VTT PUBLICATIONS NO. 243

This document is my doctoral thesis that was accepted in 1995 at the University of Oulu, Finland. What I have said on the pages of this document, is still valid. Nowadays, I write books that teach computer programming in the most natural way. You can find more information about my books at

www.naturalprogramming.com

The only thing that I do not like in this thesis is the small example of a computer program of which I show many versions. The program checks the validity of a customer number. As I am now more experienced in writing simple textbook examples of computer programs, I could produce a better example program for this thesis. So, when you read this thesis, please forgive me that the example program is somewhat artificial although it fulfils its purpose in showing different programming styles.

Oulu, Finland, October 12, 2003. Kari Laitinen.

Natural naming in software development and maintenance

Kari Laitinen

VTT Electronics / Embedded Software

Academic dissertation to be presented, with the assent of the Faculty of Science, University of Oulu, for public discussion in the Auditorium L10, Linnanmaa, on October 12th, 1995, at 12 o'clock noon.

TECHNICAL RESEARCH CENTRE OF FINLAND

ESPOO 1995

This thesis was published as a book by the Technical Research Centre of Finland (VTT) in 1995 in their VTT PUBLICATIONS series. Something like 300 copies of the book were printed by the VTT offset printing facility. VTT may still be selling copies of the printed book. For more information, go to the Internet addess http://www.vtt.fi

The following information was given on the second page of the printed book.

ISBN 951-38-4781-0

ISSN 1235-0621

UDC 681.3:681.3.06

Keywords: computer systems programs, computer programs, computers, programming, software development, software maintenance, software documentation, programming, naming, natural languages, human factors, documents

ABSTRACT

The understandability of source programs and other types of software documents is important for several reasons. Software developers have to read documents written by their colleagues, and software maintainers often need to study old source programs about which they have no previous knowledge. Naming is one important factor that affects how understandable source programs are. In general, natural naming means avoiding abbreviations in software documentation. In the context of source programs, natural naming means that program elements such as variables, constants, tables, and functions should be named using whole natural words of a natural language with respect to the grammatical rules of the same natural language.

This thesis introduces methods and tools to facilitate the use of natural naming in software development and maintenance. To support the use of natural naming in programming, source program elements are classified and specific naming rules provided for different program elements. An analytical name creation method is provided to make natural names in source programs consistent with written text in other types of software documents.

Commonly used programming languages do not require any specific naming rules to be followed. For this reason, an experimental programming language is introduced in this thesis. The language is designed to support the use of natural naming. Existing source programs usually contain many abbreviated names. To make existing programs more maintainable, intelligent disabbreviation tools have been developed as part of this study. Disabbreviation means replacing abbreviated names with natural ones in existing source programs.

The naming methods and tools have been evaluated by testing them in laboratory experiments and in practical software development and maintenance situations. The natural naming principles have been taught to software developers in different organizations. According to feedback received from users and the data collected in the experiments, natural naming is a promising approach to increase the understandability of software documents, and the methods and tools introduced in this thesis facilitate the use of this naming approach.

PREFACE

PERSONAL HISTORY

In 1978, I started studying electrical engineering at the University of Oulu in Finland. The first programming course at the university introduced Fortran. That was a shock to me because I understood virtually nothing about it. Only after studying the textbook by Hakalahti et al. (1978) a couple of years later, did I start to understand the secrets of computing. Actually, I learned so much from that book that I decided to orient my studies towards computers and software. I was then, and still am, interested in writing. I thought that computers were an interesting field because they can be commanded through writing.

After my studies I worked at a company called Oy Edacom Ab (now part of ICL Edacom Oy) as a software developer for cash terminal systems. I participated in many software projects, learned several programming languages, and developed many kinds of source programs ranging from database management to communications software. The work also involved software maintenance. I consider that I got a very good overview of software development while I was working at Edacom. That company also provided me the opportunity to work abroad at other companies which were developing software for Edacom cash terminals.

In September, 1988, I had been working about one and half years at the Dutch Edacom representative, Computercentrum C. van de Velden in Arnhem, The Netherlands. Then, in the morning of September 21, 1988, I wanted to write one piece of source code that would enhance the system we had been developing about half a year. When I wrote that program, I decided not to use any abbreviations in the names I used in the program. It turned out to be possible to write such a program, and that particular program turned out to work very well. (Part of that program is shown in Figure 3 in this thesis.) I cannot tell exactly why I tried this kind of naming style. I had already been using quite long names earlier, but perhaps living in the liberal Dutch society made me willing to try new programming styles. Anyway, I have used this naming style, which I now call natural naming, ever since.

I was able to continue the practical experiments with natural naming when I later worked at the British Edacom representative, Edacom Data Systems in Stansted, England, during the winter and spring of 1989. At Edacom Data Systems I wrote a manual entitled "Suggestions for Software Production and Documentation". That manual includes some rules for natural naming.

In June 1989, I started to work at the Computer Technology Laboratory of the Technical Research Centre of Finland (VTT). The same organization is now part of VTT Electronics. This doctoral thesis is a document that describes most of my work at VTT. My first project at VTT was a technology transfer project called SULATEK. In this project I was able to further develop my ideas about natural naming.

VTT also provided a possibility to go to work abroad. This time I wanted to explore the new world, and I spent the year 1993 at the U. S. Naval Postgraduate School in Monterey, California. There my main research subject was to study how programming languages could be enhanced to make them more suitable for natural naming.

After returning to VTT from the U.S.A. I started working in an ESPRIT III project called AMES, Application Management Environments and Support. This doctoral thesis was completed while I was working on the AMES project. A software maintenance tool called InName was the main result of my work in AMES.

During the time I have been working at VTT, I have simultaneously carried out postgraduate studies at the University of Oulu. These studies have resulted in this doctoral thesis. Although I originally studied electrical engineering, I have carried out my postgraduate studies in the Department of Information Processing Science of the University of Oulu. The reason that I switched to another department within the university is that my subject is rather far from electrical engineering.

ACKNOWLEDGMENTS

Prof. Veikko Seppänen, currently the head of VTT Electronics / Embedded Software, has greatly contributed to my work as he has read nearly all scientific texts I have written. He has a superb ability to give comments, make remarks, and suggest improvements in papers he reads.

Prof. Pentti Kerola from the University of Oulu has served as the supervisor of my postgraduate studies. I have had many encouraging and interesting discussions with Prof. Kerola. He has been especially helpful in increasing my knowledge about methodological aspects of scientific work.

Prof. Neil C. Rowe from the U. S. Naval Postgraduate School responded positively to my request to work abroad. He has been very important for this thesis particularly because he showed me how texts can be manipulated with Prolog, and he made the first disabbreviation tool.

Mr. Jorma Taramaa provided me the chance to work in the AMES project, and develop a better disabbreviation tool. Mr. Markku Heikkilä did excellent work while we were developing this tool.

Dr. Cornelia Boldyreff from the University of Durham, U. K., and Prof. Jorma Sajaniemi from the University of Joensuu, Finland, have served as the official reviewers of this thesis. They have provided me important feedback, and suggested clarifying improvements into this thesis.

Mr. Douglas Foxvog has checked the language in this thesis and in most of my scientific texts. In addition to advice in English, he has given many comments on the content as well. Mr. Adth van Bale has proofread the Dutch language in Figure 8(a).

I wish to express my deepest and warmest gratitude to all the people I have mentioned above.

In addition, I would like to thank the following people: Mr. Derek Alexander, Ms. Aria Arai, Mr. Juhani Eskelinen, Prof. Hannu Hakalahti, Mr. Kari Hakkarainen, Mr. Seppo Huotari, Dr. Ahmad M. Ibrahim, Mr. Ilkka Kallio, Mr. Daniel Keller, Mr. Kari Kumpulainen, Mr. Raino Lintulampi, Mr. Ilkka Marjomaa, Mr. Timo Mukari, Ms. Minna Mäkäräinen, Dr. Tapio Pietikäinen, Prof. Petri Pulli, Mrs. Heli Puustinen, Dr. Bengt-Olof Qvarnström, Prof. Samuli Saukkonen, Prof. Timothy Shimeall, Ms. Carla Stiff, Dr. Shorei Takada, Mr. Aarne Taube, Ms. Eija Tervonen, Mr. Matias Vierimaa, Dr. Seppo Visala, Prof. Dennis Volpano, and Dr. Matti Weckström. These people have helped me in various ways: some have expressed especially positive attitudes towards my work, some have commented my working papers, some have expressed interesting opinions and ideas, some have helped to make this thesis, etc. I would also like to express my thanks to the people working in the AMES project, and to my colleagues at VTT with whom I have worked over the years.

Finally, I would like to thank the organizations which have funded my research. This work has been supported by the Technical Research Centre of Finland (VTT), the Technology Development Centre of Finland (TEKES), the companies which participated in the SULATEK project during the years from 1989 to 1992 (Nokia Mobile Phones Oy, TH-Engineering Oy, Prosoft, Elektrobit Oy, Polar Electro Oy, Pohjois Piiri Oy, and Oy Edacom Ab), the Defense Advanced Research Projects Administration as part of the I3 Project under AO 8939, and the Finnish foundations Jenny ja Antti Wihurin rahasto and Tauno Tönningin säätiö.

Oulu, Finland, August 1995

Kari Laitinen

LIST OF INCLUDED PUBLICATIONS

This thesis includes six papers which have been accepted in proceedings of international conferences and scientific journals. These papers are used here with the permission of their original publishers. The included publications are the following: (**WARNING: The papers are not included in the .pdf version of the thesis.**)

I Laitinen, K. and Seppänen, V. 1990. Principles for Naming Program Elements, A Practical Approach to Raise Informativity of Programming. In: Part I of Proceedings of InfoJapan'90 International Conference. Tokyo: Information Processing Society of Japan. Pp. 79-86.

II Laitinen, K. and Mukari, T. 1992. DNN-Disciplined Natural Naming, A Method for Systematic Name Creation in Software Development. In: Proceedings of 25th Hawaii International Conference on System Sciences, Vol. II: Software Technology. Los Alamitos, California: IEEE Computer Society Press. Pp. 91-100.

III Laitinen, K. 1994. Pacific: A Programming Language Based on the Idea of Natural Naming. In: Baeza-Yates, R. (editor) Computer Science 2: Research and Applications. New York: Plenum Press. Pp. 529-540.

IV Rowe, N. C. and Laitinen, K. 1995. Semiautomatic Disabbreviation of Technical Text. To appear in Information Processing and Management.

V Laitinen, K., Taramaa, J., Heikkilä, M., and Rowe, N. C. 1995. Enhancing Maintainability of Source Programs through Disabbreviation. To appear in the Journal of Systems and Software.

VI Laitinen, K. 1995. Natural Naming in Software Development: Feedback from Practitioners. In: Proceedings of 7th International Conference on Advanced Information Systems Engineering (CAiSE*95). Lecture Notes in Computer Science, Vol. 932. Berlin: Springer Verlag. Pp. 375-388.

The author of this thesis is the principal author of all the included papers except PAPER IV. Dr. Veikko Seppänen helped in finding an appropriate structure for PAPER I. Mr. Timo Mukari provided the idea of object-oriented naming for PAPER II. PAPER IV has been mainly written by Prof. Neil C. Rowe, while the author participated in the research and provided some paragraphs, references, and suggestions to improve the paper. Mr. Jorma Taramaa and Prof. Neil C. Rowe provided suggestions and insights for PAPER V. Mr. Markku Heikkilä participated in the research described in PAPER V.

CONTENTS

(**WARNING: The papers are not included in the .pdf version of the thesis.**)

1 INTRODUCTION

1.1 SOFTWARE DEVELOPMENT: A SHORT HISTORY

Now, in 1995, we can say that modern electronic computers are about half a century old. Although computer-like devices were constructed even earlier (e.g. by Charles Babbage), only during the last five decades has the technology of computing equipment been steadily improving.

An important mathematical model for present-day computers was published nearly sixty years ago by Alan Turing (1937a). He imagined a machine which could store and read information from a memory device, a tape, and showed that this imaginary machine could solve mathematical problems. Later, about fifty years ago, Alan Turing, John von Neumann, and other scientists and engineers were already building the first real programmable computers (Hodges 1983). Like modern computers, the early computers could be programmed to perform different kinds of computations. Early programming was done by designing the machine code of an application by hand. Mathematics was used as an aid in the creation of the machine code. Alan Turing, for instance, used base-32 arithmetic when he was designing calculation operations for an early computer. Other people had difficulties in understanding what Alan Turing was doing (Hodges 1983).

From the very beginning, computer programming was seen to be a difficult task. Therefore, different means to aid program construction were developed. These means involved inventing program description rules which made it possible to use other symbols than just numbers to describe the operations of a computer, and to use the computer itself to produce its own machine code for the designed computation. Program description rules which allow machine processing are called programming languages. Simple programming languages, which allow machine instructions and memory locations to be described with symbols, are called assembly languages. More advanced languages, which allow, for instance, the use of mathematical and logical symbols in programs, are called high-level programming languages. The first high-level language, Fortran, was introduced about forty years ago. Other more advanced, and more or less popular, languages have emerged afterwards (Sammet 1972, MacLennan 1983, Horowitz 1987). Although programming with high-level languages was first considered as "automatic programming" (Balzer 1985) and new programming languages have been steadily emerging, programming still seen as a difficult task.

Nowadays it is common to use the terms software development and software engineering instead of programming. Developing an application for a computer is thus seen to comprise many activities other than just writing a program. Software is also a better word than program, because computer applications are not just single programs these days. One application may consist of hundreds or thousands of separate programs. Although there are definitions for the term software (ISO 9000-3 1991, Laitinen 1992), the term is widely used to denote the opposite of hardware. Hardware is easier to define, because it can be touched and seen, whereas software is hard to visualize. As we speak of software development these days, we also speak about more serious difficulties than programming problems. The difficulties of software development have been often labeled with the term "software crisis". That term is more than a quarter of a century old (Tichy et al. 1993). It means the fact that, although hardware technology and production have been evolving rapidly, software producers cannot satisfy the demand for high-quality software as needed. Typical problems of software development include development cost overruns, delays in delivery, and errors in delivered software products.[1]

Despite the difficulties, computers and different kinds of software systems have become enormously popular in today's society. During the first decades of computers they were used for pure mathematical and commercial computing purposes, but nowadays computers control, for instance, air traffic, nuclear power plants, and electronic equipment such as televisions and portable phones. It is possible that our society utilizes computers and software to such an extent that it could not function without them any more. As computers have become popular, the number of people who develop software has grown equally. There are hundreds of thousands, if not millions, of software developers working in the industry today. It is quite a difference, if we think that fifty years ago there were only Alan Turing and other clever mathematicians. Alan Turing did predict that utilization of computers will require many mathematicians (Hodges 1983), but he probably did not imagine that the future software developers would not identify themselves as mathematicians.

To tackle the problems of software development, the research community and industry have produced, in addition to many programming languages, various models for the software development process (e.g. Boehm 1988), methods to analyze and design software systems (e.g. Ward and Mellor 1985, Page-Jones 1988, Yourdon 1989, Coad and Yourdon 1990, Rumbaugh et al. 1991, Booch 1991), and many kinds of tools to help software develop-

1. We started this thesis by referring to Alan Turing's classic paper (Turing 1937a) which can be said to describe one kind of a software system. We cannot help noting also that the paper seems to describe a really typical software system as the first version of the paper contained some errors, which Alan Turing later "patched" with a separate note (Turing 1937b).

With this note we do not want to anyhow neglect the importance of Alan Turing's excellent work, his ideas, or his contributions to this field. The fact that there were mistakes in his classic paper is, in our view, one indication about how complex software systems usually are. Even the most capable brains fail to manage all the complexity.

ers in their work. Software development tools are often designed to support certain development methods. Although software development requires appropriate methods and tools, it is still work in which humans need to co-operate and the success of the work depends on human abilities and skills. For this reason, human factors and psychological aspects of software development have been important research topics (Weinberg 1971, Curtis 1984, Curtis 1985, Hoc et al. 1990).

