University of Colorado at Boulder

Center for LifeLong Learning & Design (L3D)

Department of Computer Science

ECOT 717 Engineering Center
Campus Box 430
Boulder, Colorado 80309–0430
(303) 492–7514, FAX: (303) 492–2844

KNOWLEDGE-BASED COMMUNICATION PROCESSES IN SOFTWARE ENGINEERING

Gerhard Fischer
Department of Computer Science and
Institute of Cognitive Science
Campus Box 430, University of
Colorado, Boulder, CO 80309
gerhard@cs.colorado.edu

Matthias Schneider-Hufschmidt
Siemens ZFE ST SN 71
Otto-Hahn-Ring 6
D-81730 Muenchen
Germany

In *Proceedings of the 7th International Conference on Software Engineering* (Orlando, FL)
IEEE Computer Society, Los Angeles, CA, March 1984
pp. 358-368

NOTE: Paper published while Fischer and Schneider were at the University of Stuttgart, Germany.

KNOWLEDGE-BASED COMMUNICATION PROCESSES IN SOFTWARE ENGINEERING

GERHARD FISCHER and MATTHIAS SCHNEIDER

Project INFORM, Department of Computer Science, University of Stuttgart, Herdweg 51, D-7000 Stuttgart, Fed. Rep. of Germany

Abstract

A large number of problems to be solved with the help of computer systems are ill-structured. Their solution requires incremental design processes, because complete and stable specifications are not available.

For tasks of this sort, life cycle models are inadequate. Our design methodology is based on a rapid prototyping approach which supports the coevolution of specification and implementation. Communication between customers, designers and implementors and communication between the humans and the knowledge base in which the emerging product is embedded are of crucial importance. Our work is centered around knowledge-based systems which enhance and support the communication needs in connection with software systems.

Program documentation systems are used as an example to illustrate the relevance of knowledge-based human-computer communication in software engineering.

Keywords: knowledge-based systems, human-computer communication, experimental programming environments, program documentation, incremental design, rapid prototyping, user interfaces

1. Introduction

Based on our research work of the last few years (building knowledge-based systems, improving human-computer communication (BAUER et al. 1982) and understanding the nature of design processes (FISCHER/BÖCKER 1983)) we are convinced that the currently dominant life cycle models of software engineering (HOWDEN 1982) are inadequate for most problems in the domains mentioned above. They are inadequate because they rest on the assumption (which is unproven for many classes of problems) that at the beginning the requirements can be stated in a precise way and that the complete specifications and the implementation can be derived from them relying primarily on formal manipulations. In reality this is not not the case, especially if we require that our systems (e.g. a user interface or a memory support system) are designed to meet real human needs.

In the first part of this paper we characterize our view of the software engineering process and propose a communication-based model for software engineering as an alternative. We demonstrate the central role of knowledge-based systems to support communication processes between all persons involved in the development, construction, modification and use of software. The second part illustrates our ideas by describing a computer-supported program documentation system which is being implemented as an important part of our research project.

2. Our view of the software engineering process

In software engineering we can differentiate at least between the following three different phases:

1. developing an intuitive understanding of the problem to be solved; the communication between the client and the designer is important in this phase
2. designing a system intended to solve the problem; the designer will look at previous solutions to similar problems and will try to find existing modules which can be used in the design
3. programming an implementation of the design; the implementor will try to show that his implementation is consistent with the specification.

In practice, these concerns are never totally separated nor entirely sequential.

2.1 Ill-structured problems

In software engineering we have to deal mostly with ill-structured problems (HAYES 1979). The problem solver or designer has to face the following tasks:

- he has to contribute to the problem definition by taking an active role in specifying the problem
- he has to make decisions to fill gaps in the problem definition
- he is required to "jump into the problem", i.e. he must attempt a solution before he fully understands it.

Situations in which the client cannot provide detailed and complete specifications are typical for ill-structured problems. Therefore many of the methodologies and tools developed in software engineering are of little use. Requirement specification languages are supposed to enable the developers to state their understanding of the user's ideas in a form comprehensible to the user -- but the user himself has only very vague ideas of what he wants.

2.2 Example domains: Human-computer communication and knowledge-based systems

HCC and knowledge-based systems are two research domains with mostly ill-structured problems. The main difficulty in these domains is not to have a "correct" implementation with respect to given specifications, but to develop specifications which lead to effective solutions which correspond to real needs. Correctness of the specifications is in general no meaningful question because it would require a precise specification of intent, and such a specification is seldom available.

Modern user interfaces (based on multiple windows, menus, pointing devices and using the screen as a truly two-dimensional medium) offer a huge design space. Within this design space little is known about how to present and structure information so that a human can make full use of it (see Figure 2-1 and 2-2).

Knowledge-based systems are an effort to put more knowledge (about the problem domain, about communication processes, about design and problem solving and about the user) into the machine. This knowledge should be used to make systems more cooperative, absorb some of the complexity and provide better support in problem solving.

