Computer   semiotics

Peter Bøgh Andersen

Department of Information and Media Science, University of Aarhus, Niels Juelsgade 84, DK-8200 Århus, Denmark.

Abstract

This paper presents semiotics as a framework for understanding and designing computer systems as sign systems. Although semiotic methods can be applied to all levels of computer systems, they view computer systems under a particular perspective, namely as targets of interpretations. When we need to see computer systems as automata, semiotics has little to offer. The main focus of the paper is semiosis, the process of sign formation and interpretation. The paper discusses different semiotic paradigms, and advocates the European structuralist paradigm in combination with the American Peircean tradition. Programming is described as a process of sign-creation, and a semiotic approach to pro-gramming is compared to the object-oriented method. The importance of the work situation as a context of interpretation is emphasized

Keywords: Semiotics, design, work language, object-oriented programming, computer-based sign

1.   Introduction

Semiotics, "the science of the life of signs within society" as Saussure (1966) defined it, is a general theoretical framework for analyzing and understanding a diverse range of phenomena: language, film, theater, pictures, architecture, clothings, gestures, etc. Their common denominator is that they are used as signs; they stand for something else than themselves.

   This paper sketches a possible discipline of computer semiotics (Andersen 1990a, Figge 1991). It is a discipline that analyze computer systems and their context of use under a specific perspective, namely as signs that users interpret to mean something.

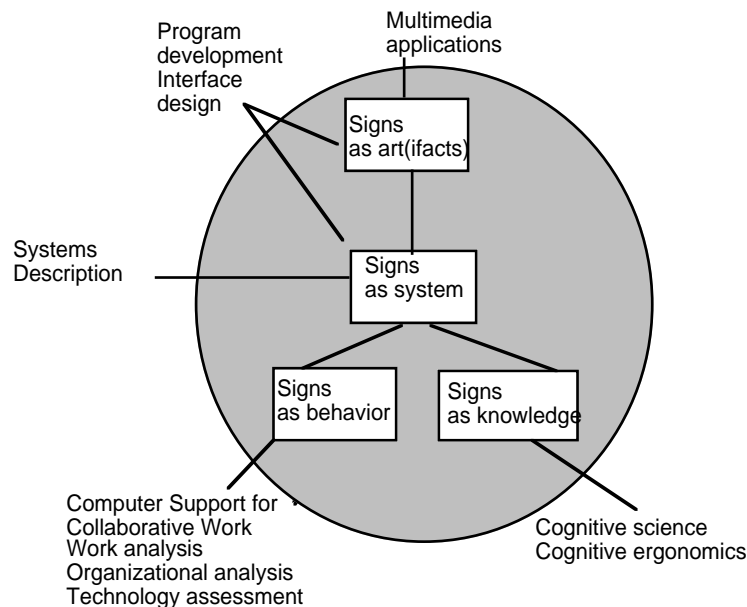Within this global perspective, we can define four local perspectives.

Fig. 1. A map of computer semiotics. Adapted from Halliday 1978.

The center is *signs as system.* The individual is considered as a creator, interpreter and referent of signs, as a user and reproducer of a common meaning potential and code, utilizing the results of a semiotic labor done by others. The focus of this box is sign systems as social phenomena with a structure that cannot be changed at will. Systems analysis, design and implementation aims at creating computer based sign systems that will typically be used by a whole organisation.

Signs as knowledge. The individual is considered as an assemblage of parts: his biological psychophysiological nature, the psychological mechanisms that enable the individual to learn, use and understand signs. In traditional linguistics these issues are treated by psycholinguistics. In the case of computer-based signs, the role is taken over by cognitive science and cognitive ergonomics that study what goes on in the mind and body of the individual.

Signs as behavior. The individual is considered as a single, indivisible entity, and the focus is on his interactions with the environment, especially that part which consists of communication with other individuals. Sociolinguistics and pragmatics work with these questions. Parts of the new field of Computer Supported Collaborative Work fills the box.

Signs as art(ifacts). The individual is considered as an innovator of code and meaning potential, as an explorer and inventor of signs. In this box, we should distinguish between code-creation in general (signs as artifacts) and its particular variants: to invent a new style of art (signs as art) is not the same as inventing new designs of articles for everyday use. Part of the design process belong here, in particular design that aims a creating new kinds of systems. The new multimedia technology requires skills of the designer that allows us to remove the parenthesis and talk about *signs as art*.

Semiotics methods have been suggested and used by several authors. (Goldkuhl & Lyytinen 1982, Winograd & Flores 1986) regard computers a tools for performing speech acts like promising, requesting, etc, predominantly viewing signs as behavior. (Rasmussen 1986) is a good representative for "the signs as knowledge" perspective; he uses semiotics for understanding the relation between the sign types needed by a control room operator and the kind of cognition he is engaged in. Signs as systems are treated by (Piotrowsky 1990, Declés 1989, Andersen 1990a). Declés describes machine architecture from a semiotic point of view, while Piotrowsky and Andersen use the structuralist school of glossematics as a theoretical basis for understanding computer systems. (Boland 1991) presents hermeneutic methods for understanding users' interpretation of the interface, and (Nadin 1988, Andersen 1991a) address aesthetic issues of interface design.

I believe semiotics to be a global perspective on computer systems; all levels of a system can be treated semiotically, but — like all perspectives — it can only treat a subset of the total set of relevant properties systematically. In this sense it is complementary to the mathematical perspective of natural science, since issues that can be treated systematically in one, is typically outside the range of the other.

The reason for this is that the two traditions are biased in different ways. Whereas natural science focuses on the mechanical aspects of the system — those aspects that can be treated as an automaton — semiotics focuses on the interpretative aspects. Semiotics must necessarily view computer systems as sign-vehicles whose main function is to be perceived and interpreted by some group of users. It has nothing to say about data in itself, only in its capacity of being interpreted and used as a source of knowledge or guide for action. If you want reliable methods for calculating time complexity of algorithms or proving correctness of programs, semiotics will be a disappointment.

Semiotic theory can only make scientific statements about phenomena that are used as signs to stand for something else for somebody. Although some semioticians will claim that all human behavior is sign behavior, I personally believe there are many aspects of behavior that cannot sensibly be classified in this way. Tacit knowledge is one example if it means: knowledge that cannot be expressed in any way. Neither is physical work whose purpose is not communication but production of commodities directly analyzable by semiotic methods, although valuable information about the workers' conception of their work can be gained by analyzing the way they talk about it.

The statement about the globality of semiotic analysis entails that it is not confined to "soft" areas like interface design and user friendliness, but that it can also treat aspects of the technical construction of computer systems.

At first this might sound implausible. After all, a computer is a machine, and we certainly do not want to use semiotics as a basis for constructing other kinds of machines like cars, vacuum-cleaners, and lawn-mowers.

However, computer systems are not ordinary machines, assembled by means of bolts and screws. They are symbolic machines constructed and controlled by means of signs. The interface of systems is of course one obvious example of a computer based sign. A sign stands for something to somebody in some respect, and since the interface of a flight reservation system stands for flights and seats to the clerk, this interface is clearly a sign. Using the system involves interpretation and manipulation of text and pictures.

But underneath the interface, in the intestines of the system, we find other signs. The system itself is specified by a program text (that is a sign since it stands for the set of possible program executions to the programmer). The actual execution involves a compiler or interpreter that controls the computer by means of the program text, and since the compiler is a text standing for the set of permissible program texts, the compiler is also a sign — in fact it is a meta-sign that — in some versions — very much resembles an ordinary grammar.

