ISSN 0105-8517

Programming Environments and System Development Environments

Pål Sørgaard

DAIMI PB – 252
May 1988

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 71 88   *Telex: 64767 aausci dk*

Mjølner Report DK-SYS-31

Abstract

This report discusses the nature of programming environments and system development environments under the common term development environments. Five dimensions for the characterisation of development environments are identified. These are:

- the system development functions supported,
- the kind(s) of work supported,
- the suitability for prototyping,
- the application area, and
- the technical properties of the environments.

These dimensions are not independent. Some schools or traditions within this field are identified using these dimensions.

It is argued that computer support can be provided for all system development functions. Care must be taken, however, to allow for the necessary interplay between the different system development functions. Technical properties of the environments like integration, incrementality, and tailorability may support this interplay. Such properties may also lead to a change in the role of programming in system development. Programming will become more integrated in all parts of system development. This will make the classical division of work between analysts and programmers less obvious.

Cooperative work is important in system development. This should be reflected, or at least not ignored, in the development environments. Cooperative work in system development can be supported in many ways, also in many ways different from electronic mail and other applications typically associated with computer supported cooperative work.

Contents

Preface

This report is about programming environments and system development environments. The starting point of the work with the report was a study group on programming environments and system development environments which was arranged during the autumn 1987 at the Computer Science Department, Aarhus University. The study group was arranged by the author in cooperation with Elmer Sandvad. In this study-group a number of classical papers on programming environments were read. There were papers on UNIX [43], the Cornell Program Synthesizer [60, 72], Interlisp [73, 74], and Smalltalk [75, 83]. In addition practical and theoretical issues from system development were treated by reading theory on system development [2], case studies of system development [41], Floyd's overview paper on prototyping [27], and a report on the use of 4th generation languages [14]. A visit was made to a company using the 4th generation language Ideal. As was to be expected this reading resulted in a conceptual chaos. There were difficulties in comparing the environments presented and in relating the virtues of these environments to the problems of practical system development. The work with this report started as an attempt to sort out some issues in this chaos.

One of the issues we believed to be central was the question on whether programming environments and development environments were the same or two different kinds of environments. We soon realised that this was not the central question. Conceptually there is no reason to distinguish between system development environments and programming environments. This does not mean, however, that all programming environments are suited for use in practical system development. Many programming environments focus on programming as an isolated activity. For these reasons the work with the report shifted to the development of a conceptual framework for development environments.

This report is based on work in the MARS project [2, 42], in the Mjølner project [21], and on work on computer supported cooperative work [65]. Elmer Sandvad and the author are both participants in the Mjølner project. The Mjølner project aims at developing an industrial prototype of an environment for object oriented programming.

The framework presented here has been developed in cooperation with Elmer Sandvad, and chapter 6 should be credited as much to him as to the author. The wording of the report, however, is the sole responsibility of the author. Elmer Sandvad should not be blamed for any errors or lacks of clarity in this report. All deficiencies are the responsibility of the author.

Besides the cooperation with Elmer Sandvad this report has also benefited from many helpful comments from Kaj Grønbæk, Riitta Hellman, Jørgen Lindskov Knudsen, Peder Christian Nørgaard, Claus H. Pedersen, Lars Bak Petersen, and from the students who followed the study group on programming environments and system development environments.

Chapter 1

Introduction

Computer support for system development has traditionally been facilities like editors, compilers, and debuggers. These facilities have provided essential support for programming, but their impact on system development as a whole has been considered to be of a limited extent. System development and compiler construction have therefore been considered as well separated subjects.

New developments in programming environments and the upshot of various facilities supporting different activities in system development have, however, widened the scope of computer support for system development drastically. This development calls for a discussion of the relationship between support for programming and support for system development. Understanding of system development is necessary in the development of programming environments in order to make the environments applicable in the work situations they address. Other kinds of computer support for system development should be seen in relation to and integrated with the support for programming.

New programming environments may lead to changed working practices in system development. Interactive programming environments increase the feasibility of prototyping. Tailorable environments can be used to provide support for many different activities in system development. Together these changes may lead to a changed role of programming in system development. Programming will to a larger extent be an integrated activity in all parts of system development.

Cooperative work is an important kind of work in system development. Cooperative work can be supported by the use of computers, but a computer system can also have negative effects on the possibilities for cooperation. Results from research on computer supported cooperative work are therefore relevant for the support for system development. Experiences and results from the support for system development can sometimes be of general interest to computer supported cooperative work.

This report is mainly theoretical. It presents five dimensions for the characterisation of development environments. These are:

- the system development functions supported,
- the kind(s) of work supported (e.g. cooperative work or not),
- the suitability for prototyping,
- the application area, and
- the technical properties of the environments.

In this report the term development environment will be used to refer to all kinds of programming environments and system development environments. This is in order to emphasise that *conceptually* there is no difference between the two although many current programming environments are of little use in practical system development. The term programming environment is not used to denote this concept because that might convey the assumption that programming support is the only kind of computer support in system development.

It is assumed that the reader of this report knows some of the discussed environments, for example from reading some of the cited papers. Throughout the report environments like Smalltalk [75, 83], Interlisp [73, 74] and UNIX [43] will be mentioned without further citations.

The emphasis in this report is on placing development environments in the broader contexts of system development, computer supported cooperative work, and prototyping. The discussion of concrete environments is therefore not as detailed as some readers could desire.

The report proceeds as follows: In chapter 2 a conceptual framework for the characterisation of system development is presented. This framework has been developed in the MARS project where the author participated. The framework has been presented at some length here since it is currently not available in English. Chapter 3 presents a brief discussion of different kinds of work, especially of cooperative work. Chapter 4 discusses the role of programming in system development and, related to this, different kinds of prototyping. In chapter 5 another dimension of development environments is presented, this is the application area of the environment, i.e. the kind of products the environment is suited to support the development of. In chapter 6 a move towards the technology is made. Here a number of central technical characteristics of development environments are discussed. Together, the chapters 2–6 present the five dimensions of the framework on development environments. In chapter 7 the dimensions are used to characterise some schools or traditions in this research area. Chapter 7 also contains a description of the Mjølner project. Finally, some conclusions are drawn in chapter 8. Chapter 8 also contains a discussion of some related work.

Chapter 2

The Context of System Development

In this chapter we will present a conceptual framework for discussing system development. This framework has its roots in Lars Mathiassen's thesis *Systemudvikling og systemudviklingsmetode* (System development and system development method) from 1981 [54]. The framework has later been revised and refined in the MARS-project, and it is presented in the textbook on system development written by the MARS project group [2].

This framework is used because an abstract framework for the discussion of system development is needed. A high level of abstraction enables a characterisation of system development processes independently of the methods and development environments being used. This is required in order to discuss the impact induced by development environments on system development practice.

2.1 Levels of description

The framework allows for a description of system development at different levels of abstraction. A distinction is made between the intention with and the actual performance of the work. This is illustrated in figure 2.1. The framework also contains a characterisation of the main components of system development.

The *process* level is the most concrete level of description. A process takes place in time and space, but it is otherwise not restricted. When describing a work process any aspect can be included, also the physical layout of the work-place, characteristics of the persons doing the work,

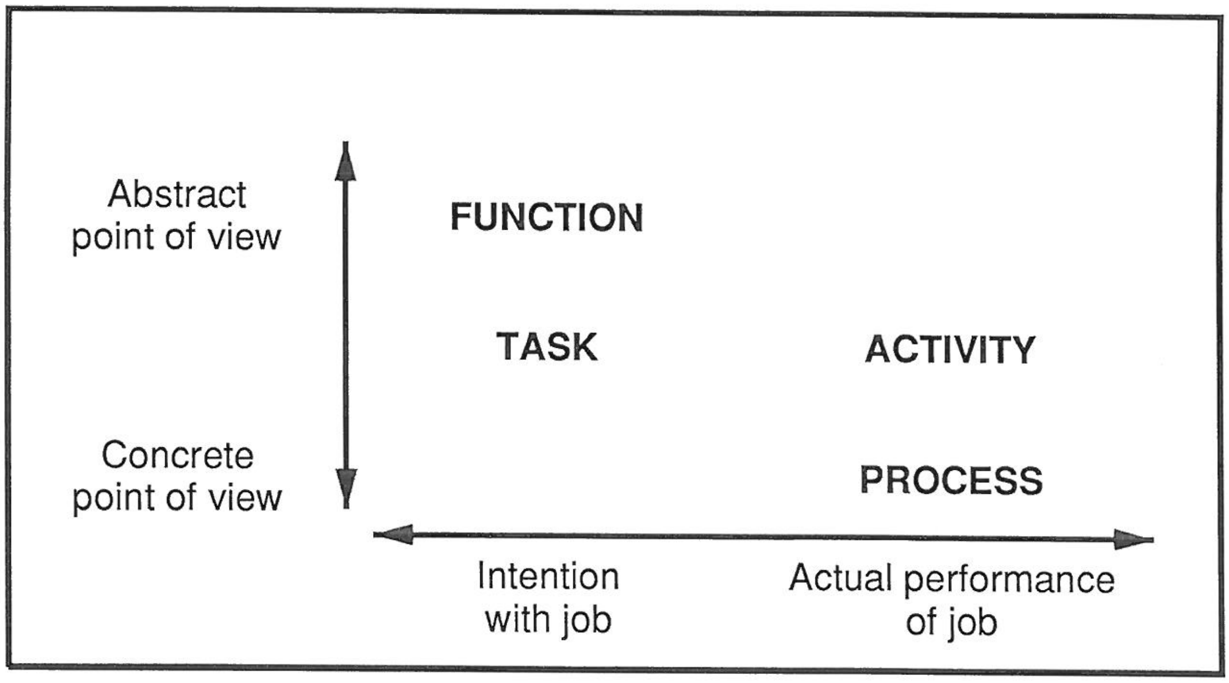

Figure 2.1: Some basic concepts for the description of system development

etc. Thus it is impossible to make a complete description of a process. For this reason it is not very convenient to describe a kind of work at the process level. Processes may be split up in subprocesses if these can be identified as taking place in a restricted part of the time and space of the superior process. On the level of concreteness where we find processes we cannot talk about the intention of the work, but only about the actual performance of the work. In the MARS framework the basic phenomenon is the system development process.

On a more abstract level we can describe the actual performance of the work as activities. An *activity* is a selected part of a process which is identified by its *content*. Examples of activities in a system development process are interviewing the users, programming a module, and revising the project plan. As we will return to in chapter 4 we see programming as an activity which takes place in the system development process.

If we remain on the same level of abstraction as where we find activities, but move to the *intention* with the work, we can identify *tasks*. When work is planned and assigned it is normally done in units of tasks. There is often a one to one relationship between tasks and activities. The programming of a module is a typical unit of work assigned to a person for a given period.

On the most abstract level we can describe the intention with the work as *functions*. A function expresses the intended result of some work with no regard to the actual performance of the work. The functions used to describe a given process are not given. A functional description expresses a theory about a class of processes.

2.2 System development functions

Mathiassen presents a functional description of system development in [54]. He identifies the functions investigation, construction, change, decision, and communication. This was the starting point for the description developed in the MARS-project. Compared to Mathiassen's description there are some differences, however. The MARS description is less abstract and more detailed. It can in fact be interpreted as a description on a functional level and as a description on an activity level. It also represents a shift of perspective from system development as seen from the outside to system development as seen from the point of view of the system developers.[1]

The functional description is based on the systematic application of three fundamental distinctions, see figure 2.2, and the identification of a more loosely defined set of general functions, see figure 2.3. The three fundamental distinctions are:

- product-orientation versus process-orientation,
- reflection versus action, and
- present realities versus visions.

In the framework of the MARS-project the product of the system development process is not only a computer system, it is a computer-based system. A computer-based system includes the computer system as well as those parts of the user organisation which contribute to or depend on the same functions as the computer system. This report is primarily concerned with computer systems.

[1]This is particularly visible when looking at Mathiassen's functions construction and change. The change function is in the new description reduced to one of several minor points under the realisation function. Construction, on the other hand, receives a more detailed treatment, and elements of construction can be found in the design and realisation functions in the new description.

