ISSN 0105-8517

A Discussion of Prototyping within a Conceptual Framework

John Kammersgaard

DAIMI PB-169
December 1983

This paper was presented at the GMD/ACM/NCC/SERC Working Conference on Prototyping, Namur, Belgium, October 1983. The proceedings of the conference will be published by Springer Verlag.

A Discussion of Prototyping within a Conceptual Framework

by *John Kammersgaard*
Computer Science Department
University of Aarhus, Denmark.

1. Summary

In this paper the notion of prototyping is discussed. The first part of the paper contains a presentation of a framework[1] used in this discussion. A discussion of programming viewed as a social activity is complex and requires an elaborated set of concepts by which we can express our opinions. The framework provides us with a set of such concepts.

In the second part of the paper prototyping is viewed »through the spectacles« introduced by the framework. This is done partly to express more precisely what is meant by the term »prototyping«, and partly to point out some problems that might rise if the prototyping-literature is used as a guideline for programming. Finally some relevant ideas and experiences from the UTOPIA-project are presented.

To sum up, the paper contains:

(1) A presentation of a framework to be used when discussing programming.

(2) A demonstration of the strength of having a framework when programming is discussed.

(3) A discussion of what is meant by the term »prototyping«.

(4) A presentation of some problems that might rise if the prototyping-literature is used as a guideline for programming.

(5) A presentation of some ideas and experiences from the UTOPIA-project.

2. A Conceptual Framework

Normally vague and unprecise terms are used in discussions of programming and programming techniques. As far as we know, no coherent framework, reflecting our world picture, exists for that purpose. In this section such a framework is presented. The framework is based on a dialectical world picture[2] expressed through the general concepts **process** and **structure** as they are defined by Lars Mathiassen [Mathiassen].

The purpose of the framework is to create a richer language to be used when discussing programming, by providing us with a set of concepts for that purpose. Such concepts are useful when discussing how programming is done in practice, as well as when discussing different programming techniques.

2.1. Basic Concepts

To create a coherent framework, we base it on some basic concepts reflecting our perspective of the phenomenon we consider. In this chapter we will present the concepts **process**, **structure**, and **function**, and we will discuss two different conceptualizations of concepts.

2.1.1. Process and Structure

Systems approaches[3] or process approaches[4] are often used as the basis for describing a phenomenon. Both of these approaches lack the possibility to consider contradictions as a central element. In dialectical thinking contradictions are central. To be able to reflect that in our framework, we introduce a process-structure approach considering contradictions between processes and structures.

The concept *process* is used to characterize properties of a phenomenon which we percieve as related to change; change in time and space, as well as development or transformation.

The concept *structure* is used to characterize properties of a phenomenon which we percieve as fixed and steady. We focus on the temporary stability, at the same time maintaining that the percieved properties are changeable.

On the basis of these two concepts it is possible to choose two different types of **process-structure views** on the relationship between processual and structural properties of a phenomenon.
Focussing on some processual properties of the phenomenon, we can illustrate that

(a) the process affects and changes inferior structures,

(b) the process may affect and change superior structures,

(c) the superior structures may limit and restrain the process.

These relationships are illustrated in fig. 1, where relation (a) is illustrated by the paranthesis, relation (b) is illustrated by the arrow, and relation (c) is illustrated by the broken arrow.

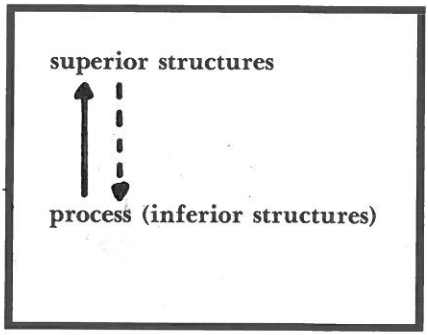


fig. 1:Focussing on a process.

Focussing on structural properties we can illustrate that

(a) inferior processes are going on inside the structure,

(b) the structure is affected by superior processes,

(c) the structure may limit and restrain the superior processes.

Corresponding to fig. 1 these relationships are expressed in fig. 2.

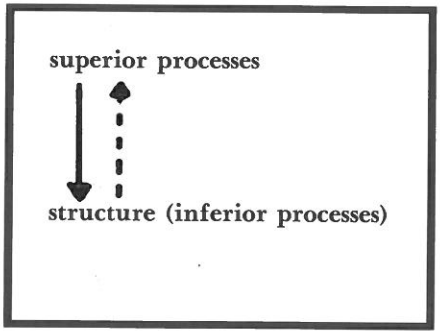


fig. 2:Focussing on a structure

2.1.2. Function

In the framework programming is considered as a work process. Work processes are carried out with certain intentions. This fact is expressed by means of **the intended function**. A *function* is an abstraction that expresses the intended result of one or more work processes, independent of how the processes actually are elaborated and carried out. A work process may contribute in carrying out one or more functions, and a function may be carried out in one or more work processes.

2.1.3. Two Conceptualizations of Concepts

Concepts are central in a discussion of programming because concepts are being realized during the programming process. We need a view on concepts that reflects the fact that ambiguities exist in different interpretations of a given concept.

Generally a concept is perceived by means of its **extension, intension**, and **designation**.
The *extension* of a concept is the collection of phenomena covered by the concept.
The *intension* of a concept is the collection of properties characterizing phenomena in the extension of the concept.
The *designation* of a concept is a common name for all phenomena in the extension of the concept.