If we continue this descent through the different layers of the system, passing through the operating system and the assembly code, down to the actual machine code, we will encounter signs most of the way. As we reach the machine code things may possibly change. The machine code is unique in that its shape physically influences the machine without any mediating layers. Chunks of machine-code may not stand for anything else than

themselves, so we seem to have arrived at a stage where signs become mere signals.

But even if it can be argued that the bottom layer of a computer system does not exhibit sign behavior, everything on top of that is clearly used as signs by some group of professionals. There are always texts that must be interpreted as statements or prescriptions about some present or future state of the system. As we change level, the concepts signified by the texts change. On the lower levels, the meaning of the signs are related to the physical parts of the machine, like *registers* and *storage cells*. As we ascend, the texts are interpreted differently, we move away from a physical interpretation, and new software concepts appear, like *run-time stacks, heaps,* and *variables*.

A total picture of the whole system will depict semiotic activities from the top down to the very bottom of the system. A computer system can be seen as a complex network of signs, and every level contains aspects that can be treated semiotically.

This globality thesis is characteristic of the approach illustrated in this paper. Since *semiosis*, sign creation and sign interpretation, takes places everywhere in the system, the paper does not only address user interpretations of systems (Section 4) but also takes an interest in the formal construction of the computer-based signs that enter into this semiosis (Sections 3.2 - 3.3). Both users and programmers interpret the system, and the clash between these two kinds of interpretations is an interesting topic. In fact, the main purpose of the paper is to contribute to a framework for connecting systems interpretation with systems design.

Another characteristics is the emphasis on empirical research and analysis. We know so little about sign systems and sign usage that speculative methods and constructed examples should be used with caution.

Finally, this paper is primarily interested in signs as systems, that is: as properties of a social group, not of the individual mind. However, I see no reason to exclude the cognitive traditions treating signs as a vehicle for individual cognition, nor signs as a means for interacting with other people or objects. These perspectives do not exclude but supplement each other.

The interest in signs as social phenomena, in empirical work, and in the technical construction of systems necessitates a discussion of suitable theoretical frameworks. This is done in the next section.

2. Which Semiotic Theory?

There are many kinds of semiotic and linguistic theories. In order to be useful as the theoretical framework we need must satisfy the following criteria, adapted from (Bannon 1989):

- it must be applicable to linguistic field work, not only constructed examples and must respect actual language usage as the basis of analysis, because we want to use it for building systems that must work in a real environment;

- it must view language as a social phenomenon used for communicating and coordinating work, not only as a phenomenon of the individual mind, because most work is social, involving cooperation and communication;

- it must be founded on a rigorous theory, because otherwise the research process will quickly degenerate;

- it must give understanding of creative use of signs, an understanding parts of which must be formalizable, since design of computer systems is a creative process, and creative use of computer-based signs involves formalization.

In the following I shall reject two popular paradigms, the generative and the logical one, and advocate a third one, the European structuralist tradition.

2.1. The Generative Paradigm

The generative paradigm is relevant because it (or variants of it, like the augmented transition networks or case-grammars) has shown very immediate uses as a basis for programming natural language interfaces or machine-translation systems. In fact, the theory of formal languages and automata it helped building has become a standard part of a computer science curricula.

The generative paradigm was founded by Noam Chomsky (Chomsky 1957, 1965, 1968, 1976, 1980) in 1957) and was originally built around one simple question: which algorithms are needed to determine whether a string is a member of a formal language, i.e. a possibly infinite set of strings? By applying descriptive techniques of hitherto unprecedented rigor, Chomsky established a set of formal grammars, and argued - against the dominating behavioristic tradition of the time - that natural language exhibited a complexity that required a much more complex grammar than posited by the behaviorists. Since Chomsky at the same time offered new and more precise descriptive techniques

and a mentalistic, non-behavioristic ideology to go with them, his ideas revolutionized the linguistic world in a few years.

The revolution consisted in insisting on explicit description, so that only those sentences actually specified by a set of rules count as described. The grammars are modelled on formal logical proofs, and consist of an axiom and a set of rules. A grammar is said to generate all and only those strings that can be derived from the axiom by applying a sequence of one or more rules to it. The grammars are defined so precisely that it becomes possible to give mathematical proofs of the descriptive capacity of different kinds of grammars (Hopcroft and Ullman 1969).

As can be imagined, this added rigor was seen as a progress by many linguists. But with time it became clear that the descriptions produced ran a serious risk of drowning possible insights in enormous masses of rules, so Chomsky initiated a serious discussion about the quality of the descriptions. This development is important since it shows that although precision may be desirable, it does not automatically provide insight in the subject.

Chomsky's strategy in recent years has been to attribute as much of the linguistic description as possible to a universal grammar, trying to show that many linguistic facts do not belong to the description of individual languages at all, but characterizes the human faculty of language, the amount of universal rules being a measure of the insight gained. Developing formal characterizations of the human faculty of language has become an important aim of the school.

Even though this short description does not do full justice to the theory, it can be concluded that the generative paradigm is not what we are looking for. Although generative grammar is built on a formalized rigorous body of theory, it is focused on the individual language user, not on the social process of communication. It is only concerned with the mental faculties, not the physical skills, and it provides few bridges from the linguistic to the non-linguistic domains. In addition, the most usual source of data is examples and counter-examples constructed by linguists, not authentic speech. Finally, the paradigm has very little to say about creative use of language, linguistic inventions, which is an important part of design. The reason is that the theory basically sees language as a rule-defined set of sentences. The critique of the generative paradigm is summarized by (Halliday 1985):

> A language is not a well-defined system, and cannot be equated
> with "the set of all grammatical sentences", whether that set is
> conceived of as finite or infinite. Hence a language cannot be
> interpreted by rules defining such a set. A language is a semiotic
> system (...)- what I have often called a "meaning potential"(...).
> Linguistics is about how people exchange meanings by
> "languaging".

Halliday 1985: 7

2.2 The logical Paradigm: Language as Reasoning

The next possibility is the logical paradigm which is really older
than the generative one. It was founded by Frege, systematized in
the thirties and forties by Rudolf Carnap and Ludwig Wittgenstein
and later given a linguistic turn by e.g. Hans Reichenbach
(Reichenbach 1947). However, only in recent time, with the logical
grammars of Richard Montague (Montague 1976), it was fashioned
in such a way as to count as a kind of linguistic theory. Like parts
of generative grammar, the logical theory of language has also
become an integrated part of computer science.

Since an important part of the logical paradigm consists in
translating natural language sentences into logical formulas on
which rules of inference can operate, it can be used as a
theoretical basis for constructing language understanding systems.
One way of constructing such systems is to represent knowledge in
terms of logical statements, to translate queries into corresponding
logical formulas, and then let the system try to prove the query
from the knowledge base.

The main objection towards using logical grammar is that it is
not a linguistic theory at all, since it does not describe factual but
idealized linguistic behavior. This is a drawback if we want to use
the analysis to learn about the concepts through which a
profession understands its work, assuming that a good
understanding of this is a prerequisite for designing usable
systems. We want to analyze language from within ("immanently"),
not as a projection of something else. If we use logic as our point
of departure, we defeat this purpose, since logic and natural
language semantics are two different systems, developed for differ-
ent aims; for example, logic uses two main conjunctions, Ÿ, often
translated *and*, and */*, translated *or*. However, the logical
conjunctions and their linguistic counterparts behave differently in
many respects. Natural language *or* normally has an exclusive
sense, meaning "one out of several", while */* means "one or both of
two". It is of course possible to define an exclusive or-conjunction

but this is beside the point. The point is that logical *or* is non-exclusive because it is convenient for constructing logical proofs to have two conjunctions that obey certain rules, e.g. de Morgans law , ¬(A ∕ B) ∫ (¬A Ÿ ¬B). But since language users do not spend their time constructing proofs, their semantic system cannot apriori be expected to contain the same semantic elements.