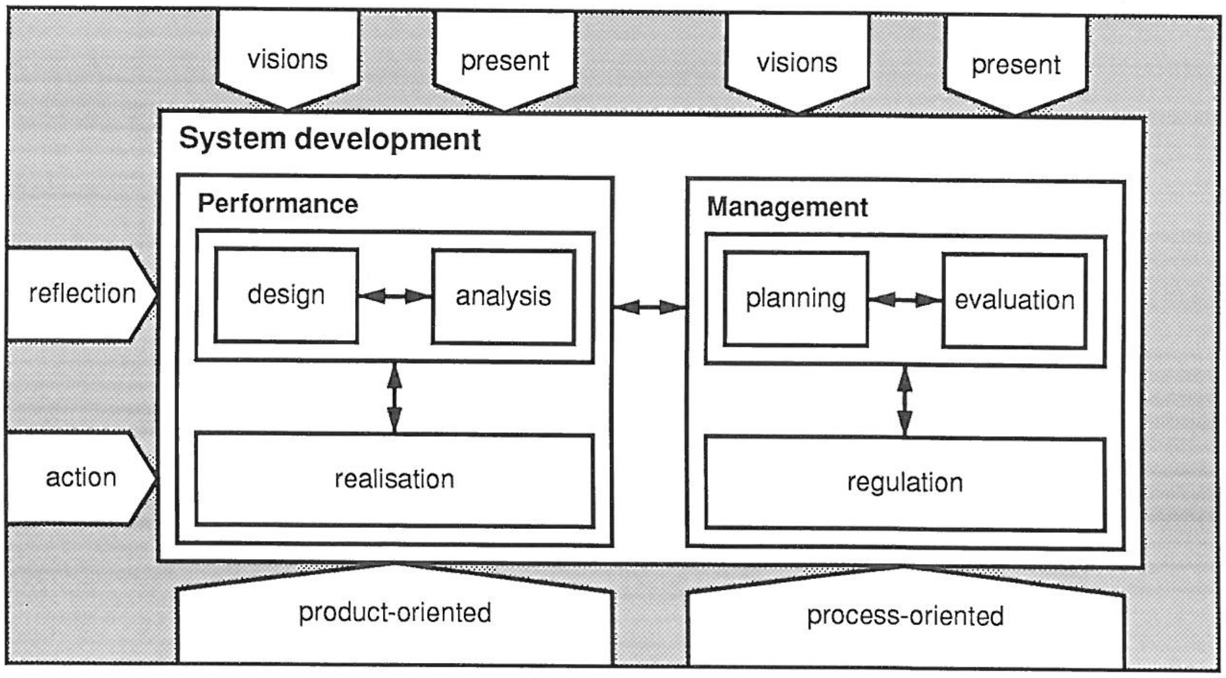

Figure 2.2: The main components of system development

The distinction between process- and product-orientation results in symmetrical descriptions of *performance* and *management*.[2] This illustrates that system development does not only deal with the creation of a product, it also deals with the creation of the process which leads to this product. A major finding in the MARS-project was that there often are severe deficiencies in the management of the process. Due to the nature of the work it is impossible to manage the process from the outside (although attempts are made), and the system developers themselves often lack the competence and the will to do it. Reasonable management is, however, a prerequisite for many other changes and improvements in system development. When addressing computer support for system development it must be stressed that the computer support also should address the process oriented functions.

The distinction between reflection and action serves to stress that although the main purpose is to construct something, there is a great need for reflection. The application of this distinction to the product oriented part of system development results in the identification of well-known functions such as analysis and design as the reflective functions

[2]The word management is here used to classify some abstract functions, and it does not denote a special group of people in the organisation. Everybody participating in a process can contribute to its management.

and *realisation* as the action. On the process-oriented side we can identify a function parallel to the realisation function. This is the function of *regulation.* Regulation is directed towards the process itself. It may be directed at the participants and their expectations, it may also be directed at the conditions and surroundings of the system development process. Regulation is sometimes invisible, for example in joint planning. Here the planning is performed in a way which commits everybody to the plan, making explicit regulation unnecessary.

Finally a distinction is made between reflection directed at present realities and reflection directed at visions of the future. This is the distinction between *analysis* and *design.* On the product-oriented side we get a parallel distinction between *evaluation* ("process analysis") and *planning* ("process design").

The arrows in figure 2.2 indicate a set of relations between the functions. In [2] a number of normative theses are stated about system development, some of these directly address the relation between the different functions. We will quote those theses here:

P 1 Analysis and design are mutually dependent, and should therefore be performed concurrently in order to support each other.

P 2 Product oriented reflection (analysis and design) and realisation affect each other, and should therefore be performed concurrently in order to support each other.

M 1 Evaluation and planning are mutually dependent, and should therefore be performed concurrently in order to support each other.

M 2 Process oriented reflection (planning and evaluation) and regulation affect each other, and should therefore be performed concurrently in order to support each other.

PM 1 A system development project should be organised in a way which ensures direct and close interaction between performance and management activities.

All these theses have implications for how development environments should be constructed. A development environment should not, for example, be constructed in a way which prevents the interplay between

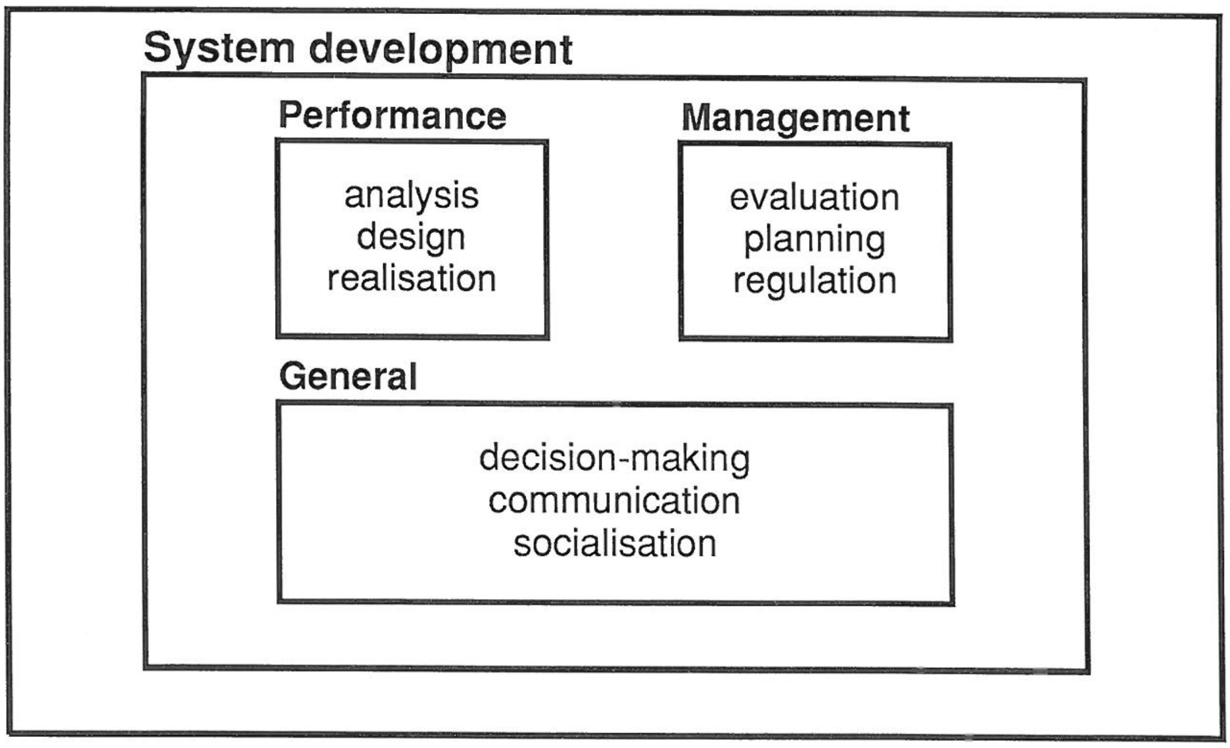

Figure 2.3: The functions of system development

different system development functions. The theses also have implications for how system development work should be organised, see chapter 3.

Finally there is a number of more general functions which have not been covered by the systematic application of the three fundamental distinctions. These functions are *decision-making, communication,* and *socialisation.* They are performed all the time during a system development process. This does not mean that they do not require time and resources, or that they should not be planned. On the contrary some decisions may take a long time, and they need to be prepared and planned. This may severely affect a realistic project plan. Communication requires resources, especially in larger groups. There are examples of projects which did not work well until the deadlines were changed and the project group was reduced to a reasonable size [26]. Socialisation is also critical to system development. People need to know and trust each other in order to work well as a team. It is one of the central recommendations in [2] that care should be taken to give the projects a good start. Planned project establishment is proposed as an activity which can contribute to this end. The nine system development functions are summarised in figure 2.3.

2.3 Support for system development functions

In this section the presented description of system development will be applied to define the first dimension in the characterisation of development environments. This dimension is the system development functions supported by the environment. In the following it will be discussed how the different functions can be supported.

When looking at the product-oriented functions, analysis, design, and realisation, the main kind of support is the support for programming, including editing, translation, debugging, and version control. This will mainly support the realisation function, but if the environment is suitable for prototyping, it may also support analysis and design. When users are confronted with one or several prototypes much previously unstated information about the user organisation is revealed. This may support analysis. Support for the production of various kinds of documents, like analysis and design reports, documentation, user manuals, and teaching material is also an important kind of support for the product oriented functions. Good text processing and facilities for making illustrations are needed. Such facilities will support all product oriented functions. Help in maintaining the relationships between different, in some sense equivalent, descriptions of the product is also needed. Such help will support all the product oriented functions and may also strengthen the interplay between these functions. Of great commercial interest are the environments for support of specific system development methods, for example specialised diagram editors. These will certainly provide some support to the analysis and design functions by making it easier to revise the diagrams.

The realisation function covers more than the "pure" construction of the system. It also covers the introduction of the new system, and as a part of this, conversion. Conceptually this is perhaps a small aspect of system development, but in practice it can be a major bottleneck in the development of new systems. Often much effort is spent in developing code which is to be used only once, and this development is made difficult by the fact that realistic tests are hard to perform. An example of conversion support is the possibility to simulate the old database to unconverted applications provided by some 4th generation languages.

The process-oriented functions, evaluation, planning, and regulation, can benefit from facilities for document preparation in the same way as the product-oriented functions can. There are many products on the market focussing on planning techniques like PERT [25] and on estimation techniques like COCOMO [10]. Another help in planning can be a shared calendar as discussed by Greif and Sarin [32]. Some aspects of project evaluation may be supported by systems keeping track of hours spent, etc., but this is a very small part of the evaluation function. This illustrates the trivial, but important, statement that some support for a function does not mean that it is fully supported. This is sometimes forgotten by system developers when they make the lack of the appropriate planning "tool" responsible for their insufficient planning. In the textbook written by the MARS project group this syndrome is diagnosed as "the myth of the tool".

The regulation function is an expression of the need to change an ongoing process in accordance with the plan. It is very hard to imagine ways to provide computer support to the regulation function except for the effects of communication support which indirectly supports many functions, among them regulation.

The general functions, communication, decision, and socialisation, can also be given computer support. Communication can obviously be supported by facilities for explicit communication like electronic mail, but a shared storage of documents, programs, etc., may also support communication.

The decision function can be supported by decision support systems and by systems making information about the process available for decision making. Meeting support systems, like those developed in Xerox PARC's Colab [70], may also support the decision function. These are examples of support for explicit decisions, but many, perhaps most, decisions are not made in explicit decision making situations. Cohen, March, and Olsen have made a study of how decisions are made in organisations. They describe decisions as coincidental meetings of problems, solutions and people [18]. It is not easy to make computer support for decisions made outside of explicit decision situations, but in general facilities allowing experimentation with alternative solutions to problems, for example in design, will support the decision function.

The socialisation function is, of course, even harder to support explicitly. It is important, however, that the project group is allowed to develop as a group, that it may invent its own practices, etc. The development environment may have impact on these issues by shaping the condition for cooperation in the project group. It is therefore important that the development environment can be adopted to the needs of the group.

The general functions go across the other functions of system development. Therefore their support will also result in support for other functions. This is especially obvious with support for communication. When the examples of support for the process-oriented functions seem "thin", it is because these functions are hard to support directly, but they may receive a lot of support from facilities supporting communication and decision-making.