As the research community is constantly producing new methods and tools for software development, researchers obviously believe that software development indeed can be made more efficient and productive. Brooks (1987), however, has expressed rather pessimistic opinions regarding this matter. He has argued that software systems are inevitably complex products which are hard to visualize. Therefore, developing these systems will remain a difficult task. Harel (1992), on the other hand, is convinced that by using graphical visualization techniques the complexity of software systems can be managed. One problem related to software development methods and tools is also that measuring the efficiency of software development is hard. Fenton (1993) and Fenton et al. (1994) have pointed out that practically none of the existing software development methods have been proved to be more efficient than some other particular methods. Potts (1993) has expressed similar opinions. Glass (1994) has described this situation as the software-research crisis. Jackson (1994) has said that we do not yet understand the nature of software development well enough.

It is obviously very difficult to measure the efficiency of software development as a whole. Otherwise, to this date, researchers would certainly have produced more convincing results. Only a few specific aspects of software development, like the effects of some documentation styles (Curtis 1985), have been measured. These measurements have usually been carried out outside real software development situations. It is possible that measuring the efficiency of software development methods and tools will remain a hard task. We have to live with this possibility.

Despite all the difficulties related to software development, we have to continue seeking new ways to facilitate software development as a human activity. We have already seen many software-based innovations which help people in their work or during their leisure time. This thesis assumes that much more can be done with software and even more clever innovations will be made, if the nature of software development can be better understood and if it can be carried out in more organized manner.

1.2 SOFTWARE DEVELOPMENT AS A DOCUMENTATION PROCESS

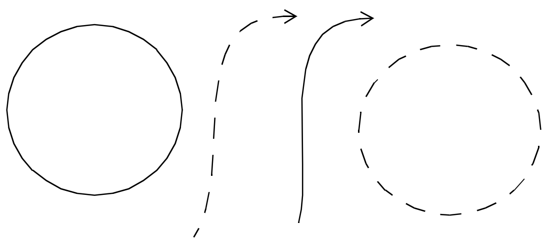
1.2.1 Source programs and other software documents

In its executable form software is a list of numerical instructions which can be directly interpreted by a processor. When executable software is generated using a programming language, a source program is written according to the rules of the programming language. Then, the source program is transformed into an executable form with a compiler. During the transformation process, known as compilation, source programs are transformed from a human-understandable form into a machine-executable form. Machine code is not, in the conventional sense, understandable to humans. This is the reason why programming languages have been invented. High-level programming languages free software developers from thinking about machine instructions or memory locations. Instead, they can concentrate on arithmetic operations, logical decisions, iterations, etc.

Source programs serve a dual purpose in software development activities. On the one hand, they are inputs for software development tools (i.e. compilers), and, on the other hand, they are descriptions which are written, read, and studied by humans. Because of the latter purpose source programs can be considered as documents. All people working in the field of software development do not necessarily have this view of source programs. For instance, Tausworthe (1992) mentions documents and code separately, although he considers both to be the same type of information products. We can, however, treat source programs as documents, because they exist to convey information to people. The importance of the documentary aspect of source programs has been emphasized by Brooks (1978) and Green (1990). In his famous book Weinberg (1971) has stressed that programs are objects which we read and write in order to learn. Also the facts that many software development organizations have documentation rules for source programs and software development standards (e.g. ISO 9000-3 1991, ESA 1991) require these documentation rules suggest that it is important to treat source programs as documents.

Before source programs are written, a typical software development project may produce other types of software documents such as requirements descriptions and various design documents. If we compare all these documents, we can find certain similarities between them. To discuss this matter, let us imagine a situation that a software development project first produces a requirement description document written in English, then a Structured Analysis (SA) model of the system (e.g. Yourdon 1989), and finally source programs written with the C programming language. Examples of the symbols that these software documentation practices employ are listed in Table 1.

Table 1. Examples of typical symbols in software documents.

Document type	Symbol category	Typical symbols
Requirements descriptions written in English	technical symbols	`. , ; : ! ? ( )`
	textual symbols	`a an and the time system must display customer message will has output transmission data command control block from during power`
Graphical-textual data-flow diagrams included in structured analysis and design methods	technical symbols	
	textual symbols	`order customer invalid detail invoice orders ship book books payment item purchase`
Source programs written using the C programming language.	technical symbols	`= == != < > <= >= . , ( ) [ ] { } >> << & * / + - ? int char long double if else do while for`
	textual symbols	`i j k ptr len display msg message number buff s1 s2 string 0 1 2 87 456 999`

By studying Table 1 we can see that usually all types of software documentation practices use some sort of textual symbols. In English they are English words; in data flow diagrams they are names of data flows, transformations, and data stores; in C source programs they are names for various program elements such as variables, tables, constants, and functions, as well as English words in comments. Textual symbols are similar in all types of documents and that makes all documents similar. What makes the difference between the document types in Table 1 are the technical symbols. In English texts, technical symbols are the conventional punctuation symbols, while the structured methods employ graphical symbols such as arrows and circles, and the C programming language has many special arithmetic and logical symbols as well as reserved words.

Although Table 1 is just an approximation about what kinds of symbols may appear in software documents, it displays the real fact that various textual symbols usually exist in every type of software document. On this basis, we can treat source programs as documents which are comparable with other documents produced in software development.

1.2.2 Documents in the development process

Software development activities can be perceived in different ways, depending on the viewpoint. The fact that so much research effort has been put to developing various tools for software development suggests that many people see software development as a technical process which is largely dependent on the tools used. Perhaps the extreme of this view is to consider software development organizations as factories which need factory-like tools (Matsumoto 1987, Cusumano 1989).

We are not claiming that considering software development a technical process would be a mistake, but there are also non-technical aspects in software development. Curtis et al. (1988) have pointed out that software development must be treated, at least partly, as a communication and learning process. We will favor this view in this thesis. People need to study and communicate with other people in order to learn. Documents form an important basis for communication in software development. By studying documents people learn as well. As documents are important in communication and learning, their understandability is an essential factor for the effectiveness of software development.

When considering software development as a learning and communication process we can see it as a documentation process. The documentary view of software development can be characterized as follows:

- Emerging documents are an indication that software developers communicate and learn in their work.

- As the development work proceeds, it produces more and more elaborate documents describing the software system being developed.

- The final stage of software development must result in documents from which the executable form of software can be transformed. We call these documents transformable. Regardless of what documents have been produced earlier, transformable documents must be produced in every software development project. Transformable documents are usually source programs.

- Software development is completed when correct transformable documents (e.g. source programs) have been created.

- The efficiency of software development depends on how quickly the developers can produce correct documents. The capability of producing correct documents is constrained by the learning abilities of the developers.

The documentary view is the basis of this thesis. A similar approach to software development has been taken by Welsh and Han (1994). This view has been described in the context of document classification by Laitinen (1992). Parnas and Clements (1986) have stressed the importance of correct documentation in software development.

In this thesis, we assume that software development always produces source programs as final documents. Because software development practices change rather slowly (Raghavan and Chand 1989), this form of software development will most likely remain a living practice for a long time. However, we must mention that there are software development tools which generate source programs automatically (e.g. ReaGeniX 1994). When these tools are utilized, software developers write descriptions for the program generator which then produces source programs according to the descriptions. In these cases, the final documents of software development are the input descriptions for the generator.

We can identify the following advantages of having the documentation-oriented view of software development:

- Studying the software development process is simpler when we concentrate only on documents. We do not have to worry about the executable machine code, because that can be regarded as a by-product which can always be generated in the development environment when we have the correct documents available.

- Software development can always be treated in the same way regardless of which development methods or tools are used. We can assume that the development methods and tools produce some sort of documents.

- The documentary view of software development is valid with all development models (e.g. the waterfall and spiral models (Boehm 1988)). There are concerns that software development cannot be divided into clearly distinguishable phases in practice (Swartout and Balzer 1982). By concentrating on the documentary outputs of the development process, we can live with the possibility that the phases of software development would always be intertwined in reality.

1.2.3 Documents vs. knowledge

Software developers need to use knowledge from various domains in their work. For this reason we have to specify how knowledge relates to our documentation-oriented approach. The concept of knowledge domain has been popular in research related to software reusability (Seppänen 1990, Prieto-Diaz and Arango 1991). The idea of knowledge domain is in harmony with the approach of considering software development a documentation process. Software development is a process during which the developers learn and integrate knowledge from different domains and incorporate the knowledge they consider necessary into different types of software documents. A similar view of software development has been proposed by Tausworthe (1992) who suggests that people develop software by producing and interacting with information. Program understanding has also been modeled as a process in which software developers use knowledge from various domains (Brooks 1978, Brooks 1983).

As knowledge is an abstract concept which is difficult to measure in exact quantities, it is also hard to give an exact definition for the concept of a knowledge domain. Knowledge can be regarded as existing in peoples' minds (Suitiala 1993), whereas different forms of information (e.g. speech, gestures, and all kinds of written documents) can be regarded as representations of the knowledge in some peoples' minds. By a knowledge domain we mean knowledge related to particular objects or phenomena in the world. In software development, a knowledge domain can be, for instance, knowledge related to a certain communications chip or knowledge of a particular programming language. In order to be able to write programs, one needs to possess knowledge of at least one programming language domain. A person who writes, for example, a program that controls a communications chip, has to integrate knowledge from, at least, two domains: the communications chip domain and a programming language domain. The resulting source program is information that represents at least these two knowledge domains.

Figure 1 illustrates the integration of different knowledge domains in software development. On the left side of the figure various knowledge domains are represented. From these domains information is extracted and transformed into different kinds of software documents shown on the right side of the figure. In order to construct different types of software documents, we need domains for knowledge representation, which are illustrated in the middle of the figure.

According to Figure 1, there is no exact relation between documents and knowledge domains. What kind of knowledge a document represents depends on the writer of the document. However, high-level documents (e.g. requirements descriptions) usually represent fewer knowledge domains than low-level documents (e.g. source programs). High-level documents do not deal with implementation details. Therefore, they do not have to represent the knowledge of implementation domains (e.g. programming languages or communications chips).

1.2.4 Documents in software maintenance

Potts (1993), for instance, has noted that the border between software development and maintenance is vague. People start speaking about software maintenance when a software system is being modified after it has been delivered to customers. However, if an error needs to be corrected in a software system, the work to be done is in most cases the same regardless to whether it is done before or shortly after the delivery of the system. When a system is modified a rather long time after its delivery, the situation is different, because the original developers of the system may not be available.

Bennett et al. (1991) discuss four different types of software maintenance: (1) corrective maintenance means correcting evident errors in a software system; (2) perfective maintenance aims at developing new features for an existing system; (3) adaptive maintenance is being done when a software system is modified to comply with an external interface (e.g. different operating system); and (4) preventive maintenance means that an existing system is modified to make it more maintainable, i.e., to make it more amenable to future maintenance activities.

Any of the maintenance types listed above requires that software maintainers need to study and understand software documents. A common problem in software maintenance is that maintainers cannot be sure whether all software documents are up to date. For this reason, source programs are often the only reliable documents for industrial software maintainers (Bennett et al. 1991, Suitiala 1993). Although comment lines in source programs may not always be up to date, the actual source code lines must be correct because the machine code is generated from them.

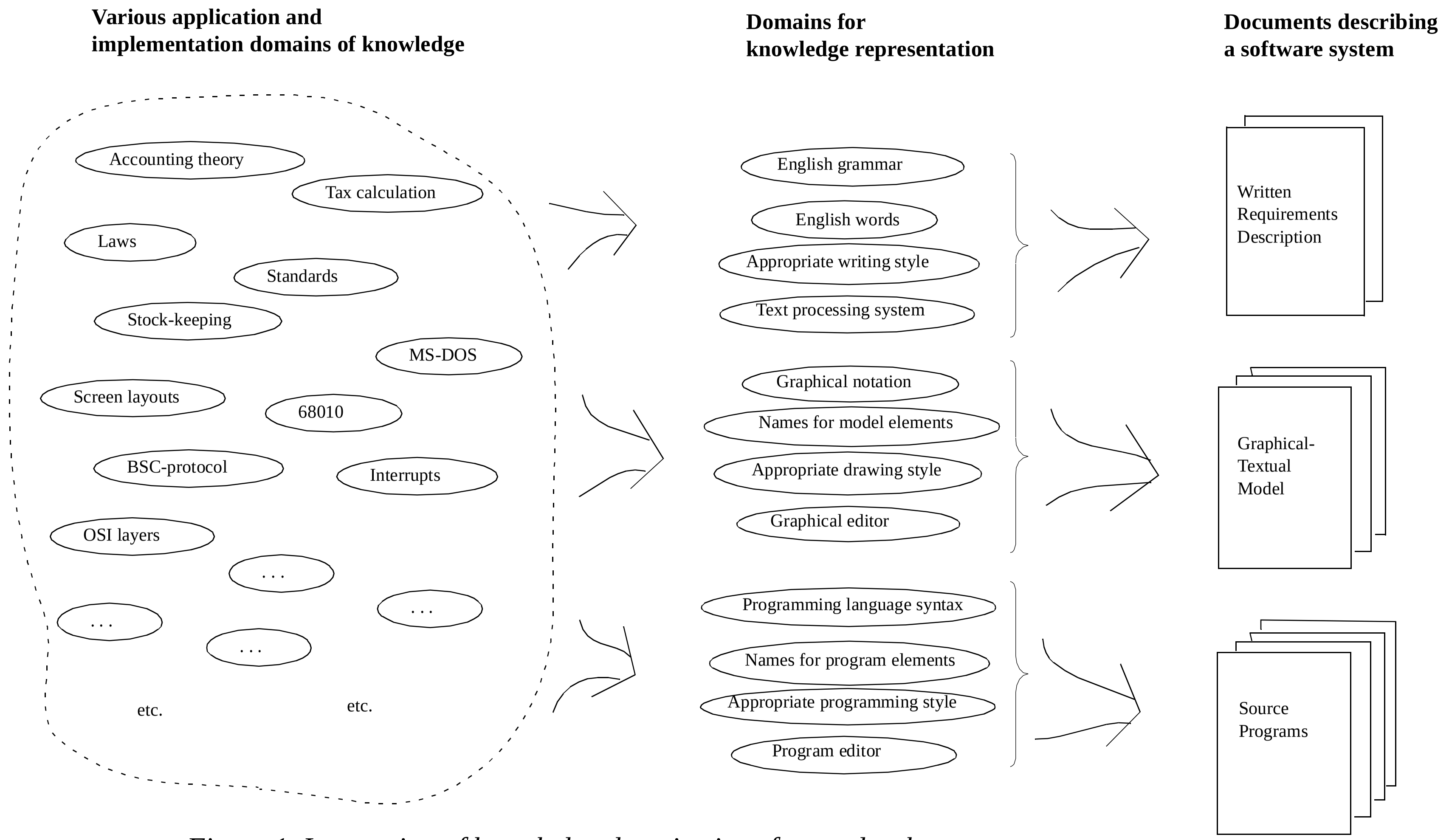


Figure 1. Integration of knowledge domains in software development.

1.3 THE IDEA OF NATURAL NAMING

Source programs are important software documents for the following reasons:

- If we assume that software is being developed using programming languages for implementation, source programs need to be written during software development regardless of what other software documents are written before them.
- Source programs describe a software system completely and exactly as it is, because the executable machine code is generated from them.
- Source programs are, for sure, reliable documents for software maintainers. (We exclude here the possibility that the machine code has been patched.)

For these reasons, this thesis focuses primarily on source programs and secondarily on other software documents. This does not mean that we would not consider the higher-level documents important. On the contrary, we agree that the first phases of software development, and therefore the corresponding documents, are crucial for the success of software development (Jokela 1991). Our primary interest is in source programs, because they are usually the most complex and the least understandable documents in software development.

Figure 2 illustrates two example procedures from a real software system. Those procedures are written with the PL/M programming language provided by Intel Corporation. By studying the program text in Figure 2, we can see that it differs from conventional writing at least in two ways: the text contains special short symbols (e.g. "+", ">=", and "=") as well as special textual symbols (e.g. "NXTBLO", "SEQTBL", and "WLEN2"). At first sight, the text in Figure 2 appears rather incomprehensible. If we study the program text in Figure 2 more carefully, we can note that the textual symbols consist of abbreviations. For instance, the textual symbol "NXTBLO" is obviously an abbreviation of "NEXT BLOCK". If we compare the text in Figure 2 to some more conventional texts like newspapers, novels, and the text in this paragraph, the text in Figure 2 appears to be much less readable and understandable than conventional texts.