Traditionally concept structures have been defined by means of Aristotelian logic. In the Aristotelian view on concepts, the intension is considered as a collection of properties common to *all* phenomena within the extension. As a consequence of this view, concepts can be ordered hierarchically. Furthermore the intension gives a complete definition of the concept, which means that it is always possible to determine whether or not a given phenomenon belongs to the extension of a concept.

The Aristotelian view on concepts does not correspond to empirical observations on human communication and language use [Larsen]. First of all, it is not always possible to determine whether a phenomenon belongs to the extension of a concept. Secondly, different phenomena are more or less typical members of the extension of a concept, which means that different members of the extension might have different intensions.
A good example is the concept vegetables. Is the phenomenon garlic member of the extension of this concept? Probably biologists would say yes, but how about a cook? He would never think of garlic as a vegetable but as a spice.

These observations motivate another view on concepts where the intension is thought of as a collection of properties not necessarily valid for all phenomena in the extension. All that is required is that each property is valid for some phenomena in the extension of the concept.
In this **fuzzy** view on concepts, determination of membership of the extension of a concept is based on assessment, which means that individuals can percieve concepts differently.

The two views on concepts are not incompatible. On the contrary, the fuzzy view is an augmentation of the Aristotelian view in the sense that Aristotelian concepts are covered by the fuzzy view.

2.2. Fundamental Structures

A discussion of programming by means of the fundamental concepts *process* and *structure* requires that we are able to identify and characterize some fundamental structures related to the programming process. In this chapter we will identify and describe some of these structures, namely

- the problem area,
- the techniques,
- the tools,
- the machine,
- the vision.

2.2.1. The Problem Area

Programming requires knowledge. A part of the required knowledge is independent of the intended function of the product, e.g. knowledge about computers, programming languages, operating systems, etc. Another part of the required knowledge depends on the intended function of the product. This second type of knowledge is denoted **the problem area**. Often the product of a programming process is intended to be used to change parts of an organization. In these cases it is valuable to be able to talk about **the object area** as the part of the organization influenced by the change.

Thesis 1: *The problem area* is defined by one or more functions carried out in *the object area*.

From thesis 1 follows that *the problem area* is expressed through technical terms (concepts) related to *the object area*.

Concepts with relation to *the problem area*, which we denote **problem-oriented concepts**, are important because some of these concepts are going to be realized during the programming process. Problem-oriented concepts are not necessarily Aristotelian.

2.2.2. Techniques and Tools

Guidelines for carrying out programming is, among other things, given by means of **techniques** and **tools** [Mathiassen].

A *technique* indicates how a certain work process can be carried out. A *technique* focuses on the performance of a function; it ties knowledge about the product to an understanding of how to carry out the process. *Techniques* ignore to a great extent problems related to the context in which the process is to take place (e.g. ressources, conflicts etc.). Introducing a new *technique* means creating a new type of work process.
By this definition a programming *technique* is a guideline that ties knowledge about programs to an understanding of how to carry out programming processes.

Tools are facilities supporting work processes. *Techniques* make use of a number of *tools* elaborated to support one or more subprocesses. *Tools* are constructed as means to optimize work processes, which means that *tools* are connected to one or more *techniques*.

Thesis 2: The development of *techniques* and *tools* takes place in dialectical interaction.

Thesis 2 expresses that the construction of a *tool* requires a notion about a *technique* as well as the introduction of a new *tool* leads to experiences changing the foundation on which existing *techniques* are formulated.

2.2.3. The Machine

The **machine** is a special tool in the sense that we always use a *machine* during a programming process. By the notion *machine* we do not only mean the physical machine(s). On the contrary, *machine* is a common notion for the available hardware and software.
The *machine* is understood by means of concepts. Among these concepts some can be interpreted automatically (i.e., executed). These concepts are denoted **realized concepts**.

Thesis 3: *Realized concepts* are Aristotelian.

Thesis 3 points out the fact that a single unambiguous interpretation of concepts realized on a computer exists.

Realized concepts are primarilly defined by the programming language(s) used. A secondary source of realized concepts is existing programs (source code) relevant to the programming process.
The machine can be more or less general in relation to the problem area. The generality depends on the amount of problem-oriented concepts among the realized concepts (i.e., the amount of **realized, problem-oriented concepts**).
For many purposes, the so-called high level languages will lead to relative general *machines*, whereas profession oriented languages [Nygaard] will lead to rather specialized *machines*.

2.2.4. The Vision

As pointed out by Dahl and Hoare [Dahl et al.], programming means bridging a conceptual gap between a machine and a **vision** about program executions. Bridging this conceptual gap is done by developing realized, problem-oriented concepts in a process where problem-oriented concepts are defined in terms of realized concepts, thereby giving them an Aristotelian interpretation.
At any time the *vision* denotes our notions about program executions related to the problem area. The *vision* is not fixed; it develops in interaction with the development of descriptions of the *vision* as we permanently seek to bring the descriptions and the *vision* into correspondence.

Thesis 4: Ambiguities and conflicts characterize *the vision*.

Thesis 4 expresses, that normally no agreements on changes in the object area exist.
Descriptions of the *vision* are important documents in a programming process. A lot of different types of such descriptions exist (like "systems requirements", "systems specifications", etc.). All of these are interesting because of their effect on the *vision*. Descriptions which can be executed directly are denoted **programs**. Execution of a program leads to realization of a *vision*.

When we study a specific programming process, we choose to consider a *vision* as **the initial vision**. *Visions* are characterized by their **generality**, **precision**, **form**, and **conceptual distance to the machine**. For *the initial vision* these characteristics are especially interesting.

Generality is relative to the problem area. One *vision* is more general than another, if the problem area of the second is part of the problem area of the first.