Chapter 3

Kinds of Work in System Development

In this chapter the second dimension of the framework on development environments will be presented. This dimension is the kind of work supported by the environment. Every computer system is based on some assumptions about the work context where it is going to be used. These assumptions are often implicit. We will distinguish between four kinds of work: *individual, cooperative, bureaucratic,* and *market-organised* work. This classification is inspired by the transaction cost school [15, 16, 57, 69, 81] and by the field of computer supported cooperative work [19, 65]. All these kinds of work can be found in system development. The development environments we mention in this chapter may all support the system development process, but their applicability is restricted to specific kinds of work in system development.

Many computer systems do not fit well with the work context where they are used. Much criticism has been raised of the naive or false assumptions behind these systems. Examples of this criticism can be found in Ackoff's classical paper "Management Misinformation Systems" [1] and in Ciborra's criticism of the data- and decision-oriented views [16]. We can also observe that the use of computer systems have influence on work patterns, but this influence is not deterministic. The same system may have different impacts on different work contexts. Blomberg has described how the same computer system led to different changes in two different organisations [6].

3.1 Individual work

It is not so that individual work is totally different from various kinds of organised work. All kinds of organised work are after all based on the work performed by individuals. Individual work is, however, included in the classification because many environments tend to ignore that system development requires the effort of many people. Such environments focus on programming as an individual activity. Central examples are Interlisp and Smalltalk. One reason for the implicit focus on individual work is probably that these environments have been developed in an exploratory way with the aim of supporting exactly this style of work. In addition Smalltalk has its origin in the Dynabook concept [40], a vision of the kind of individual computer support that could be expected in the future.

The facilities of environments addressing individual work are editors, browsers, "smart" help or correction facilities, etc. Many of these facilities are referred to as "tools". In this report the term tool is used in a more specific meaning. A computer system cannot be a tool in itself. A computer system can be used as a tool if the user considers it as an extension of him/herself. An editor becomes a tool when the user can stop thinking explicitly about how to use the editor and move the primary attention to the object being edited [7, 8]. Tool use is normally individual, but there are examples of collective tools and tool use [66].

3.2 Cooperative work

Computer supported cooperative work has recently received much attention. Cooperative work can tentatively be defined in the following way [65]:

- people work together due to the nature of the task,
- they share goals and do not compete,
- the work is done in an informal, normally flat organisation, and
- the work is relatively autonomous,

where "organisation" refers to the actual organisation of the work unit performing the work. This work unit may of course be a part of a larger, less cooperative, hierarchical organisation. System development has a strong component of cooperative work. It is normally organised in projects. These are relatively autonomous and provide the context of a relatively flat organisation. Using the theses in chapter 2 this way of organising system development can be seen as a consequence of the nature of the work. Thesis PM 1 states that the management of the process cannot be separated from the performance of the work, and the other theses state that the different functions should be performed so that they can support each other. Another argument for the importance of cooperative work to system development is the high uncertainty characteristic to system development [2, p. 57, thesis M 3]. High uncertainty is characteristic to cooperative work [65, 69].

Computer supported cooperative work can be defined as a quality of the *relation* between the context of the work and the computer system [66]. This relation can be of many kinds. Three of these: medium, shared material, and tool are of special interest to computer supported cooperative work.

Facilities for electronic mail can obviously be used as a communication medium. There are some examples of the integration of electronic mail in a programming environment, see the use of mail in Interlisp described by Teitelman [73]. Malone et al. have developed the Information Lens, a tailorable mail system running on Interlisp-D [51, 52]. The system allows for the definition of different letter-types, which can be classified in a class hierarchy with inheritance of properties.

Shared material is of clear interest to system development. One obvious problem is the administration of different versions and alternatives of programs, while still allowing the work to proceed in a relatively informal way. This can be supported by facilities like the Source Code Control System (SCCS) [61] and the Revision Control System (RCS) [76]. Kaiser et al. present an example of a system which maintains the consistency of concurrent updates to different modules of the same system [38]. This work is based on an extension of the Cornell Program Synthesizer. The work on databases for computer aided design data will hopefully result in systems which are well suited to be used as shared material. Katz et al. propose a version server for design data which is based on a distinction between private versions (individual work), group versions (cooperative work), and more official releases (bureaucratic work) [39].

Tool use, be it individual or not, can be very supportive to a cooperative work process. A central example for system development is how environments like Interlisp and Smalltalk can be used to create prototypes for the illustration of a design idea.

3.3 Bureaucratic work

System development does, especially in large and long-lived projects, also have a clear bureaucratic nature. The process needs to be managed carefully to deliver the right products in due time. Thus much of the support for bureaucratic work will focus on the process-oriented parts of the work, for example planning. There are also some bureaucratic aspects on the product-oriented part, for example configuration control. Systems like SCCS and RCS, which restrict and log access to the different modules of the product, can support this aspect of the work.

3.4 Market-organised work

System development projects are often regulated by a set of commercial contracts. Large systems are often developed by several contractors who in their turn use sub-contractors. Dowson presents a system for maintaining control with the different commercial contracts for the development of large systems [24].

The kind of work supported is an important dimension of development environments. Much research in programming environments has focused on individual work. This research has, however, resulted in environments which have many technical properties, for example incrementality, that are badly needed in system development. It appears to be much easier to obtain these properties if a restriction is made to individual use.

This report emphasises the importance of cooperative work in system development. It is possible, however, that the argument about the importance of cooperative work in system development is specific to the Nordic countries. In a comparative study by Friedman et al. of the ICON project it has been documented that the degree of external control, enforcement of methods, etc., in system development work is much higher in the Netherlands than in Denmark [28]. The United Kingdom is found to have intermediate characteristics. Friedman et al. have not found any associated differences in the kind of systems developed or in the quality of the product. This does not mean that it is invalid to focus on cooperative work in system development, but it does imply that one should not consider cooperative work as the only kind of work relevant to system development.

There are also tendencies towards a stronger emphasis on the bureaucratic aspects of system development. Large software buyers like the Department of Defence in the United States tends to make specific requests about how the software is developed. This may include the use of specific quality assurance techniques and other measures which are primarily of a bureaucratic nature.

Chapter 4

The Role of Programming in System Development

This chapter discusses the nature of programming, especially the relationship between programming and system development. An understanding of this relationship is needed to be able to discuss the impact induced by new development environments on system development practice. The role of programming is different in iterative than in phase oriented system development. The development environments may restrict the possibility to practice iterative system development. The discussion in this chapter is used to introduce another dimension in the framework on development environments. This is the *suitability for prototyping.*

Several important references on programming environments do not relate their work to system development, for example the references on Smalltalk [75, 83], Interlisp [73, 74], and on the Cornell Program Synthesizer [60, 72] used in this report. The people behind UNIX have, however, been showing some concern for system development issues. Kernighan and Mashey discuss the support provided by UNIX for the software life cycle [43]. For obvious reasons there are not many published attempts at relating programming to the theory about system development presented in chapter 2. Two master's theses from the Computer Science Department, Aarhus University, have tried to relate programming to the theory on system development presented by Mathiassen [54]. These reports will be briefly reviewed. Thereafter, in section 4.2 programming is defined as an activity in system development. In section 4.3 prototyping will be discussed, especially the way prototyping has impact on the role of programming in system development. This leads to the identification of another dimension in the framework on development environments: the suitability of the environments in supporting prototyping. Finally, in section 4.5, some predictions about the future role of programming are stated.

4.1 Some earlier work

In this section we will briefly review two master's these which have tried to relate programming to theory on system development. These are the theses by Jørgensen and Kammersgaard [37] and by Borup et al. [11].

In their thesis Jørgensen and Kammersgaard introduce a conceptual framework for the characterisation of programming processes [37]. They define three subfunctions of the programming process: formulation, refinement, and realisation. These subfunctions are related to the subfunctions of system development described by Mathiassen. The subfunctions of programming are seen as overlapping with several of the subfunctions of system development. The main strength of Jørgensen and Kammersgaard's work is that they base their discussion of programming on an empirical case, and not on normative theory about how programming should be nor on introspective studies of their own programming.

In this report programming is seen as a part of system development. This becomes difficult when programming is described as a process. As argued in section 4.2 programming is not a subprocess of system development, but an activity. It would, of course, be perfectly valid to consider programming as a process, and then define a set of subfunctions. This would not, however, give much help in a discussion of the dynamic role of programming in system development. There would also be a risk to end up with a characterisation of the programming process which was based on implicit assumptions on the role of programming in system development. The work of Jørgensen and Kammersgaard is therefore valid in its own right, but it is hard to apply in the context of this report.

In Borup et al. a conceptual framework for program development is presented [11]. Program development is seen as a smaller category than programming, taking its starting point in a situation where the programmer knows the purpose of the program, its expected behaviour, and to some extent how it can be implemented (p. 9). Borup et al.'s purpose is to design and implement a programming environment, and therefore they restrict the definition of program development to the parts of the programming activity which can be supported by a programming environment. More intellectual activities like formulation of visions and reflection about how the vision can be implemented are explicitly excluded (p. 9–10).

Borup et al. identify four subfunctions of program development: program construction, program examination, program documentation, and program administration.

When relating program development to system development Borup et al. relate program development to the subfunctions of system development as described by Mathiassen. They conclude that the program development process only deals with the construction function of system development, that it is only concerned with the computer system, and that it is not concerned with the organisation in which the system is to be used (p. 19). Thus the report contains an explicit delimitation from the wider context of system development. The focus is on programming as an isolated activity.

Borup et al. define a program development system as the hardware and all the program development tools used in the program development process. The term programming environment is defined as synonymous with a program development system (p. 22).

This definition of programming environments is very operational for the purposes of Borup et al., but it is clearly too restricted to be used in the discussion in this report. If the definition were applied here it would lead to a strong distinction between programming environments and system development environments, where facilities supporting experimental design and communication would be seen as facilities only belonging to the category of system development environments. Borup et al.'s restricted definition does not allow for a discussion of how the role of programming may change because of the possibilities provided by new development environments.

Another reason for not applying any of the two mentioned attempts to relate programming to system development is that they have used the framework by Mathiassen and not the newer framework from the MARS project. This framework was developed after Jørgensen et al. and Borup et al. wrote their theses. The framework from the MARS project does to a higher degree reflect the perspective of the system developers. This should make it easier to discuss the role of programming in system development.

4.2 Programming as an activity

It is important to be able to express that the role of programming in system development is dynamic. Partially dependent on the properties of the development environment, programming may support different subfunctions of system development. In principle no subfunction is excluded. This is because all system development functions can be given some computer support, and if the development environment is tailorable the programming activity may interfere with all system development functions. Therefore it has little meaning to discuss on a general level which subfunctions are those supported by programming.

By programming we refer to the work of developing programs. Using the concepts presented in chapter 2 we see programming as an *activity* in system development. Programming is not a subprocess of system development since it is not possible to delineate when and where programming takes place in a concrete process evolving in space and time. Examples of subprocesses could be coding or discussions with users. Programming is not a subfunction of system development. Functions have a purpose, and as programming may contribute to many different purposes in system development, we find it unreasonable to identify programming as a purpose in its own right. To see programming as an activity is the only choice if we want to stick to the concepts presented in chapter 2. This characteristic is also close to our intuition.

It is not all programming that takes place within the context of system development. Typical examples are students doing their programming exercises and computer scientists programming as a part of their research. Experiences from these settings cannot tell much about the role of programming in system development, but they may still have important contributions to the discussion. The researchers may, for example, be working on an experimental programming environment facilitating rapid prototyping in a compiling environment.

Some consider systems programming as a kind of programming which takes place outside a system development context. This is not in line with the view of programming presented in this report. From a systems programmer's point of view the users are the application programmers, not the "end-users". This does not reduce the need for documentation, stability, quality, etc. The main difference is perhaps the way the work is organised. System development is normally organised in projects. Systems programming is more like a permanent maintenance activity. One may also expect that much system development in the future may be the development of specialised programming environments for the users. Such system development will be a mix of systems programming and classical system development.

Programming by the users is another example of a kind of programming some look upon as taking place outside the system development context. This kind of programming is not too common yet, but it is seen in research environments where scientists, not computer scientists, make their own programs for the analysis of data and for controlling experiments. This programming activity can be seen as a part of an ongoing system development process where new software is developed by exploration and evolution.