The use of different kinds of abbreviations is one factor which makes the text in Figure 2 differ from conventional texts. Figure 2 is, although the program is about ten years old, a typical example of a computer program. Textbooks and research papers as well as real software systems contain similar programs which are written with many abbreviations. There seem to be no strict rules about how to write source programs. We say that the program in Figure 2 is correct because it works and it can be successfully compiled. It is not usually considered an error in writing if a program contains strangely written words such as "NXTBLO". In contrast, the correctness of texts written in English can be judged in terms of the spelling rules and grammar of English. Having these rules in mind, we are able to quickly perceive the tiniest spelling mistakes when reading, for example, a newspaper.[1]

The use of abbreviations in source programs and other software documents will be our concern in this thesis. Abbreviations harm the readability and understandability of source programs. Figure 3 shows another program example in which abbreviations are not so widely used. We claim that the text in Figure 3 is more understandable than the text in Figure 2 because abbreviations have been avoided in Figure 3.

The textual symbols of source programs are called names. For this reason, we use the term "natural naming" to refer to the idea of avoiding abbreviations. The word "natural" is used to emphasize that instead of abbreviations we should use names that consist of natural words. Giving an accurate definition for natural naming is difficult, because the term "natural word" is somewhat inaccurate. New words constantly enter natural languages (Fromkin and Rodman 1988) and even some abbreviations belong to natural languages. For instance, the abbreviations "MAC" and "IDU" are used in a comment and in some names in Figure 3, because these abbreviations are generally used terms in the application domain of the program in Figure 3.

By using natural names in source programs we aim at increasing program understandability, which in turn facilitates software development and maintenance. Having natural words in source programs brings these documents terminologically closer to other types of software documents such as written English texts and graphical-textual models. Natural names in source programs should thus make the entire software documentation simpler.

1. If this paragraph were written in the same way as source programs usually are, the text could look like the following:

The use of diff knds of abbr is one factor which mks the txt in Fig 2 diff from conv txts. Fig 2 is, althg the progr is abt ten yrs old, a typcal exmpl of a comp progr. Txtbks and rsrch papr as well as softw syst cntn smlr progrs whch are writn with many abbr. There seems to be no stct rules abt how to writ src progrs. We say that the progr in Fig 2 is corr bse it wrk and it can be succful comp. It is no uslly consrd an err in writ if a progr cont stngly writ wrd such as "NXTBLO". In contr, the cornss of txt writ in Engl can be jdgd in trms of the spellg rules and gramm of Engl. Having these rules in mnd, we are able to qckly percve the tniest spllng mstks when readng, for exmpl, a npaper.

```
$EJECT
/**********************************************************************/
/*                                                                    */
/*  NXTBLO : READ NEXT BLOCK                                          */
/*                                                                    */
/**********************************************************************/
NXTBLO : PROCEDURE(FILE);
DECLARE FILE BYTE;

CALL BUBWRI(SEQTBL(FILE).LNRBLO,.BWORK0);

IF SEQTBL(FILE).LNRBLO = SEQTBL(FILE).LSTBLO
   THEN SEQTBL(FILE).LNRBLO = SEQTBL(FILE).FSTBLO;
   ELSE SEQTBL(FILE).LNRBLO = SEQTBL(FILE).LNRBLO + 1;

CALL BUBRED(SEQTBL(FILE).LNRBLO,.BWORK0);

END NXTBLO;
$EJECT
/**********************************************************************/
/*                                                                    */
/*  WRIMOV : MOVE DATA FROM WORK BUFFER TO THE BUBBLE BUFFER          */
/*                                                                    */
/**********************************************************************/
WRIMOV : PROCEDURE(FILE,WRILEN,BUFPOI);
DECLARE FILE BYTE,WRILEN ADDRESS,BUFPOI ADDRESS,
        BUFFER BASED BUFPOI(1)BYTE,WLEN1 ADDRESS,WLEN2 ADDRESS;

IF (WRILEN + SEQTBL(FILE).LNRBYT) >= BLOLEN
   THEN DO;
      WLEN1 = BLOLEN - SEQTBL(FILE).LNRBYT;
      WLEN2 = WRILEN - WLEN1;
      IF WLEN1 > 0
          THEN CALL ZMOVE(.BWORK0(SEQTBL(FILE).LNRBYT),.BUFFER,WLEN1);
      CALL NXTBLO(FILE);
      IF WLEN2 > 0
          THEN CALL ZMOVE(.BWORK0,.BUFFER(WLEN1),WLEN2);
   END;

   ELSE CALL ZMOVE(.BWORK0(SEQTBL(FILE).LNRBYT),.BUFFER,WRILEN);
END WRIMOV;
```

Figure 2. Examples of procedures of a real software system.[1]

1. This program is printed here with the permission of its writer. We use this program as an example in which abbreviated names are used. We are not criticizing the quality of the program or the writer of the program. This program has been proved to work reliably in real environments.

```
/*------------------------------------------------------------------*/

GENERAL_MAC_GENERATOR: PROCEDURE( MESSAGE_POINTER,
                                  MESSAGE_LENGTH_WORD )  BYTE PUBLIC;

/*------------------------------------------------------------------*/

/*  This procedure shall be called before the message is sent to bank
    with the send_message_to_bank - program.

    First the procedure constructs a message and sends it to the IDU.
    The IDU responds with a message which contains the MAC.
    The MAC is joined to the message to be sent to the bank host.
*/
    DECLARE   MESSAGE_POINTER   POINTER,  MESSAGE_LENGTH_WORD   WORD;
    DECLARE   RETURN_CODE       BYTE;
    DECLARE   MESSAGE_INDEX     BYTE,     BINARY_MESSAGE_TYPE   WORD;
    DECLARE   READY             BYTE;

    DECLARE   MESSAGE_DATA  BASED  MESSAGE_POINTER   ( 1 )   BYTE;

    DECLARE   NUMBER_OF_ITEMS_TO_COMPRESS     BYTE;
    DECLARE   MAC_CALCULATION_STATUS          BYTE;

    DECLARE   IDU_MESSAGE_TRAILER_POINTER     POINTER ;

    DECLARE   IDU_MESSAGE_TRAILER BASED  IDU_MESSAGE_TRAILER_POINTER

        STRUCTURE(

        MESSAGE_SEPARATOR           BYTE,

        NUMBER_OF_ITEMS_TO_COMPRESS          BYTE,
        START_POSITION_OF_FIRST_ITEM   ( 3 ) BYTE,
        LENGTH_OF_FIRST_ITEM           ( 3 ) BYTE,
        START_POSITION_OF_SECOND_ITEM  ( 3 ) BYTE,
        LENGTH_OF_SECOND_ITEM          ( 3 ) BYTE,

        MESSAGE_TERMINATOR          BYTE  ) ;
```

Figure 3. Variables declared using natural names.

The main reason why natural naming has not traditionally been used is that the early compilers of programming languages had restrictions on name lengths. Because programmers could use only names that did not exceed a certain number of characters (e.g. six characters in Fortran), they had to use abbreviations. During the early days of computing, there was a need to abbreviate all data that were stored in the memories of computers (Bourne and Ford 1961), because memories were expensive and their capacity was small. By putting limits on name lengths, the designers of the early programming languages could save the memory needed in compilation. These reasons for using abbreviations are no longer valid because memory technology is not expensive today and longer names are now permitted.

Another reason for using abbreviations in source programs is probably that programming has its roots in mathematics. Fortran -- the first high-level programming language -- was designed mainly for mathematical calculations (MacLennan 1983). The first ideas of computing were presented by applying them to a mathematical problem (Turing 1937a). The first computers were used mainly to solve mathematical problems (Hodges 1983). Because mathematicians have traditionally used short symbols, they took this tradition to computer programming as well.

We see no reason not to try to break the tradition of using abbreviations. This tradition may be so strong partly because it is often hard to invent descriptive names for a source program, and because it takes more time to type longer names. It is possible that there are programs (e.g. in mathematical software) in which abbreviations are useful. During the time when people started the tradition of using abbreviations in source programs, software systems were much smaller than today. Considering that the source programs of large software systems may contain more than one thousand different names, it cannot be a bad idea to try to distinguish these names as clearly as possible.

1.4 OUTLINE OF THE THESIS

In the second chapter of this thesis we discuss the structure of source programs more thoroughly. The term programming style is used to denote various factors which affect the understandability of source programs. We show that naming is one of the most important programming style factors. We also justify the use of natural naming.

In the third chapter we present the research problem of this thesis. Basing on the presented problem, appropriate research activities and methods will be defined. The methodological approach will be justified.

Chapter Four discusses related work. We focus on other approaches to increase understandability and maintainability of source programs. Because naming is using a natural language, we will discuss other scientific fields such as linguistics and philosophy.

The fifth chapter is an introduction to the previously-published papers which are included in this thesis. Each paper is a description of one research activity defined in the third chapter.

In the concluding sixth chapter, we evaluate the results of this work, and discuss possibilities for further work.

2 PROGRAMMING AND NAMING STYLES

2.1 THE CONCEPT OF PROGRAMMING STYLE

When we consider source programs in a purely technical sense, i.e. from the viewpoint of compilation, they can be treated as lists of coded characters which are processed by a compiler (Aho et al. 1986). Different characters and character combinations have a clearly defined meaning to the compiler. For example, in the case of the C programming language, the equal sign "=" is an assignment operator, and two adjacent equal signs "==" mean a logical operator to compare equality. Because compilers set only technical rules for writing source programs, programs can be written in many ways which are all equally acceptable in the technical sense. The issue here, however, is that different ways of writing source programs are not equally readable and understandable to humans.

The factors that affect the readability and understandability of source programs are denoted by the concept of programming style. Oman and Cook (1991) classify programming style factors into the following major categories:

- General programming practices, defined as rules and guidelines pertaining to the programming process that directly affect the style of the software product.
- Typographic style, defined as style characteristics affecting only the typographic layout and commenting of source code.
- Control structure style, defined as style characteristics pertaining to the choice and use of control flow constructs, the manner in which the program or system is decomposed into algorithms, and the method in which the algorithms are implemented. Control structure style excludes data structure aspects.
- Information structure style, defined as style characteristics pertaining to the choice and use of data structure and data flow techniques, i.e., to the manner in which information is manipulated throughout the program or system.

As shown in Figure 4, the major style categories can be divided further into macro and micro style concerns. Macro concerns are those pertaining to a set of source program modules of a system, while micro concerns are those relative to a single program module or statement. Programming style can be considered analogous to "writing style" which means that programming style researchers are interested in the written outputs of the programming process, the source programs, and not in the programming process itself.

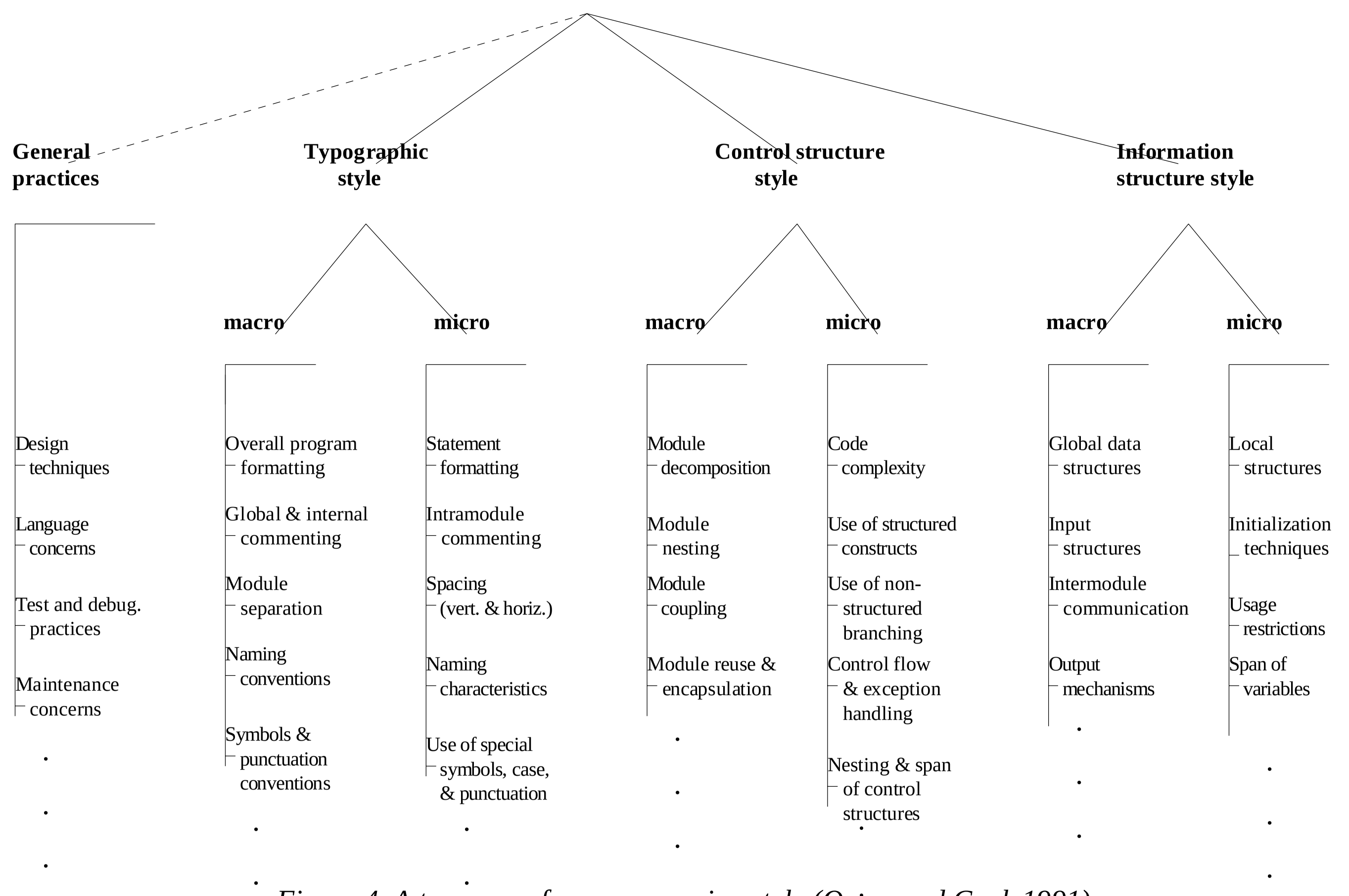


Figure 4. A taxonomy for programming style (Oman and Cook 1991).

Although different programming styles affect the understandability of source programs, understandability is hard to measure in exact quantities. If a person tries to read and understand a source program, his or her background determines how well he or she is able to succeed in this task. Understandability is also a very subjective matter: what is considered easily understandable by one person may be poorly understandable for another. Typically, educational as well as professional experience affect how well a person can understand a particular source program. We can say that specialized people with appropriate education or experience can understand computer programs, but we find it difficult to judge how well a particular specialized person can understand a particular source program. Due to the obvious abstractness of the concept "understandability" we will not attempt to define or measure it explicitly. Rather, we will study the appearance of example source programs and discuss their visual understandability.

To demonstrate that a source program may have forms which are different in terms of understandability and the same in terms of functionality, three program examples are illustrated in Figure 5. A compiler produces identically functioning outputs from any of these program versions. However, the program versions visually appear quite different and they may evoke different kinds of intuitions about their functionality when an observer studies them. To a literate observer, the program version (a) in Figure 5 contains familiar characters and numbers, but the characters form no familiar words, whereas the program version (b) contain strings of characters that at least resemble English words. The program version (c) incorporates many English words which, if the observer is familiar with computer programming, should enable him or her to learn quite much about the program's functionality and purpose. Figure 6 shows more functionally equivalent versions of the same program as is used in Figure 5. In Figure 6(a) the program contains exactly the same English words as the program in Figure 5(c), but the visual appearance is different because of different organization of English words and special characters. The program in Figure 6(b) is a commented version of the program in Figure 5(b) and Figure 6(c) presents the program from Figure 5(b) in a visually different form. From these examples we see that there is a great variety of possibilities how we can organize a program.