Logicians perform other tasks than other language users, so their professional language is under constraints absent in other kinds of language.

An immanent structuralist semantics cannot apriori accept the logical concepts like quantifiers " x, $ x, connectors Ÿ, ∕, etc. as basic semantic units, since the purpose of the semantic research is precisely to establish the identity of these self-same units. The only way of establishing the units is to investigate whether they have a function in language. Using logic as a method for describing the semantics of natural language is like trying to use the tonal scale to describe its phonology. In both cases, we impose distinctions on language from activities that are different from those performed by language users. Here logic, there music. Therefore, although logic grammar may masquerade as a sort of linguistic theory by using linguistic terms like syntax, semantics, and lexicon, it is not a linguistic discipline.

From this point of view, logic and logic grammar should be viewed as a professional activity that seeks to discover general laws of valid inferences. Montague grammar can be seen as a good method for comparing results from formal logic to results from semantics, comparisons that can yield valuable insights in both areas, just as comparisons between a linguistic analysis of color terms and a psychological description of our visual discrimination capabilities can do.

To summarize: in spite of their popularity in linguistic work with computers, both the generative and logical paradigms have been rejected as a general theoretical framework, not because they are bad theories, but simply because their general goals are different from the ones we want to accomplish.

This does not mean a wholesale rejection of the approaches and insights they represent, only a delimitation of their range of applicability. Just as logical grammar is useful in systems where logical proof procedures are relevant, so is generative grammar relevant in those cases where formal manipulation of structured strings of characters is demanded.

2.3. The European Structuralist Paradigm: Language as Creation of Meaning

The generative and the logical paradigm - and their many variations - have had a large impact on linguistic debate from the sixties and onward. However, in our connection a better choice seems to be the European structuralist paradigm characterized by such names as Ferdinand de Saussure, Louis Hjelmslev, Michael Halliday, and Umberto Eco.

In particular *systemic grammar* founded by Halliday is interesting because of the following properties:

- Its data is to a large degree gathered from authentic language usage in real situations, and tape recordings are often used. Thus the paradigm is well suited for field-work.

- Language is basically seen as a social phenomenon and is described according to the functions people use it for in real life. There exist theories that relate language to situations and social roles and classes. This makes the theory suitable for describing communication in work situations.

- The semantic aspect of language has highest priority. Language is seen as a meaning potential, a set of alternatives from which language users can choose when they create meaning.

- Although the emphasis is on semantics, there are fairly explicit rules for relating meanings to observable expressions, and the lifeline to the traditional body of knowledge about syntax and morphology is kept intact. This means that the theory can be used in practical textual analysis.

Because of its emphasis on the language use and its attempts to relate language to social structures, systemic grammar can serve many, but not all of the needs mentioned in the beginning of this section. One problem is that systemic grammar does not offer support for active design and creative use of signs. Another inadequacy is that systemic grammar, with a few exceptions, is not used to describe other codes than verbal ones and it is difficult to judge if it can be generalized to cover non-verbal signs also. Technological developments in recent years has made this a real drawback, since pictorial codes are now commonly used in interfaces, and multimedia applications, involving both sound, animations and video seem to be just around the corner.

One place to look for relevant theories is in European semiotic research because it treats creative sign-usage in a broad spectrum of codes: literature (Greimas, Eco), photographs (Barthes), and

film (Metz) and shares a minimum of theoretical background. As an example, let us take Umberto Eco's "A Theory of Semiotics" (Eco 1977). The book is relevant to design since its second part is devoted to a theory of sign production. The basic concepts in this part is the dichotomy between form and substance that, in a creative artistic context, is important, since most art consists in giving form to a substance. The sculptor gives form to his stone, the painter to his colors, the dancer to body movements, and the poet to linguistic material:

> We may define as invention a mode of production whereby the producer of the sign-function chooses a new material continuum not yet segmented for that purpose and proposes a new way of organizing (or giving form to) it in order to map within it the formal pertinent elements of a content-type.

Eco 1977: 245

The book describes and classifies many ways of producing signs. At one end of the scale, we have everyday sign production and interpretation where the expression-tokens (e.g. sounds) can easily be assigned to its proper type (e.g. phonemes), and where standard coding rules exist that enable the listener to interpret its meaning. At the other end, we have radical inventions, like e.g. the birth of impressionism, where the artist strives to perceive reality in a new way, and fixes the result of his labor on a canvas in the form of an expression-token (paint) that does not belong to any established type (unlike a sound that belongs to a definite phoneme) and is not referred to by any socially accepted coding rule (like the rules that relate the words and sentences of language to meanings).

Although systems designers may not be van Gogh's, they are faced with similar problems when building systems: should we base our system on a metaphor that users understand in order to ensure understandability, but running the risk of constructing a system that really do not give users new opportunities, or should we invent new ways of doing and looking at things, risking that nobody will understand it?

In fact, it is possible to stretch the analogy even further: it is not only sculptors that give form to substance. Designers and programmers of computer systems do the same thing, only the substance they mould does not consist of stone but of program executions (viz. sequences of system states), and the tools they use to shape the substance are not chisels but programming environments. The term *structure* in the following quotation in fact has some similarities to the form concept:

> A *process* is a development of a part of the world
> through transformations during a time interval.

> *Structure* of a process is limitations on its set of
> possible states (and thus on its possible sequences of
> states). In most programming languages, the
> transformations are prescribed by *imperatives* (and
> thought of as *actions*) in structure descriptions called
> *programs*.

Nygård & Sørgård 1986: 380

3. What Is a Computer-Based Sign?

3.1. The Sign Concept.

If we have decided to look at computer systems as sign-vehicles,
the next question is naturally: what is a sign and what is a
computer-based sign?

This section presents two variants of the sign-concept, namely
the structuralist variant defined by Saussure and later elaborated
by Hjelmslev and Eco, and the Peircian variant. The structuralist
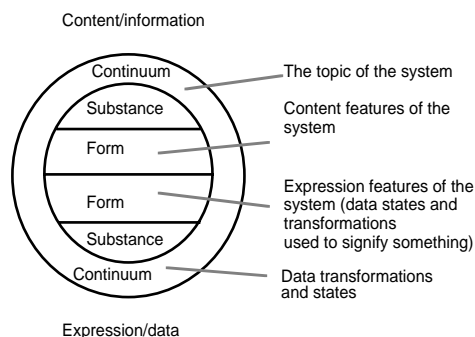variant can be depicted as follows:

Fig. 2. The structuralist sign concept.

The structuralist sign is mainly a description of a sign *type*. It
focuses on its social and institutional aspects. The sign has two
main planes, the *expression* and the *content* plane, also called the
signifier and the signified. Both planes have two aspects, a *sub-
stance* and a *form* aspect.

The substance of the sign is a part of the continuum articulated
by the form of a sign occurrence. When a sign token is produced in
some situation, two continua are articulated: significant
distinctions are introduced into the expression continuum, e.g.
sound in spoken language and screen pixels in computers; these

form elements are — via the sign function — correlated with se-
mantic form elements, which in their turn establish distinctions in
the content continuum. In both language, the content continuum is
the meaning of the sentences, in computer systems it is the domain
of the system.