4.3 Prototyping

Christiane Floyd presents an overview of different approaches to prototyping in [27]. She identifies three main types:

- Exploratory prototyping. This approach is very informal, the programmer is playing with ideas on the computer. It is normally performed alone, and could be characterised as introspective programming. Prototypes developed in this way are typically very incomplete, they are only vehicles for some other activity. Such prototypes are typically thrown away.

- Experimental prototyping. In this approach more emphasis is put on a thorough evaluation of the prototype. This approach to prototyping is the one most resembling the use of prototypes in other industries. It can be characterised as use of planned experiments.

- Evolutionary prototyping. This approach, which perhaps should not be referred to as prototyping at all, is based on letting the system evolve through continuous use or testing in a realistic environment. It implies a circular, or spiral shaped, development process.

Floyd concludes her paper by stating that the word prototyping could be dropped entirely without changing the message of the paper. What we do have are different styles of experimentation. Experiments are used to support different functions of the system development process. The distinction between the three types is not very sharp, and different types of prototyping may very well be combined. For the purposes of this discussion, and in order to make the distinctions clearer, we will use the terms *exploratory programming, planned experiment,* and *evolutionary development.*

Exploratory programming

Exploratory programming does not take its starting point in a well formulated vision. Instead the starting point is the playful programmer and his/her computer equipped with a suitable development environment. The programmer typically focuses on his/her own needs for computing support, playing with ideas of what kind of computing support it would be nice to have and with how some of these ideas could be implemented. The vision tends to be formulated on the way. Certainly such endeavours often, perhaps in most cases, do not produce anything useful except a programmer with a better knowledge of the development environment. In fact, exploratory programming is the recommended, and perhaps the only way to learn Interlisp. This style of programming requires that it is easy to modify and extend existing systems; keywords are incrementality and tailorability. This style of programming is hard to describe with the concepts of Jørgensen and Kammersgaard, i.e. as a movement from formulation, via refinement, to realisation, or with the characterisation by Borup et al., i.e. as the construction of a vision. The problem and the solution are in fact developed concurrently.

Exploratory programming is common in teaching situations and in some research. Exploratory programming may be useful in practical system development when it is important to develop new visions. There is, however, a contradiction between the development of a specific product and the free generation of visions. If used in system development exploratory programming will mainly support the design and parts of the realisation function. One can, however, imagine exploratory programming involving the users. Experimenting with a potential computer application together with a user can be a very efficient way to provoke the user to tell more about his/her work, for example explaining why a certain idea would not work. In this way exploratory programming may also support analysis. Exploratory programming with the users may give the users an idea of what can be accomplished on a computer, thus enabling a valuable interplay between knowledge of working practices and technological phantasy. In the UTOPIA project [8, 78], which did not use exploratory programming, experience has been obtained on design *with* users. In UTOPIA the emphasis was on creating a mutual learning process where the system developers and the users learned about the skills and technology of the other group. Clearly some development environments are more suited for such mutual learning processes than other. The "wizardry" style of programming typical to environments like Interlisp may be unsuited in a process where the users are to learn what can be done on a computer.

Exploratory programming may also be used to develop computer support for the system developers, in other words to develop the project's own development environment. In this way any system development function may be supported.

Planned experiment

The planned experiment has, in terms of the system development functions addressed, a narrower focus than exploratory programming. Its main focus is on the design function, more specifically on the evaluation of one or more concrete designs in an experimental setting. It may also support the analysis function by the feedback provided on the prototypes and it will of course support the communication between the system developers and the users. It is assumed that a separate implementation process will follow the experiments. During this process it may turn out to be necessary to change the design. The design function can therefore not be totally supported by use of planned experiments.

Every strategy based on experimentation poses special requirements on the development environment. Conventional programming environments are not very well suited for the development of prototypes. The main problems being the lack of incrementality, i.e. the possibility to make changes incrementally to an existing system, the difficulties in running incomplete programs, and the weak possibilities for experimenting with different ways of interaction.

Planned experiments should be used with an awareness of what one can test in an experimental setting. Göranzon argues that many effects of introducing new technology are long-term effects [31]. Some systems are hard to learn but good to use for those who know them proficiently. Sometimes the patterns of cooperation in the user organisation are changed due to the introduction of new technology. All these aspects are hard to assess in a test set-up, they require long-term real use as the basis for a reasonable evaluation. Planned experiments therefore have their strength in the evaluation of issues which can be observed within a short time span. Care must be taken in the evaluation of an experiment to avoid a bias towards "pure" interface issues and systems that are easy to learn.

Evolutionary development

Evolutionary development is a strategy which has very little to do with the use of prototypes in the normal sense of that word. On the other hand it is the strategy that has the strongest impact on the system development process since it implies a cyclic shift of attention between analysis, design, and realisation.

Evolutionary development may to a varying degree be combined with the other strategies. In one extreme it may look like an iterated waterfall process. In the other extreme it may use exploratory programming and planned experiments to support analysis, design and realisation. Evolutionary development can be very hard to manage. It may be hard to find a point to terminate a cyclic process. The users will not be impatient. They already have a useful system, and they can always find points of possible improvement. This implies that the distinction between development and maintenance may fade away. The main difference will be the frequency with which the system goes through the development cycle. In order to maintain some control of the development process it is important that some evaluation criteria for the system are laid out beforehand. This could, for example, be a set of specifications to be met before the system is considered finished. In this way it is easier to determine that the original goals of the project in fact are met. Care must be taken, however, to formulate these evaluation criteria in such a way that the interplay between analysis, design, and realisation is not disrupted. Predetermined specifications may cause the developers to focus on these and not on the possibilities which they discover during the process. There is therefore an inherent contradiction between the need to control the process and the desire to let ideas evolve during the process. Any set of evaluation criteria for the product represents a compromise on this contradiction.

It is only in an evolutionary process that the system gets tested in real use during a longer time span. This is the optimal solution with respect to getting the most qualified response from the "future" users, and thus also getting the best possible input to the next development cycle.

Evolutionary development also introduces a number of complications. It may be hard to get the users to participate in the evaluation of a prototype, but it may be even harder to make them work using an incomplete, and perhaps erroneous system. In this situation the users may have to put an enormous amount of resources into the development process. Another complication is the difficulties imposed by running test versions of a system on production data. Often this will be simply unacceptable, thus changing the process to a kind of repeated planned experiment. In many cases the way out is to run the old and the new system in parallel, with suitable safe gateways between the two systems, so that the old system will receive all transactions concerning "its" data, and so that the new system can take care of all the new functionality and perhaps perform its own version of the old functionality. The ultimate solution is an environment allowing incremental execution of a changing system, allowing old data — old objects — to be treated in the old way, assuring that data created with the new version of the system will be treated by this version at later occasions. In addition facilities for (semi-)automatic conversion between new and old data are needed.

In an evolutionary process the task of controlling and maintaining multiple versions of the system becomes extra critical. Conversion between versions must be possible, and it must be possible to reconstruct earlier versions.

Evolutionary development is not without its problems. In a company a 4th generation language was used for evolutionary development. It was considered important that final code could be generated for every alternative presented to the users; the use of prototypes for idea generation only was discouraged. One problem in this company was that the prototypes for security reasons were not allowed to manipulate real data. The users *used* the prototypes for actual work, this meant that data had to be reentered when the system was put in operation.

4.4 Suitability for prototyping

The discussion of the role of programming leads to the identification of another dimension in the framework about development environments: *the suitability for prototyping.* Different environments are suited for different development strategies both with respect to the *degree* and *kind* of prototyping. Interlisp and Smalltalk are definitely suited for prototyping, primarily for exploratory programming, but these environments are also useful as prototype generators for planned experiments. These environments are, however, less suited for evolutionary development because their use in evolutionary development necessarily implies that the final product is limited to what can be implemented in these environments. HyperCard [30] will also be useful for planned experiments. Examples of environments well suited for evolutionary development are hard to find, but many 4th generation languages give some support for this kind of prototyping. Christensen et al. have documented, however, that this potential seldom is exploited [14].

Many environments are unsuited for prototyping. One bottleneck is the difficulties in generating testable systems. In a company a new mini-computer based system for handling of customer services in bank offices was developed. One major problem in testing was the parallel development of the new hardware and software, another was the difficulty in making tests with real data. A major part of the system was communication with central book-keeping systems, and special solutions had to be found in order to test functions like a deposit of money without entering erroneous data in the central system.

4.5 The changing role

The role of programming in system development is not static. The main problem in system development was in the beginning the development of programs, then it shifted to be the organisation of the development process, and now it is to understand the work and the organisation where the system is to be used [2, p. 43]. In this view the importance of programming is falling. At the same time different prototyping strategies and development environments extend the role of programming to new system development functions. We claim that the role of programming in system development is partially dependent on the capabilities of the development environment.

Since the lack of suitable development environments is one of the reasons for the low use of prototyping in system development we expect an increased use of prototyping as the development environments improve. Another claim we make is that this will result in an extension of the role of programming in system development to more, perhaps all, subfunctions. At the same time the relative importance of programming in system development may continue to fall since programming productivity increases with improved environments and since the problems of understanding the user organisation accentuates. The old and the new role of programming in system development is sought illustrated in figure 4.1 and figure 4.2. In these figures programming is depicted with grey.