2.3 What can we do without complete specifications?

We have to accept the empirical truth that for many tasks system requirements cannot be stated fully in advance - in many cases not even in principle because the user (nor anyone else) does not know them in advance.

The development process itself changes the user's and designer's perceptions of what is possible, increases their insights into the application environment, and indeed often changes the environment itself.

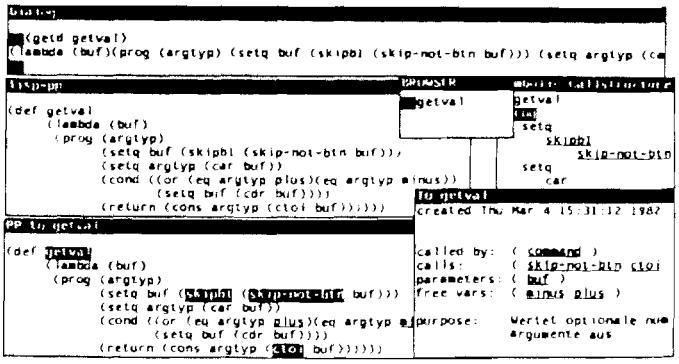

Figure 2-1: Screen layout of a program analysis system

The screen displays multiple windows, in which different perspectives are shown: a dialogue window for typein; two windows showing "pretty-print" structures; a window showing the symbolic calling structure and a window showing the descriptive structures computed by our system (e.g. called by, calls, free variables, etc). The information describing the calling structure is used to drive the editor if a change is made to the external structure of a procedure (FISCHER et al. 1981).

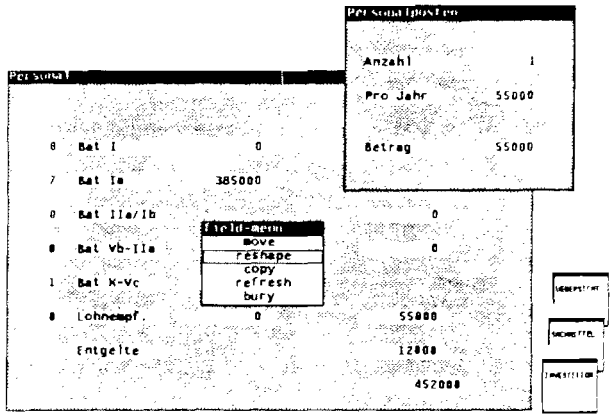

Figure 2-2: Screen layout for a financial planning system

The screen shows menus (the commands contained in them can be activated with a pointing device) and windows (which are viewers into the same knowledge base); the program can be considered as a knowledge-based version of Visicalc (i.e. each field has its own parser and is part of a dependency network) (RATHKE 1983)

The following (not mutually exclusive) possibilities exist to cope with this situation:

1. Development of experimental programming systems (DEUTSCH and TAFT 1982; SHEIL 1983) which support the

coevolution of specifications and implementations. Prototypical implementations allow us to replace anticipation (i.e. how will the system behave) with analysis (i.e. how does it actually behave), which is in most cases much easier. Most of the major computing systems (operating systems, editors, expert systems, software development systems) have been developed with extensive feedback (based on their actual use) which continually contributed to improvements as a response to discrepancies between a system's actual and desired state.

2. Heavy user involvement and participation in all phases of the development process. The user should be able to play with the preliminary system and to discuss the design rational behind them. An existing prototype makes this cooperation between designer and user much more productive, because the user is not restricted to reviewing written specifications to see whether or not the system will satisfy his needs for the right functionality and ease of use.

3. Let the end-user develop the systems; this would eliminate the communication gap altogether. He is the person who knows most about the specific problem to be solved and by giving him the possibility to change the system there is no necessity any more to anticipate all possible future interactions between user and system. "User tailorability" (e.g. to define keyboard macros in text processing systems or to create forms forms with a general form kit; HERCZEG 1983) is a first step towards "convivial systems" (FISCHER, HERCZEG, MAIER 1983), which give the user the possibility to carry out a constrained design process within the boundaries of the knowledge area modelled.

4. Accept changing requirements as a fact of life and do not condemn them as a product of sloppy thinking: we need methodologies and tools to make change a coordinated, computer-supported process.

Knowledge-based systems combined with modern techniques for human-computer communication (see Figure 4-1 below for the general architecture of such a system and Figure 2-1 and 2-2 for two implemented prototypical systems), are the most promising approaches to cope with this situation.

3. The changing needs in software engineering

Design without final, precisely definable goals is possible (SIMON 1981) and in many cases inevitable. Constructing complex designs in software engineering which are implemented over a long time and continually modified in the course of implementation has much in common with other creative activities (like writing, painting, composing or building models with technical construction kits (FISCHER/BÖCKER 1982).

