974].

When we create tools for dealing with multiple versions of an object and with (usually) partial maps of objects, it may be useful to insist that we from the outside of an object always may assume that it is in a representative state.

If we are to deal in general with persistent object, assuming that we only may interact with them in representative states, we must also provide for the preservation of information about their stage of execution, so that they may be reactivated and proceed with the execution of their associated actions. This, and other related problems, are of course addressed by researchers considering the notion of "live objects" existing on a variety of substrates. We will have to take into account the results already achieved by other researchers in this area.

6 *Authoring and Staging:* Programming from the Small to the Large and Systems Development

6.1 New modes of expression

In this section I will discuss matters relating to the writing of scripts:

- Different kinds of imperatives, introducing property descriptors, extending the imperatives of a programming language, dealing with the structuring of actions by human actors.
- The notion of open and closed imperatives, allowing for situations in which human actors may act (manouevering) according to motivations and rules that are not describable in programming language terms.
- Having a new go at the notion of contexts, trying to find a solution better than the half-baked ones offered by using inheritance.

- Linking the BETA fragment system alternative to multiple inheritance, program module reuse and program libraries to the new possibilities to defining contexts.

In DELTA the notion of actions had to be extended [DELTA 75, p. 172]: "The actions of a component (object) may be divided into the categories:

- time consuming actions, as, e.g. the heating of ore to the melting point within a furnace, or the traversing of a crane from one point to another, being executed during a time interval.
- instantaneous actions, as, e.g., the leaving of a queue, or the selection of which ship in a queue of waiting ships should be allowed to occupy an empty quay position in a busy harbour, being initiated, executed and completed at a discrete point in time."

(Remember that DELTA is a system description language in which both compilable and computer executable as well as non-compilable and non-computer executable description elements are available.) Since continuous changes of state had to be described, concurrency imperatives with property clauses were introduced, containing property descriptors. A property descriptor is set of relations, separated by commas and enclosed by braces. Examples of concurrency imperatives are:

- WHILE {temperature ≤ melting point} LET {temperature = start-temperature + F(Energy-supply(time))};
- WHILE {time < delivery time} LET {candidates work, each in isolation, on their written exam};
- LET {Evaluation of exam} DEFINE passed, mark;
- LET {x ** + y ** ≤ r **} DEFINE x, y;
- LET {Position of cardinals be inside conclave}; WHILE {no pope is elected} LET {Negotiations go on. Emit black smoke after each indecisive vote}; WHILE {Voting slips are burning} LET {Emit white smoke};

Concurrency imperatives and imposed property descriptors are discussed carefully in the DELTA report, including the duration of the imposition of a property descriptor upon its system environment. These language elements have to be reexamined in the GOODS project. By associating property clauses with objects in an action stack, a dynamically changing set of imposed condition may be described.

In programming languages, declarations are of course examples of property descriptors. More general constraints, described by property clauses have been treated in Alan Borning's work (see [Borning, A., 1981]), and extended availability of such tools in programming languages would be very welcome. However, this mode of expression often is the most natural one when describing the actions of people within the operation of an information system.

A performance of an information system will come to a halt if, at some step in the action sequence of a piece of computing equipment, no definite and precise next step is indicated. This will result in inactivity until a triggering interrupt arrives, from either

some other artifact or from a human actor. For a human actor this is different. People may themselves decide which next action to choose in such situations, within the limits imposed. When no such choice exists, we will say that the next action (for a person or some artifact) is closed. When there is choice, the action is open. The actions within the performance then become depending upon the choices made by people in the situation, by what I call their manoeuvering. I think this term is quite suitable, as we may see from some of its meanings in a dictionary:

- A controlled change in movement or direction of a moving vehicle or vessel, as in the flight path of an aircraft.
- A movement or procedure involving skill and dexterity.
- A strategic action undertaken to gain an end.
- Artful handling of affairs that is often marked by scheming and deceit.

(The American Heritage® Dictionary of the English Language, Third Edition copyright © 1992 by Houghton Mifflin Company.)

The situation in the theatre corresponding to what is illustrated, is what is usually referred to as improvisation.