Precision has to do with our understanding of the product. The more precise the initial *vision* is, the better is our possibility to imagine the product. Precission increases with the degree of detail, and the degree of formality [Munk-Madsen].

The initial *vision* can be presented in different *forms*. Sometimes it only exists inside a human brain, and sometimes it is presented either orally or in some solid form (e.g., paper, film, magnetic tape, etc.).

The conceptual distance to the machine (i.e., the size of the conceptual gap between the initial *vision* and the machine) indicates how hard it is to express the initial *vision* in terms of realized concepts.

In a discussion of programming it can be useful to distinguish between the **structural** and the **functional** part of the *vision*. The functional part of the *vision* denotes our notion about the function of the product. It is expressed and understood by means of problem-oriented concepts. The structural part of the *vision* denotes our notions about the structure of the product. During the programming process we build up an understanding of the structural part at least partly based on the problem-oriented concepts that we are trying to realize.

2.3. Programmers

A **programmer** is a human-being, who takes actively part when programming is carried out. *Programmers* bring knowledge into the programming process through their skills as well as through investigations made during the process.
Insight into the problem area and knowledge about the vision, the machine and relevant parts of computer science are prerequisites for performing programming. If this knowledge and insight is not present, it must be build up. Knowledge about the vision and insight into the problem area and the object area are normally not present beforehand, so it must be build up during the process. Knowledge about the machine and other relevant parts of computer science is normally brought into the process by the *programmers*, but further investigations within this area might be (and often are) necessary. An important part of these types of qualifications is knowledge about techniques that may be used to bridge the conceptual gap between the vision and the machine.
The above observations lead to the formulation of thesis 5, expressing that the *programmers* build up new knowledge during the programming process.

Thesis 5: The programming process is a learning process.

The sum of qualifications present is important, but also the distribution of qualifications between the *programmers* plays a role, because it sets limitations to how programming can be organized.

2.4. The Programming Process

As mentioned, one way to characterize a process is by the intended *function*. Saying that programming means carrying out the programming function does not tell very much about what is going on. If we want to speak more explicitly about the contents of the programming process, it is insufficient to speak about the programming process as a whole - we need to make distinctions between various subfunctions.

In the following we postulate and describe some subfunctions, which independent of the actual elaboration of a programming process, necessarily must be carried out during the process. Furthermore we use the process-structure diagrams to describe the relations between the structures described in chapter 2.2 and the processes in which the described functions are carried out.

2.4.1. Subfunctions

In choosing subfunctions we intend to capture as much as possible of the totality involved in programming. Furthermore we intend to choose subfunctions, which can be separated conceptually, although they do not have to describe separatable parts of the programming process. Finally we intend to choose subfunctions which relate to substantial parts of the programming process.

On the basis of the above criteria we make two basic distinctions. On one hand we distinguish between elaboration of descriptions using or not using realized concepts. On the other hand we distinguish between elaboration of descriptions that determine either the functional or the structural part of the vision. These distinctions make it possible to postulate **formulation**, **specification**, and **realization** as subfunctions of programming. We define the subfunctions by the scheme shown in fig. 3. So for instance formulation is defined to be a function in which the functional part of the vision is described by means of concepts that are not realized on the machine.

subfunction	description by means of	description of
formulation	concepts not realized	the functional part of the vision
specification	concepts not realized	the structural part of the vision
realization	realized concepts	the structural part of the vision

fig. 3.: Chosen subfunctions

Formulation

Formulation takes place on the basis of a vision. By performing *formulation* the functional part of the vision is elaborated and described by means of concepts that are not realized on the machine. Typically a lot of these concepts are problem-oriented.
Programming requires knowledge. Insight into the problem area and understanding of the object area are prerequisites for performing programming. *Formulation* is carried out to build up this knowledge, thereby emphasizing that the programming process is a learning process.

Formulation is carried out in a wide variety of subprocesses, e.g., generation of systems requirements, communication with professionnals with relation to the problem area and/or the object area and literature studies. It is common to all these processes that the intension is to increase the knowledge of the participants. Furthermore they contribute to the development of the functional part of the vision.

The problem area, the object area, and the vision are all conditioning factors when *formulation* is performed. On one hand the problem area and the object area set limitations to the possible ways in which the elaboration af the functional part of the vision can take place; on the other hand *formulation* might result in demands for changes in the object area, or in a changed understanding of the problem area. The vision restrains *formulation*. On one hand new visions must be consistent with the existing vision, but on the other hand existing ideas might be rejected. By means of process-structure diagrams these relationships can be expressed as shown in fig. 4.

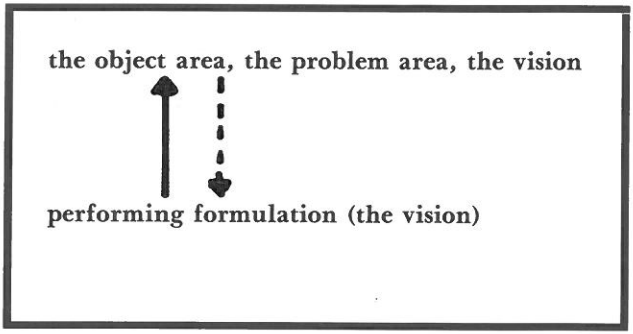


fig. 4.

Specification

Specification takes place on the basis of the functional part of the vision. When performing *specification*, the structural part of the vision is described without use of realized concepts. Typically a lot of these concepts are problem-oriented. *Specification* results in descriptions of how to realize the vision, without using realized concepts.