We thus conclude that if there are two versions of a functionally equivalent source program and these versions have different visual appearance, it is very likely that the versions evoke different processes in an observer's mind and the observer understands the two versions in somewhat different manner. The understandability of the programs can then be different, and one version of a program can be more understandable than another. Experiments with human subjects support this (Shneiderman 1980, Curtis 1985).

```
#define   C0001       13
#define   C0002       0
#define   C0003       1

/*-----------------------------------------*/

f0001 (  char  s0001 [],

         int  *i0001  )

/*-----------------------------------------*/
{
    int   i0002,   i0003 ;

    *i0001  =  C0002  ;

    i0003  =  strlen ( s0001 ) ;

    if ( i0003  >  C0001 )
    {
        *i0001  =  C0003 ;
    }
    else
    {
       for (i0002 = 0; i0002<i0003; i0002++)
       {
          if(( s0001[ i0002] < '0') ||
             ( s0001[ i0002] > '9') )
          {
             *i0001  =  C0003 ;
          }
       }
    }
}
```

Figure 5 (a)

```
#define   CNUMMAX   13
#define   VALID     0
#define   NVALID    1

/*-----------------------------------------*/

isvalid (  char  cnumbr [],

           int  *rcode  )

/*-----------------------------------------*/
{
    int   i,   len ;

    *rcode  =  VALID  ;

    len  =  strlen ( cnumbr ) ;

    if ( len  >  CNUMMAX )
    {
        *rcode  =  NVALID ;
    }
    else
    {
       for  ( i=0 ; i<len ; i++ )
       {
          if (( cnumbr[ i] < '0') ||
              ( cnumbr[ i] > '9') )
          {
             *rcode  =  NVALID ;
          }
       }
    }
}
```

Figure 5 (b)

```
#define   MAXIMUM_CUSTOMER_NUMBER_LENGTH       13
#define   CUSTOMER_NUMBER_IS_VALID              0
#define   CUSTOMER_NUMBER_IS_NOT_VALID          1

/*--------------------------------------------------------------------------*/

check_customer_number_validity (  char  possibly_valid_customer_number [],

                                  int  *success_code  )

/*--------------------------------------------------------------------------*/
{
    int   customer_number_index,   customer_number_length ;

    *success_code  =  CUSTOMER_NUMBER_IS_VALID  ;

    customer_number_length  =  strlen ( possibly_valid_customer_number ) ;

    if ( customer_number_length  >  MAXIMUM_CUSTOMER_NUMBER_LENGTH )
    {
        *success_code  =  CUSTOMER_NUMBER_IS_NOT_VALID ;
    }
    else
    {
       for  (  customer_number_index  =  0  ;
               customer_number_index  <  customer_number_length ;
               customer_number_index  ++    )
       {
          if(( possibly_valid_customer_number[ customer_number_index] < '0') ||
             ( possibly_valid_customer_number[ customer_number_index] > '9') )
          {
             *success_code  =  CUSTOMER_NUMBER_IS_NOT_VALID ;
          }
       }
    }
}
```

Figure 5 (c)

Figure 5. Functionally equivalent versions of the same program.

```
#define MAXIMUM_CUSTOMER_NUMBER_LENGTH 13
#define CUSTOMER_NUMBER_IS_VALID 0
#define CUSTOMER_NUMBER_IS_NOT_VALID 1
check_customer_number_validity_(
char possibly_valid_customer_number[],int*success_code)
{int customer_number_index,customer_number_length;
*success_code=CUSTOMER_NUMBER_IS_VALID;customer_number_length
=strlen(possibly_valid_customer_number);if(customer_number_length
>MAXIMUM_CUSTOMER_NUMBER_LENGTH){*success_code=CUSTOMER_NUMBER_IS_NOT_VALID;
}else{for(customer_number_index=0;customer_number_index
<customer_number_length;customer_number_index++){if((
possibly_valid_customer_number[customer_number_index]<'0'
)||(possibly_valid_customer_number[customer_number_index]
>'9')){*success_code=CUSTOMER_NUMBER_IS_NOT_VALID;}}}}
```

Figure 6 (a)

```
/*  A program to check the validity of the customer number. */

#define   CNUMMAX   13   /* Maximum length of customer numer  */
#define   VALID     0    /* Code for valid customer number */
#define   NVALID    1    /* Code for invalid customer number */

/*-------------------------------------------------------------*/

isvalid_ ( char  cnumbr [],    /*  Customer number string  */

           int  *rcode  )      /*  Return code for caller  */

/*------------------------------------------------------------*/
{
    int   i ;      /*  Index to access the customer number */
    int   len ;    /*  customer number length  */

    *rcode  =  VALID  ;   /*  Assume that customer number is valid. */

    len  =  strlen ( cnumbr ) ;   /*  Get the lenght of string. */

    if ( len  >  CNUMMAX )
    {
        *rcode  =  NVALID ;       /*  String too long. Not valid. */
    }
    else
    {
       for  ( i=0 ; i<len ; i++ )
       {
          if (( cnumbr[ i] < '0') ||   /* Are there just numerical */
              ( cnumbr[ i] > '9') )    /* digits in the string ?   */
          {
             *rcode  =  NVALID ;       /* No. */
          }
       }
    }
}
```

Figure 6 (b)

```
#define CNUMMAX 13
#define VALID 0
#define NVALID 1
isvalid__(char cnumbr [],int *rcode){int i,len;*rcode=VALID;len
=strlen(cnumbr);if(len>CNUMMAX){*rcode=NVALID;}else{for(i=0;
i<len;i++){if((cnumbr[i]<'0')||(cnumbr[i]>'9')){*rcode=NVALID;}}}}
```

Figure 6 (c)

Figure 6. More equivalent versions of the program in Figure 5.

```
#define             13
#define             0
#define             1

--------------------------------------------

        (  char         [],

           int   *       )

--------------------------------------------
{
    int     ,       ;

    *       =         ;

         =  strlen (        ) ;

    if  (      >          )
    {
        *       =         ;
    }
    else
    {
       for  (  =0 ;  <    ;  ++ )
       {
          if ((        [  ] < '0') ||
               (       [  ] > '9') )
          {
             *       =         ;
          }
       }
    }
}
```

```
          CNUMMAX
          VALID
          NVALID




isvalid          cnumbr

                 rcode




          i    len

     rcode     VALID

    len                  cnumbr

         len     CNUMMAX

         rcode     NVALID



              i     i len   i

                cnumbr  i
                cnumbr  i

              rcode     NVALID
```

Figure 7. Names separated from a program.

Considering that there are so many programming style factors that may affect the understandability of programs, we are confronted with the question: what is the programming style factor that bears the most significant correspondence with understandability? Because understandability is such a subjective and largely unmeasurable concept, and because different programming style factors may not be completely separable, we will not attempt to give a justified answer to that question. Rather, it is assumed that naming is one of the most important programming style factors, and, therefore, we focus our attention on it. The examples of differently written programs in Figures 5 and 6 illustrate the importance of naming style.

As shown in Figure 5, different naming styles greatly affect the physical appearance of the program. There is also some empirical evidence that naming may contribute most to the understandability of programs. Gellenbeck and Cook (1991) have found that the meaningfulness of names in programs affect the understandability more than such typographic signals as different fonts for different kinds of program elements. The importance of naming can be perceived rather well in an intuitive observation. In Figure 7 the program version (b) from Figure 5 is shown in two different forms. In the left-hand part the names have been removed from the program and the right-hand part presents only the names of the program in their relative positions. We can easily perceive that there is not much "meaning" left in a typical program when its names are taken away. On the other hand, we see that neither do the names alone make much sense, although they display some details of the functionality of the program.

2.2 DIFFERENT NAMING STYLES

Names in source programs are strings of letters and numbers which are uniquely identified by the compiler. Some compilers make a distinction between uppercase and lowercase letters while others do not. Usually, a name must start with a letter, but the subsequent characters may be also numbers. Many compilers allow also underscore characters "_" in names. Underscores can be used as word separators when a name consists of more than one word.

Naming in programming means giving names to different source program elements such as variables, constants, tables, functions, and procedures. Different programming languages may have different kinds of elements which need names. As shown in Figure 4, naming is part of the typographic style. Naming does not deal with the functionality of programs, since the typographic style only dictates how a source program appears to an observer. Figure 4 makes a distinction between the macro typographic concern "naming conventions" and the micro typographic concern "naming characteristics". Naming characteristics deal with issues such as should we use an underscore "_" or a capital letter to separate words of a single name. Because naming characteristics play a minor role in making the names understandable, we are mainly interested in naming conventions.

As compilers set only technical constraints for naming, a programmer is free to use many kinds of names. Figure 5 above shows some possibilities:

- The names in Figure 5(a) are made of a single letter which identifies the type of the name. The names are made unique by adding sequentially different numbers after the first letter.

- Most of the names in Figure 5(b) are abbreviations of one or more English words. The name "VALID" is a single English word. The name "i", which is used as an index variable, is a symbol commonly used in mathematics.

- Figure 5(c) contains natural names which consist of several natural English words.

Intuitively, natural names such as the ones in Figure 5(c) appear to be the most understandable. Therefore we will study this naming approach. Natural names can also be formed in different ways. Figure 8 illustrates two different ways for constructing natural names:

- The names in Figure 8(a) are about the same as the names in Figure 5(c), but they are written in Dutch. The reader should be competent in this language in order to properly understand the program in Figure 8(a).

- Figure 8(b) has natural names written in English, but these names are arbitrarily chosen expressions which do not relate rationally either to each other or to the functionality of the program.

Basing on the examples in Figure 8, we can clarify the concept of natural naming with two amendments:

- When natural names are used, a decision must be made about which natural language to use as the naming language. English is usually a good basis for naming, because programs usually contain other items, such as reserved words, which are English words.

- Natural names used in source programs should describe the functionality of the program.

Names in source programs can denote various matters depending on the programming language in question. In procedural programming, names are needed to refer to certain locations in the memory of a computer (e.g. the name of a variable refers to the memory location reserved for that variable and the name of a procedure refers to the memory location in which the machine code of the procedure starts). The terms "symbol", "identifier", and "label" are sometimes used instead of the term "name" in the literature. For instance, the table into which a compiler collects all the names found in a source program is called a "symbol table" (Aho et al. 1986).

When the meaningfulness and understandability of names are discussed in the literature, the term "mnemonic name" is often used (e.g. Sheppard et al. 1979, Shneiderman 1980). Mnemonicity should help memory (Webster's 1989), but the mnemonicity of names is hard to define explicitly. Obviously names like those in Figure 5(a) cannot be considered mnemonic. When Sheppard et al. (1979) wanted to use non-mnemonic names in psychological experiments, they made them of two randomly chosen characters. On the other hand, abbreviations consisting of a few letters can be mnemonic. For instance, three-letter instructions of some assembler languages (e.g. "MOV", "LDA", and "STA") are called mnemonics (Intel 1979). To make a distinction between mnemonic and natural names, we can conclude that mnemonicity of names is a different concept than naturalness of names, since natural names have to be formed using natural words.

```
#define   GROOTSTE_CLIENT_NUMMER_LENGTE       13
#define   CLIENT_NUMMER_IS_GOED               0
#define   CLIENT_NUMMER_IS_NIET_GOED          1

/*------------------------------------------------------------------------*/

is_client_nummer_goed (  char  mogelijk_goed_client_nummer [],

                         int   *success_code  )

/*------------------------------------------------------------------------*/
{
    int   client_nummer_index,   client_nummer_lengte ;

    *success_code  =  CLIENT_NUMMER_IS_GOED  ;

    client_nummer_lengte  =  strlen ( mogelijk_goed_client_nummer ) ;

    if ( client_nummer_lengte  >  GROOTSTE_CLIENT_NUMMER_LENGTE )
    {
        *success_code  =  CLIENT_NUMMER_IS_NIET_GOED ;
    }
    else
    {
       for  (  client_nummer_index  =  0  ;
               client_nummer_index  <  client_nummer_lengte ;
               client_nummer_index  ++    )
       {
          if(( mogelijk_goed_client_nummer[ client_nummer_index] < '0') ||
             ( mogelijk_goed_client_nummer[ client_nummer_index] > '9') )
          {
             *success_code  =  CLIENT_NUMMER_IS_NIET_GOED ;
          }
       }
    }
}
```

Figure 8(a). Natural names written in Dutch.

```
#define   ALL_WORK_NO_PLAY              13
#define   BREAD_IS_HEALTHY              0
#define   SMALL_BABIES_SLEEP_A_LOT      1

/*------------------------------------------------------------------------*/

are_sundays_really_boring (  char  breakfast_is_an_important_meal [],

                             int   *war_and_peace  )

/*------------------------------------------------------------------------*/
{
    int   michael_jordan_and_carl_lewis,   this_silence_is_ ;

    *war_and_peace  =  BREAD_IS_HEALTHY  ;

    this_silence_is_  =  strlen ( breakfast_is_an_important_meal ) ;

    if ( this_silence_is_  >  ALL_WORK_NO_PLAY )
    {
        *war_and_peace  =  SMALL_BABIES_SLEEP_A_LOT ;
    }
    else
    {
       for  (  michael_jordan_and_carl_lewis  =  0  ;
               michael_jordan_and_carl_lewis  <  this_silence_is_ ;
               michael_jordan_and_carl_lewis  ++    )
       {
          if((breakfast_is_an_important_meal[michael_jordan_and_carl_lewis]<'0')||
             (breakfast_is_an_important_meal[michael_jordan_and_carl_lewis]>'9'))
          {
             *war_and_peace  =  SMALL_BABIES_SLEEP_A_LOT ;
          }
       }
    }
}
```

Figure 8(b). Natural names not referring to the functionality of the program.

2.3 JUSTIFYING THE USE OF NATURAL NAMING

In the program examples above we have shown that naming is an important programming style factor. Naturally named programs appear to be more understandable than programs containing abbreviated or completely non-mnemonic names. The concept of natural naming can thus be defined as follows:

> *Natural naming means that all kinds of names needed in source programs should be formed by using, preferably several, natural words of a natural language so that the grammatical rules of the used natural language are respected, and the names describe the functionality of the program.*

This definition is somewhat vague because the term "natural word" is vague. We argue, however, that it is hard to give much more accurate definitions for natural naming. It is a known linguistic fact that natural languages have changed during the history and they are changing all the time (Fromkin and Rodman 1988). Therefore, it is possible that new words and even new grammatical rules can be invented, and new meanings can be assigned to existing natural words. Natural naming is against the use of abbreviations, but it is a fact that many abbreviations are already accepted as symbols in our natural languages. Because it is not always clear what is a known natural word, the definition of natural naming given above has its limitations.

It is also hard to say when a name is meaningful enough. The definition above suggests that more than one word should be used to make a name meaningful, but it is difficult to set an upper limit for the number of words. For instance, all the following names could represent the same variable:

```
(1)  n
(2)  nbytes
(3)  bytes
(4)  byte_count
(5)  number_of_bytes
(6)  number_of_bytes_in_buffer
(7)  number_of_bytes_in_reception_buffer
(8)  current_number_of_bytes_in_buffer
```

The names (1) and (2) are not acceptable as natural names, because they do not consist of natural words. The name (3) is not very descriptive because it is a single word. The rest of the names from (4) to (8) can all be considered natural although they are much different from each other. It depends on the context in which a name is used and also on the observer's personal taste which natural name can be considered to be suitably long and meaningful. Newsted (1979) has pointed out that personal taste affects which names are considered appropriate.

Although the concept of a natural name is somewhat vague, we can justify this research by the fact that in software documentation it is very common to use non-natural and abbreviated names which are hard to understand. The following points can be seen to support the use of natural naming:

- Terminology control in software documentation is important (Boldyreff et al. 1990). Higher-level software documents (e.g. requirements descriptions) are usually written using natural words and a natural language. If source programs also contain natural names, they are terminologically closer to higher-level software documents, and thereby more understandable. As higher-level software documents are usually read by other people than software developers (customers, end users, sales personnel, etc.), we can assume that software developers can more easily communicate with non-experts when the same terminology is used in every software document.