We need methodologies and tools which are adequate to cope with situations in which the specifications are unavailable, incomplete, change over time or have such an immense size that it is impossible to understand them fully.

It is natural to look for such methodologies and tools in subject areas which have dealt with this situation for a long time. Artificial Intelligence (AI) research has always tried to solve ill-structured problems. In AI, one of the main objectives to write programs was to get a deeper understanding of the problem. In addition, intelligent activities are complex, therefore AI programs are complex and large. Tools were needed to absorb some of the complexity. The efforts to create good programming environments (SANDEWALL 1978, TEITELMAN and MASINTER 1981, SHEIL 1983) have been a major focus of AI research during the last twenty years. The creation of good programming environments was easier for the AI community, because in LISP programs and data have the same representations and lend themselves easily to program manipulation.

Similarly the development of new user interfaces has encountered some of the same problems. Until the appearance of the STAR and the LISA machines only few people have done research in this area (the SMALLTALK development at Xerox PARC has been the most notable exception).

The AI view of programming has been for a long time that a program should not only be regarded as a piece of text understood by a compiler ("a program is more than its listing") but as a complex knowledge-structure with includes (see Figure 3-1)

- the program-text
- documentation information
- design information
- knowledge about a complex artifact put together from pieces.

```
Sample Data Structure
(defobject prem
  (name prem)
  (supers function-description)
  (status ANALYZED)
  (code
   (def prem
    (lambda (key1 key2)
     (let ((i (phashit key1 key2)) (a nil))
       (setq a (passociation i key1 key2))
       (cond (a
              (store (put-get-hash-table i)
                     (cond ((eq
                             (car (put-get-hash-table i))
                             a)
                            (cdr (put-get-hash-table i)))
                           (t
                            (delq
                              a (put-get-hash-table i)))
                     ))))))))
  (in-package pputget)
  (is-called-by)
  (calls passociation phashit)
  (type function)
  (parameters ((key1) (key2)))
  (local-variables (i (TYPE NUMBER))
                   (a (TYPE NUMBER)))
  (free-variables)
  (see-also (pputget-description))
  (history ((DEFINED 10/14/1983
                     (programmer HDB)
                     (reason
                      " "))
            (MODIFIED 12/12/1983
                      (programmer HDB)
                      (reason
                       "prem didn't work if the property
                       to be deleted was the CAR of the
                       appropriate bucket"))))
  (version 2)
  (side-effects (PUTACCESS put-get-hash-table))
  (purpose "removes properties from the hashtable")
  (description
   "this function removes the appropriate association-list
    entry from the hashtable. If the right entry is the
    first entry of the association-list, delq won't work,
    so catch this event first.")
   (scratch-pad " "))


                    Conventions
1. Reverse Video: slot names of our knowledge units
2. Underlined: data that can be interpreted, used and
               updated by the system
3. Normal font: knowledge generated by the user,
                commentaries etc.
4. CAPITALS: system-generated information
```

Figure 3-1: A sample function-description

AI has developed a set of tools for coping with knowledge-based systems:

- general purpose knowledge-base dependency analysers, e.g. tools for monitoring all changes made to objects in the knowledge base. The calling structure of Figure 2-1, for example, can be used to drive the editor to update our programs after we have changed the number of parameters of a procedure.

- indexing tools (e.g. Browsers) for classifying and retrieving objects on the basis of selected properties

- inference and truth maintenance mechanisms (see Figure 5-2 where the callees-slot has been filled automatically)

- multiple views, generated by user-definable filters (see Figure 5-2 and 5-4)

- constraints (e.g. to enforce the consistency between different representation; e.g. between the program text and its description).

Within AI research new methodologies were developed like structured growth (e.g. a partially implemented system can be run and tested) and programming by specialization (which is supported by the inheritance mechanism in object-oriented languages).

The experimental, error-correcting approach which is characteristic for a rapid prototyping methodology is not an excuse for being unable to think clearly enough but it is a respectable scientific activity (see Popper's remarks about the "critical method to eliminate errors" (POPPER 1959), Simon's theory of bounded rationality (SIMON 1981) and Alexander's claim that we see good fit only from a negative point of view (ALEXANDER 1964)).

4. Knowledge-based models for communication

As an alternative to the life cycle model we propose a communciation model. The knowledge-based systems which we develop for software engineering with respect to this model support the following two activities:

1. the communication between a human and the knowledge base which represents the emerging product

2. the communication between the different classes of humans (e.g. designers, users) : in this case the computers serves as a structured media for communication.

Human communication and cooperation can be used as a model to define the general characteristics of a knowledge-based system of this sort. What can humans do that most current computer systems cannot do? Human communication partners

- do not have the literalism of mind which implies that not all communication has to be explicit; they can supply and deduce additional information which has not been explicitly mentioned and they can correct simple mistakes

- can apply their problem solving power to fill in details if we give statements of objectives in broad functional terms

- can articulate their misunderstanding and the limitations of their knowledge

- can provide explanations (especially for the information which gets exchanged over the implicit communication channel, see Figure 4-1 below).