Conceptualization and elaboration of descriptions are important subprocesses in which *specification* takes place. It is common to all such subprocesses that creativity is required and that the amount of insight into the problem area is important in relation to the quality of the product.

Through *specification* new knowledge is build up, as it becomes possible to understand the structural part of the vision by means of problem-oriented concepts.

The problem area and the vision are conditioning factors when *specification* is performed. On one hand, the problem area restricts the possible ways of describing the structural part of the vision using problem-oriented concepts; on the other hand *specification* might result in a changed understanding of the problem area. The vision restrains the development of the structural part of the vision, but nevertheless *specification* might result in changes in the existing understanding of the vision.
By means of process-structure diagrams these relationships can be expressed as shown in fig. 5.

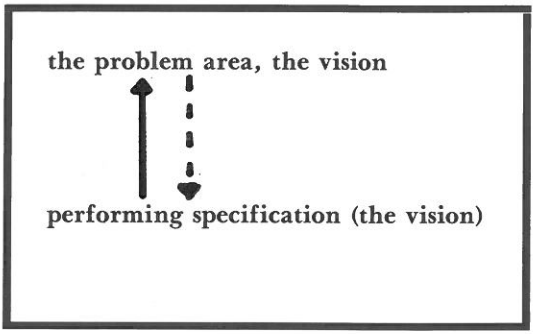


fig. 5.

Realization

Realization takes place on the basis of the structural part of the vision described without using realized concepts. By performing *realization* the structural part of the vision is described by means of realized concepts.

Decomposition and classification [Mark] are essential techniques when *realization* is carried out. Concepts that are going to be realized are identified with realized concepts through classification (e.g., identifying a problem-oriented concept with a procedure). Descriptions of the intension of a concept can be made by decomposition (e.g., creation of a procedure body). Programs and program executions, which are the results of *realization*, are intended to realize the vision.

The machine is an important conditioning factor when performing *realization*. On one hand the machine represents the given realized concepts that can be used when carrying out *realization*; on the other hand the machine is augmented with new realized concepts through *realization*. As for specification the vision represents a conditioning factor.
By means of process-structure diagrams these relationships can be expressed as shown in fig. 6.

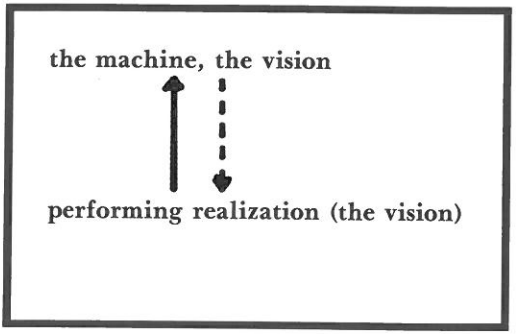


fig. 6.

The Main Direction of the Programming Process

As part of each programming process the intended function of the resulting program executions is **formulated**. Descriptions containing not-realized concepts are developed through **specification**, and finally **realization** results in executable programs realizing the vision.
As indicated above the programming process has a main direction. This is expressed in thesis 6.

Thesis 6: The programming process follows the main direction
formulation ⇝ specification ⇝ realization

The main direction does not indicate a cronological separation of the functions. The main direction indicates: (1) how the importance of the functions displaces as the process goes on, and (2) a time dependence between the functions. As an example it is possible that formulation might take place even at a time where realization is the dominating activity, for instance as an effect of the discovery of ambiguties in existing descriptions of the functional part of the vision.

3. Prototyping

From the way prototyping is presented in the literature[McNurlin, Floyd81, Falster, Söderström] it seems reasonable to claim that the activities involved, with a few exceptions as for instance education of users, are covered by the view on programming presented in section 2. With this observation in mind we can discuss prototyping by means of the framework. This will be done in the following by describing *techniques, programmers, problem area, vision, machine and other tools* in relation to prototyping.
Each of the following subsections consists of four parts. The four parts contain descriptions of:

1) The **intentions** behind the use of prototyping.
2) The **prerequisites** for using prototyping.

3) Some **advantages and drawbacks** related to the use of prototyping.

4) **Experiences** from, and **intentions** in **the UTOPIA-project.**

To support the reader, and to make the strength of the framework more explicit, the concepts defined in the framework are highlighted in *italics* throughout this section.

3.1. Techniques

As pointed out by Floyd[Floyd83] »prototyping« is an ambigous term, which has been used in literature to denote different types of programming *techniques*. In the following we will discuss different alternative *techniques* by focussing on the products intended to be developed during the process.

3.1.1. Intentions

In our view all prototyping-*techniques* suggest that the end-product is produced in a sequence of programming processes, each resulting in a prototype. So, to develop a prototype *formulation, specification*, and *realization* are carried out.
Remark that, although I am aware of the inconsistent use of the term compared to the use within engineering, I use the term »prototype« to denote all computer systems intended to be provisional. This is also inconsistent with the way in which the term is used by Rzevski[Rzevski], who distingushes between prototypes and pilot systems.

Apart from the development of the first prototype, *formulation* is intended to be supported by evaluations of a running prototype. By using the prototype, users can check to which extent *the vision* is *realized*, and at the same time be motivated to *formulate* new *visions*. By means of prototypes, users have the possibility to check whether the *Aristotelian* interpretation of *problem oriented concepts*, made during the development of the prototype, corresponds to their own interpretation.

Contrary to traditional programming processes, *specification* is a subfunction intended to have minor importance when prototypes are developed. *The machine* is expected to contain a lot of *problem-oriented concepts* which make it possible to describe *the structural part of the vision* nearly

exclusively by means of *realized concepts*, making *specification* nearly superfluous.