The theoretical role of the continuum is solely that of an
amorphous mass that can be formed. We cannot say anything
about it, since in that case the continuum has already been formed
and has turned into substance. Still, it helps us defining the
concept of a computer-based sign since, according to the analysis
in (Piotrowsky 1990), we can identify the expression continuum of
computer-based signs with the storage cells of the computer. If we
disregard the program, the only thing that can be said about the
cells is that they are different and can be in different states but are
able to contract meaningful relations when structured by a
program. Similarly, like the continuum of the sign concept, they
are extremely versatile, being able to manifest widely different
kinds of meaning-bearing forms: word processors, spread-sheets,
data-bases, etc, etc. I will add the peripherals, e.g. screen and
loudspeaker, to Piotrowsky's substance concept.

Since programming imposes a form onto the cells, programming
is really sign-creation.

In a flight reservation system the content continuum is airplanes,
passengers, flights, departure and destination location and times.
This continuum can be articulated in many ways, and a flight
reservation system imposes quite special distinctions into it,
reflecting the particular tasks of the clerk. For example, the
number of motors of the plane is non-distinctive, whereas the
number of seats is.

3.2. Simple Signs.

The two planes of a sign are built out of smaller units. The smallest
expression units in language are the phonemes. We can use them to
build syllables and syllables can again be parts of words, which
again are parts of tone-groups, utterances. In a similar vein, we
must look for the smallest elements out of which to build
computer-based signs. In the following, I present some very rough
outlines of the nature of computer-based signs and their methods
of combination.

The prototypical computer-based sign is composed of three
classes of features:

A *handling feature* of a computer-based sign is produced by the
user and includes key-press, mouse and joystick movements
that cause electrical signals to be sent to the processor.

A *permanent feature* of a computer-based sign is generated by the computer. It is a property of the sign that remains constant throughout the lifetime of a sign token, serving to identify the sign by contrasting it to other signs. Examples: icons in picture based systems, or letter sequences in textual systems.

A *transient feature* of a computer-based sign is also generated by the computer, but unlike permanent features, it changes as the sign token is used. It does not contrast primarily to other signs, but only internally in the same sign, symbolizing the different states in which the sign referent can be. Examples: location, hilite.

In addition we must incorporate the notion of *action* in the definition of the sign types, since our interpretation of a computer base sign also depends upon what it can do to other signs. For example, the fact that the pencil sign can leave black traces on the paper sign of the screen enables the pencil and the paper to build a composite computer based sign that can be paraphrased "I am painting the paper with the pencil". Similarly, a *sort* command would not be interpreted as "sorting" if it did not change its object in a specified way.

The following classification of computer-based signs is based upon three criteria

• which features does the sign possess?

• can the sign perform actions that affect features of other signs?

• what is its possible range of meaning?

		+action		-action
		+handling	-handling	
+permanent	+transient	Interactor	Actor	Object
	-transient	Button	Controller	Layout
-permanent		Ghost sign		

Fig. 3. Classification of computer based signs.

The *Interactor* exploits features from all three dimensions. It is distinguished from the other signs by permanent features like e.g. size and shape, and during its lifetime it can change transient properties like e.g. location and color, these changes being functionally dependent upon its handling features. In most cases it can perform actions that change transient features in other signs. As Fig.4 shows, it is used for signifying tools in tool-like applications, and represents the user in games.
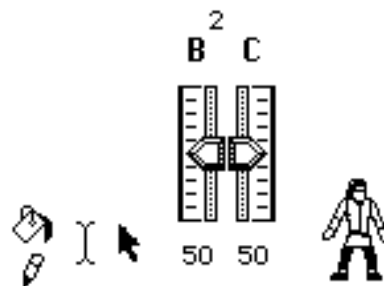

Fig. 4. Interactors. Cursors from Canvass, sliders from Digital Darkroom, and the protagonist from the adventure game Dark Castle

Buttons resemble Interactors, only their transient features are rudimentary, e.g. consisting of highlighting. *Actors*, lacking handling features, are autonomous processes that cannot be interfered with once started. In a word processor, they include operations like sorting, creating a table of contents, or saving the text to disc. In games, actors represent the antagonist. *Controllers* lack transient and handling features but can still act on other signs. Windows are often divided into areas, e.g. at tool palette and a work space, whose borders are controllers that change the cursor Interactor. If the cursor is within the palette it becomes an arrow

for selecting a tool, while it changes into a tool when moved into the work space. As the name indicates, *objects* are passive items lacking handling and action features. They are typically acted on by means of actors or interactors. Examples are signs for work objects like pictures and texts. *Ghosts* are signs that can neither been seen nor handled, but still influence other signs. Examples are hidden typographical codes for *new line* and *soft hyphens* in word processors. In games ghosts can represent hidden traps. Finally, Layouts lack all features except permanent ones, being mere decorations. Examples are constant headings in textual systems, and boxes and colors in graphical systems.

It is important to notice that this definition of computer-based signs is not a definition of interface elements, since a semiotic analysis of the form of computer-based signs will be perpendicular upon the standard distinction between functionality and interface. In a semiotic approach separation of interface from functionality is theoretically invalid: on the one hand, there will be features of the interface that do not belong the form but to substance, since they play no role in the creation of meaning. On the other hand, according to the argument about the pencil sign or the sort-command above, many features of the algorithms providing the functionality of the system will belong to form, since the meaning of the signs depends upon them.

For example, the sort-sign will incorporate those part of the sort algorithm that distinguishes it from e.g. processes that are interpreted as merging-processes. It encompasses what in some traditions are called the *invariants* of the sort-process.

Suppose that x and y are elements that can be ordered in a list. Let C(x) be the contents of the elements, and suppose that an order is defined on the contents also, e.g. an alphabetical order. Let $P(x,y) =_{\text{def}} C(x) < C(y) \rightarrow x < y$ be an ordering principle demanding that the two orders must agree. An important part of the meaning of sorting is that before sorting at least two element did not satisfy P, while afterwards all elements satisfy P. This description would capture the essentials of the meaning of "sorting" — its content form that distinguishes it from other kinds of processes.

The specific method of sorting, however, belongs to the substance of the sort-sign, since exchanging one method for another does not alter the interpretation of the process.

3.3. The semiotics of object-oriented programming.
The concepts of object-oriented programming, OOP, (Kirstensen et al. 1991, Korson & McGregor 1990, Rumbaugh et al 1991) are close to the semiotic concepts presented above. In the object-

oriented paradigm, the system is seen as a model of the application domain. Its components, viz. classes and objects, stand for things, events, and interactions in the domain. Object-oriented analysis can be seen as a componential semantic analysis of the signs we wish to enter into the system.

The characteristic of the object-oriented paradigm, the concepts of classes and specialization, is a standard in semantic analysis at least going back to Aristotle. The hierarchy of classes, known as the *Porphyrian tree,* was developed by the semioticians of the middle ages (Eco 1984: 46 ff).

The semiotic form-substance distinction has its analogue in the object-oriented *encapsulation* concept. The form describes the distinctive properties of a sign — the features that contribute to creation of meaning — while its substance describes a particular way of realizing the form. In object-oriented programming, the interface of an object can be made to correspond to form, while its inner data and processes are designed as substance, a particular way of implementing the interface concepts: "The fourth guideline requires that to belong to the class, each operator must represent a behavior of the concept being modeled by the class" (Korson & McGregor 1990: 53).