Often the situation is characterised by a manoeuvering in which there also exists a repertoire of available structured action sequences to choose between. This is a category of systems which semi-seriously may be referred to as PASTA systems (from PArtially STructured Activities). It is useful to point out that this mode of system description opens up for the inclusion very subjectively motivated actions, the details of which are unknown to those making the description.

6.2 From Simulation to Mediation

SIMULA, DELTA and BETA are languages that describe phenomena and concepts in the real world using objects and classes (patterns in BETA). Thus, object orientation is used to model the real-world domain that the information system is intended to maintain information about. Lately, there has been an attention to also capture aspects beyond this domain, and address the usage world, e.g., aspects relating to actors, communication, articulation of work, collective work arrangements, task flow, and work procedures. This is due to a shift of perspectives regarding the role of the computer in work settings; from a focus on the computer as means of control and administration of a real-world domain, to a focus that also include the computer as a mediator in the usage world, e.g., as in groupware or workflow applications.

Much work is invested in applying object-oriented technology to analysis and design in system development. We feel that more may be achieved than today.

In the GOODS project we will develop and use a conceptual framework for modelling computers incorporated into work arrangements based on activity theory ([Leontjev, A. N., 1983], [Engeström, Y., 1987]). Activity theory is used as a bridging link between the social concerns and the technical concerns as it address human work in a social context and has a strong emphasis on how artifacts (like computer systems)

mediate human activity. Further, the theory distinguishes between four aspects of an activity:

production
(the production of the outcome of the activity, like a service or goods),
consumption
(the use of goods or services by clients/customers or citizens),
communication
and
distribution
(aspects of the collective work done in order to realise production). We explore using the notion of

real-world modelling
to address the production and consumption aspects and a
notion of a theatre performance
to address the distribution and communication aspects. (This work has already started, see [Smørdal, O., 1996], [Smørdal, O., 1997])

6.3 In the large: Authoring and Staging

In the GOODS project we regard the concept of distribution in two ways:

- Multiple locations of stages. Still we may need to regard activity on multiple stages as belonging to one logical performance.
- Division of responsibilities. We want to address the responsibilities on objects to the performance as a whole. This include how actors interpret their role.

We argue that both of these interpretations of the concept of distribution are necessary to cope with complex systems, because notions of a whole (a performance), combined with layering (actor-role relationships) are powerful abstractions.

This naturally implies that "programming in the large" becomes an issue, because layering involves attaching various pieces of software to fulfil some function in the performance.

In the discussions of reuse, "programming in the large" has been a slogan. Now the notion of "design patterns" is attracting attention. Other approaches, like Ray Wellands "design zones", are also addressing the same set of problems. The GOODS project should provide a good basis for discussing these problems, and also contribute to solutions.

We will address the structuring of performances at two levels: Authoring and staging. This is due to a need within systems development to separate design of of-the-shelf software and software standards from the adaptation and integration of software in an organisation to fit the work.

When our general view is adopted, one also observes that this will lead to a way of specifying the proper place in the total picture of a number of programming tools, in particular program libraries. Since we will be using the same language approach for all aspects of the total process, we should achieve the "seamlessness" in concepts for the

various stages of systems staging, operation and maintenance. Also, as far as we can see, a solution seems to be opening to the old problem of creating unions of contexts without resorting to multiple inheritance. (See, e.g., [Møller-Pedersen, B., 1977])

6.4 Research approach

In some of the main object-oriented language developments (SIMULA, Delta, BETA) a rich set of diverse examples were used to develop and test ideas. In the GOODS project the main examples, or scenarios, will be:

- The work in the Planning and Building Authority in the City of Oslo.
- The planning and production in a shipyard.
- The staging and performing of plays in a theatre.
- The development of concepts, the multitude of discussions and the writing of reports by research teams scattered geographically and in various kinds of locations (offices, meeting rooms, homes etc.).

The first scenario will be used extensively in close contact with the administration and specialists working in the Authority. The second and third will mainly serve to provide and test ideas. The fourth we want to establish as a factual networked cooperation system.