When prototypes are developed, *realization* is intended to be done relatively fast because of a relatively small *distance between the vision and the machine*.

One way to distinguish between different prototyping-*techniques*, is to consider different intended end-products. The two basic alternatives are:

a) **Production systems**. Development of a sequence of prototypes finally results in a production system, i.e., a computer system ready to be used for production within *the object area*.

b) **Concrete visions**. Development of a sequence of prototypes finally results in a system realizing (parts of) *the functional part of the vision*.

Another way to distinguish between the different *techniques*, is to consider the intended nature of each prototype. Here the following categories seem appropriate:

1) **System sketches**, intending to exhibit selected functional properties of the end-product. System sketches are only intended to *realize* parts of *the vision*.

2) **Models**, intending to exhibit all functional properties of the end-product as they can be formulated without experiments in *the object area*.

3) **Realistic systems**, intending to exhibit all functional properties of the end-product. Realistic systems are based on experiments in *the object area*.

Each of the programming processes carried out to produce the end-product may in principle result in prototypes in any of the three categories. If the end-product is intended to be a *vision*, the following prototyping-*techniques* may be used:

Formulation by sketching
The end-product is a *vision*, and all involved programming processes result in a system sketch. Formulation by sketching is illustrated i fig. 7(b1).

Formulation by modelling
The end-product is a *vision*, and models are produced at least in some of the latter programming processes. Formulation by modelling is illustrated in fig. 7(b2).

Formulation by piloting
The end-product is a *vision*, and realistic systems are produced at least in some of the latter programming processes. Experiments in *the object area* are done in parallel with normal production. Formulation by piloting is illustrated in fig. 7(b3).

Using the terms introduced by Floyd[Floyd83], we can say that use of the above *techniques* leads to **exploratory prototyping**.

If the end-product is intended to be a production system, the following prototyping-*techniques* may be used:

System modelling
The end-product is a production system, and models are produced in at least some of the latter programming processes. System modelling is illustrated in fig. 7(a2).

System piloting
The end-product is a production system, and realistic systems are produced at least in some of the latter programming processes. Experiments in *the*

object area take place in parallel with the existing production system. System piloting is illustrated in fig. 7 (a3.1).

Versioning
This *technique* is similar to system piloting apart from the nature of the experiments in *the object area*. In versioning the production takes place on the »experimental system«. Although versioning covers traditional *techniques*, where systems are used for a period, then changed and so on, we only want to denote versioning as prototyping if other prototyping-*techniques* have been used before the system is brought into *the object area*, and if there is consciousness about the production system as a prototype. Versioning is illustrated in fig. 7(a3.2).

Using the terms used by Floyd[Floyd83] we can say that, use of versioning leads to **evolutionary prototyping**, whereas use of system modelling or system piloting leads to **experimental prototyping**.

In each of the following figures there are different levels, each indicating a state in the development process. At the level **vision**, we are in a state where a *vision* is formulated. At the level **computer system**, the *vision* is realized on a *machine* as a prototype. At the level **pilot experiments**, the computer system is used for pilot experiments, and at the level **production**, the computer system is used for production in *the object area*.
Each figure illustrates moves between the levels as time goes by, thereby describing a *technique*. Shaded areas mean that any of the preceeding *techniques* might be used an arbitrary number of times. The figures are inspired by Bansler and Bødker[Bansler et al.].