Knowledge-based systems are one promising approach to equip machines with some of these human communication capabilities. The work in our research project INFORM (BAUER et al., 1982) is guided by the model shown in Figure 4-1.

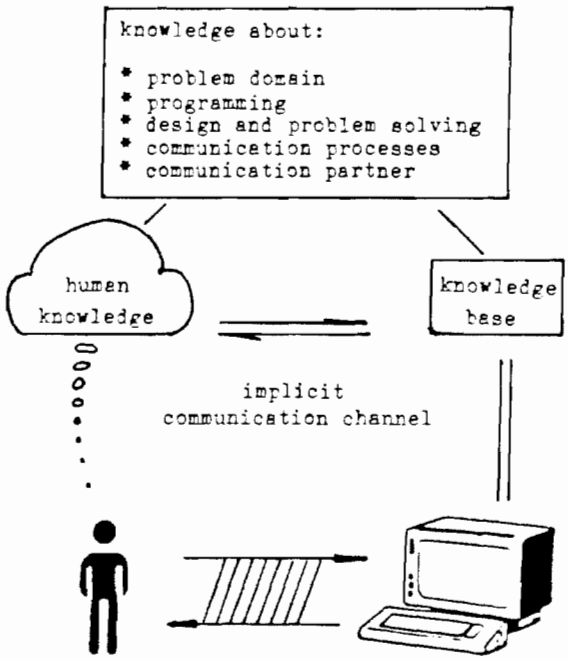

Figure 4-1: Architecture of a knowledge-based system to support HCC

The system architecture from Figure 4-1 has two major components:

1. The explicit communication channel is widened. Our interfaces use windows with associated menus, pointing devices, color and iconic representations: the screen is used as a design space which can be manipulated directly (see Figure 2-1 and 2-2).

2. Information can be exchanged over the implicit communication channel. Both communication partners have knowledge which eliminates the necessity that all information has to be exchanged explicitly.

The five domains of knowledge shown in Figure 4-1 have the following relevance for software engineering:

1. knowledge of the problem domain: a programming system must have knowledge about the task domain, otherwise a domain expert cannot communicate with it in a natural form (BARSTOW 1983)

2. knowledge about programming (RICH, SHROBE and WATERS 1979): such knowledge consists of program fragments (abstracted to various degrees), each associated with a variety of propositions about the behavior of the schema and indexed so that they can be applied to large classes of problems

3. knowledge about design and problem solving: it is important not only to retain the finished product but the important parts of the process: we should be able to explore alternative designs in different contexts and merge them if necessary (GOLDSTEIN and BOBROW, 1981: FISCHER and BÖCKER 1982)

4. knowledge about communication processes: the information structure which controls the communication should be made explicit, so the user can manipulate it; scripts (actions which are in general carried out in a sequence) should be available

5. knowledge about the communication partner: the designer of a program wants to see quite different parts compared to a programmer who wants to use the module only as a package. For the user information will be relevant which helps him to create a consistent model of the system, to know how to invoke certain subsystems and to link the behavior of the system to the underlying design rationale.

A knowledge-based architecture and new, two-dimensional interfaces are important components for convivial systems (see section 2.3). The additional freedom for the user of a convivial system (giving him the possibility to carry out a constrained design process within the boundaries of the knowledge area modelled) increases the functionality and the complexity of the communication process; novel interfaces are required that the user can take full advantage of the possibilities given to him. The representation of arbitrary information structures in knowledge-based systems allows that

the communication process can be shaped dynamically, which implies

- the user can modify a program according to his specific needs and the designer does not have to anticipate every interaction in advance
- static documentation structures (traditionally stored as canned text) can be generated dynamically and can serve as a user- and problem-specific explanation.

We believe that the development of convivial tools will break down an old distinction: there will be no sharp borderline any more between programming and using programs -- a distinction which has been a major obstacle for the usefulness of computers. In the open systems of the future (as they are needed in domains like office automation) a convivial system is a "must", because the system designer cannot foresee in detail all the specific needs which the users will have in the future.

5. Program documentation systems

We consider a program documentation system to be the heart of a software engineering environment, because it serves as the communication medium between different users and the knowledge base of the system.

The importance of a documentation system stems from the large range of different tasks a documentation is useful for:

- to enhance the designers understanding of the problem to be solved and to assist him in improving the problem specifications
- to support the designer during the implementation of his solution
- to enable a programmer to reuse a program and to extend existing systems to tool kits
- to maintain a programming system
- to make a program portable and sharable in a larger community

A program documentation system is a knowledge base containing all of the available knowledge about a system combined with a set of tools useful for acquiring, storing, maintaining and using this knowledge (SCHNEIDER 1981).