A main difference between the semiotic and the object-oriented approach lies in the treatment of the "modelleling" relation. From a semiotic point of view, the modelling relation between system and domain is a very imprecise one in the object-oriented tradition. I agree that the system stands for — represents — the domain, but a sign does not just stand for something abstractly. It stands for something to somebody in some respect.

Here the European structuralist and the object-oriented paradigm have a common problem: they lack one important concept, namely that of the interpreting person and his reactions to the sign. This is a problem, since we build computer systems in order for the users to interpret and use it.

Peirce's variant of the sign concept can help us clarify this point: in his framework it does not make sense to say that signs occur except when they are interpreted by some interpreter. Furthermore, in this process the interpreter cannot help reacting to the sign, producing a new sign called an *interpretant*. Therefore Peirce's sign contains three parts: the *representamen* (analogous to the expression plane), the *object* (analogous to the content plane), and the *interpretant* (which has no analogue in the structuralist sign concept).

Peirce's sign concept represents an important corrective to the structuralist approach and motivates the question: to *whom* is the system a model? *Who* is going to make *which* interpretants? The

system as a whole cannot be a sign to the users, since they do not normally see its elaborate Porphyrian tree, but only bits and pieces of system executions. But a sign has to have an interpreter to exist, and I believe that the implicit interpreter, the person who sees the system standing for something, is the designers and programmers. The problem is that these people get paid to produce signs for the users, not for themselves!

In Section 4 we shall see an example of users that definitely do not interpret their system by means of classification and aggregation. They use space instead.

This is an argument for revising the view of design and programming. The essence of programming in a semiotic perspective is not to make models for the benefit of the designer; it is rather to use the machine to try to tell people something. This point is well made in a paper by (Nardi and Miller 1991) on user programming of spreadsheets

> Many spreadsheets are destined from the start for the
> boardroom or the boss's desk or the auditor's file. In
> our study, spreadsheet users were very aware of the
> importance of presenting their spreadsheets to others —
> Laura stated "I usually think in terms of my stuff [her
> spreadsheets] as being used by somebody else" and
> users constructed spreadsheets with effective pre-
> sentation in mind.

> Nardi and Miller 1991: 170

Making a spreadsheet is programming, and the purpose of this programming is communication — sharing domain knowledge as they call it. Therefore spreadsheets signs must be seen as parts of other kinds of communication. For example, they enter into conversations and influence them:

> Her comments show that the spreadsheet organizes
> discussion, as we have seen in the preceding example.

> Nardi and Miller 1991: 171

But if users enter computer-based signs into conversations, the former must fit into the latter. I return to this issue in Section 4.

Instead of thinking of model making, I advocate the metaphor of stage direction. A system is viewed as a kind of theater, its executions are performances that are interpreted by an audience, and the designer is a stage director whose success does not ultimately depend on the look of the props and set-pieces from backstage, but on their communicative effect on the audience.

The typology of computer-based signs presented in the preceding section, was designed with this objective in mind. The

individual classes are designed so that they have meaningful roles in "computerized" dramas. In a tool system for example, Interactors work well as signs for tools, Objects perform excellent as work material, and Controllers and Layout signs are skilled producers of illusions of spaces and locations. With suitable elaboration and a better empirical foundation, I think the typology could become useful as a general classification of object classes, encouraging the designer to think more consciously about the performance he is directing.

Another suggestion relates to the level structure of programs. I suggest that system executions are divided into levels, each level being defined by a set of possible plots, actors, set-pieces and props. The requirement is that the plots of a level must be of the same types, and that only set-pieces and props with a function relative to the plots are allowed at this level. The program text is divided into similar levels, and the individual level is not allowed to make distinctions that lacks meaning in its corresponding execution. These principles are intended to make easier for the programmer to interpret the program text as a script for meaningful performances.

The following two examples show consequences of these principles. Consider first a direct manipulation interface. At the user level execution, the plot is about hitting, grasping, dragging, and letting go of material. Hitting can be softer (click) or harder (double click). If the following line

```
if ticks-oldticks>10 then return "Pause"
```

is visible in the code defining clicks and double clicks it is a programming error. The reason is that *ticks* (returns the current system time) and *oldticks* (stores the time of the last mouse event) are not part of the user-level story, and that replacing 10 with 12 does not produce a new story. The line belongs to a level below that tells stories about mouses and clocks.

In this example, the semiotic and the object-oriented guidelines probably agree. The next example shows that they can be at cross purposes.

Consider classes in an accounting system at the level where customers withdraw and deposit money and occasionally change addresses:

```
Class  account
   Balance:  integer;
    procedure  Deposit(CustomerId,Amount);...
    procedure  Withdraw(CustomerId,Amount);...
end   account

Class  customer
   Address:  string;
    procedure  ChangeAddress  (NewAddress)
      Address := NewAddress
    end  ChangeAddress
end   customer
```

Fig. 5. Account and Customer classes of a banking system.

The Account class possesses Deposit and Withdraw operations, since they both effect Balance, the internal variable of the class. For a similar reason, Customer contains the ChangeAddress operation. This object design, which follows normal guidelines, entails code like this:

```
ACustomer.ChangeAddress("A  new  address").
AnAccount.Deposit(Acustomer,1000).
```

According to the criteria above, this is bad code, since it introduces a syntactic distinction between depositing/withdrawing (subject-verb-object) on the one hand, and changing addresses (object-verb-subject) on the other hand which has no interpretation at the current plot level, where the customer is the subject of both actions: customers change adresses as well as deposit money. The underlying principle is that differences of object structure should have an interpretation in the domain.

According to OOP the code is good because it obeys the principle of strong cohesion between operations and properties of a class. The principle is that the Deposit operation should not be declared in the Customer class, because it affects the Balance property of the Account class.

In this case, the OOP principle of operation/data cohesion and the semiotic demand for interpretability do not agree.

The criteria for judging the two pieces of code is based on one of the basic principles of sign analysis called the *commutation test*. The test is used to empirically determine what belongs to the form of a sign and contributes to creation of meaning and what does not: a feature is a part of the expression or content plane of a sign if and only if exchanging the feature for another one causes a change of features on the other plane. Therefore /male/ is a part

of the contents of the word *boy* since exchanging it for /female/ causes the expression to change to *girl*. /large/ is not a feature of *boy* since a small boy is still a boy. On the other hand, /large/ is a part of the meaning of *forest*, since a small wood is not a forest.

The recommendations in the preceding simply boils down to the guideline that the program text should pass the commutation test, but a closer inspection of the sign-processes reveals that there are at least three different kinds of signs we can apply the test to:

domain

contents   contents

expression

expression

program   **execution**
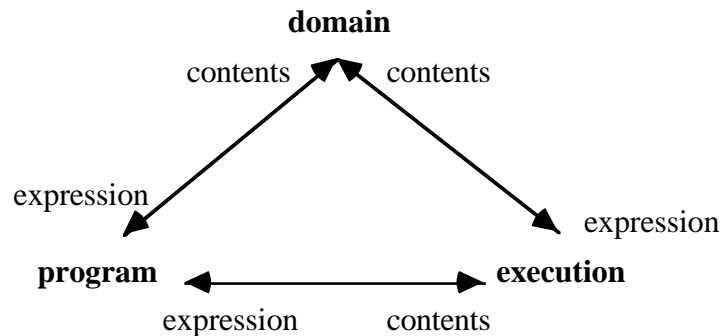
expression   contents

Fig. 6. The sign relations of program text, execution, and domain.