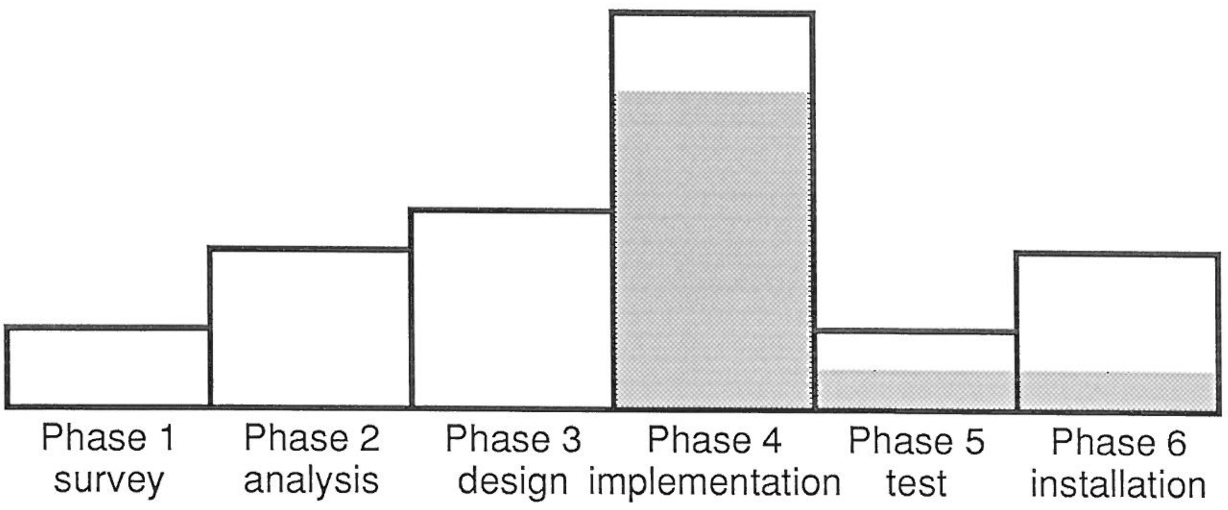

Figure 4.1: The role of programming in the waterfall model

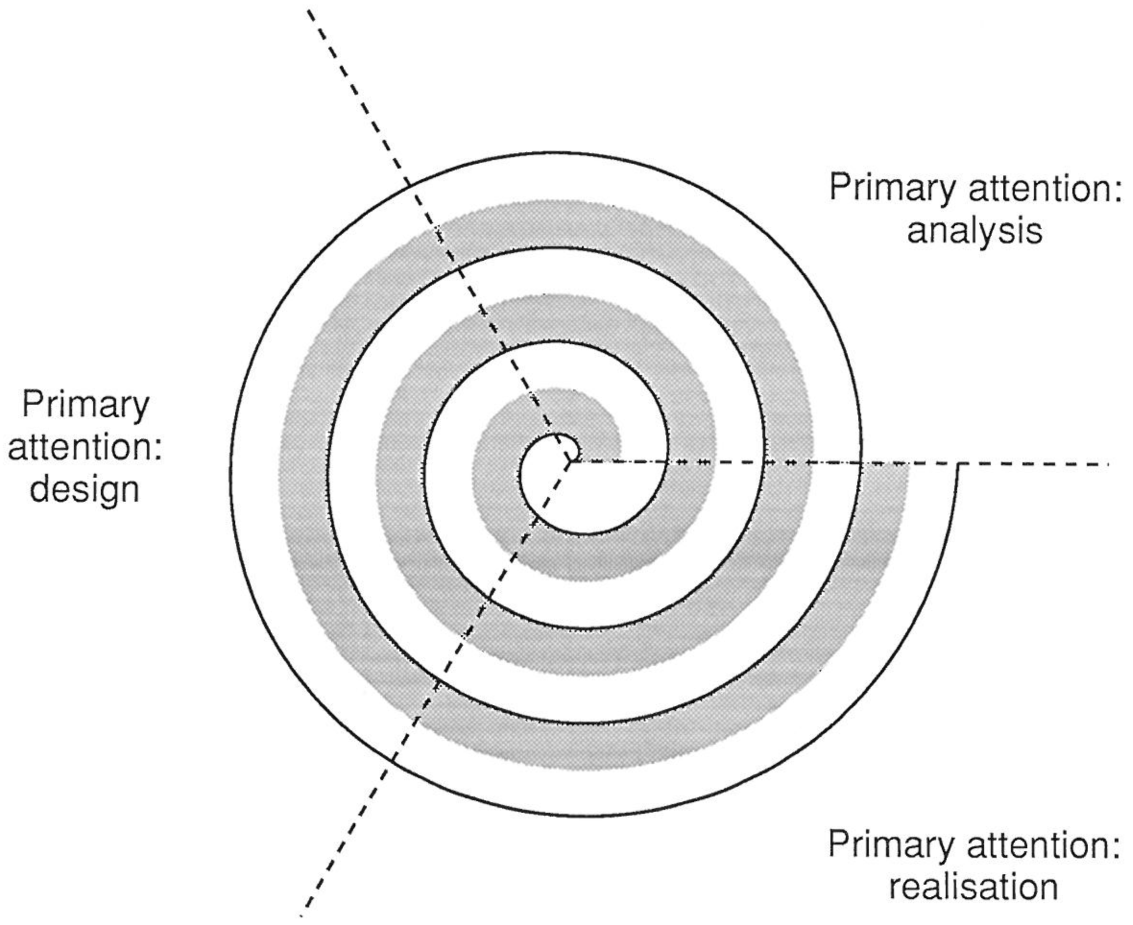

Figure 4.2: The role of programming in cyclic system development

Chapter 5

The Application Area

In the three previous chapters three dimensions of development environments which may be used to describe the work the environments can support have been introduced. In this chapter the product of the work, i.e. the computer systems developed with the aid of the environments, is characterised.

Mathiassen uses the application area as one of three main characteristics of system development methods [54], see also [2]. When discussing and comparing system development methods it is important to be aware of the application areas of the methods since the qualities of the methods are closely related to their application areas. The application area is, however, seldom stated explicitly in the methods. This can be due to commercial reasons or it can be because the authors prefer to believe that their experiences are valid also outside the field where the experiences have been obtained.

When discussing development environments to be used in system development a similar distinction has to be made. It is not very interesting to discuss whether UNIX is better than Interlisp without making it clear what kind of computer system which is to be built. A given environment is only suited for the development of certain kinds of computer systems. We define this to be the *application area* of a development environment.

A thorough discussion of the application areas of various environments would require a comprehensive classification of all kinds of computer systems. This is an enormous task clearly beyond the scope of this report. Two distinctions which may be useful in describing the application area of a development environment will be introduced. The first of these distinctions is whether the system is going to be integrated in a work process or not. The terms work-near and technical systems will be used. The second distinction goes between single-user and multi-user systems.

Typical examples of *technical systems* are an operating system kernel, a traditional compiler, and the software in a disc-controller. The problems these systems are supposed to handle are well understood and can be considered separately from the organisation of the work. The basic requirement to these systems is that they do not fail. It is often possible, and sometimes useful, to make formal specifications of such systems. The development of technical systems often follows a phase oriented approach. Parnas and Clements have argued, however, that there is a need for experimental strategies in this field too [58]. Technical systems are traditionally developed in languages like Fortran, C, Pascal, assembly language, and various specialised languages.

An environment for the development of technical systems should of course provide general services for compiling, debugging, etc. Some facilities are more specific, however. Examples are program provers and support for formal specification techniques. Since the actual surroundings of the product may be hostile for debugging or not even exist at the time of development, it is important to be able to simulate this surrounding. IBM's Virtual Machine operating system makes it possible to develop operating systems on a virtual computer. It makes testing a lot easier and cheaper.

Work-near systems are computer systems which have to fit with the user organisation and with the way the users perform their work. Such systems cannot be seen in isolation and it is therefore necessary to focus on the entire computer-based system [2, 54]. Good examples of work-near systems are a text processor or an inventory system. Systems for the support of cooperative work are also work-near systems. The user organisation and the way the users actually perform their work are often poorly understood. The requirements are often vague. This calls for the use of prototypes, not only to elicit requirements, but also as a part of the analysis of the user organisation.

The development of work-near systems is characterised by a high degree of uncertainty. High uncertainty makes it hard to perform the work according to a specific procedure. Use of prototyping is one way to cope with high uncertainty. The organisation of the development process should also reflect the high level of uncertainty. Using the transaction cost theory [57, 81] one can argue that high uncertainty leads to more use of group organisation or cooperative work in the development process

[69]. Therefore development environments aiming at the development of work-near systems ought to support cooperative work in system development. Mantei and Teorey make a similar conclusion about changes in the system life cycle caused by the incorporation of human factors in the development process, although they use a totally different frame of analysis [53].

The second distinction is between *single-user* and *multi-user systems.* This distinction is central since there are many difficulties which only arise when there are several users. Many of these difficulties are of a technical nature, but there are also many problems in the interplay between an organisation and its computer systems which only arise when multi-user systems are being used.

Some environments are only useful for the development of single user systems, examples are HyperCard [29], MacApp [64], and the many database packages for personal computers, for example REFLEX [59]. It also appears that Smalltalk and Interlisp are best suited for the development of single user systems.

The development of multi-user systems requires the presence of some means to handle concurrency problems. Examples are transaction processing kernels like IBM's CICS and the facilities in many database systems. Such facilities are also provided by most operating systems. Among multi-user systems we can further distinguish between systems supporting various kinds of organisations, i.e. between systems supporting cooperative work, bureaucracies, and market organised work.

Many 4th generion languages can be seen as specialised development environments with a very narrow application area. They typically focus on classical, administrative applications that support market organised or bureaucratic work [69]. Many classical system development methods also focus on markets or bureaucracies, for example by their focus on formal organisation, data flow, etc. Environments for the support of specific system development methods will of course inherit the application area of the supported method. Specialised diagram editors are a good example of method support. We do not know of any environment which is especially suited for the development of systems supporting cooperative work.

There are also some development environments which do not have any application area at all according to the definition given here. These are the teaching environments. The systems developed during education are not going to be put in real use. Some teaching environments have, however, been very influential. A good example is the Cornell Program Synthesizer. Many of the qualities of a good teaching environment are also relevant for "serious" software development, for example facilities for high-level debugging and rapid prototyping. But there are also many properties that are not desirable for a professional user, for example extreme "user-friendliness" and verbose messages. Many teaching environments include a syntax-directed editor. It can be questioned, however, what need a professional programmer has of a syntax directed editor. The developers of teaching environments have also been allowed to ignore many issues that are critical in other contexts, like the ability to handle large programs, integration with existing applications, and efficiency. This implies that the solution strategies chosen in teaching environments cannot generally be transferred to environments for professional use.

Chapter 6

Technical Characteristics of Development Environments

In this chapter the fifth and last dimension of the framework on development environments will be presented. This dimension is the *technical characteristics* of the development environment. Concrete facilities of environments will not be discussed. Instead a number of characteristics which can be used when comparing different environments will be introduced. At the end of the chapter some concepts which the author has found less useful for this purpose will be listed.

6.1 Uniform metaphor

The power of many environments has been achieved by viewing *all* phenomena as instances of *one* general concept or metaphor. Examples of such concepts are lists, objects, and files. We will refer to this as the *uniform metaphor* of an environment.

The uniform metaphor of UNIX is the file. A typical UNIX program produces output on one file according to input from another. All files are considered as sequential files. The uniform metaphor of Smalltalk is the object. The whole system is seen as a collection of objects passing messages to each other. The user's interaction with the system is also regarded as message passing. In a Lisp environment the uniform metaphor is the list. All data and programs are represented as lists. The user's interaction with the system is regarded as passing lists to the system for evaluation. The result of the evaluation is also a list.

Consistent use of a uniform metaphor results in a simple and homogeneous architecture. When the metaphor has been learned it can be used to combine existing facilities in powerful and predictable ways.

The major disadvantage of basing an environment on a uniform metaphor is that the metaphor can be too narrow. This will restrict the application area of the environment. It will also make it difficult to introduce certain facilities in the environment. One example is that it will be very hard to introduce incremental compilation in UNIX. Since everything is represented as files, small changes in a program requires that the corresponding file has to be recompiled. To achieve an effect close to incrementality the program has to be split up in many files. Under any circumstances the file will be the unit of compilation.

6.2 Persistence and sharing

Persistence is the ability to support datastructures across program executions. All development environments must support persistence in some way so that the data (the programs) can be retained between the sessions with the system. If the environment is based on a uniform metaphor the ideal is that the uniform metaphor also is the unit of persistence, since if this is not the case the usefulness of the uniform metaphor will be reduced. In UNIX it is simple to let the uniform metaphor be the unit of persistence, because the uniform metaphor is the file. An alternative to persistence is to make a residential environment. A residential environment is an environment where the primary copy of the program resides in the environment itself as data structures [62]. In a residential environment the whole system is loaded in memory when the user is running. The user's core image is saved between sessions with the system. There are some limitations with residential environments. One is that it typically will be necessary with a supplementary kind of permanent storage, typically textfiles. This kind of storage may be in violation with the uniform metaphor. A second limitation is the difficulty in allowing sharing of data.

By *sharing* we refer to the sharing of data (in this context often programs) between several users. As mentioned in chapter 3 shared material is important in the support of cooperative work. In order to provide shared material the sharing and the persistence must be in natural units. In an environment with a uniform metaphor the sharing should be in terms of the metaphor. UNIX satisfies this criterion. In UNIX there is further support for sharing and version control by facilities like SCCS [61] and the RCS [76]. Dart et al. also point at the importance of persistence and sharing. They use this in their critique of what they call structure-based environments [23]. Among these environments are the Cornell Program Synthesizer. Persistence and sharing are fundamental properties of a development environment. If these properties are not present in the development environment it is very unlikely that they will be provided in the products made with the environment. This means that environments that do not support persistence and sharing have a very narrow application area. This does, for example, apply to residential environments.

6.3 Metaprogramming

Metaprogramming is to write programs that manipulate other programs. Support for metaprogramming is clearly relevant for those who develop the development environment. If a development environment is to be adaptable or tailorable it is necessary that it supports metaprogramming. If the programs are represented according to a uniform metaphor it is easy to support metaprogramming. Such environments are often able to "describe themselves". This capability is especially well known from Lisp-systems where the uniform metaphor — the list — is used for the representation of data as well as programs. This has made it easy to write Lisp programs which manipulate other Lisp programs. In Interlisp this has been used to develop facilities like Masterscope and Do What I Mean [74]. Conceptually the same capability can be found in UNIX, since the uniform metaphor — the sequential file — is used for representing data as well as programs. This metaphor is, however, much weaker than the list metaphor since structure cannot be given a reasonable representation. This makes it much harder to copy the power of Interlisp to UNIX, see the earlier discussion on incremental compilation. The weakness of the uniform metaphor in UNIX can to some extent be compensated with the available utilities for handling textfiles with a given structure. See the description of the `yacc` and `lex` utilities by Kernighan and Pike [44]. In Smalltalk the uniform metaphor is the object. Program code is not, however, represented in an object structure. For this reason the basic means of expression in the Smalltalk language cannot be used to manipulate Smalltalk programs in the same direct way as this can be done in Interlisp.