The knowledge base is

- in part interpreted by the computer to maintain the consistency of the acquired knowledge about structural properties; it supports the user in debugging and maintaining his program system
- in part only useful for the user, i.e. not directly interpretable by the machine. In this case the machine serves as a medium for structured communication between the different users. The computer can support the user to maintain the non-interpretable information in the knowledge base, do user-guided change of information (by driving the editor (see section 3)), suggest the updating of possibly inconsistent data in the knowledge base, etc. (see Figure 3-1).

A documentation system should support the entire design and programming process. A valid and consistent documentation is of crucial importance during the programming process itself. The information structures that are accumulated around a program (see Figure 3-1) can be used to drive an evolutionary and incremental design process. Documentation should not only be done at the end of the implementation but throughout the whole design process (see Figure 5-1).

In writing large manuscripts, document preparation systems (e.g. SCRIBE; REID and WALKER 1980) offer many services (e.g. automatic generation and maintenance of a table of contents and an index) which are of crucial importance during the process of writing. The amount of work that has to be done to keep the documentation of a large program or manuscript up-to-date with each incremental change is far too large to be done manually by the designer. Support is necessary for

- creating information about the structural properties of a system
- keeping the whole documentation consistent.

This task should be done by a documentation system.

By improving the specifications of a problem the designer gets new ideas about how to solve his problem. A documentation of a program that is usable during the design process can therefore be a driving force for the synthesis of new ideas and implementations (see Figure 5-2).

We gain the full benefit of a program documentation system only, if it is an integral part of an integrated programming environment. A program documentation produced as a separate document by a word processing system has at least the following disadvantages:

- it is impossible to provide pieces of information automatically

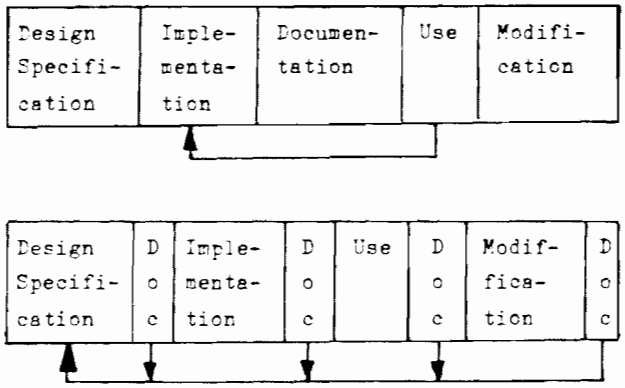

Figure 5-1: Alternative views on the position of documentation in the programming-process

In the traditional view documentation is created at the end of the programming process; in our model (SCHNEIDER 1981) documentation serves as the communication medium for all people involved with a software product. Documentation is useful throughout the entire process and serves as a starting point for new solutions of the problem. The purpose of a documentation in this view is comparable to that of a proof in mathematics: a crystallization point for new ideas (LAKATOS 1977).

- it is impossible to maintain consistency between the program and its documentation automatically (or at least semi-automatically)
- it is impossible to generate different external views dynamically from one complex internal structure (e.g. to read a documentation either as a primer or as a reference manual)
- it is impossible to create links between the static description and the dynamic behavior.

Modern systems for human-computer communication (see Figure 2-1 and 2-2) which are dynamic (i.e. the screen is constantly changing) can only be insufficiently described with a static medium like pencil and paper. Therefore it is difficult (if not impossible) to give a detailed and precise specification and documentation of such systems using less powerful media.

5.1 Program documentation for whom?

Program documentation has to serve different groups who try to perform different tasks. Therefore the amount and quality of information offered to these groups of people has to be different. We distinguish the following groups and their tasks:

- the designer of a system during the programming process (see Figure 5-1 and 5-2). He has to have access to his design decisions and the different versions of the system. He also needs information about the state of his work in the whole design process. At any point we should be able to ask questions of the following kind:
  * What has still to be done?
  * Which parts of the problem are still unsolved?
  * How far did the designer get in his plans and with respect to his task?
- the programmer who is trying to reuse or modify a program that he does not know yet. He first wants to understand the purpose and algorithms of the program to decide which parts of it have to be changed to fit his needs. He needs information about design-decisions (in order to avoid known pitfalls) as well as a thorough documentation of the existing code.
- the client who is trying to find out whether the implemented system solves his problem. He wants to improve his own understanding by working with a prototypical version of the system and is therefore not interested in any programming details but in design decisions.
- the user wants to see a description in terms of "What does it do? How can I achieve my goals?"; for end-users the documentation has to offer different views of the system: a primer-like description for the beginner and manual-type explanations for the expert (see Section 5.4, Figure 5-2 and 5-4).