7 Conclusion

This manuscript has been prepared under extreme time pressure, since the invitation arrived very late. Other researchers whom I have consulted have stated forthrightly and gleefully that they were looking forward to nail me at ECOOP '97 because of the blunders to be expected in a paper not carefully prepared. Hopefully some of the less well-considered proposals will have been improved when the lecture is delivered. But certainly not all.

Also I may at that time be better oriented about the work of others in the same field.

I have, however, used the opportunity offered to give my account of the outline of the GOODS project at an early stage. The other team and reference group members might have preferred a different selection of points to be presented.

I hope that the lecture at least will be useful in bringing us in good and close cooperation with others sharing our interests.

References

Belsnes, D., 1982: "Distribution and Execution of Distributed Systems", Report NR- 717, Norwegian Computing Center. Oslo 1982.

[BETA, 1993] Madsen, O. L., Møller-Pedersen, B. and Nygaard, K., 1993: "Object Oriented Programming in the BETA Programming Language", 357 pp. Addison-Wesley/ACM Press, ISBN 0-201-62430-3, 1993.

Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B.and Nygaard, K., 1973: "SIMULA begin". Studentlitteratur, Lund and Auerbach Publ. Inc., Philadelphia, 1973.

Borning, A., 1981: "The Programming Language Aspects of ThingLab: A Constraint-Oriented Simulation Laboratory", ACM Transactions on Programming Languages and Systems 3(4), pp. 353-387, Oct. 1981.

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

Dahl, O.-J., Myhrhaug. B and Nygaard, K., 1968, 1970, 1972, 1984: "SIMULA 67 Common Base Language", Norwegian Computing Center 1968 and later editions.

Dahle, H. P., 1981: "Observasjon av BETA-systemer" ("Observation of BETA Systems"), Report NR-707, Norwegian Computing Center, Oslo, 1981.

[DELTA, 1975] Holbæk-Hanssen, E., Håndlykken, P. and Nygaard, K., 1975: "System Description and the DELTA Language". Norwegian Computing Center, 1975.

Dijkstra, E. W., 1974: "Self-stabilizing Systems in Spite of Distributed Control", Comm. ACM 17, 11 (Nov. 1974), pp. 643-644.

Engeström, Y., 1987: "Learning by Expanding. An Activity-theoretical approach to developmental research". Orienta-Konsultit Oy, Helsinki, 1987.

Haugen, Ø., 1980: "Hierarkier i programmering og systembeskrivelse" (Hierarchies in Programming and System Description"), Master Thesis, Department of Informatics, University of Oslo, 1980.

Hoare, C. A. R., 1968: "Record Handling". In Genuys, F., ed., "Programming Languages", pp. 291-347. New York, Academic Press.

Holbæk-Hanssen, E., Håndlykken, P. and Nygaard, K., 1975: "System Description and the DELTA Language". Norwegian Computing Center, 1975.

Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1977: "BETA Language Development Survey Report, 1. November 1976" (Revised Version, September 1977). DAIMI PB-65, September 1977, Dept. of Computer Science, University of Aarhus.

Kristensen, B. B., Madsen, O. L. and Nygaard, K., 1976-80, "BETA Project Working Note" 1-8. Norwegian Computing Center, Oslo and Computer Science Department, Aarhus University, Aarhus, 1976-80.

Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B and Nygaard, K., 1983: "Syntax Directed Program Modularization". In "Interactive Computing Systems" (Ed. Degano, P. and Sandewall, E.), North-Holland 1983.

Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B and Nygaard, K., 1983: "The BETA Programming Language". In "Research Directions in Object-Oriented Languages" (Ed. Shriver, B., and Wegner, P.), MIT Press, Cambridge, Massachusetts 1987.

Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B and Nygaard, K., 1986: "Dynamic Exchange of BETA Systems", Unpublished manuscript, Oslo and Aarhus, 1986.

Leontjev, A. N., 1983: "Virksomhed, bevidsthed, personlighed" (English: "Activity, Consciousness, Personality"(?)). Forlaget Progress, Denmark, 1983.

Lindsjørn, Y. and Sjøberg, D., "Database Concepts Described in an Object-Oriented Perspective". In Proceedings of the European Conference on Object-Oriented

Programming (Oslo, 15th–17th August 1988), Gjessing, S. and Nygaard, K. (editors), pp. 300–318, Lecture Notes in Computer Science 322, Springer-Verlag, 1988.