fig. 7(b1):Formulation by sketching

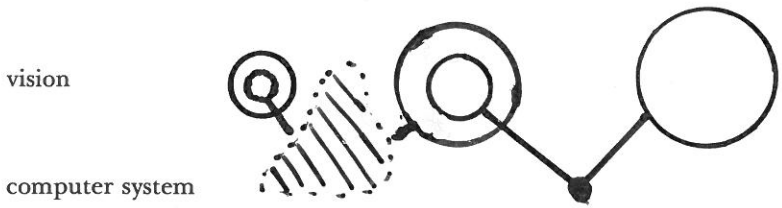


fig. 7(b2):Formulation by modelling

fig. 7(b3):Formulation by piloting

fig. 7(a2):System modelling

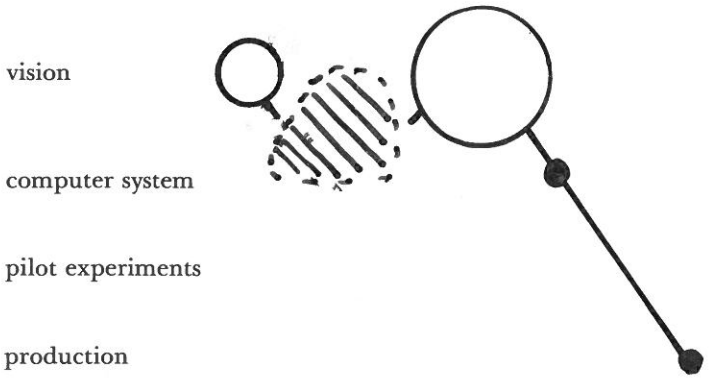


fig. 7(a3.1):System Piloting

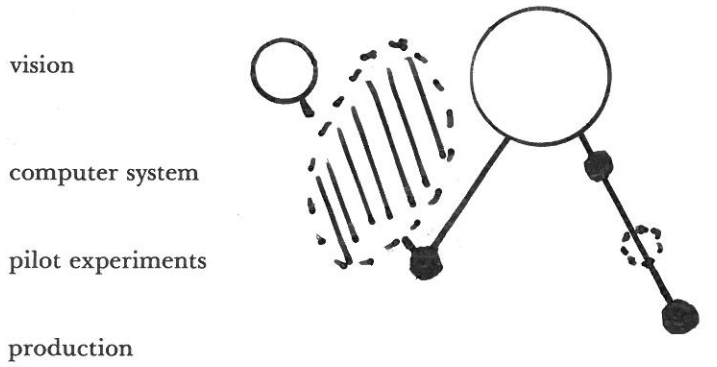
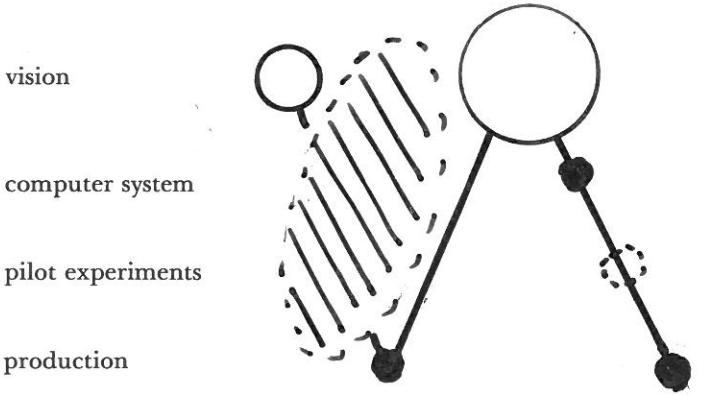


fig. 7(a3.2):Versioning



3.1.2. Prerequisites

All the described prototyping-*techniques* require some kind of user involvement. **Formulation by sketching** may for instance be done with rather few users involved, whereas **versioning** requires involvement from all users working in *the object area*.

Requirements to the organization are also different for the different *techniques*. Most organizations will be able to do **formulation by sketching**, whereas production with unfinished versions as required by **versioning** might cause a lot of changes in *the object area*, making this *technique* unsuitable for many organizations.
A detailed work on organizational prerequisites for prototyping is done by Nosek[Nosek].

Techniques, by which **production systems** are intended to be developed, require *tools*, which make it possible to keep a reasonable program structure during the process, whereas *techniques* intended to produce **concrete visions** may be used without such *tools*.

3.1.3. Advantages and Drawbacks

A main advantage of prototyping is the possibility to let *formulation* be supported by evaluation of running prototypes. By doing that, users can get a more concrete understanding of the proposed computer-system, making it realistic that users and data processing professionals can build up knowledge by common efforts.
Floyd[Floyd81] points out that no *techniques* are known to do such evaluations. In our opinion no formal *techniques* for doing that can be given, as too many individual considerations must be taken. Interviews and dialogues seem to be sufficient.

Only by bringing the system into *the object area*, all functional demands can be tested. First of all, because some functional demands may not show up before work organization is tried out; secondly, because some functional demands like response time, security, etc., cannot be realistically evaluated unless the system is running in its real environment. Finally, as pointed out by Helms Jørgensen[Helms Jørgensen], users are less likely to notice shortcommings in the system, if experiments are made in artificial environments. These observations point out that some **piloting** and/or **versioning** must be done.
A main drawback is that *techniques* for prototyping in *the object area* are nearly unknown. At least no attention is paid to such *techniques* in literature.

3.1.4. UTOPIA:Experiences and Intensions

In the UTOPIA-project we have done **formulation by sketching** to be able to *formulate* more precise *visions* about isolated parts of computer-based systems for newspaper production. In the comming year we will continue doing that, especially in relation to the interaction between man and machine. Up till now the experiences are very limited because of the lack of a *machine* close to *the problem area.*

A technique denoted **stepwise structuring**[5] has shown to be helpful in our experiments. To be able to do experiments as easy and fast as possible, we have chosen not to structure information more than required for a given experiment. This means that some information only is represented as bit-maps, making a lot of operations extremely troublesome; but some fundamental aspects can be tried out anyway. As more insight into the problems is gained, we can do structuring stepwise by going from bit-maps to text representation, and maybe even to more complex data structures, allowing some operations to be done easily, but at the same time restricting the possibility to do other operations.

Some kind of **piloting** is intended to be done during the comming year. In the project we have worked out descriptions of demands on parts of an integrated text- and image processing system for newspaper production. These demands are at least partly taken into account in the development of a commercial system called TIPS[6]. During the next year the TIPS-system will be tested on different newspaper plants in Sweden, and the UTOPIA-project is guaranteed to be allowed to make experiments with the organization of work. In this way we hope to be able to *formulate visions*, which we were unable to imagine without bringing the system into *the object area.*

3.2. Programmers

3.2.1. Intentions

One of the characteristics of prototyping is the intention to let users act as *programmers*, although data processing professionals normally are intended to be involved too. A *machine* very close to *the vision*, makes it possible for the users to participate when performing all subfunctions, whereas more general *machines* might allow them only to participate when *formulation* is carried out.
The degree of user involvement also depends on the prototyping-*technique* used. *Techniques*, in which the intended end-product is a *vision*, make it possible for the users to participate in performing all subfunctions, whereas *techniques*, leading to a production system, reserve some of the necessary *specification* and *realization* to the data processing professionals.

3.2.2. Prerequisites

Data processing professionals with other qualification than usual today are required, because prototyping leads to a situation where more discussions with the users and less coding is needed. The primary qualifications required from the data processing professionals are the ability to discuss, question, and re-think *visions formulated* by the users. Furthermore, as pointed out by Helms Jørgensen[Helms Jørgensen], it is required that data processing professionals are willing to discard their own work, if experiments show that the users want a completely different system.