- Natural names are commonly used in the graphical-textual descriptions which need to be written when certain software development methods are used. For instance, the data-flow diagrams shown by Ward and Mellor (1985) and by Yourdon (1989), and the various object diagrams shown by Coad and Yourdon (1990) contain names which consist of natural words. It is possible that the use of natural names in these diagrams is one reason why they are considered useful in software development. Figure 9(a) illustrates a typical data-flow diagram with natural names. The diagram in Figure 9(b) is less typical and it appears to be less understandable.

- Several writers recommend that computer programs should be described with a language close to a natural language before they are written with a programming language. Caine and Gordon (1975) recommend the use of structured English which is a language using the English vocabulary together with a syntax of a structured programming language. Some software development methods (e.g. Page-Jones 1988, Yourdon 1989) recommend the use of so-called pseudocoding, which means describing programs with a language that is somewhere in between a natural language and a programming language. When natural naming is used in programming, source programs themselves become descriptions which are close to a natural language.

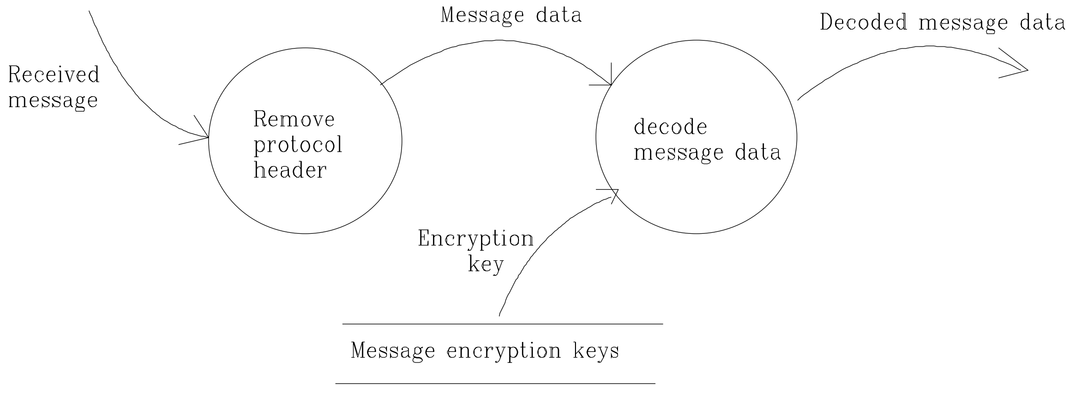


Figure 9(a). An example of a data-flow diagram

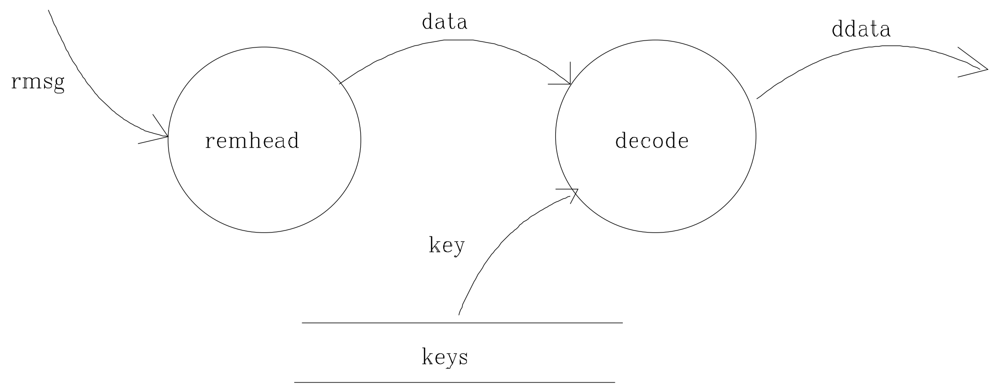


Figure 9(b). The same diagram with less informative names.

Figure 9. Data-flow diagrams with different naming styles.

- Some researchers have stressed the importance of having specifications of software systems written in a natural language, and they provide methods to transform the natural language specifications into more elaborate descriptions (Abbott 1983, Saeki et al. 1989, Yonezaki 1989). Using documents written in pure natural language is thus seen to be advantageous in software development. If we assume that specifications are written with natural language, the use of natural names in source programs should narrow the gap between specification and implementation.

- Tests with the users of an interactive text editor suggest that it is easier to control an interactive system with longer natural commands than with short abbreviated commands (Ledgard et al. 1980). In many software tools, natural words in user interfaces are becoming popular. For instance, in graphical operating systems for personal computers and workstations, the operating systems are being controlled by using the mouse to select natural language expressions from various menus. In older operating systems, we need to remember and type short abbreviated commands on the command line. The use of abbreviations seems thus to be decreasing in the context of modern graphical operating systems. This gives us a reason to consider reducing the use of abbreviations in other contexts as well.

- The use of abbreviations has been criticized in other contexts than software documentation. Some guides for technical writing warn against the overuse of abbreviations (Kauranen et al. 1993). Ibrahim (1989) criticizes acronyms and Logsdon and Logsdon (1986) show that acronyms neither shorten texts nor make them more readable.

- It is possible to formulate a philosophic-linguistic theory to support the use of natural naming (Laitinen and Taramaa 1994, Laitinen 1995), although this theory has not (yet) acquired wider acceptance.

Several authors (Weissman 1974, Curtis et al. 1979, Sheppard et al. 1979, Shneiderman 1980, Teasley 1994) describe experiments in which the effect of different naming styles, among other programming style factors, has been tested with human subjects. Students have been the usual subjects in these experiments in which different groups of people have been asked to study source programs written with different programming styles. The understandability of different versions of source programs has been evaluated by letting the subjects modify or memorize the programs or by asking them questions about the programs.

Although the mentioned experiments have revealed that some programming style factors affect significantly the understandability of source programs, the effect of naming is still somewhat obscure. Usually, the subjects who have been exposed to programs containing mnemonic or natural names have had a better performance than the subjects who have studied the same programs with less clear names. The problem is, however, that the results of the experiments have not always been statistically significant. It thus seems that it is hard to measure how different naming styles affect understandability.

Despite the fact that the mentioned experiments have not been able to convincingly prove the usefulness of having mnemonic or natural names in source programs, our working hypothesis remains that the natural naming approach is useful in practical work. The mentioned experiments were carried out by using small examples of source programs which were quickly studied by students. Practical software development work differs substantially from these kinds of experimental arrangements. Industrial software developers and maintainers study programs which may contain hundreds of names, they may have to spend days to understand a certain problem in a software system, and they may have to study and modify the same programs several times during a longer time period. It is likely that natural names are helpful in real software development situations. Teasley (1994) also points out that it is possible that such a programming style factor as naming could have a greater impact in practical software development work than it has in classroom experiments.

The reason why controlled understandability tests have been carried out only in classrooms is obviously that it is not easy to do controlled experimentation with staff employed in practical software development projects. Theoretically, we could let two groups of software developers build the same software system using different documentation practices (e.g. using abbreviations vs. natural names), and then we could compare the performance of the groups by observing which group get the job done in shortest time. However, if we had only two groups to compare, it would be impossible to get statistically significant results. For statistical reasons, we should have many more groups performing the same job. It is hard to imagine that somebody could provide funds for this kind of experimentation where many groups of expert people would be doing the same real-size software projects.[1]

Because controlled experimentation with practical software development projects may be infeasible, we do not aim to unambiguously prove that the use of natural naming really makes software development and maintenance more effective. As explained above, there are sufficient grounds to believe that natural naming has the potential of easing the cognitive tasks of software developers.

1. We have actually done this kind of experimentation in small scale with non-experts. During the fall 1990, we organized a test with two groups of students. Both four-member groups had to build the same software system. These software development projects were part of a course given at the University of Oulu in Finland. We gave a course on natural naming to one student group and told nothing about the subject to the members of the other group. It turned out that the student group which was taught and used natural naming in their work finished their software project earlier than the other group. However, because we had only two samples of performance, we cannot be sure whether the use of natural naming was the only reason why the first group was more successful. It is possible that the members of the better group were simply more clever and co-operative than the people in the other group.

3 RESEARCH PROBLEM

3.1 PROBLEM DEFINITION AND RESEARCH ACTIVITIES

We suppose in this thesis that by the use of natural naming, source programs and other software documents can be made more understandable, which facilitates software development and maintenance. The poor understandability of source programs is thus the underlying problem considered in this work. Poor understandability is a non-desirable aspect. Therefore, it can be considered as a problem. Thinking this way, natural naming can be regarded as one solution to alleviate the problem of poor understandability. It can also be seen as a means of eliminating the risks of misunderstanding.

The understandability of source programs and other software documents is important when we consider software development as a documentation process. It seems that the importance of having understandable documents is increasing as software quality systems require that software documents need to be read and inspected formally (Ackerman et al. 1989, ISO 9000-3 1991). Understandability of documents is perhaps the most important in software maintenance which comprises a sizeable proportion of the entire software business (Parikh and Zvegintzov 1983, CSTB Workshop Report 1990, AMES 1993, Taramaa and Oivo 1993).

As we explained in the previous chapter, it may be quite hard to empirically prove that using natural naming is advantageous in every application. Still we have sufficient reasons to believe that this naming approach is useful in practice. Moreover, even if we could empirically show that it is usually advantageous to use natural naming, we would not have much practical use of this knowledge as we had little experience of using it in practice. For this reason, we consider that it is more important to gain practical experience in the use of natural naming, rather than trying unambiguously to prove its advantage. We thus define our research problem as follows:

How can we facilitate the use of natural naming in software development and maintenance?

The purpose of this thesis is to find means for making natural naming more attractive and easy-to-use for people doing practical software development and maintenance work. We define the following research activities to produce solutions to the research problem defined above:

(a1) Because the basic idea of natural naming is rather simple (i.e. avoiding abbreviations), produce more detailed principles for using natural naming in source programs.

(a2) Natural names used in source programs should be similar to textual expressions used in other software documents. For this reason, formulate a method which enables the creation of names at an early stage of the development process, in such a way that the names used in programs correspond with the names and textual expressions in other types of software documents (e.g. requirement descriptions).

(a3) Because the habit of using abbreviations is the most common in source programs, and because existing programming languages have not been specifically designed for the use of natural names, investigate how we could redesign programming languages to be more suitable for the use of natural naming.

(a4) Because hard-to-understand source programs pose a difficult problem for software maintainers, investigate how already abbreviated names could be replaced with natural ones in existing source programs which need maintenance.

(a5) To encourage the use of natural naming, seek to produce more practical evidence of the usefulness of natural naming.

Natural naming can be applied by using existing software development practices and tools. The only requirements for using natural naming in programming are that the compiler in use must be able to distinguish relatively long (e.g. at least 30 characters in length) names, and that separate words in a name be distinguishable. Because using natural naming does not require any especially modern tools, we will concentrate in commonly used software development practices in this work. Natural naming will be studied mostly in the context of procedural programming. Although object-oriented programming is gaining popularity, we can assume that procedural programming will still remain important. Because we are also interested in software maintenance, it is relevant to concentrate on procedural programs, which are usually the ones being maintained.

At least theoretically, any written natural language can be exploited as the basis for natural naming. Because there are about 6000 different spoken languages in the world (Fromkin and Rodman 1988), there are hundreds if not thousands of natural languages for which a writing system has been invented. As it would be rather hard to study all written languages as potential software documentation languages, this study will limit its concern to English, which is probably the most common natural language used in software documentation.

3.2 RESEARCH METHODS

This work can be characterized as constructive research which involves studying existing source programs and other software documents and interacting with people who are doing practical software development work. We will justify this research approach more profoundly in the subsequent section. For each research activity from (a1) to (a5) defined in the previous section, we describe a set of steps to be followed. These steps form the methods for the research activities. Although the research described in this thesis has been carried out by following the steps below, we will not explicitly describe the execution of every step.

The method for research activity (a1) consists of the following steps:

- Analyze existing source programs in order to capture the knowledge of what kinds of names are usually needed.
- Develop naming principles for each type of name found through the above analysis in accordance with commonly used naming conventions.
- Apply the naming principles in practical software development tasks and refine the principles as appropriate in light of practical experience.
- Let expert software developers review the naming principles; refine the principles according to their comments.

The method for research activity (a2) consists of the following steps:

- Having done (a1), analyze the naming principles and consider which types of names can exist in source programs as well as in other types of software documents.
- Develop name derivation rules for those names that can appear both in source programs and in other types of software documents.
- Consider systematic ways for applying the name derivation rules in practical application domains and develop a systematic name creation method.
- Let expert software developers review the name creation method and refine it as appropriate.

The method for research activity (a3) consists of the following steps:

- Apply the natural naming principles in programming with a commonly used and widely accepted programming language.

- Analyze the resulting naturally named programs and seek ways for simplifying and improving the programming language so that the naturally named programs would be more readable.
- Specify a new programming language according to the analysis in the previous step.
- Implement a prototype compiler for the new programming language and refine the language to make it suitable for compilation.

The method for research activity (a4) consists of the following steps:

- Analyze typical abbreviation patterns and information in comments of existing source programs.
- According to the analysis, define methods for disabbreviating the names in existing source programs.
- Basing on the disabbreviation methods, develop a computer tool which can process existing source programs, suggest natural replacements for the abbreviated names in the programs, and produce more understandable naturally-named versions of the source programs.
- Evaluate this tool by using source programs from several application domains, and, if possible, modify the tool and disabbreviation methods to increase the efficiency of the tool.

The method for research activity (a5) consists of the following steps:

- Provide educational material and arrange courses on the natural naming principles for different groups of software developers.
- Let the software developers use the naming principles for different periods of time.
- Survey the software developers using a formal questionnaire.
- By analyzing the answers from software developers, assess the usefulness of the naming principles.

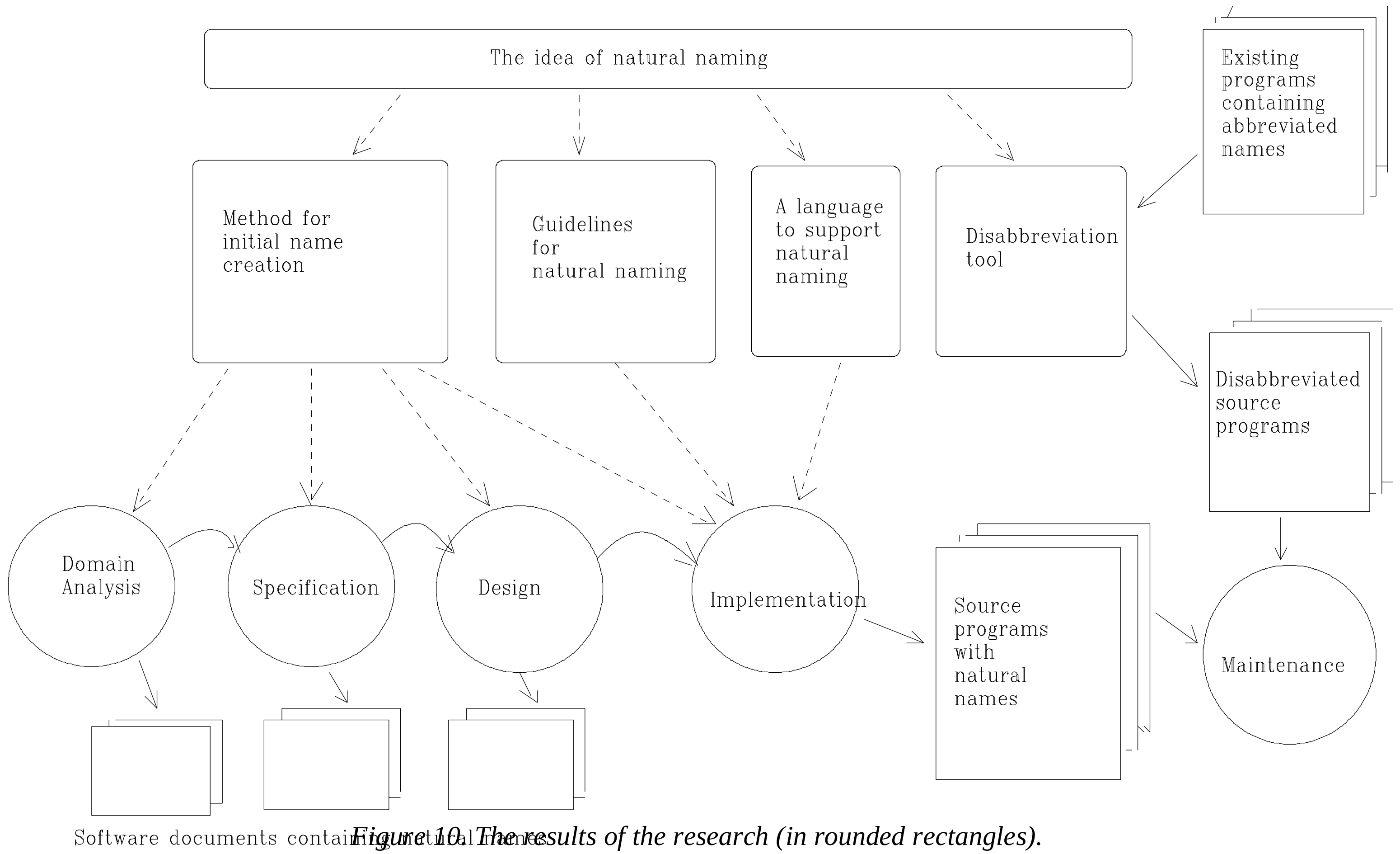

The idea of natural naming
Method for initial name creation
Guidelines for natural naming
A language to support natural naming
Disabbreviation tool
Existing programs containing abbreviated names
Disabbreviated source programs
Domain Analysis
Specification
Design
Implementation
Source programs with natural names
Maintenance
Software documents containing natural names

Figure 10. The results of the research (in rounded rectangles).