The execution is intended as a sign for objects and processes in the domain, but the program text is an ambiguous sign whose contents can *both* be the execution and the domain. In the banking example, the syntax and identifiers of a code like *ACustomer. ChangesAddress ("A new address")*   invites an interpretation relating to the domain, while the implementation inside the Customer object *Address := NewAddress* refers to variables and therefore stands for properties of the execution. It seems to me that OOP focuses on the program/domain sign as the important part of the system — the modelling relation — and seeks to hide the program/execution relation by means of encapsulation.

My criticism of this arrangement was that the interpreters of the program/domain signs are programmers, while users interpret the execution/domain sign. If the programmer is to be a successful director of executions, he should work as carefully with the program/execution sign as a director works with movements, gestures, lights, and set-pieces. But this is difficult if the program/execution signs are hidden and encapsulated!

On the basis of this analysis, I would shift the focus from the program/domain relation to the program/execution relation, and require of a program text that it clearly expresses the form-elements in the execution that contribute to creating the execution/domain signs. I conjecture that the typology in Fig. 3, suitably elaborated, could be the basis of a programmer's palette for creating the required signs. This shift of course requires the de-

signer and programmer to assume a more humble attitude: we d o
not mirror reality in our systems, we only direct a play that t h e
users under lucky circumstances can use to gain knowledge a n d
insight.

3.4. Composite Signs.

 As in language or paintings, the smallest sign-parts are g r o u p e d
into larger wholes called *syntagmas* or *phrases*.   Formally, a
syntagma can be described as a sequence of slots that can be filled
according to rules that specify what material can be used as f i l l e r
and whether the slot must be filled or can be empty.

 Traditionally,   we   distinguish   between   three   main   type o f
syntagmas, *constellations, subordinations,* and *interdependences*.
They can be defined formally in this way:

- In a *constellation* the slots can be filled or empty,

- In a *subordination* one slot (often called the *head*) must b e
  filled while the other (the *modifier*) needs not, and

- In an *interdependence*   all slots must be filled.

Although   the   formal   definitions   have   analogies   in logic (a
constellation corresponds to a *disjunction*, a subordination to a n
implication, and an interdependence to an *equivalence*), the t h r e e
syntagmas are not identical to logical constructions,   since t h e y
have a different semantics. It is is as follows:

- In a   *constellation*   both   parts   relate   to   the   context; t h e
  individual part can play the same semantic role as does t h e
  whole construction. In "I saw a house and a car" both *h o u s e*
  and *car* relate to *saw*, since it makes sense to say "I saw a
  house" and "I saw a car".

- In a *subordination* only the head makes a direct c o n n e c t i o n
  to other meanings in the text. The modifier does not directly
  relate to the text outside the subordination. In "I saw a large
  tree" *large* relates only to *tree*, not to *I* or *saw*.

- In an *interdependence* the effect of the whole cannot b e
  reduced to the effect of one of its parts, so none of the t w o
  parts relates individually to the surrounding text. Only t h e i r
  whole does so. In "I said he came" *he came* as a whole r e l a t e s
  to *said*. It makes no sense to say "I said he" or "I said came".

Constellations express   contingent   relations,   e.g.   an o p e n
enumeration of elements that can be continued: "I saw a house, a

car, a train, ...". In opposition to this, *interdependences* express necessary relations, and their parts form a closed set. The relation expressed by "He came" must contain two elements, while that of "He gave the book to me" needs three. We cannot add or subtract elements randomly as with constellations.

Finally, *subordinations* organize the material in subordinate and superordinate elements. For example, it follows from the definition that only the head of a subordination is active in creating the narrative structure of a text.

These concepts can be used as a guide to structure the program code. For example, suppose we are building a teaching system always containing conceptual explanations in one window, and sometimes examples in another window. Then we could say that the main syntagma of the system is a subordination with the Explanation window as the head and Example window as the modifier:
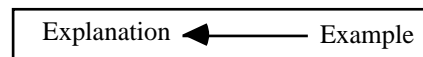
Explanation ◄——————— Example

Fig. 7. Subordination. The arrow points from the modifier to the head.

Descriptions like Fig. 7 could be entered into programs as constraints the system must always obey (see Freeman-Benson, Maloney & Borning 1990). Suppose in addition to the two windows above, we have two others, Text and Picture, that must contract interdependence, and suppose furthermore that the user can open and close windows as she wants via menus or buttons. Then the following procedures, which are called every time something happens to the system, ensure that the constraints are obeyed.

```
Procedure   KeepWatch
     Subordination  Explanation,  Examples
     Interdependence  Text,  Picture
end   KeepWatch
```

```
Procedure  Subordination  Head,  Modifier
    if  Opened(Modifier)  then
          Open(Head)
    end  if
    if  not(Opened(Head))  then
          Close(Modifier)
    end  if
end   Subordination

Procedure  Interdependence   Window1,Window2
    Subordination  Window1,  Window2
    Subordination  Window2,  Window1
end     Interdependence
```

Fig. 8. Implementation of subordination and interdependence.

The *KeepWatch* procedure is called every time an event happens, maintaining subordination between Explanation and Examples and interdependence between Text and Picture. The *Subordination* procedure checks if the Modifier is open; if so, it opens the Head. If the Head is not opened, it closes the Modifier. *Interdependence* is implemented as a double Subordination.

Like the simple signs, composite signs have a dual definition, one part of it referring to the expression plane (the formal definition), the other one to the content plane, and it is an error to implement a formal subordination that does not also satisfy the semantic requirements.

Thus, if we make a system where a text window is formally subordinate to a picture window, then the main structure of the system must be interpretable solely via the picture window, and the text window should only play a secondary role as a commentary to the picture window (Andersen 1990b). As emphasized in the preceding section, the programming process always have one foot in the expression plane (the formal aspect) and the other foot in the content plane (the interpretation aspect).

4. The context of interpretation.

A computer-based   sign acquires   part of its meaning   from its relation to other signs the user sees in the interface; but the u se situation contains other kinds of signs it must relate to. In fact, it combines   with the global semiotic system of the workplace t h e most important   part of which is the work language of the users, but which also includes e.g. diagrams and plant layout. In order f o r the computer-based   sign to amalgamate   with its semiotic c o n t e x t of use, the structuring   principles underlying the two semiotic systems must be similar.

    This is not always easy to achieve, since users o f t e n conceptualize and talk about their work in ways that are different from those of the designers. Sometimes the conceptualization is fairly common, sometimes they use distinctions a designer c o u l d never get at except by listening carefully.

    This section gives an example of both cases. The first one is t h e use of space as a way of talking about computer systems.

    As an example I quote some of the data from a project w h e r e Berit Holmqvist and I investigated how users interpreted a n e w administrative   system at the Postal Giro Office in S w e d e n (Holmqvist & Andersen 1991). The system was based on of t h e OOP principles of aggregation and classification, but   the w o r k e r s disregarded the system and used space and time as principles f o r naming   their computerized   work processes.   The following quotations are from a speaking a loud session, where we asked a worker to tell us what she did while she worked.