If facilities for metaprogramming are present, and especially if these are based on a uniform metaphor, it will be possible for the programmers to use the same competence and creativity in the development of the product as in the development of their own computer support.

6.4 Integration

Integration of facilities is important for several reasons. One reason is the need for integration which arises because several facilities are to be used together in an activity. This is the case when a programmer needs to browse existing programs, read design reports, and code a new program at the same time.

A second reason is the need for integration which arises because several related activities are to be performed concurrently. The environment should not prevent this. One example of such activities are editing, compiling, testing, and debugging of programs. It is much to prefer that the debugger is able to point at the place in the source program where something went wrong. In general integration is needed to allow the interplay between various system development subfunctions.

A third reason for integration is that very few systems are developed entirely from scratch. It is crucial to be able to use existing software for standard task like databases and communication. In general one cannot expect all these facilities to be written in the same language. Some sort of object-code or run-time compatibility is therefore needed. Environments like Smalltalk and Interlisp cannot provide this kind of integration. This leads to a severe limitation in the application area of these environments.

6.5 Interactivity and incrementality

Interactivity has for a long time been considered important in development environments. A prototypical example of interactivity in a development environment is the kind of interaction known from residential Lisp systems [62]. In these systems all effects of a change take place immediately. An environment like UNIX is not fully interactive since it still requires the programmer to perform the edit-compile-run cycle for every modification of a program. For a large system this cycle can take considerable time.

The crucial aspect of interactivity in a development environment is *incrementality,* i.e. the ability to perform stepwise modification of a program. We can distinguish between the following kinds of incrementality (adapted from Magnusson and Minör [50]):

- *Incremental editing* means that the whole program does not have to be supplied each time, today a fairly conventional feature.
- *Incremental check of context free syntax* is typically achieved by a syntax directed editor restricting the edit operations to those modifications of the program which are allowed according to the grammar.
- *Incremental check of context sensitive syntax,* often referred to as semantic check, means that it is continuously controlled that variables are declared, types are compatible, etc.
- *Incremental code generation* means that the machine code representation of the program is continuously kept equivalent to the source code.
- *Incremental linking* (and loading) means that a complete executable program is available all the time. Incremental code generation and linking are not very common. They are very hard to implement.
- *Incremental execution* means that it is possible to modify a running or temporarily suspended program. This is certainly useful for debugging purposes and for the execution of incomplete programs. In section 4.3 we pointed at the need for this kind of incrementality in evolutionary development. Another important use is modification of real time systems that must run 24 hours a day.

Incrementality is often, and relatively easily, implemented by interpretation. In Interlisp, which uses interpreted as well as compiled code, new or modified code is interpreted, whereas unmodified compiled code is executed as directly as allowed by the semantics of Lisp. Incrementality is more easily achieved for certain languages, namely languages without blockstructure and static scope. This is because in these languages certain checks always occur at runtime, even when the code is compiled.

The checks needed at "edit-time" are only local. The effects of a small change, however, need not be local. This is only discovered at runtime with dynamically scoped languages.

The usefulness of incremental code generation and linking, without incremental execution, may be questioned. If the computer is fast enough these operation may be performed behind the curtain in virtually no time. What is needed is that all possible errors are detected interactively. In this sense incrementality is a less central concept than interactivity.

When we turn to incremental execution, however, we make a conceptual leap. Here it is not enough to replace incrementality by a fast compiler behind the curtain. The old and the new code need to be related in some way. This implies that the version handling mechanisms in the development environment must be integrated with the runtime system. We do not think that incremental execution can be implemented without any restrictions on the unit of incrementality. More precisely we believe that one should, in object oriented environments, restrict incremental execution to deal with multiple versions of classes, allowing instances of different versions of the same class to exist in the system concurrently. Such a mechanism would, if combined with facilities for persistent objects provide the functionality asked for in section 4.3 on evolutionary development.

6.6 Adaptability and tailorability

There is a growing awareness that computer systems should be *adaptable* to the needs of the users. This for several reasons. One reason is that different users have different preferences with respect to many details which need not be the same for all users. A second reason is that use of computers is becoming a more and more integrated part of human work. This makes it much harder to make a final system before it is taken in use. A third reason is that no work is static. It is therefore necessary to be able to change the system as the work changes. Trigg et al. distinguish between four ways in which a system can be adaptable. They distinguish between flexible, parameterised, integratable, and tailorable systems. They define a system to be tailorable if it "allows users to change the system itself, say, by building accelerators, specialising behaviour, or adding functionality" [77].

Incremental execution is the ultimate solution to the implementation of tailorability. A less radical solution is possible if the system is based on the implementation of a basic model with some primitive operations which can be combined in different ways as need arises [68]. This idea is based on Jackson System Development [36].

The ability to build tailorability into the product is of special interest for the implementation of work-near systems and for the support of cooperative work. It may be used to overcome some of the difficulties in adapting the system to the users' work. In this way tailorability may contribute to an extension of the application area of the environment.

As mentioned earlier tailorability in the development environment may open up for computer support for all system development functions. This may contribute to a changed role of programming in system development. Tailorability in the development environment may also improve the support of cooperative work in system development.

6.7 The programming languages supported

The programming language or languages supported is a central part of a development environment. In order to support development of programs in a given language the environment must have some representation of knowledge about the language.

Some environments are single language environments. Central examples are Interlisp and Smalltalk. For these two examples a distinction between the programming language and the environment is of little interest. The Smalltalk programming language would not be of much value without the Smalltalk environment.

Some environments can be parameterised with the necessary description of a language. The Synthesizer Generator [60] is a generalisation of the Cornell Program Synthesizer [72] which is parameterised with an attribute grammar for the programming language in question. Smalltalk, Interlisp, and the Cornell Program Synthesizer can all exhibit a high degree of integration, largely due to the focus on one programming language or the explicit representation of knowledge about the supported language. In the Cornell Program Synthesizer it is, for example, possible to execute a program in the same units as those used in the editing of the programs.

Some environments are almost language independent. This means that the representation of knowledge about the programming language is spread around at different locations in the environment, typically in the editor, in the compiler, and in the debugger. As a consequence there is little integration between these facilities. Although UNIX in principle is a language independent environment it has a strong bias towards languages which fit well with its uniform metaphor. One such language is the UNIX shell, a language which is strongly geared towards the manipulation of files. Another language especially fit for UNIX is the C programming language. UNIX contains several facilities, for example `yacc` and `lex`, which can only be used with C [44]. The scope rules of C reflect the fact that C programs are to reside in files. The scope of the `static` storage type is the file where the declaration is made. This is very convenient for the packing of related C functions in one file.

The discussion in this section illustrates a dilemma. There ought to be independence between the programming language and the development environment in order to support the integration of programs written in different languages. There should on the other hand be a well-designed relationship between the facilities provided by the language and the facilities provided by the environment. Kristensen et al. propose that the issue of modularisation should be regarded as an issue supported by the environment rather than by the specific language [47].

6.8 Concepts not used

There is a plethora of concepts about development environments. The discussion in this chapter reflects one view of development environments. It therefore contains one selection of concepts about environments. Some concepts excluded from the discussion are:

- Openness
- Flexibility
- User friendliness
- Consistency
- Comprehensibility
- Understandability

- Modularity
- Good design
- Factoring

The concepts selected have been selected according to two criteria: fundamentality and (relative) objectivity. A concept like user-friendliness is too subjective. To compare development environments according to user-friendliness will result in lots of disagreements due to differences in personal taste. The presense of tailorability is much easier to agree upon. Some concepts are more fundamental than others. Openness can be provided by tailorability, hence tailorability is a more fundamental concept. Another example is that consistency, comprehensibility, and understandability can be provided by means of a uniform metaphor.

Chapter 7

Relations between the dimensions

In the previous chapters we have presented five dimensions for the characterisation of development environments. These dimensions are not independent: support for the interplay between analysis, design, and realisation is typically achieved by support for prototyping; sharing of data is central to the support for cooperative work in system development. In this chapter we will use the dimensions to identify some schools or traditions in the area of development environments. Thereafter we will use the dimensions to describe the Mjølner project.

7.1 Schools in development environments

The first school is the tradition represented by Smalltalk and Interlisp, with focus on the product-oriented functions and on individual work. In this tradition there is heavy emphasis on prototyping, especially on exploratory programming. The application area is more diffuse since this tradition does not really address the full context of system development. In terms of the distinctions made in chapter 5 these environments aim at the development of work-near single user systems. These systems will like the development environments be highly interactive. A typical technical property of these environments is a strong uniform metaphor closely connected to one programming language. The environments provide tailorability and incrementality. Systems developed in these environments are hard to integrate with other products.

A second tradition is the class of 4th generation languages. This tradition also emphasises the product-oriented functions, but support for planning is sometimes mentioned in advertising material. These environments typically contain facilities for sharing of programs, version control, etc. There is no specific support for cooperative work, but these environments are not primarily directed towards individual work either. The emphasis on prototyping is not as strong as in the Smalltalk/Interlisp tradition, but these environments can be used for planned experiments and also for evolutionary development. Most of these environments aim at a very narrow application area: classical administrative data processing, i.e. work-near multi-user systems with a bias towards support for bureaucracies and markets. Several studies have shown that these environments put clear limits to what kind of applications which can be built. See, for example, the experiences from the Florence project [5]. In terms of technical characteristics these environments are not very advanced. They are not highly interactive nor very adaptable. Support for metaprogramming is typically absent. These environments typically support a specific 4th generation language, often supplemented with support for an interactive query language. A major strength of these environments is integration. Tasks like screen-layout and database definition are often supported in an integrated way so that the systems developer can refer to specific fields in the database when designing a screen layout.

A third school is defined by the software engineering environments. Often these environments focus on the support for Ada, see, for example, [13, 71]. In this school there is also a focus on support for the product oriented functions, but many also address project management, configuration control, quality assurance, etc. There is a strong tendency, however, to support a purely phase oriented style of development. Clearly related to this is a focus on bureaucratic rather than on cooperative work. This can to some extent be explained by the enormous size of the projects which this school aims at supporting. The role of programming is here seen to be relatively isolated to realisation, since they are based on a view of system development as an engineering discipline, where products are specified before they are constructed. The application area is complex and large technical systems like those embedded in modern weapons. The set of facilities to be included in these environments is very comprehensive, but there is little or no emphasis on facilities which could support group work or prototyping. This is natural since there is a focus on phase oriented development, and, as pointed out by Lennartsson [48], methods based on formal specification are very popular in this school. Method support, or method enforcement, is central in software engineering environments. Dart et al. present a view which we believe is typical for this school. They claim that progress towards the ideal software development environment [23, p. 26]

> "requires a better understanding of the specification and design process and the development of formal methods that appropriately capture information and decisions. A supportive environment must capture and reason about the semantics embedded in the method, and must process information incrementally to assist the designer in exploring design alternatives. A better formal understanding of the derivation of efficient implementations from a specification permits more automation of this process".

Or in other words: Support for the whole system development process can only take place if the environment is based on a model covering all aspects of the system development process. We disagree with this view. We do not think the goal of Dart et al. can be reached without severely limiting the interaction between various activities and without destroying the imagination and intuition needed. Critique of this tradition has also been raised by, for example, Parnas and Clements [58], Ciborra and Lanzara [17], and Hanseth [34].

In addition to these well-known traditions some emerging traditions can be identified. These have a stronger focus on cooperative work.

One of these traditions has emerged from the UTOPIA project [8, 78]. Within the research programme on "computer support in cooperative design and communication" at Aarhus University work is going on to design a development environment based on the experiences from the UTOPIA project [3, 9]. This tradition focuses on design performed in cooperation between system developers and users. The environment which is to be built is described as an application simulator. The application simulator shall facilitate "cooperative envisionment" of future computer applications. There is therefore a strong focus on prototyping, in the terminology of this report primarily on planned experiments. The intended application area is work-near systems.

Another activity in the above-mentioned research programme is to develop support for communication in system development [3, 67]. In this activity system development is seen as cooperative work and it clearly aims at support for the communication function.