As the defined research methods indicate, the result of this research is a set of guidelines, methods, and tools to take natural naming in practical use. Figure 10 illustrates the results of this work. The methods and tools which will be produced are in rounded rectangles and the broken-line arrows symbolize how they affect the software development processes shown as circles.

3.3 JUSTIFYING THE METHODOLOGICAL APPROACH

In this research we are interested in software development work as it is being carried out in industrial organizations. Software development is studied from the viewpoint of individual software developers and maintainers. We are mostly concerned how easily or effectively a software developer or maintainer is able to perform in his or her work. Although the quality of developed software systems is important, we are less directly concerned about that in this thesis. Here, we are more interested in the software process than in software products. However, we believe that when a developer or maintainer can easily understand the software documents being studied, he or she gains a better understanding about the software system or product as well.

As our perspective is that of a software developer or maintainer who has to produce, study, and understand software documents in his or her work, this research can be described as belonging to the software engineering and computer science research traditions. We speak about software systems, by which we mean systems in which software is executed by processors or computers. Software is considered to mean only computer software. This distinction is necessary because some manufacturers of electronic entertainment equipment use the word "software" to mean music on compact discs and motion pictures on video tapes. Software systems include all systems for which executable software has been generated from source programs or from other types of descriptions processed with software generation tools. The term "software system" can thus be applied to traditional computer systems such as accounting or text processing systems, as well as to less traditional systems such as portable telephones and factory automation systems.

Iivari (1991) lists software engineering research as one research paradigm for information systems development. This work could thus be considered as research related to information systems. However, this is not the main emphasis of this work. Information systems research is usually concerned about much higher-level issues, such as efficiency of an organization which is using an information system. An information system can be seen as a much wider concept than our concept of a software system. Development models for information systems cover many human and organizational aspects that traditional software engineering methods do not deal with (Kerola and Freeman 1981). Information systems are human systems. They do not necessarily even include software systems. Because studying natural naming in the context of information systems would widen the scope of this research tremendously, we will speak about software systems and follow the software engineering research tradition.

Then, why do we want to do constructive research while studying the problems related to natural naming? Why are we constructing a new reality by introducing guidelines and methods for natural naming, and providing experimental tools to facilitate the use of natural naming in software development and maintenance? First, we would like to note that the constructive approach is the traditional research method used in the software engineering research tradition (Iivari 1991). Moreover, we point out the following rather practical reasons:

- Because using abbreviations is a tradition in programming and software development, we could not explain the ideas and potential benefits of natural naming without constructing material in which natural naming is used. To provide appropriate material about natural naming, we need to apply natural naming in a systematic manner. This requires constructing some guidelines and methods for natural naming.

- It is unlikely that people could improve their software documentation practices without being taught through practical examples and methods. In the case of naming and other programming style factors, some people seem to be somewhat blind to what they are doing. For example, before starting to use natural naming on September 21, 1988, the author of this thesis, who now claims to be an expert in avoiding abbreviations, had been using abbreviations in programming for about eight years without clearly realizing that he was using abbreviations.

- In the beginning of this research, we had only the idea of natural naming, and we had been using natural names in practical software development work for about half a year. We had seen that natural names simplify software documentation, but we were unsure about all possible consequences of the idea. Doing constructive research was a way to explore this new idea.

- In the previous chapter we explained that other researchers have failed to prove whether different naming styles significantly affect the understandability of source programs. For this reason, it is the author's belief that it is important to first lay firm grounds for the use of natural naming. After constructing an "infrastructure" for natural naming, it will be possible to gain more knowledge of the practical application of natural naming to assess its real effect on understandability.

Constructive research is the main characteristic of this work, but we are not claiming that this is purely constructive. We can find traits of several other research approaches in this work (Järvinen and Järvinen 1993):

- Explorative research means studying a phenomenon about which little or no previous knowledge exists. This work can be seen to be explorative, because in the beginning of this research natural naming was a new idea which had not been systematically used in software development.

- The research method for activity (a5) defined in the previous section can be seen as a field study, as we are asking the opinions of software developers after they have been exposed to the ideas of natural naming. With the field study we try to find more evidence to support the use of natural naming.

- In action research a researcher participates in the process which is the research subject. In this work, software development and maintenance are the processes being studied. Because the author of this thesis has about five years of previous experience on software development in commercial organizations, there are certainly some aspects of action research involved here.

Kuhn (1962) introduced the concept "scientific paradigm" which can be characterized as a certain way of thinking among the members of a scientific community. Kuhn sees science, and especially such natural sciences as physics and chemistry as processes in which revolutionary discoveries are made from time to time. The scientific revolutions change the thinking of entire scientific communities: people start communicating with new concepts, measuring new things in the world, studying new problems, etc. Scientific revolutions are changes in scientific paradigms.

There are also other interpretations for the word "paradigm" (Iivari 1991) which is one reason why we would not like to place this work under a certain paradigm. Another reason is that it is possible that the Kuhnian paradigms are not so relevant in research related to software systems. Although attitudes in the software engineering community have changed and new research trends emerged during the short history of computers, we cannot say that this research community has reached stable agreements about the nature of software development. The recent papers by Jackson (1994) and Glass (1994) indicate that no strong paradigms in the Kuhnian sense exist in our community. Jackson (1994) has the opinion that the nature of software development work is not yet profoundly understood. Moreover, Glass (1994) has said that our research community is in crisis, largely because researchers are not able to measure the real efficiency of software development methods and tools (Potts 1993, Fenton 1993, Fenton et al. 1994).

We consider that this research can be done without bothering about paradigms in strict sense. Much of this work (e.g. developing natural naming principles for programming) could have been done much earlier, even when the first high-level programming languages were introduced. This is a third reason not to tie this with time-dependent paradigms. Feyerabend (1975) has recommended prejudiceless attitudes towards scientific work. We take this attitude by not tying ourselves to paradigms.

4 RELATED WORK

4.1 INTRODUCTION TO RELATED WORK

When we are interested in finding ways for making source programs and other software documents more understandable to people (e.g. through natural naming), we can find many connections between our research and other research carried out in the field of software engineering. Although the word "understandability" is not always explicitly mentioned, most software engineering research aims at helping software developers to understand user needs, market requirements, technical limitations, technical possibilities, etc. more easily, and thereby to develop more satisfactory software systems in a cost-effective manner. We could thus cover quite a large number of related works, but in the following we will only discuss work which is most closely and interestingly related to our research.

As the use of natural naming has the greatest impact on source programs, we will discuss other approaches to make source programs more understandable. These approaches include various programming guidelines; tools which can show source programs in different format than conventional editors and produce high-quality listings; and programming languages which are designed to be easy to read.

The use of natural naming should make source programs more maintainable. For this reason, we will discuss tools built to increase maintainability of programs or to help in understanding source programs in software maintenance.

Names in source programs, regardless of whether they are abbreviated or natural, are constructed using the elements of natural languages (e.g. natural words, abbreviated natural words, and letters used to form the words of natural languages). Research on naming means studying the use of natural languages in software documentation. Therefore, we will discuss other research areas related to natural languages. Natural languages and their use are addressed in many other scientific fields. These include linguistics, semiotics, and subfields of philosophy. As the essential idea in natural naming is to avoid abbreviations, we will discuss how the use of abbreviations has been taken into account in the above mentioned scientific fields.

4.2 APPROACHES TO INCREASING UNDERSTANDABILITY OF SOURCE PROGRAMS

4.2.1 Guidelines for naming and programming style

Several programming guidelines have been published for enhancing the understandability of source programs (e.g. Ledgard and Tauer 1987, Plum 1984). As well as public guidelines, written internal programming style guidelines exist in many software development organizations. In addition to aiming at understandable source programs, the published programming style guides aim at enhancing portability and efficiency of source code. In many cases these issues seem to be considered more important than understandability. Oman and Cook (1991) have carried out an extensive study on published programming style rules. They have found that there exist many contradictory rules, which indicates that many of the rules are personal opinions of individuals, and more research effort should be devoted to programming style issues.

Naming in programming is, however, a subject that has not been widely addressed by the software engineering research community. Rowe (1985) has published a rather general article dealing with the issue, but he does not provide practical solutions to the problems. Neither have Oman and Cook (1991) been able to find very accurate naming rules from the literature. Usually, programming style guidelines as well as textbooks for programming and software engineering suggest that clear and descriptive names should be used, but they do not advise how the names can be made descriptive.

It is hard to know why naming has received so little attention, although it is obviously one of the most important factors that contributes to the understandability of programs (Gellenbeck and Cook 1991). One reason may be the amazing pace with which computers and programming languages evolve. As there are so many interesting things to be researched in the evolution of technology and in the social impact that computers have made, such things as the names in programs are easily forgotten.

Many researchers are interested in the syntactic and notational matters in programming languages, but pay less attention to the syntax and semantics of names. These are traditionally considered to be outside of the syntax and semantics of programming languages. The meaning of names is important to humans, but in compiler theories there do not exist such concepts as syntax and semantics of names (Aho et al. 1986). It is well known that compilers treat all kinds of names similarly without caring about how understandable the names are. Many of the people who have been involved in defining and developing programming languages possess a mathematician's background. Therefore, they are inclined to favor short notations, and different approaches, such as natural naming, do not necessarily attract them. Similarly, many people carrying out practical software development have a technical background, and their education has not paid much attention to human issues, such as the understandability of documentation.

Although naming as a research subject has not attracted many, there are, nevertheless, publications that deal with naming in programming. Keller (1990) published natural naming rules at the same time that Paper I of this thesis was in press. Although both publications contain quite similar ideas and their objectives are the same, they were written totally independently. Keller does not provide as deep a classification of program elements as we have done in Paper I.

Some naming rules and general discussion about the subject have been published by Anand (1988), Carter (1982), Johnson (1987), and Marca (1981). An interesting notion expressed by Carter (1982) is that the number of different words that we need for constructing the names in the programs of a large software system is likely to be calculated in hundreds rather than in thousands.

Table 2 provides a comparison between the natural naming principles presented in this thesis and the naming rules which are given in the publications which are mentioned above. As can be seen by studying Table 2, the most essential difference between this thesis and other publications is that we clearly consider the overuse of abbreviations harmful. For this reason, we do not provide any rules for abbreviating.

Commenting is a form of documentation that uses the same means to express information as the names in programs, namely the letters and words of a natural language. An essential finding in this thesis, also made by Keller (1990), is that some comments become superfluous when natural names are used. This proves that comments and names are closely related programming style factors. Both appropriate commenting and naming aim at enhancing the understandability of programs. Therefore, publications that deal with commenting are related to the subject of this thesis. However, there is no clear consensus about appropriate commenting. Many published commenting rules have been found contradictory (Oman and Cook 1991), and varying opinions and ideas about appropriate commenting have been expressed (e.g. Grogono 1989, Kaelbling 1988).

Table 2. Comparing publications related to naming.

FEATURES IN THE PUBLICATIONS	Marca (1981)	Carter (1982)	Rowe (1985)	Johnson (1987)	Ledgard (1987)	Anand (1988)	Keller (1990)	PAPER I (1990)	PAPER II (1992)
The publication concentrates mainly on naming.		x	x			x	x	x	x
It is emphasized that an appropriate naming style is important for readability and understandability of programs.		x	x	x	x	x	x	x	x
The use of abbreviations is strongly criticized.								x	x
Rules for forming abbreviations are provided.	x	x			x		x		
The opinion that names should not be very long is maintained.				x	x				
Naming rules for different kinds of program elements are given.					x	x	x	x	x
Techniques for distinguishing related names are provided.							x	x	x
Word lists for naming are provided.		x						x	x
Examples of appropriate names are provided.	x	x	x		x		x	x	x
Examples of appropriately named programs are given.		x	x		x		x	x	x
The relation between names and comments is discussed.						x	x	x	x
Historical reasons for poor naming styles are discussed.			x					x	
It is suggested that people should agree about appropriate naming style before a software development project starts.		x							x
A systematic name creation method is provided.									x

4.2.2 Program visualization tools

At present, electronic forms of source programs are usually stored as ASCII files which may contain only the basic set of ASCII coded symbols. Source programs that are accepted by commercial compilers may not contain any special information, such as graphics or information about fonts and colors. However, there are experimental systems that allow source programs to be stored, viewed, and printed in visually more appealing forms. Baecker and Marcus (1990) present this kind of a system which brings similar features into program editing as we have in the present systems for text processing.

The following are the essential features in the program visualization system introduced in Baecker and Marcus (1990):

- A program can be viewed, edited, and printed in a typographically readable form from which the compilable source code form can be automatically derived.

- Different fonts can be used for different kinds of elements in programs, e.g., function names can be boldface Helvetica and function arguments small Helvetica.

- There can be different kinds of comments (e.g. external and internal) and they can be highlighted in various ways, e.g., by using boxes or gray shading around comments.

- Special marginal comments can be added beside the actual source code.

- Advanced text processing features, such as page headers, are available to make source programs more document-like.

The programs created with program visualization systems could look like the one in Figure 11 (see (Baecker and Marcus 1990) for better examples.) It is very likely that programming environments will develop to allow different kinds of fonts, colors, etc. to be exploited in order to make programs more readable. This kind of development is merely technical, whereas the natural naming approach is human-oriented. We emphasize that programmers need to have human abilities to make their programs understandable and we have created naming principles for programmers to enhance their documentation abilities. Neither a fully automatic tool nor another person can make someone else's programs understandable, since the program writers are the best experts who understand their own work completely. Hence, they are the right people to take the responsibility of making their programs understandable to others.

```
                                  A program to validate a customer number.


                                  #define   CNUMMAX   13
                                  #define   VALID     0
                                  #define   NVALID    1
```

isvalid (

```
Customer number string.           char  cnumbr [],
Return code for validation.       int    *rcode  )

                                  {
Index for string access.              int   i ;
Length of the customer number.        int   len ;

                                      *rcode  =  VALID  ;

                                      len  =  strlen ( cnumbr ) ;

                                      if ( len  >  CNUMMAX )
Checking for appropriate length.      {
                                          *rcode  =  NVALID ;
                                      }
                                      else
                                      {
                                         for  ( i=0 ; i<len ; i++ )
                                         {
                                            if (( cnumbr[ i] < '0') ||
Checking that the string                        ( cnumbr[ i] > '9') )
contains only digits.                       {
                                               *rcode  =  NVALID ;
                                            }
                                         }
                                      }
                                  }
```

Figure 11. A program with enhanced typographic style.

Gellenbeck and Cook (1991) have carried out empirical studies which indicate that the meaningfulness of names and the existence of appropriate comments contribute more to the understandability of programs than certain typographic signals, such as different fonts. On this basis, naming seems to be more important than different fonts. However, developing such systems as the one by Baecker and Marcus is important, since we should consider all possible aids to make the usually complex source programs more friendly for human eyes.