When using prototyping, users are not required to be able to give a complete *formulation* of their *visions* in one shot. Instead users are required to be able to evaluate running prototypes, and to *formulate* new *visions* on the basis of the experiences gained. More extensive user involvement might even require that users are familiar with *the machine*, as well as with some *techniques* and *tools*.

3.2.3. Advantages and Drawbacks

A main advantage of prototyping is the intention to let users participate during the programming process, making it possible to build up knowledge by common efforts from users and data processing professionals, but a drawback in literature on prototyping is that no explicit considerations is taken as to which users to involve, and how to handle contradictions in user demands. According to literature, managers might represent users giving no real influence to the groups of people affected by the systems developed. Furthermore, literature ignores the possibility of conflicts in user demands.

Another serious point is the way prototyping affects the work of the data processing professionals. It is not possible to predict what will happen yet, but some possibilities can be pointed out.
As *machines* directed towards specifik *problem areas* are being developed more and more work in application development will become almost trivial. If some data processing professionals are supposed only to do that kind of work, then prototyping leads to dequalification. Another possibility is that users will take over a great deal of application development, leaving development of *tools* as a job for data processing professionals. Finally, as pointed out earlier, prototyping might be used by users and data processing professionals together, leaving a lot of that kind of work to the data processing professionals.

The conclusion is that it is too early to say how prototyping will affect the work of the data processing professionals. It will be extremely dependent on the division of labour within the organizations using prototyping.

3.2.4. UTOPIA:Experiences and Intentions

In the UTOPIA-project skilled workers participate. As no *machine* directed towards *the problem area* is present, the graphical workers only participate in the evaluation of prototypes and in the *formulation* of *visions* based on these evaluations. The rest of the programming is done by computer scientists and computer science students.

3.3. The Problem Area

3.3.1. Intentions

According to literature prototyping is intented to be used in connection with all types of *problem areas*. As the basic idea behind prototyping is to support *formulation*, the approach will be especially useful in connection with *problem areas* where *formulation* of *visions* cannot be done without a lot of investigations.

3.3.2. Prerequisites

Prototyping requires a *machine* close to the vision. This means that prototyping is hard to do without a *machine* directed towards *the problem area*.

3.3.3. Advantages and Drawbacks

The majority of experiences with prototyping have up till now been gained in the development of computer-based systems for information storage and retrieval. This fact can be explained by the above-stated prerequisite.
Due to the development of information management systems[7] with a lot of facilities like file-management, security control, recovery control, screen layout design, sorting, non-procedural query-languages, and report generators, this prerequisite has been fulfilled to an extent that has made prototyping possible in relation to *problem areas* containing potential *visions* about information storage- and retrieval systems.

In relation to most other *problem areas* nearly no experiences in developing commercial systems have been gained yet, but some experiments in developing so-called expert systems have been going on within research laboratories[8].

A main drawback is the lack of *machines* directed towards special *problem areas*. As explained below, this is one of the main problems in the use of prototyping within the UTOPIA-project. To solve this problem, research in[9] and development of profession oriented languages is necessary.

3.3.4. UTOPIA:Experiences and Intentions

In the project, where *the problem area* is connected to printing and setting, we are using prototyping to support the *formulation* of *visions* about alternative electronic text- and image processing systems. The main problem in doing that is the lack of a *machine* directed towards *the problem area*, and the lack of a powerful programming environment. In the current situation we have to develop *the machine* ourselves making prototyping very cumbersome.

3.4. The Vision

3.4.1. Intentions

Traditionally a lot of descriptions of *the vision* are made on paper. When using prototyping it is intended that the different prototypes shall represent the (only) materializations of *the vision*. Furthermore it is the intention that *formulation* of new *visions* shall take place when evaluating *realizations* of existing *visions*.

3.4.2. Prerequisites

When developing production systems, prototyping requires, either that the *structural part of the initial vision* can be neglected, or that very powerful *tools* are available. The reason for that is, that it must be considered very troublesome to take structural considerations when production systems evolve from a series of prototypes, if traditional *tools* are used. As neglection of *the structural part of the vision* might lead to products that are hard to develop further, development of more powerful tools is necessary, if we want to use prototyping to develop production systems.
Another prerequisite is a small *conceptual distance between the vision and the machine*.

3.4.3. Advantages and Drawbacks

A main advantage of prototyping is the possibility of visualizing some aspects of *the realized vision*, making a concrete understanding of *the vision* possible earlier than when using traditional *techniques*. Furthermore *initial visions* in a non-solid *form* and with a low degree of *precision* may be handled without having to write a lot of documents.

Getting a lot of spontaneous reactions on a prototype seems to be an advantage at first sight, but a possible drawback is, that this strategy will lead to a situation where noone thinks about the long-term consequences of the proposed system.

3.4.4. UTOPIA:Experiences and Intentions

Formulation of *visions* on the basis of evaluation of prototypes developed by ourselves has, as described earlier, only been done as **formulation by sketching** in relation to very limited parts of our *visions*. Our *formulation* has instead been based on studies of different commercial systems, and evaluation of aspects tried out on "wood and paper prototypes", i.e., mock-ups.
These *techniques* have made it possible for us to *formulate visions* on a general level on systems for page make-up and systems for image processing. To be able to *formulate* more precise *visions* we intend to do **formulation by sketching**, especially in connection with our discussions about man-machine interaction.

3.5. The Machine and other Tools

3.5.1. Intentions

Prototyping intends to reduce the number of necessary *tools* by letting *the machine* support *formulation* as well as *realization*.