In the area of computer aided design much work with relevance to cooperative work is going on. One example is the version server for design data described by Katz et al. [39]. Such facilities are relevant for the support of cooperative work since there is great emphasis on how to handle design data in a way which allows a group to work together on a design. Since computer aided design typically addresses less flexible technologies than software there is little emphasis on evolutionary development. There is, however, emphasis on other kinds of prototyping. Computer aided design focuses on technical products. A central example is the design of VLSI-chips.

7.2 The Mjølner project

The Mjølner project aims at developing an "industrial prototype" of an environment for object oriented programming. The aim is to make an incremental, interactive, and integrated environment that essentially makes the strong properties of environments like Interlisp and Smalltalk available for industrial programming. The main components of Mjølner are compilers for Simula [20] and BETA [46], a metaprogramming system [49], a syntax directed editor [12], a fragment library [45, 47], a program database [33], a window package [22], and an editor for O-SDL diagrams [55]. Mjølner clearly focuses on the product-oriented functions design and realisation, i.e. the functions where programming traditionally takes place. The proposal about a mail handler [67] is so far only a proposal. It is a clear aim, however, to make an environment which can be used outside the laboratory, in actual system development.

No explicit assumptions about the nature of work in system development have been made in Mjølner, but some of the planned components clearly address issues which are of interest to cooperative work. This applies especially to the fragment library and the program database. Mjølner aims at supporting prototyping, inspired by the exploratory style of programming common to Interlisp and Smalltalk. Support for evolutionary development is also aimed for by the emphasis on incrementality. The Mjølner environment will support tailorability in various ways. The syntax-directed editor can be tailored to or specialised for various purposes. The support for metaprogramming provides a general support for user-programmed facilities for the manipulation of programs. The uniform metaphor in Mjølner is the object. Programs are represented as abstract syntax trees. The metaprogramming system defines an object oriented view on abstract syntax trees.

The Mjølner environment can support a variety of programming languages since the facilities in Mjølner are parameterised with a grammar of the supported language. The tailorability in Mjølner is of course in terms if the implementation languages Simula [20] and Beta [46].

Chapter 8

Final remarks

In this last chapter we will discuss some related literature and conclude the discussion of the report.

8.1 Related work

Dart et al. present a taxonomy for software development environments [23]. They identify four categories of environments: language-centered environments, structure-oriented environments, toolkit environments, and method-based environments. Prototypical examples of these four kinds of environments are Interlisp, the Cornell Program Synthesizer, UNIX, and Software through Pictures[1] [80].

Many of the issues addressed by Dart et al. are also addressed in this report. They point at the importance of the data and program representation for the integration of the facilities in the environments. This issue is covered by the discussion of uniform metaphor in this report. Dart et al. also state that commercial use of the language-based environments like Interlisp is restricted to the development of prototypes.

Dart et al.'s taxonomy is based on knowledge of a larger number of environments than ours. In an overview they mention 46 different environments. Their classification is able to put a number of the central examples used in this report in separate boxes, see the prototypical examples above.

Compared to Dart et al. the framework presented in this report is more theoretical. Dart et al.'s work is not based on a theory about system development. They do not address the issue of different kinds of work in system development. They do, however, clearly focus on the applicability

[1]The use of this example is due to Dart et al.

of the environments in actual system development. They use that to criticise the limited applicability of environments such as Interlisp and the Cornell Program Synthesizer. Their central distinction is programming in the small versus programming in the large. They argue that many experiences obtained with environments for programming in the small do not scale to programming in the large. Dart et al. do not discuss the application area of development environments. It appears, however, that they make an implicit restriction to technical systems. Even though they mention a large number of environments they do not include 4th generation languages or PC-development environments like MacApp in their discussion. Thus their scope is much narrower than ours. See also the discussion in the previous chapter of Dart et al.'s view on system development.

Ole Hanseth has made a theoretical investigation on "development of software systems for support of software development processes" [34]. The theoretical basis of his work is very large, its most important parts being Israel on dialectics [35] and Winograd and Flores, and their interpretation of Heidegger, on design [82]. The discussion remains, however, theoretical. There is no discussion of existing environments in the report.

Hanseth discusses factory work and assembly lines. He concludes that system development cannot be organised that way. Instead he considers various theories on cooperation and office work. This leads him to advocate support by providing systems which can be used as media and tools, but he also emphasises the strong institutional nature of the entire development environment. He argues strongly against support for specific methods, stating that "the complexity and dynamics of software development processes make it necessary to provide as much flexibility as possible and to build as few presuppositions as possible about the process into the environment" [34, p. 76].

Hanseth has a strong focus on design. He states that "all tasks (in software development) are *essentially* design work" [34, p. 39, our italics], and he argues that software development is design plus reuse [34, pp. 52–54]. Although we agree that design is underemphasised in current methods and practices we disagree with Hanseth's bias towards seeing everything as design. This leads to ignorance of tedious analysis, the need for disciplined coding, documentation, and almost all process oriented activities. In our view the art of system development lies in finding a balance between the need for more emphasis on design and the need for standardisation, documentation, careful management, etc. Hanseth would not disagree to this since he concludes his report this way: "Our conclusion is that we think that software development in the future will be a dialectic between the analytic and the dialectic approach" [34, p. 77]. The main difference between Hanseth's work and this report is therefore a difference in emphasis. This report has a bias towards programming support. This bias can be explained by the way this report has come to be, especially by the relationship to the Mjølner project. This bias has been justified, however, since it is traditionally in programming where most of the computer support for system development has taken place. A further justification is that new kinds of programming support may lead to a change of the role of programming in system development.

Lennartsson discusses the distinction between programming environments and software engineering environments [48]. He sees programming environments mainly as a research area whereas he considers software engineering environments as an attempt to provide the ultimate solution to phase-oriented system development. He argues that because of the phase-oriented style of work, because of the need for compatibility, and because of the educational level it will be very hard to transfer the research based programming environments to the software industry. With an analogy to expert-system shells and application generators he sees a possible answer in a multi-level approach where the computer specialists develop "tools" which the application specialists use.

Compared to our work Lennartsson uses a more specific definition of programming environments. This results in a dichotomy which is very illustrative in terms of characterising the narrowness of much research and the conservativism of the industry. We find this point interesting, but we have a different view on system development than the view of those Lennartsson refer to talking about software engineering environments. Many of the theses of the MARS project stress that system development cannot be made in a strictly phase-oriented way, and we therefore see many efforts in the development of phase-oriented environments as futile. By our broader class of environments we can also observe that environments which are better suited for prototyping, for example 4th generation languages and some PC-environments, do make their way into industry.

Concerning the multi-level approach we feel closely related to it. In Mjølner there is much emphasis on metaprogramming and tailorability, and the final environment will hopefully be useful for "tool" construction as well as for application development.

8.2 Conclusion

To some people a discussion of development environments is interesting in itself. The discussion does, however, have a wider relevance. Many of the properties of development environments are also desirable in other contexts. The point we make here is that the ability to program and modify should be present in any environment. Sandewall et al. go as far as proposing that the system delivered to the users should be a modified programming environment [63]. This idea becomes even better if the application is written in a profession- or application-oriented language [56]. If this is the case many standard procedures can be implemented as programs in this special language, such that they lend themselves to modification and evolution. Tailorability and incrementality will be important properties of such environments, hence some of the properties we have discussed in chapter 6 are relevant outside the scope of this report. This analogy to other areas than system development has an important weakness, however. Everybody is not interested in or able to learn programming. This is therefore an idea which should be used with care and it should not be used as an excuse for making bad products.

Cooperative work is an important kind of work in system development. This report can be seen as a study of computer supported cooperative work within a specific field. There are also other kinds of work in system development. A definition of cooperative work should therefore not divide the world in two, it must reflect the observation that cooperative work is an aspect of the work which is present at various degrees. Another lesson to learn is that computer supported cooperative work is not only to provide facilities like mail and meeting support, it is just as much to construct the computer system in such a way that patterns of cooperation are not disturbed or destroyed. There are reasons to fear that much effort in software engineering environments will have a negative impact on cooperative work. An example is the tendency to support specific development methods or models. This may result in stricter control and less autonomous work in system development.

Ole Hanseth writes in the abstract of his report: "We will analyze how software development is carried out, just as we analyze an arbitrary field we intend to computerize" [34]. We do not, unfortunately, believe this to be true. With the risk of overestimating this report, we claim that it is not normal that system development is based on such comprehensive analyses of the field to be computerised as Hanseth's and this report. Secondly we are faced with another problem. How do we use the understanding achieved in the analysis when we build systems? We hope that we have succeeded in making some connections between our analysis and the technical properties of the development environments, but we feel that much remains to be done.

In aiming at comprehensive support for system development work we are faced with a contradiction: More comprehensive support may lead to environments which fix the way work is to be performed, see the quotation of Dart et al. in chapter 7. Our problem is that we want to develop support which goes beyond being a set of independent facilities, while at the same time allowing the process to proceed undisturbed.

Since this report represents a mix of two different areas of research it is appropriate to draw some conclusions specific to the two areas involved.

For the system development field a central observation is that new environments will lead to changes in the way system development can be performed. Programming, or "coding", can no longer be considered as a separate discipline which belongs to another field. Also a fortunate development in the possibilities for achieving interaction between various activities in system development may be expected. Prototyping can perhaps be used as an instrument not only in design and realisation, but also in analysis.

Research on programming environments should be related to the context of system development. Many experiences cannot be scaled to industrial programming, and the negative consequences of supporting specific methods have been commented upon. Often programming environments contain strong implicit assumptions about the nature of the work supported, for example its organisation, as well as assumptions about the application area. These assumptions need to be made explicit and taken into consideration. The failures of classical information systems need not be repeated in the development of support for the system development process.

This report has presented a framework for development environments. Five dimensions of development environments have been identified:

- the system development functions supported,
- the kind(s) of work supported,
- the suitability for prototyping,
- the application area, and
- the technical properties of the environments.

The framework will hopefully contribute to the discussion about development environments by pointing out the relationships between issues from the different research fields involved.

Bibliography

[1] Russel L. Ackoff. Management misinformation systems. *Management Science*, 14(4):B–147–B–156, December 1967.

[2] Niels Erik Andersen, Finn Kensing, Monika Lassen, Jette Lundin, Lars Mathiassen, Andreas Munk-Madsen, and Pål Søgaard. *Professionel systemudvikling*. Teknisk Forlag, København, 1986.

[3] Peter Bøgh Andersen et al. *Research Programme on Computer Support in Cooperative Design and Communication*. IR 70, Computer Science Department, Aarhus University, Århus, 1987.

[4] David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors. *Interactive Programming Environments*. McGraw-Hill, New York, 1984.

[5] Gro Bjerknes and Tone Bratteteig. *Å implementere en idé — samarbeid og konstruksjon i Florence-prosjektet*. Florence-report 3, Institute of Informatics, University of Oslo, Oslo, September 1987.

[6] Jeanette Blomberg. The variable impact of computer technologies in the organization of work activities. In *Proceedings from the Conference on Computer Supported Cooperative Work*, MCC Software Technology Program, Austin, Texas, December 1986.

[7] Susanne Bødker. *Through the Interface — A Human Activity Approach to User Interface Design*. PB 224, Computer Science Department, Aarhus University, Århus, April 1987. PhD thesis.

[8] Susanne Bødker, Pelle Ehn, John Kammersgaard, Morten Kyng, and Yngve Sundblad. A UTOPIAN experience: on design of powerful computer-based tools for skilled graphic workers. In Gro Bjerknes, Pelle Ehn, and Morten Kyng, editors, *Computers and Democracy - A Scandinavian Challenge*, pages 251–278, Avebury, Aldershot, England, 1987.

[9] Susanne Bødker, Pelle Ehn, Jørgen Lindskov Knudsen, Morten Kyng, and Kim Halskov Madsen. Computer support for cooperative design. In *Proceedings of the Conference on Computer-Supported Cooperative Work, September 26–29, 1988, Portland, Oregon*, pages 377–394, ACM, New York, 1988. ACM order number 612880.

[10] Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[11] Karen Borup, Kurt Nørmark, and Elmer Sandvad. *EKKO — An Integrated Program Development System.* IR 51, Computer Science Department, Aarhus University, Århus, November 1983.

[12] Karen Borup and Elmer Sandvad. *Editor Specification.* Mjølner report DK-SYS-10, Sysware ApS, Aarhus University, Aalborg University Centre, 1986.