4.2.3 Literate programming

A famous approach to enhance the readability and understandability of programs is literate programming introduced by Knuth. In his introductory article about the approach he expresses deep enthusiasm about the invention (Knuth 1984). A central idea in literate programming is that programs should not be considered just inputs for compilers, but they should be regarded as writings, comparable to the works of literature. Then, the activity of programming should be regarded as a writing process in which style issues play an important role. The task of a programmer is thus to produce descriptions which are meant to be read by humans, and compilation is a secondary issue.

The objectives of literate programming are clearly the same as those of this thesis. We have stressed that it is important to regard source programs as documents. We have personally experienced that when we concentrate on writing programs with a style such that we spend time in choosing appropriate words for names and carefully comment the entire program module, we manage to produce programs with enhanced quality which contain few errors and need little time for debugging.

Literate programming is based on special literate programming environments which have tools that support the approach such that programs are created together with their documentation. Literate source programs are written with a special notation which allows natural language texts and statements written with a programming language to be mixed into a single file. This literate source program can then be processed with special tools which produce the actual compilable source program and a separate literate program document. Figure 12(a) is an imaginary example of a literate source program made with similar notation as used by Cordes and Brown (1991). Figure 12(b) shows the corresponding literate program document generated automatically from the program in Figure 12(a).

```
@* Check customer number.
This program checks the validity of a customer number.

@* Routine for customer number validation.
@d
@<Constant definitions of the program@>
@c
isvalid ( char  cnumbr [],
          int  *rcode  )
{
   @<Internal variables for the program@>
   @<Check customer number validity@>
}
@* Constants.
The following constants are defined:
   Maximum length for the customer number,
   return code when customer number is valid, and
   return code when customer number is not valid.

@<Constant definitions of the program@>=

   #define   CNUMMAX      13
   #define   VALID        0
   #define   NVALID       1

@* Variable declarations.
An index for the customer number string and a variable
  to store the length of the customer number are
  declared as internal variables.

@<Internal variables for the program@>=

  int i,  len  ;
```

```
@* Actual program code.
The validity of the customer number is checked
  according to the following criteria: The length of
  the customer number may not exeed the maximum limit
  and it may contain only numerical digits.

@<Check customer number validity@>=

   @<Initialization of the variables@>

   if ( len > CNUMMAX )
   {
      @<Customer number is not valid@>
   }
   else
   {
      @<Check that string contains numerical digits@>
   }

@<Check that string contains numerical digits@>=
    for  ( i=0 ; i<len ; i++ )
    {
       if (( cnumbr[ i] < '0') ||
           ( cnumbr[ i] > '9') )
       {
          @<Customer number is not valid@>
       }
    }

@* Setting values for variables.

@<Initialization of the variables@>=
    *rcode  =  VALID ;
    len     =  strlen( cnumbr ) ;

@<Customer number is not valid@>=
    *rcode  =  NVALID ;
```

Figure 12(a). An example of a literate source program.

Several experimental literate programming environments have been built. However, the programming community has not widely accepted the literate programming approach. Cordes and Brown (1991) discuss the reasons why literate programming has not gained wide popularity. One obvious reason is that, although a literate programming environment aims to help in software documentation, it also brings extra complexity to the programming work. Literate programmers have to learn to manage a literate programming language, in addition to an ordinary programming language.

Cordes and Brown (1991) point out that it is usually not the lack of text processing skills or tools that makes software documentation difficult, but merely the general difficulty in finding words to explain what a source program does and the difficulties in organizing one's thoughts into an intelligible form. One can create programs which are badly documented even in a literate programming environment. It seems thus quite obvious that special tools cannot automate software documentation. Tools can only help to overcome some practical difficulties, but the real origins for understandable programs are human skills. For example, in Figure 12(b) we can see that a literate programming environment automatically produces a document in which the program is organized into numbered sections. But, however, it is the responsibility of the person who writes the real program, like the one in Figure 12(a), to decide the appropriate division of the program.

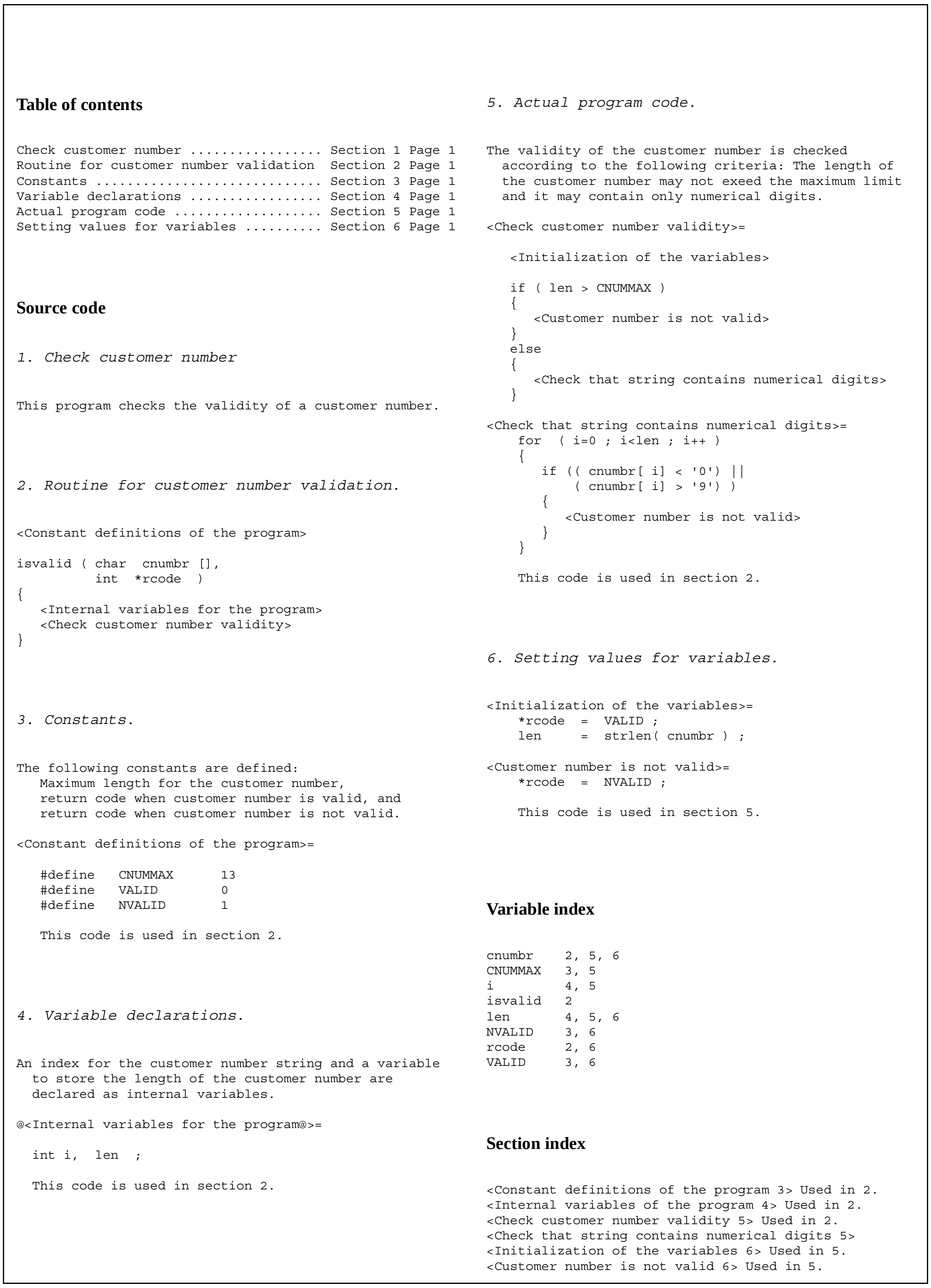


Table of contents

```
Check customer number ................ Section 1 Page 1
Routine for customer number validation  Section 2 Page 1
Constants ............................ Section 3 Page 1
Variable declarations ................ Section 4 Page 1
Actual program code .................. Section 5 Page 1
Setting values for variables ......... Section 6 Page 1
```

Source code

1. Check customer number

```
This program checks the validity of a customer number.
```

2. Routine for customer number validation.

```
<Constant definitions of the program>

isvalid ( char  cnumbr [],
          int  *rcode  )
{
   <Internal variables for the program>
   <Check customer number validity>
}
```

3. Constants.

```
The following constants are defined:
   Maximum length for the customer number,
   return code when customer number is valid, and
   return code when customer number is not valid.

<Constant definitions of the program>=

   #define   CNUMMAX      13
   #define   VALID        0
   #define   NVALID       1

   This code is used in section 2.
```

4. Variable declarations.

```
An index for the customer number string and a variable
  to store the length of the customer number are
  declared as internal variables.

@<Internal variables for the program@>=

  int i,  len  ;

  This code is used in section 2.
```

5. Actual program code.

```
The validity of the customer number is checked
  according to the following criteria: The length of
  the customer number may not exeed the maximum limit
  and it may contain only numerical digits.

<Check customer number validity>=

   <Initialization of the variables>

   if ( len > CNUMMAX )
   {
      <Customer number is not valid>
   }
   else
   {
      <Check that string contains numerical digits>
   }

<Check that string contains numerical digits>=
    for  ( i=0 ; i<len ; i++ )
    {
       if (( cnumbr[ i] < '0') ||
           ( cnumbr[ i] > '9') )
       {
          <Customer number is not valid>
       }
    }

    This code is used in section 2.
```

6. Setting values for variables.

```
<Initialization of the variables>=
    *rcode  =  VALID ;
    len     =  strlen( cnumbr ) ;

<Customer number is not valid>=
    *rcode  =  NVALID ;

    This code is used in section 5.
```

Variable index

Section index

Figure 12(b). An example of a generated literate program document.

In our view, the essential difference between conventional and literate programming is that in literate programming the words and sentences of a natural language can be treated in a different manner. A literate programming environment allows a program writer to arrange natural words and sentences according to a different notation than that of a conventional programming language. The literate program in Figures 12(a) and 12(b) is the same as the naturally named program in Figure 5(c). By comparing these versions of the same program, it can be noticed that by using long natural names we can similarly put long sentences into a program as can be done in a literate programming environment.

For example, in Figure 5(c) we have a name

```
check_customer_number_validity
```

which is uniquely distinguished by the compiler, and in Figure 12(a) there is the expression

```
<Check customer number validity@>
```

which could be uniquely distinguished in a literate programming environment. Moreover, many of the written texts that are in the literate program in Figure 12(a) could be put into comments in a conventional program. Thus, although literate programming environments offer many features which help software documentation, we can claim that literate programs can also be written by using appropriate comments and long natural names with conventional programming tools.

The literate programming approach is important for the research related to software documentation, though the approach has not, at least to date, become especially popular. It is essential to stress that source programs are writings that need to be studied by humans, and, therefore, an appropriate program writing style is important.

4.2.4 Easy-to-read programming languages: COBOL and SNAP

The difficulty of computer programming was realized during the early decades of the history of computers. Only specialized people could write computer programs and people who would benefit from the programs could not know for sure whether the programs were doing correct computations. Because of these problems the widely-used programming language COBOL (Common Business-Oriented Language) was defined in the late fifties (Sammet 1981). That language tried to bring features of natural languages into computer programming. SNAP (Stylized Natural Procedural Language) was another language, although not widely known, which tried to make computer programming resemble writing in a natural language (Barnett 1969). Because natural naming also attempts to make computer programs more readable, the mentioned programming languages are related to this work. It is also interesting to note that during the first decades of computing there were discussions about using a pure natural language to dictate the behavior of computers (Sammet 1966).

COBOL is a programming language that has its origins in the fifties and has been very widely used since the late sixties. COBOL is intended to be used in business-oriented applications and the motivations for its definition involve the following (Sammet 1981):

- Due to the time and cost of programming, there was a need for a language that would be easy to learn and use.
- There was a need to broaden the base of those that can state problems to computers.
- There was a desire for people without a programmer's education to be able to read and write computer programs. It should also be possible for managers to be able to read programs in order to check that various kinds of financial calculations are performed correctly.
- There was a need to have programs that could be run on computers from different manufacturers.

The developers of COBOL responded to the needs listed above by defining a language which is in many ways close to natural English. The solution is thus related to the principle of natural naming.

The natural features of COBOL include such aspects as having keywords ADD, SUBTRACT, MULTIPLY, and DIVIDE instead of short mathematical symbols such as "+", "-", "*", and "/". Using natural words instead of mathematical symbols raised many disputes among computer scientists, and, therefore, COBOL variations allowing the use of mathematical symbols also exist.

Many documentation-related aspects were considered when COBOL was defined. The first official version of the COBOL definition included the feature that names could be long and natural names were intended to be used. A COBOL program is intended to be a document itself. The language has documentation related keywords such as AUTHOR, DIVISION, and SECTION which should help the program writer to incorporate relevant documentary information and to divide a program into logical parts.

COBOL has been widely used, although it has been neglected by many computer scientists (Shneiderman 1985). Whether or not COBOL has succeeded in being a language that yields readable and understandable programs is too difficult to be answered here. In our opinion, the writing skills of programmers are still needed to achieve appropriate under-standability of programs. Of course, COBOL programs can be made difficult to read and abbreviated names have been used in programming with COBOL.

SNAP is another programming language which tries to imitate natural languages. SNAP is similar to COBOL in the sense that it includes a large repository of reserved words borrowed from English. As listed by Barnett (1969), the objectives in the development of SNAP were largely the same as in the development of COBOL. SNAP did not, however, become a widely-used or well-known language. Probably the language never had commercially-available compilers; it was used mainly by students. SNAP was designed for applications involving mainly text processing, file reading, and printing. Its narrow applicability is one reason why it was only used in small circles.

Obviously, the developers of COBOL and SNAP thought that by incorporating many reserved words from a natural language into the programming language, the programs written in that programming language become easy-to-read and understand. Using abbreviations has been and still is common in COBOL programming. The SNAP programs shown by Barnett (1969) contain many abbreviations. Therefore, the approach of making programs understandable is different in the case of these languages than in the case of using natural naming. Having many reserved words borrowed from natural languages does not necessarily make source programs more understandable. Reserved words are symbols which need to be repeated over and over again in source programs, and they are the same in every program. For this reason, it should not really harm if short special symbols (e.g. "+" and "-") are used instead of longer reserved words (e.g. ADD and SUBTRACT), because these symbols have to be used very often and need therefore to be kept in mind. Names, on the other hand, are unique in every program. They are more likely to show how a program differs from other programs.

It is certain that most programming languages have been designed to make computer programming easier in some way. It is impossible to think that somebody would have created a programming language which he or she would not have thought to be better in some fashion than some existing languages. Because there exist hundreds of programming languages (Sammet 1972), we cannot discuss all of them here. However, to our knowledge, there is no programming language which has been specifically designed to incorporate the use of natural naming.

4.3 TOOLS TO AID IN SOFTWARE MAINTENANCE

In the previous section we discussed techniques for making source programs more understandable and thereby more maintainable. In this section we will introduce slightly different techniques which help to maintain different source programs and other software documents which have been found hard to maintain. Here too, we will mostly concentrate on source programs, because they are usually the most reliable documents for software maintainers (Bennett et al. 1991).

Perhaps the simplest tools to make source programs more maintainable are so-called prettyprinters which can produce program listings in which different text fonts are used and reserved words of the programming language are printed with different fonts than the rest of the program text. The term is also used for tools which can adjust alignment and indentation in source programs.

More advanced systems are browsing tools which can be used to study a set of source programs for detecting dependencies between different source program files. A browsing tool can, for example, find all places in all source files where a certain function or procedure is being called or a certain variable used. With a browsing tool a software maintainer can easily jump from one source program to another to study interdependent parts of an application. An example of a browsing tool is Sbrowse by Computer Enterprises. Suitiala (1993) introduces another browsing tool.