3.5.2. Prerequisites

Although prototyping might be done with traditional computer systems supporting file-handling, compiling, text editing, etc., in a non-integrated way, the new integrated programming environments, removing a lot of the troublesome administrative programming jobs, will make the approach more attractive. By using such systems a lot of *tools* are integrated into a single *tool* - the computer making automation of a lot of the necessary work possible.

3.5.3. Advantages and Drawbacks

Given a *machine* directed towards the *the problem area*, an integrated programming environment and a reasonable environment for experimenting, prototyping has the potential to support the integration of development and use. The main drawback is that these prerequisites at present only are fulfilled within very restricted areas.

3.5.4. UTOPIA:Experiences and Intentions

Formulation by sketching is done on a Perq graphical work-station. Using this work-station requires a lot of programming to create a *machine* close to our *visions*. Furthermore the programming environment lacks a lot to support us well, first of all because no integration of the different *tools* is done, but also because some of the *tools*, as for instance the debugger and the source-code library facilities, are of poor quality.

4. Conclusions

To conclude this paper we relate to the purposes. First of all prototyping is presented by describing intentions and prerequisites using the framework. By doing that we also intended to demonstrate the strength of the framework - we will leave it to the readers to judge whether we were successful or not.
Secondly we pointed out some advantages and drawbacks. The main drawbacks were:

- The lack of explicit considerations in literature as to which users to involve, and how to handle contradictions in user demands.
- The lack of machines and other tools to support prototyping when realizing visions about other systems than information storage- and retrieval systems.
- The lack of techniques for prototyping in the object area.

The main advantages of prototyping pointed out were:

- The possibilities of letting users participate in performing all sub-functions.
- The possibility to integrate development and use.
- The possibility to build up knowledge by common efforts from users and data processing professionals.
- The possibility to give users a concrete understanding of the proposed computer-system.

5. Notes

[1]: The framework represents a summary of the substantial part of a master-thesis by John Kammersgaard and Troels Møller Jørgensen [Jørgensen et al.].

[2]: The dialectical world picture can be expressed by the following three rules

reality is an organized and structured totality,

reality is undergoing a continous process of transformation,

contradictions are central and incentive for the process of change.

[3]: See for instance »System description and the Delta language«, Holbæk-Hanssen et al., Norsk Regnesentral, Oslo 1975.

[4]: See for instance [Munk-Madsen].

[5]: This designation is due to a research group at the Software System Research Center, Institute of Technology, Linköping, Sweden. See »Stepwise Structuring - A style of Life for Flexible Software«, Sandewall et al., Proceedings of the 1983 NCC Conference, IEEE 1983.

[6]: The TIPS-system will be marketed by Liber System AB.

[7]: As for example ADMINS devoloped at MIT.

[8]: As for example the diagnose-system MYCIN and *the machine* on which it is based - EMYCIN.

[9]: Some such research will be done within the SYDPOL research-project. A project description is to be published by NORDFORSK, the Nordic co-operative organization for applied research.

6. Literature

Bansler og Bødker: Experimentelle Teknikker i Systemarbejdet, (Danish) Master-thesis, Comp. Science Dept., University of Copenhagen, 1983.

Dahl, Dijkstra and Hoare: Hierarchical Program Structures, in: Structured Programming, Academic Press, 1972.

Falster: Eksperimentel Systemudvikling - 80'ernes edb Fremgangsmåde? (Danish), Technical University of Denmark, 1981.

Floyd: On the Use of »Prototyping« in Software Development, HP3000 Int. user meeting 1981.

Floyd: A Systematic Look at Prototyping, Proceedings from the GMD /ACM/NCC/SERC Working Conference on Prototyping, Springer Lecture Notes, 1984.

Helms Jørgensen: On the Psychology of Prototyping, Proceedings from the GMD/ACM/NCC/SERC Working Conference on Prototyping, Springer Lecture Notes, 1984.

Jørgensen og Kammersgaard: Et Begrebsapparat til Karakteristik af Programmeringsprocesser (Danish), DAIMI IR-38, Comp. Science Dept., University of Aarhus, 1982.

Larsen: Egocentrisk tale, Begrebsstruktur og Semantisk Udvikling (in Danish), Nordisk Psykologi 1980, 32(1).

Mark: Repræsentation (in Danish), DAIMI IR-40,Comp. Science Dept., University of Aarhus, 1982.

Mathiassen: Systemudvikling og Systemudviklingsmetode (Danish), DAIMI PB-136, Comp. Science Dept., University of Aarhus, 1981.

McNurlin: Developing Systems by Prototyping, EDP-analyzer vol 19,9, 1981.

Munk-Madsen: Systembeskrivelse med Brugere (Danish), DUE-note nr. 9, Comp. Science Dept., University of Aarhus, 1978.

Nosek: Organization Design Choices to Facilitate Evolutionary Development of Prototype Information Systems, Proceedings from the GMD/ACM/NCC /SERC Working Conference on Prototyping, Springer Lecture Notes, 1984.

Nygaard: Systemspecialistenes Situasjon i 1980-årene (Norwegian), Data nr. 6, 1979.

Rzevski: Prototypes versus Pilot Systems: Strategies for Evolutionary Information System Development, Proceedings from the GMD/ACM/NCC/SERC Working Conference on Prototyping, Springer Lecture Notes, 1984.

Söderström: Experimentel Systemudveckling - ESU som Metodik (Swedish), Data nr. 3, 1982.

UTOPIA: The UTOPIA-project - On Training, Technology and Products Viewed from the Quality of Work Perspective, Comp. Science Dept., University of Aarhus, 1982.