5.2 Knowledge Acquisition and Updating

The information structures which are used in our system come from two sources:

- the analyzing system PAMIN (FISCHER et al., 1981) provides information about the structural properties (cross references, side effects) of a program. This system's functionality is similar to the one of MASTERSCOPE (TEITELMAN and MASINTER 1981). The user doesn't have to provide information that can be created automatically, so he is free to concentrate on the creative aspects of his work.
- the programmer has to provide semantic information about the different parts of the program, information about the internal (semantic) structure of his system, descriptions of the used algorithms etc.

Most of the analysis done by the system is done at read-time (see Figure 5-2). This means that we have to do the analysis after each alternation of the program code. The system knows about possible dependencies between knowledge units and , if necessary, reanalyzes the units in question. It informs the programmer about possible inconsistencies in the knowledge base. These techniques help us to maintain the consistency between different representations of the information.

```
prem (FUNCTION) FILTER: normal
in-packages:
 pputget
callers:

callees:
 phashit passociation
purpose:
 removes properties from the hashtable
description:
 this function removes the appropriate association-list
 entry from the hashtable. If the right entry is the
 first entry of the association list, delq won't work,
 so catch this event first.
code:
 (def prem
   (lambda (key1 key2)
     (let ((i (phashit key1 key2)) (a nil))
          (setq a (passociation i key1 key2))
          (cond (_
see-also:
```

Figure 5-2: Documentation during programming

When the user starts to define a new function (or package) the system creates a new knowledge unit (as shown in Figure 3-1) and inserts inferred information as soon as possible. If the user deletes already typed code the system extracts the information derived from this piece of code (LEMKE and SCHWAB 1983)

The way the system decides if a knowledge unit has to be updated is the following:

- the system knows that it has to change certain structural information (e.g. calls - is-called-by relations) automatically. The system is able to alter information by using its cross-reference knowledge. This knowledge can also be used to guide the user to places where he possibly wants to change information.

- for each unit the user can provide a list of other knowledge units he wants to inspect and possibly alter if a unit has been updated (see the "see-also"-slot in Figure 5-4; this information cannot be created by automatic inspection of the code.)

5.3 Object-oriented knowledge representation

Object-oriented knowledge representations have the following advantages:

- for many problems an object-oriented style of description (e.g. like in SMALLTALK or OBJTALK (RATHKE and LAUBSCH 1983)) provides a good model for the designer's understanding of the domain

- inference mechanisms to deduce properties of the program are bound to objects (in our case knowledge units which represent the knowledge about a small part of the documented program) by means of methods; therefore the effects of these inferences can be kept local and reduce the complexity of the deduction process

- effects of changes are kept local by using the same technique of defining methods to propagate necessary changes to other objects

- objects own methods that define different views on their knowledge; new views on a knowledge unit to fit the needs of a user are created by defining other methods by the user himself.

Our system uses OBJTALK (RATHKE and LAUBSCH 1983) as the implementation language for the knowledge base. It is a good descriptive mechanism to model our problem domain (documentation of LISP-programs). The basic units are frame-like structures (MINSKY, 1975) that incorporate different kinds of knowledge about the analyzed items. Information is organized around the concepts of packages (the largest package being the entire system) and functions. Additionally there is a concept called a filter (see section 5.4) which provides the user with the possibility to create his own filtered views on the information units. Figure 3-1 shows a sample knowledge unit for an analyzed and documented function.

5.4 Using the available knowledge

A knowledge-based program documentation system is only useful if the relevant information can be easily obtained. The following two requirements must be supported:

1. availability: the knowledge about the system (incorporating the consequences of all changes) must be available at any time (see Figure 5-1).

2. views of reduced complexity: the structures in our knowledge base are too complex (see Figure 3-1) to be used directly. A filter mechanism where the filters can be defined by the user (see Figures 5-3 and 5-4) allows to generate views of reduced complexity showing only the information which is relevant for a specific user at a specific time.