    In her speech work objects can *go there, come, come forth,*

    you know, everything goes there

    this one is no good because then all cards come

    and she presents her own actions as a journey in a landscape o f cards and tasks, describing the surroundings from her p r e s e n t location: *come to, go to, go back to, be, sit*

    because I came to the last card on this one

    when I sit in "comp" then it is "completing"

    Her own actions on the work objects either removes them f r o m her: *put aside, send back, place there, take out, take away*

    then I put aside

    then I have to take this signal away

or brings it nearer to her: *fetch*

    I could have fetched it to

    In addition to the spatial distinction between *here* and *not here,* the worker uses another basic contrast, namely between *have* a n d *not have.*

These verbs can be arranged in a semantic system structured by three sets of features:

location versus possession

positive versus negative versus modal

(result of) process versus state

where positive location = here, negative location = there, not here. The feature *modal* means that the sentence does not signify a state of affairs but a wish or an obligation.

	location		possession		
	positive	negative	positive	negative	modal
(result of) process	COME, FETCH hämta, komma till, fram	PUT, SEND lägga åt sidan, ta bort, skicka	GET, TAKE få, få fram, ta,	ABANDON överge, inte få	WANT, MUST vill ha, ska ha
state	BE, SIT vara,sitta	NOT EXIST finns inte	HAVE ha, ha framme	LACK inte ha, sakna	

Fig.9. Semantic system of task designations in work-language.

For example, *put aside* differs from *get* by being concerned with location instead of possession, from *come* by by leaving the object *there* instead of *here*, from *exist* by being a process and not a state, and from *want to have* by expressing a state of affairs and not a wish for the future.

Kim Halskov Madsen and I got the same result in another project where we looked at librarians' interpretation of their system. Although users use other metaphors, the space metaphor is very frequent.

The reason why the desk-top metaphor of Xerox and Apple has been so successful is probably that it uses space as the main structuring principle, and the designers of the system at the Postal Giro might have got good design ideas from knowledge of the workers' principles of structuring meaning.

In this case, the programmers' Porphyrian tree was foreign to the users, and should probably not have been let lose in the interface. However, although the spatial metaphor seems to be a general metaphor, we should be cautious not to over-generalize. It is perfectly possible that classification and aggregation are natural concepts in other professions, e.g. librarianship. We have to verify

these issues empirically, since the complexity of reality exceeds our current theories.

The next example from the same research project illustrates this unpredictability. The computer system was built as a mirror image of the paper flow and therefore the distinction between computer-based and paper-based material was important. The system and the manuals tried to make the distinction systematically, but it turned out that the users only upheld it in verbs, not in nouns.

Verbs for manipulating work objects are divided into two different categories, one used about paper (*skriva på, skriva dit, skriva under, måla, måla över, [write, sign, paint])* and another used for manipulating the electronic data *(slå, slå in, trycka, trycka på, trycka ner, slå fel [hit, press, enter]).* But in the nouns they used the same words for both categories:

> But you understand, you have to put the *card* [Sw:*kort,*
> paper card] in the flier box, if you have put a *card* [*kort,*
> electronic card] in the flier, and then I'll have to look in
> this flier box.

The first occurrence of the word *card* refers to the paper card, the second to the electronic card, since *flier box* denotes a physical plastic tray, and only paper cards can be placed there, while *flier* denotes a file, where only electronic cards can be placed.

The reason for this unpredictability is that work language evolves closely connected to tasks. The here and now pressure of daily work changes the language continually; if a worker needs to make a verbal distinction in order to do work, she just creates it out of whatever linguistic material is at hand. Work languages are not neat and logical but look more like a patchwork blanket.

The use of domain specific concepts as a basis for designing classes is recommended in object-oriented design; the lesson from the preceding examples is that they do not just lay around ready to pick; it sometimes requires close attention and knowledge of tasks and organization to discover the semantic system of the workplace.

Another point worth of emphasizing is that signs have specific contexts of uses. Concepts do not just spend their life hanging in the scaffold of the *Porphyrian tree,* but are constantly put to work in conversations (Winograd & Flores 1966). Investigating recurrent patterns of conversations gives information about when information is needed and in which form. For example, problem solving conversations as the following were recorded in the Postal Giro project

E: That is an adding mistake.

Y: Yes.

E: Yes, it is.

…

E: Card missing?

X: Well, I didn't have the card then, you know, then I put it to one side, for the time being, though we'll sit on it, now we *fetch job*, *enter job number 1*, it was the first job, you see.

E: You'll have to destroy it or hand it in the box.
X: Right, that's it.
E: He's, he's...
X: Well, let's see now, then I should, then I'll destroy the whole batch.E: Let's see, hmm, it's sure to be scan, this one's going to be a real mess.

…

E: Do it like this, I think: take a little slip, and then put it on the C card, and tell them that it's a C card, then when you get it back, then you put it all in this box here.

It bears the hall-mark of rule-governed work since it does not aim at inventing methods for solving the problem, but seeks one out a fixed set of rules. Its start with a problem definition *(adding mistake)*, followed by attempts to determine the error *(card missing)*. Then follows a lengthy method discussion *(You'll have to destroy it)* that eventually ends with picking a rule *(do it like this)*. The recurrent structure of this type of conversations is described in the diagram below:
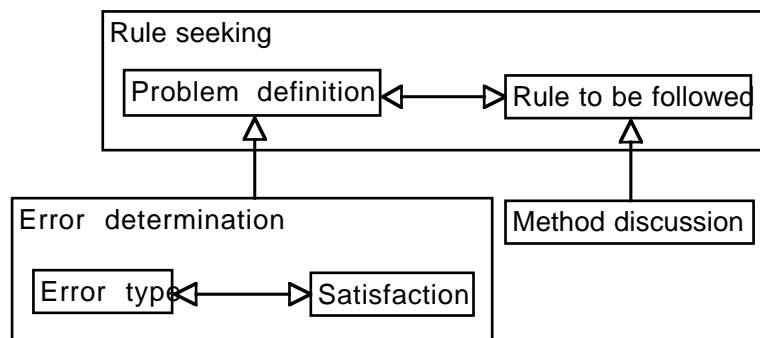
Fig. 10. Interdependences are represented as double arrows, cf. the implementation in Fig. 8.

This diagram tells us which kind of information they handle in the conversation and describes the relations between types of information: for example, Problem Definition and Rule to be Followed always occur together in our data, whereas the Error Determination and Method Discussion were optional.

An obvious guideline is to use work language as a basis of design, not by copying it, but rather as an inspiration for creativity, in the same sense as a filmmaker can use a novel as a basis for a film.

In the case of Fig.10, we could base the design of problem solving support on a copy of the conversation structure. It will consist of three windows. The main window, Rule Seeking, contains two parts, a List of Problems, and suggestions for Rules to be Followed. These two items always pop up because our data shows that workers always seem to need this information. The two other windows, Error Determination and Method Discussion, are subordinate to the Rule-Seeking window, which may therefore contain two buttons for opening them.

The relation between work process and design is discussed further in (Holmqvist & Andersen 1991, Andersen 1990a).

5. Summary.

The key concept of this paper is semiosis, the processes whereby signs emerge and are interpreted.

Computer systems are seen as sign-vehicles, and since signs consist of contents as well as expressions, one should neither investigate user interpretations without considering the technical structure of the system, nor treat design and implementation divorced from the intended interpreters of the system. A computer semiotics must integrate an understanding of programming as a semiotic process with empirical field work investigating the actual interpretants of the users. It means that the theoretical framework must contain concepts for understanding formal structures as well as concepts for analyzing the non-formal signs of the users. It has been argued that the European structuralist tradition is broad enough to meet these demands, provided its bias towards structural descriptions is balanced by a Peircean understanding of sign processes.

In order to suggest possible contributions of a semiotic approach concretely, example solutions based on object-oriented programming has been compared to solutions based on semiotic theory. In some cases they agree, in others they disagree. The two

approaches agree in seeing the computer system as a sign system standing for events and objects in the application domain. The main difference between them is that object-oriented programming seems to focus on the designer's and programer's concept of the domain; the main task is to build a model of the domain which is then later used as a basis for providing information users need. Thus, model first, interfaces afterwards.