Madsen, O. L., Møller-Pedersen, B. and Nygaard, K., 1993: "Object Oriented Programming in the BETA Programming Language", 357 pp. Addison-Wesley/ACM Press, ISBN 0-201-62430-3, 1993

Møller-Pedersen, B., 1977: "Proposal for a Context Concept in DELTA". DAIMI PB-83, DELTA Project Report No. 7. Department of Computer Science, University of Aarhus, Denmark, 1977

Nygaard, K., 1963: "Opparbeidelse av kompetanse innenfor Real-Time Systemer" ("Building Competence on Real-Time Systems"). Working Note September 19, 1963. Norwegian Computing Center.

Nygaard, K., 1970: "System Description by SIMULA - an Introduction". Norwegian Computing Center, Publication S-35, Oslo 1970.

Nygaard, K., 1986: "Program Development as a Social Activity", Information Processing 86, pp.189-198, Proceedings of the IFIP 10th World Computer Congress, North Holland, 1986.

Nygaard, K., 1992: "How Many Choices Do We Make? How Many Are Difficult?", pp. 52-59 in "Software Development and Reality Construction", Floyd, C., Züllighoven, H., Budde, R., and Keil-Slawik, R., editors. Springer-Verlag, Berlin 1992.

Nygaard, K., 1996: " "Those Were the Days"? or "Heroic Times Are Here Again"?" The Scandinavian Journal of Information Systems, Vol. 8.2, 1996.

Nygaard, K. and Håndlykken, P., 1980: "The System Development Process". In "Software Engineering Environments" - Proceedings of the Symposium held in Lahnstein, Germany, June 16-20, 1980. Hünke, H., editor, North Holland, Amsterdam 1981

Nygaard, K. and Sørgaard, P., 1987: "The Perspective Concept in Informatics", pp. 371-393 in "Computers and Democracy", Bjerknes, G., Ehn, P., and Kyng, M., editors, Abury, Aldershot, UK, 1987.

Smørdal, O., 1996: "Soft Objects Analysis, A modelling approach for analysis of interdependent work practices". In Patel D and Sun Y (eds.) Third international conference on object-oriented information systems (OOIS'96). (London, UK), Springer-Verlag, pp. 195-208, 1996.

Smørdal, O., 1997: "Performing Objects —A conceptual framework for object oriented modelling of computers incorporated into work arrangements". (Forthcomming).

Nygaard, K., 1992: "How Many Choices Do We Make? How Many Are Difficult?", pp. 52-59 in "Software Development and Reality Construction", Floyd, C., Züllighoven, H., Budde, R., and Keil-Slawik, R., editors. Springer-Verlag, Berlin 1992.

Nygaard, K., 1996: " "Those Were the Days"? or "Heroic Times Are Here Again"?" The Scandinavian Journal of Information Systems, Vol. 8.2, 1996.

Nygaard, K. and Håndlykken, P., 1980: "The System Development Process". In "Software Engineering Environments" - Proceedings of the Symposium held in Lahnstein, Germany, June 16-20, 1980. Hünke, H., editor, North Holland, Amsterdam 1981

Nygaard, K. and Sørgaard, P., 1987: "The Perspective Concept in Informatics", pp. 371-393 in "Computers and Democracy", Bjerknes, G., Ehn, P., and Kyng, M., editors, Abury, Aldershot, UK, 1987.

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

[SIMULA 67, 1967] Dahl, O.-J., Myhrhaug. B and Nygaard, K., 1968, 1970, 1972, 1984: "SIMULA 67 Common Base Language", Norwegian Computing Center 1968 and later editions.

Smørdal, O., 1996: "Soft Objects Analysis, A modelling approach for analysis of interdependent work practices". In Patel D and Sun Y (eds.) Third international conference on object-oriented information systems (OOIS'96). (London, UK), Springer-Verlag, pp. 195-208, 1996.

Smørdal, O., 1997: "Performing Objects —A conceptual framework for object oriented modelling of computers incorporated into work arrangements". (Forthcomming).