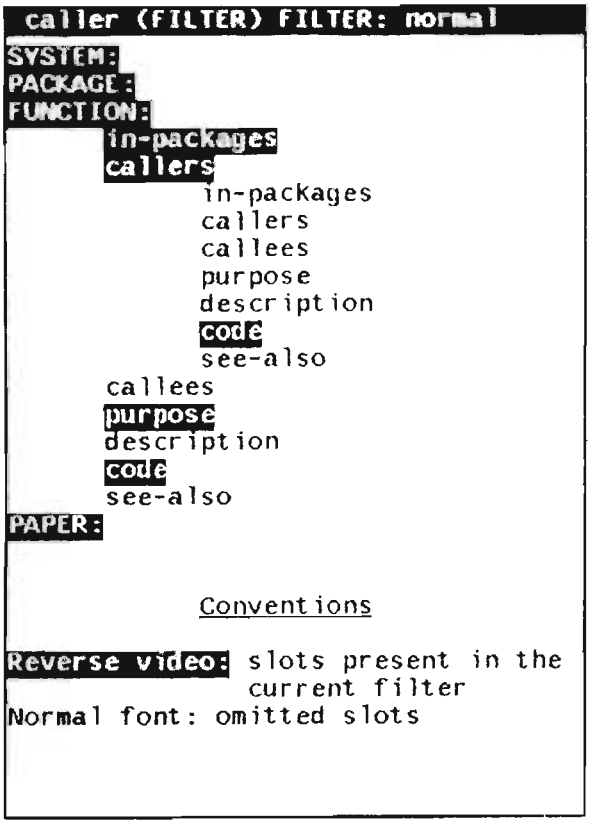

Figure 5-3: Definition of a filter

The user can create his own views of a knowledge unit. In the example given the user wants to see information about called functions (LEMKE and SCHWAB 1983).

5.5 Human-Computer Communication (HCC) techniques to enhance program documentation

The broad functionality of future computer systems can only be achieved with complex programs. The descriptive structures around them will be even more complex. Communication between users of these large systems and their knowledge bases will be impossible without an easy-to-use human-computer interface.

Our research on HCC shows several ways to enhance the communication between users and the documentation.

1. The user may decide which information he wants to insert or inspect. Knowledge acquisition can be user driven (by selecting and filling appropriate knowledge units) or guided by the documentation system (by asking for necessary information).

2. Default values for certain knowledge units enable the user to concentrate on areas that are of interest to him.

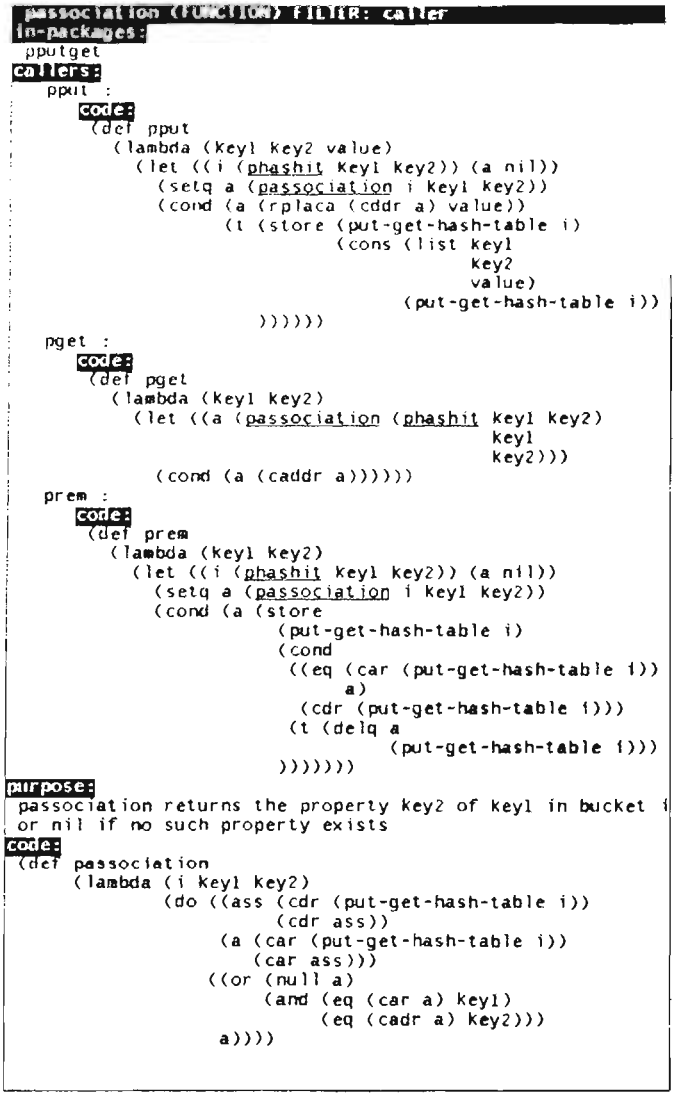

Figure 5-4: A filtered view on a function

After having defined a filter for a knowledge unit (see Figure 5-3) the system generates a representation of the information structure showing the code of the function and its callers. This view allows the user to easily alter the name or parameters of the described function (LEMKE and SCHWAB 1983).

However, the user is always able to change these default values.

3. Multiple windows are used to focus the attention on specific issues: they provide different contexts to avoid confusion.

4. Filters generate user- and context-specific information structures of reduced complexity (see 5.1). The user can define his own filter to see just the information he wants to see (see

Figure 5-3); these techniques help the user to build models of the system at different levels of abstraction.

6. Conclusions

Experimental programming environments, built as knowledge-based systems and accessible by a good human-computer communication will not be a luxury but a necessity to make the software engineer more productive and to develop systems which serve real human needs.