In opposition to this, the semiotic approach focuses on the execution viewed as sign-vehicles users interpret in their context of work. It does not advocate the opposite course of action, interface first, model afterwards, simply because in this approach interfaces should not be separated from functionality. Instead the recommendation is to view the execution in analogy with a theater performance, and to design the classes and objects of the system as actors, props and set-pieces in this drama.

How different the two approaches really are is difficult to judge. After all, one of the founders of the object-oriented paradigm, Kristen Nygaard, is rumored to have been fond of puppet-theater as a child!

References

Andersen, P. Bøgh and B. Holmqvist, (1990a). Narrative computer systems. The dialectics of emotion and formalism. In J. F. Jensen, editor, *Computer-kultur — computer-medier — Computer-semiotik [Computer culture — computer media — computer semiotics].* Nordic Summer University, Aalborg.

Andersen, P. Bøgh and B. Holmqvist, (1990b). Interactive Fiction. Artificial intelligence as a mode of sign production. *AI and Society,* 4(4): 291-313.

Andersen, P. Bøgh, (1990a). *A Theory of Computer Semiotics. Semiotic Approaches to Construction and Assessment of Computer Systems.* Cambridge University Press, Cambridge.

Andersen, P. Bøgh, (1990b). Towards an aesthetics of hypertext systems. A semiotic approach. In A. Rizk et al, editors, *Hypertext: concepts, systems, and applications.* Cambridge University Press, Cambridge.

Andersen, P. Bøgh, (1991a). Vector spaces as the basic component of interactive systems.Towards a computer semiotics. VENUS report no 7. Department of Information and Media Science, University of Aarhus. Niels Juels gade 84, 8200 Aarhus N. To appear in *Hypermedia.*

Andersen, P. Bøgh, (1991b). A semiotic approach to construction and assessment of computer systems. In: Nissen, Klein & Hirschhaim, editors, *Information Systems research: Contemporary Approaches & Emergent Traditions,* 465-514. North Holland.

Bannon, L., (1989). From Cognitive Science to Cooperative Design. In: N. O. Finnemann, editor, *Theories and Technologies of the Knowledge Society.* 33-59. Center for Cultural Research, University of Aarhus: Aarhus.

Boland, J. R., (1991). Information system use as a hermeneutic process. In: Nissen, Klein & Hirschhaim, editors, *Information Systems research: Contemporary Approaches & Emergent Traditions,* 439-458. North Holland, Amsterdam.

Chomsky, N., (1957). *Syntactic Structures.* Mouton, The Hague.

Chomsky, N., (1965). *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Mass.

Chomsky, N., (1968). *Language and Mind.* Harcourt, Brace & World, New York.

Chomsky, N., (1976). *Reflections on Language.* Fontana/Collins.

Chomsky, N., (1980). On Binding. *Linguistic Inquiry* 11: 1-46.

Declés, J-P., (1989) Intermediate representations in the cognitive sciences. *Semiotica* 77(1/3): 121-135.

Eco, U., (1984). *Semiotics and the philosophy of language.* Indiana University Press, Bloomington.

Eco, U., (l977). *A Theory of Semiotics.* The MacMillan Press, London.

Figge, U. L., (1991). Computersemiotik. *Zeitschrift für Semiotik* 13(3/4): 321-330.

Freeman-Benson, B. N., J. Maloney & A. Borning, (1990). An Incremental Constraint Solver. *Comm. of the ACM,* 33(1): 54-63.

Goldkuhl, G. & K. Lyytinen, (1982). A language action view of information systems. *SYSLAB report no. 14.* Göteborg, Sweden: Department of Computer Sciences, Chalmers University of Technology and the University of Göteborg.

Halliday, M.A.K., (1976). *System and Function in Language.* Oxford University Press, Oxford, U.K.

Halliday, M.A.K., (1978). *Language as Social Semiotic. The Social Interpretation of Language and Meaning.* Edward Arnold, London.

Hjelmslev, L., (1963). *Prolegomena to a Theory of Language.* Menasha. The University of Wisconsin Press, Winsconsin. Translated from "Omkring Sprogteoriens Grundlæggelse". University of Copenhagen 1943, reprinted and published by Akademisk Forlag, Copenhagen 1966.

Holmqvist, B. and P. Bøgh Andersen, (1987). Work-language and information technology. *Journal of Pragmatics* 11: 327-357.

Holmqvist, B. and P. Bøgh Andersen, (1991). Language, perspective, and design. In J. Greenbaum and M. Kyng, editors, *Design at Work.* Earlbaum, Hillsdale.

Holmqvist, B., (1989). Work-language and perspective. *Scandinavian Journal of Information Systems* 1: 72-96.

Hopcroft, J. E. and J. D. Ullman, (1969). *Formal languages and their relation to automata.* Addison-Wesley Publ. Comp., Reading, Mass.

Kaasbøll. J., (1986). *Intentional development of professional language through computerization. A case study and some theoretical considerations.* Paper for the conference on System Design for Human Development and Productivity: Participation and Beyond. Humboldt-Universität zu Berlin, Berlin.

Korson, T. & J. D. McGregor, (1990). Understanding object-oriented: A unifying paradigm. *Comm. of the ACM* (33/9) : 40-60.

Kristensen, B. B, O. L. Madsen, B. Møller-Pedersen & K. Nygaard, (1991). *Object oriented programming in the BETA programming language.* University of Aarhus, Dept. of Computer Science, Aarhus.

Mathiassen, L. and P. Bøgh Andersen, (l984). Semiotics and informatics: the impact of edp-based systems upon the professional language of nurses. In van der Veer, editor, *Readings on Cognitive Ergonomics - Mind and Computers.* 226-247. Springer-Verlag, Berlin. Also in *Journal of Pragmatics* 10(1986): 1-26.

Montague, R., (1976). *Formal Philosophy. Selected papers of Richard Montague.* Yale University Press, New Haven and London.

Nadin, M., (1988). Interface design: A semiotic paradigm. *Semiotica* 69: 269-302.

Nardi, B. A. and J. R. Miller, (1991). Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *Int. J. Man-Machine Studies,* 34: 161-184.

Nygård, K. & P. Sørgård, (1987). The persepctive concept in informatics. In Bjerknes, G. et al., editors, *Computers and Democracy.* Aldershot, Avebury. 357-379.

Ouellet, P. Semiotics, cognition, and artificial intelligence. Special issue of Semiotica. *Semiotica* 77. 1989.

Piotrowsky, D., (1990). *Structures Applicatives et Language Naturel. Recherches sur les fondements du modele: "Grammaire Applicative et Cognitive".* Ph.D thesis, Ecole des Hautes Etudes en Sciences Sociales, Paris.

Rasmussen, J., (1986). *Information processing and human-machine interaction.* North-Holland, New York.

Reichenbach, H., (1947). *Elements of symbolic logic.* New York.

Rumbaugh, J. et al., (1991). *Object-oriented modeling and design.* Prentice-Hall: Englewood Cliffs.

Saussure, F. de. (1966). *Course in General Linguistics.* McGraw-Hill: New York.

Stamper, R., (1991). The Semiotic Framework for Information Systems Research. In Nissen, Klein & Hirschhaim, editors, *Information Systems research: Contemporary Approaches & Emergent Traditions,* 515-528. North Holland, Amsterdam.

Winograd, T. & F. Flores, (1986). *Understanding Computers and Cognition.* Ablex: Norwood, New Jersey.