[13] John N. Buxton and Larry E. Druffel. Requirements for an Ada programming support environment: rationale for STONEMAN. In Horst Hünke, editor, *Software Engineering Environments*, pages 319–330, North-Holland, Amsterdam, 1981.

[14] Søren Christensen, Kaj Grønbæk, and Tove Rolskov. *Arbejdsformer under anvendelse af 4. generationsværktøjer.* IR 69, Computer Science Department, Aarhus University, Århus, May 1987.

[15] Claudio U. Ciborra. Information systems and transactions architecture. *International Journal of Policy Analysis and Information Systems*, 5(4):305–324, 1981.

[16] Claudio U. Ciborra. Reframing the role of computers in organizations — the transaction costs approach. *Office: Technology and People*, 3(1):17–38, May 1987. Paper presented at the 6th International Conference on Information Systems, Indianapolis, December 16–18, 1985.

[17] Claudio U. Ciborra and Giovan Francesco Lanzara. True stories and formative contexts in information systems development. In *Information Systems Development for Human Progress in Organizations*, Atlanta, May 1987.

[18] Michael D. Cohen, James G. March, and Johan P. Olsen. People, problems, solutions and the ambiguity of relevance. In James G. March and Johan P. Olsen, editors, *Ambiguity and Choice in Organizations*, pages 24–37, Universitetsforlaget, Oslo-Bergen-Tromsø, 1976.

[19] *Conference on Computer-Supported Cooperative Work*, MCC Software Technology Program, Austin, Texas, December 1986. Proceedings.

[20] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language.* Pub. S-2, Norwegian Computing Center, Oslo, 1967.

[21] Hans Petter Dahle, Mats Löfgren, Ole Lehrmann Madsen, and Boris Magnusson. The MJØLNER project. In *Software Tools: Improving Applications: Proceedings of the Conference held at Software Tools 87*, pages 81–87, Online Publications, London, 1987.

[22] Hans Petter Dahle, D. Menikosy, Georg Ræder, and Terje Rød. *An overview of GUNGNE: the Computer Human Interface for Mjølner.* Mjølner report N-EB-1.3, EB Technology/Norwegian Computing Center, July 1986.

[23] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Software development environments. *Computer*, 20(11):18–28, November 1987.

[24] Mark Dowson. Integrated project support with IStar. *IEEE Software*, 4:6–15, November 1987.

[25] Salah E. Elmaghraby. The theory of networks and management science. *Management Science, Application*, 17(2), 1970.

[26] Preben Etzerodt, Finn Kensing, Bo Bagger Laursen, Kurt Kirkedal Laursen, Lars Mathiassen, Birgitte Nielsen, Jørgen Holm Nielsen, and Carsten Underbjerg. *Systemudvikling i praksis: Regnecentralen af 1979, Århus.* MARS-report 3, Computer Science Department, Aarhus University, Århus, June 1984.

[27] Christiane Floyd. A systematic look at prototyping. In Reinhard Budde, Karin Kuhlenkamp, Lars Mathiassen, and Heinz Züllighoven,

editors, *Approaches to Prototyping*, pages 1–18, Springer-Verlag, Berlin-Heidelberg, 1984.

[28] Andrew Friedman, Jens Hørlück, Harrie Regtering, and Bernard Riesewijk. Work organization and industrial relations in data processing departments: a comparative study of the United Kingdom, Denmark and the Netherlands. Report for the Directorate of the European Community — General Employment, Social Affairs and Education, September 1987.

[29] Danny Goodman. *The Complete HyperCard Handbook.* Bantam Books, New York, 1987.

[30] George O. Goodman and Mark J. Abel. Communication and collaboration: facilitating cooperative work through communication. *Office: Technology and People*, 3(2):129–145, August 1987.

[31] Bo Göranzon. Bakgrunden. In Bo Göranzon, editor, *Datautvecklingens filosofi*, Carlsson & Jönsson, Stockholm, 1983.

[32] Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211, April 1987.

[33] Anders Gustavsson and Mats Löfgren. *Yggdrasil, concepts and programming interface.* Mjølner report S-LTH-17.1, University of Lund/Lund Institute of Technology, August 1987.

[34] Ole Hanseth. *Development of Software Systems for Support of Software Development Processes.* Report nr 803, Norwegian Computing Center, Oslo, November 1987.

[35] Joachim Israel. *The Language of Dialectics and the Dialectics of Language.* Munksgaard, Copenhagen, 1979.

[36] Michael Jackson. *System Development.* Prentice-Hall, Englewood Cliffs, 1983.

[37] Troels Møller Jørgensen and John Kammersgaard. *Et begrebsapparat til karakteristik af programmeringsprocesser.* IR 38, Computer Science Department, Aarhus University, Århus, August 1982.

[38] Gail E. Kaiser, Simon M. Kaplan, and Josephine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 4(6):58–67, November 1987.

[39] R. H. Katz, M. Anwarrudin, and E. Chang. A version server for computer-aided design data. In *23rd Design Automation Conference*, pages 27–33, ACM/IEEE, 1986.

[40] Allan Kay and Adele Goldberg. Personal dynamic media. *Computer*, 33–41, March 1977. Also in [79].

[41] Finn Kensing, Jette Lundin, Andreas Munk-Madsen, Henning Simonsen, Jytte Sørensen, and Mogens Sørensen. *Systemudvikling i praksis: Jydsk Telefon-Aktieselskab*. MARS-report 2, Computer Science Department, Aarhus University, Århus, June 1984.

[42] Finn Kensing, Lars Mathiassen, and Andreas Munk-Madsen. *MARS A Research Project on Methods for Systems Development*. MARS-report 1, Computer Science Department, Aarhus University, Århus, July 1984.

[43] Brian W. Kernighan and John R. Mashey. The UNIX programming environment. *Computer*, 14(4):25–34, April 1981. Also in [4].

[44] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[45] Bent Bruun Kristensen. *The Program Fragment Library: A Preliminary Specification*. Mjølner report DK-SYS-12.3, Sysware ApS, Aarhus University, Aalborg University Centre, November 1987.

[46] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48, MIT Press, Cambridge, Massachusetts, 1987.

[47] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. Syntax directed program modularization. In Pierpaolo Degano and Erik Sandewall, editors, *Integrated Interactive Computing Systems*, pages 207–219, North-Holland, Amsterdam, 1983.

[48] Bengt Lennartsson. Programming environments and paradigms, some reflections. In Henning Christiansen, editor, *Workshop on Programming Environments — Programming Paradigms*, Roskilde, Denmark, 1986.

[49] Ole Lehrmann Madsen and Claus Nørgaard. An object-oriented metaprogramming system. In Bruce D. Shriver, editor, *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences, Volume II Software Track*, IEEE Computer Society Press, January 1988. Also available as PB 236, Computer Science Department, Aarhus University, Århus, November 1987.

[50] Boris Magnusson and Sten Minör. III: an integrated interactive incremental programming environment based on compilation. In *Proceedings of the ACM Symposium on Small Systems, Boston, May 2–3, 1985*, 1985.

[51] Thomas W. Malone, Kenneth R. Grant, Kum-Yew Lai, Ramana Rao, and David Rosenblitt. Semistructured messages are surprisingly useful for computer-supported cooperation. *ACM Transactions on Office Information Systems*, 5(2):115–131, April 1987.

[52] Thomas W. Malone, Kenneth R. Grant, Franklyn A. Turbak, Stephen A. Brobst, and Michael D. Cohen. Intelligent information-sharing systems. *Communications of the ACM*, 30(5):390–402, May 1987.

[53] Marylin M. Mantei and Toby J. Teorey. Cost/benefit for incorporating human factors in the software lifecycle. *Communications of the ACM*, 31(4):428–439, April 1988.

[54] Lars Mathiassen. *Systemudvikling og systemudviklingsmetode.* PB 136, Computer Science Department, Aarhus University, Århus, 1981. Also DUE-report nr. 5.

[55] Birger Møller-Pedersen, Dag Belsnes, and Hans Petter Dahle. Rationale and tutorial for OSDL: an object oriented extension of SDL. *Computer Networks*, 13(4):97–117, 1987.

[56] Kristen Nygaard. User oriented languages. In Roger, Willems, O'Moore, and Barber, editors, *Proceedings of Medical Informatics Europe 84*, Brussels, 1984.

[57] William G. Ouchi. Markets, bureaucracies, and clans. *Administrative Science Quaterly*, 25:129–141, March 1980.

[58] David Lorge Parnas and Paul C. Clements. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.

[59] *REFLEX for the mac.* Borland International, Scotts Valley, California, 1986.

[60] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 42–48, May 1984. Published as ACM Software Engineering Notes 9(3) and ACM SIGPLAN Notices 19(5).

[61] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):363–370, December 1975.

[62] Erik Sandewall. Programming in an interactive environment: the LISP experience. *ACM Computing Surveys*, 10(1):35–71, 1978. Also in [4].

[63] Erik Sandewall, Claes Strömberg, and Henrik Sörensen. Software architecture based on communicating residential environments. In *Fifth International Conference on Software Engineering*, San Diego, March 1981. Also in [4].

[64] Jonathan Simonoff. *MacApp Programmer's Guide.* Technical Report, Apple Computer, 1987.

[65] Pål Sørgaard. A cooperative work perspective on use and development of computer artifacts. In Pertti Järvinen, editor, *The Report of the 10th IRIS (Information Research seminar In Scandinavia) Seminar*, pages 719–734, University of Tampere, Tampere, 1987. Also available as PB 234, Computer Science Department, Aarhus University, Århus, November 1987.

[66] Pål Sørgaard. *A Framework for Computer Supported Cooperative Work.* PB 253, Computer Science Department, Aarhus University, Århus, May 1988.

[67] Pål Sørgaard. *HEIMDAL: a Mjølner mail handler.* Mjølner report DK-15.1, Computer Science Department, Aarhus University, November 1986.

[68] Pål Sørgaard. Object oriented programming and computerised shared material. In Stein Gjessing and Kristen Nygaard, editors, *ECOOP '88 European Conference on Object-Oriented Programming, Oslo, Norway, August 1988, Proceedings*, pages 319–334, Lecture Notes in Computer Science 322, Springer Verlag, Heidelberg, 1988. Also available as PB 247, Computer Science Department, Aarhus University, Århus, May 1988.

[69] Pål Sørgaard. *Transaction supporting systems and organisational change.* PB 248, Computer Science Department, Aarhus University, Århus, May 1988.

[70] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chalkboard: computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

[71] Vic Stenning, Terry Froggatt, Roger Gilbert, and Ellis Thomas. The Ada environment: a perspective. *Computer*, 26–36, June 1981.

[72] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981. Also in [4].

[73] Warren Teitelman. A display-oriented programmer's assistant. In [4].

[74] Warren Teitelman and Larry Masinter. The Interlisp programming environment. *Computer*, 14(4):25–34, April 1981. Also in [4].

[75] Larry Tesler. The Smalltalk environment. *BYTE*, 6(8), August 1981.

[76] Walter F. Tichy. RCS: a revision control system. In Pierpaolo Degano and Erik Sandewall, editors, *Integrated Interactive Computing Systems*, pages 345–361, North-Holland, Amsterdam, 1983.

[77] Randall H. Trigg, Thomas P. Moran, and Frank G. Halasz. Adaptability and tailorability in notecards. In H.-J. Bullinger and B.

Shackel, editors, *Human Computer Interaction — INTERACT'87*, pages 723–728, North-Holland, Amsterdam, 1987.

[78] The UTOPIA project group. *An Alternative in Text and Images.* GRAFITTI 7, Swedish Center for Working Life, Stockholm, 1985.

[79] Anthony I. Wasserman, editor. *Software Development Environments.* IEEE Computer Society Press, New York, 1981.

[80] Anthony I. Wasserman and Peter A. Pircher. A graphical, extensible integrated environment for software development. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto, California, December 9–11, 1986*, pages 131–142, January 1987. Published as *SIGPLAN Notices*, 22(1).

[81] Oliver E. Williamson. The economics of organization: the transaction cost approach. *American Journal of Sociology*, 87(3):548–577, 1981.

[82] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition.* Ablex Publishing Corp., Norwood, New Jersey, 1986.

[83] The Xerox Learning Research Group. The Smalltalk-80 system. *BYTE*, 6(8):36–48, August 1981.