We have developed a framework to pursue some of the stated problems. The program documentation system that we have implemented serves currently mostly as a memory support system. Many parts of our knowledge structures are not interpreted by the machine but are presented to the human at the right time, under the desired perspective and with the appropriate level of detail. We have moduls which automate some of the knowledge acquisition (mostly by analyzing the program code) and we have simple mechanisms to maintain the consistency among different representations. All components are embedded in a LISP environment and can be accessed by a uniform, high-bandwith interface using windows, menus and a pointing device.

To investigate the real potential of our approach many more problems remain to be solved. To mention some of the important ones:

- more parts of our knowledge structures must be formalized that they can be manipulated by the computer
- evolutionary, incremental development and modification must be supported by the computer (e.g. a dependency network must propagate the implications of a small change our knowledge structures)
- the computer needs more knowledge about: specific subject domains, design, programming, users and communication processes; more support is necessary to acquire, represent and utilize this knowledge.

We hope to contribute with this work to one of the key problems of our time: to understand the potential of computers in domains that call for human understanding and intelligence; an area where Artificial Intelligence and Software Engineering can benefit from each other.

References

Alexander, C. (1964): "The Synthesis of Form", Harvard University Press

Bauer, J., H.-D. Böcker, F. Fabian, G. Fischer, R. Gunzenhäuser and C. Rathke (1982): "Wissensbasierte Systeme zur Verbesserung der Mensch-Maschine Kommunikation", MMK-Memo, Institut für Informatik, Stuttgart

Barstow, D. (1983): "A Perspective on Automatic Programming", Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, pp 1170-1179

Deutsch, P.L. and E.Taft (eds.)(1980): "Requirements for an Experimental Programming Environment", Xerox Corporation, Palo Alto, California

Fischer, G. and H.-D. Böcker (1983): "The nature of design processes and how computer systems can support them", in P. Degano und Erik Sandewall: "Integrated Interactive Computer Systems", North Holland, Amsterdam, pp 73-86

Fischer, G., J. Failenschmid, W. Maier and H. Straub (1981): "Symbiotic Systems for Program Development and Analysis", MMK-Memo, Institut für Informatik, Stuttgart

Fischer, G., M. Herczeg and D. Maier (1983): "Knowledge-based systems for convivial computing", MMK-Memo, Institut für Informatik, Stuttgart

Goldstein, I. and D. Bobrow (1981): "An Experimental Description-Based Programming Environment: Four Reports", Xerox Corporation, Palo Alto, California

Hayes, J.R. (1978): "Cognitive Psychology. Thinking and creating", Dorsey, Homewood, Illinois

Herczeg, M. (1983): "DYNAFORM: ein interaktives Formularsystem zum Aufbau und zur Bearbeitung von Datenbasen", in Balzert, H. (ed): "Software Ergonomie", Teubner Verlag, Stuttgart, pp 135-146

Howden, W.E. (1982): "Contemporary Software Development Environments", Communications of the ACM, Vol.25, Nr. 5, May 1982, pp 318-329

Minsky, M. (1975): "A Framework for Representing Knowledge", in: P.H. Winston (Ed) "The Psychology of Computer Vision, McGraw Hill, New York, pp 211-277

Lakatos, I. (1977): "Proofs and Refutations", Cambridge University Press, Cambridge

Lemke, A. and T. Schwab, (1983): "DOXY: computergestützte Doku- mentationssysteme" Studien-Arbeit Nr. 338, Institut für Informatik, Stuttgart

Popper, K.R.(1959): "The Logic of Scientific Discovery", New York

Rathke, C. (1983): "Wissensbasierte Systeme: Mehr als eine attraktive Bildschirmgestaltung", in: Computer-Magazin, Heft 3, 1983, pp 40-41

Rathke, C. and J. Laubsch (1983): "OBJTALK: eine Erweiterung von LISP zum objektorientierten Programmieren", in H. Stoyan and H. Wedekind (eds.): "Objektorientierte Software- und Hardwarearchitekturen", Teubner Verlag, Stuttgart, pp 60-75

Reid, B.K. and J.H. Walker (1980): "SCRIBE User Manual", Unilogic, Pittsburgh

Rich, C.H., H. Shrobe and R. Waters (1979): "Computer Aided Evolutionary Design for Software Engineering", MIT AI Memo 506, Cambridge, Massachusetts

Sandewall, E. (1978): "Programming in an Interactive Environment: the LISP Experience", in ACM Computing Surveys, Vol 10, No 1, March 1978, pp 35-71

Schneider, M. (1981): "Rechnerunterstützte Dokumentationssysteme für Software", Diplom-Arbeit Nr. 165, Institut für Informatik, Stuttgart

Sheil, B. (1983): "Environments for exploratory programming", in Datamation, February 1983

Simon, H.A. (1981): "The Sciences of the Artificial", MIT Press, Cambridge, MA, 2nd Edition

Teitelman, W. and L. Masinter (1981): "The Interlisp Programming Environment", Computer, pp 25-33