In addition to these tools there are more complex reverse engineering tools used to make existing applications more maintainable. These are discussed for example in IEEE Software (Vol. 7, No. 1), in the proceedings of software maintenance conferences (e.g. ICSM 1994), and in the proceedings of workshops related to program comprehension (e.g. WPC 1993). As there are many both experimental and commercial tools available, we will discuss them according to a known taxonomy. Chikofsky and Cross (1990) classify the reverse engineering activities and tools as follows:

- *Reverse engineering* tools and techniques aim at producing a higher-abstraction-level description of an existing system. Practical reverse engineering tools can, for example, produce graphical-textual descriptions, such as data flow diagrams, from existing source programs. Reverse engineering is a process of examination and it does not include modifying an existing system.

- *Design recovery* is a special case of reverse engineering. Design recovery produces higher-level abstractions from existing systems, but it also adds external knowledge to the higher-level abstractions being produced. Biggerstaff (1989) has written an important article of design recovery.

- *Restructuring* means renovating an existing application without changing its external behavior. Restructuring does not switch from one abstraction level to another. In practice, restructuring can be, for example, modifying source programs so that goto statements are no longer needed.

- *Redocumentation* means changing a system's documentation without making any functional changes to the software system. We see redocumentation as a special case of restructuring, although Chikofsky and Cross (1990) have listed it as the simplest form of reverse engineering. Replacing abbreviated names with natural ones is one possible form of redocumentation.

- *Reengineering* is the process in which a system is modified after having been reverse engineered.

Chikofsky and Cross (1990) admit that some of the above-mentioned terms can be vague in some cases, but those terms are certainly important for clarifying different activities and tools used to make existing systems more maintainable. The existence of reverse engineering tools is an indication that source programs being maintained are usually hard to understand.

New kinds of tools to help in software maintenance are being developed in a project called Application Management Environments and Support (AMES 1993). The research activity (a4) defined in Section 3.1 above was partly carried out in the AMES project. Software maintenance is included in a wider concept called application management in the AMES project (Boldyreff et al. 1994). The project is producing tools which use a special database containing information about the application being maintained. The new kinds of tools are the following (AMES 1993):

- *Impact analysis tools* try to show a software maintainer which other documents need to be modified if a certain modification takes place in one software document.

- *Navigation and display tools* allow a user to navigate among a set of software documents which can be documents other than source programs. Above, we discussed browsers which can be used to view dependencies among source program files. Navigation and display tools can be characterized as browsers which allow other documents in addition to source programs.

- *Application understanding tools* aim at forming a combined representation of different types of software documents. Through the use of application understanding tools, a user should be able to view documentation of a software system from different points of view.

4.4 FIELDS OUTSIDE SOFTWARE DOMAIN: LINGUISTICS, SEMIOTICS, AND PHILOSOPHY

Although linguists, semioticians, and philosophers have not directly studied how natural languages should be used in software documentation, we shall briefly discuss their fields here because our research is related to natural languages. Linguists study natural languages. Semioticians study the symbols of natural languages among other symbols used in human communication. Philosophers have also been interested in how natural languages correspond with the reality around us. As the main idea in natural naming is to avoid abbreviations, we will try to find out what has been said about abbreviations in these fields.

An excellent textbook of basic linguistics has been written by Fromkin and Rodman (1988). The field of linguistics studies the syntax and semantics of languages, as well as word formation (morphology), the sounds of language (phonology), and sound production (phonetics). The history of natural languages and their role in society are also linguistic subjects. Because natural languages have first had a spoken form with writing systems having been invented later, linguistics is in many ways more concerned with spoken languages. In software documentation, we are more interested in written languages.

Linguists admit that the nature of our natural languages is not yet fully discovered and it perhaps never will be (Fromkin and Rodman 1988). Natural languages are indeed very complex, although some similarities can be found between all human languages. Due to the complexity of languages, it is natural that there are several schools of linguistics. Traditional linguistics has been criticized by Yngve (1986) who thinks that languages should be studied merely as a result of human activities, not as objects per se. Traditional linguistics tends to view languages without referring to human activities behind the languages.

Although linguistics sees abbreviating as one way to coin new words in natural languages, we have not been able to find opinions or data about the usefulness of abbreviations in linguistic publications. It is well known that some abbreviations have been used about two thousand years (Hall-Quest 1979), but the benefits or disadvantages of their use have not been widely discussed. In fact, we have found the only articles dealing with abbreviations and efficiency of communication in the field of "technical linguistics", namely in the publications of IEEE Professional Communication Society (Logsdon and Logsdon 1986, Ibrahim 1989). Neither of these publications refer to any other articles which dealt with the use of abbreviations in communication. Thus, we have good reason to believe that the use of abbreviations has not been a popular research topic. It is seen as a problem only in the fields of technical documentation and communication.

To find articles dealing with abbreviations and communication, we have conducted extensive searches in databases containing abstracts of articles from linguistics, informatics, and several other fields. The results of the searches were interesting, though we did not find any material which could be strongly related to this thesis. Concerning the use of abbreviations, we found articles which, for example, reported that the growing popularity of abbreviations causes problems in teaching foreign languages (Ching 1983), and that one of the hardest word types for people to understand are abbreviations (Smith and Taffler 1992).

Although linguistics has not provided answers to the question whether abbreviations are useful or not, there is one linguistic finding which is valuable for this research. The finding is that all human languages have changed during the history of mankind (Fromkin and Rodman 1988). Our languages are not stable. They are changing all the time. Perhaps the most obvious change is that new words emerge within our languages. It is assumed that new words emerge partly because new technological innovations are being made. Words like "microprocessor", "Prolog", and "mainframe" did not exist before computers were invented. To communicate about computers and software, many popular abbreviations have been coined: RISC, PC, WWW, etc. The abbreviations in source programs and other software documents can also be seen as new symbols in our languages. In this thesis we thus consider that the overuse of previously unknown abbreviations changes our languages too rapidly and in an uncontrolled way, and can therefore be considered harmful.

As we are concerned about the overuse of abbreviations in software documentation, we are interested in the change and the future of our languages. The language of software documentation is, in our opinion, developing in the wrong direction because too many abbreviations are entering the language. This is the essential difference between this thesis and linguistic research. Linguists are more interested in the history and present state of natural languages. The historical change of languages has been an important research subject in linguistics. It is understandable that linguists have not studied the usage of natural languages in technical documentation, because in order to do that one needs to be at least a partial expert in the technical field in question.

Benjamin Lee Whorf is a linguist who deserves to be mentioned here. He has proposed a famous hypothesis which states that the natural language we use dictates how we can think and how we perceive the world around us (Carroll 1956). Whorf concluded this after studying the languages of American Indians, and comparing their languages to the Indo-European languages. Soloway (1986) has also noted the Whorfian hypothesis while studying the mental processes of programmers. Subjectively, we have found that, compared to using abbreviations, using natural names in programming helps a programmer to think about the problems being solved. This supports the Whorfian hypothesis, if we consider that abbreviations are symbols of one language and natural names are symbols of another language.

Semiotics is a scientific field that includes linguistics. Semiotics also studies symbol systems other than just natural words. Non-linguistic symbols which have meanings to people include, for example traffic signs and lights, music, and paintings. Even buildings can be considered to transfer meanings. Because the semiotic world includes so many potential research subjects, some people are doubtful whether semiotics can be considered a science (Tarasti 1990). An influential person in the field of semiotics is Umberto Eco. His books about semiotics (e.g. Eco 1984, Eco 1990) provide numerous references to philosophy, which indicate that natural languages have been subjects of interest to people who have been pondering the most fundamental questions about living as a human being in this world.

It seems that semioticians have not studied the problems related to abbreviations either. The situation is the same as in linguistics. Semiotics deals with existing symbols and how these symbols relate to the human behavior. To our knowledge, semioticians have not studied whether it is useful, or not, for efficient communication to form new symbols by abbreviating existing natural words, or to construct acronyms from the first letters of a list of natural words. Andersen (1990) presents a specific theory for computer semiotics, but he does not deal with the goodness of symbols. Andersen's semiotic theory is directed towards using a computer and making computers more user-friendly, not towards documenting computer software. Therefore, he does not specifically address the problems of software documentation.

A central question related to languages is how the elements of languages (e.g. words) actually relate to the real world around us. Nelson (1992) writes that we actually know as little about these matters as was known by the philosophers of ancient Greece. Nelson (1992) surveys many different theories of linguistic reference. His study shows that there is little consensus in regard to these matters. One question in this research is that what does a name in a software document mean or refer to. Our claim is that a name like "customer_number" refers more effectively than a name like "cnumbr" in a source program. But if the name "customer_number" refers well, where does it refer to? If it refers to a location in a computer's memory where a customer number is stored, what does the customer number indicate? If each customer has a number, what does that number mean? Why do customers have numbers? Why aren't they just called by their names?

These are difficult questions. Answering them would require much knowledge about the software system in which the customer numbers are used.

The referent of a name depends on the context, i.e. on the software system in question. However, if we use the name "customer_number", that name is in its textual appearance closer to the question "What is a customer number?" than an abbreviated name like "cnumbr" would be. To clarify this matter, let us imagine two discussions between A and B. The discussion with an abbreviated name could be:

A: "What is cnumbr?"
B: "It is an abbreviation for customer number."
A: "What does the customer number mean?"
B: "Well, in this particular application the customer number ..."

The discussion with a natural name could go as follows:

A: "What is customer_number?"
B: "Well, in this particular application the customer number ..."

From these imaginary discussions we can see that, although the actual meaning of a name may be difficult to explain, it is possible that the discussion is at least one question shorter when natural names are in use.

It is a complex philosophical question how a language relates to the real world. What are the meanings of the words and other symbols? Can the meanings be explained? The philosopher Ludwig Wittgenstein has provided an answer to the latter question. According to Wittgenstein (1953), the meanings of words and other symbols become evident in the situation in which they are used. Some words have quite stable and usual meanings, but the meanings of some less frequently used words depend much on the context of usage. Wittgenstein (1953) uses the term "language game" to denote the process in which we associate meaning to words and other symbols. Therefore, a name like "customer_number" in some source program means something specific in that program or in the larger software system in which the program is used. Developing a software system can, in the Wittgensteinian sense, be regarded as playing a language game in which certain names and word combinations gain a special meaning. We can exploit Wittgenstein's theory to support the use of natural naming by noting that the language game of a software system becomes more complex when too many abbreviations are in use.

To our knowledge, Ludwig Wittgenstein is a philosopher who has worked more on the philosophy of languages than other philosophers. He has also been noted by other writers in the field of computer science (e.g. Zemanek 1974, Sowa 1990). In some ways, Wittgenstein's life and work can be seen as a proof about the complexity of human languages. He was a rather radical and extraordinary person (Jarman 1993). During his younger years, he published a well-known theory related to logic and languages (Wittgenstein 1921). Having invented this theory he considered that he had solved the essential problems of philosophy, and he therefore abandoned philosophical work for years. Later, however, he discovered that his earlier theory did not match with the real world and natural languages. He returned to philosophy, but, according to the preface of Wittgenstein (1953), he was disappointed that he was unable to formulate a clear and elegant theory about languages.

It seems that the meanings of all textual symbols cannot be exactly described. The meanings of some symbols become evident from the way they are used (Wittgenstein 1953). To still investigate the problems of meaning, let us take a look at the cartoon in Figure 13. Although that drawing was not intended for serious study, it is interesting for us. The man in the figure is using paint and a paintbrush to mark the objects in his environment with their appropriate names. There is, however, something essential in the figure that the man cannot mark with its name. It is hard to write on air or on the sky. Even the paintbrush is hard to mark, but on the paint itself the man cannot write anything. Paint is a special object because it is used to mark the other objects. Paint shows us the names of the other objects, but its own meaning we can only guess.

One could also ask whether the marked things are clear in Figure 13. We answer that things can always be made clearer, but they never become completely clear. Something is always left unexplained and is considered evident. The man in the figure supposes that everybody knows what the word "house" means. To make things even clearer, the man could, for example, write on the wall that the word "house" means "a building for a person or a family to live in" (Webster's 1989). He could continue by telling more about the meanings of the words "building", "person", and "family". He could write a really long story about what the verb "to live" means. Actually, he could spend the rest of his life explaining the nature of the objects which he has just marked with paint. The problem whether the things are clear in the figure is the same as whether natural names are clear enough. In Section 2.3 we said that it largely depends on the context which natural names can be considered sufficiently informative. In this sense, we can say that things are certainly clear enough in Figure 13. Normally we can recognize houses and trees even if they are not marked as in the figure.

The original thesis has a Gary Larson’s drawing on this page. Because of copyright reasons the drawing is not included in the .pdf version of the thesis. In the drawing there is a man who has written “THE DOG”, “THE CAT”, “THE HOUSE”, “THE TREE”, etc. on the corresponding objects in his environment. To find more information of Gary Larson’s drawings, please visit www.thefarside.com

Figure 13. A man who made things clear.

4.5 DISCUSSION OF RELATED WORK

To summarize the previous sections of this chapter, we state the following:

- There are many approaches to increasing the understandability and maintainability of source programs and other software documents, but the issue of naming has not been explicitly taken into account in most cases.

- In the same way as natural naming, the other approaches to increasing understandability lack hard evidence about their usefulness. For this reason, it is difficult to compare this thesis and the related research. This is a common problem in our field. Glass (1994) describes this type of research as "advocacy research".

- Considering the discussion on linguistics, semiotics, and philosophy, our main conclusion is that the researchers inn those fields are mostly addressing different problems than we are. The researchers outside the domain of computers are interested in the history and present state of languages, whereas we are interested in the future of languages. The use of abbreviations in software documentation can be seen to change the natural language of software documentation. If too many abbreviations are used, that may cause future understandability problems for the readers of that documentation.

This thesis deals with the understandability of source programs and other software documents, and with the use of natural languages in software documentation. Although natural languages and understanding are important research issues in many branches of computer science and software engineering, most research regarding these issues is not very closely related to this thesis. Below, we discuss two of these research branches and explain how they differ from this work:

- Special techniques have been developed to make computers understand natural languages (Allen 1987, Smeaton 1992). If these techniques became popular in practical applications, it would certainly make computers even more useful to mankind. Natural language understanding techniques can be applied, for instance, in multimedia systems (Rowe and Guglielmo 1993). Computerized natural language understanding is not related to our work, because these techniques are not used in the case of software documentation.

- Different kinds of models for human mental activities in program understanding have been proposed (Curtis 1985, Hoc et al. 1990). Perhaps the most famous work in this area has been done by Soloway and Ehrlich (1984) who have shown by empirical human experiments that professional programmers possess certain kind of mental programming plans, according to which they develop and understand source programs. It is very important to do this kind of psychological research. However, we do not consider much of this research on mind models closely related to our work, because the researchers have not specifically studied how different styles of natural language usage affect the working of the human mind.

5 INTRODUCTION TO THE INCLUDED PAPERS

Six papers, which have been accepted in scientific journals and conference proceedings during a five-year period, are included in this thesis.[1] The papers provide solutions to the research activities which we defined in Section 3.1. The papers correspond with the defined research activities as shown in Table 3.

Table 3. Relations between research activities and included papers.

Research activity code	Description of the research activity	Number of paper	Publication year
(a1)	Produce detailed naming principles.	I	1990
(a2)	Formulate a method for early name creation.	II	1992
(a3)	Investigate how programming languages could be made more suitable for the use of natural naming.	III	1994
(a4)	Investigate how already abbreviated names could be replaced with natural ones in existing source programs.	IV, V	1995, 1995
(a5)	Produce evidence of the usefulness of natural naming.	VI	1995

5.1 PAPER I: GUIDELINES FOR NATURAL NAMING

This paper introduces a set of natural naming principles for programming with the C programming language, though most of the naming principles are applicable to other procedural programming languages as well. The essential idea in the paper is the principle of natural naming. That principle is derived from the notion that it is useful to avoid abbreviations in order to minimize the risk that something is misunderstood. The ultimate goal is the avoidance of abbreviations all together, and thus we have the principle of natural naming. The term "natural naming" is, however, not used in Paper I. That term was first used by Keller (1990), at the time when Paper I was in press.

The natural naming principles presented in Paper I are meant to guide programmers rather than provide absolute naming rules which programmers should follow. It is pointed out that natural naming principles cannot be very strict, because natural languages are informal and not always accurate means for presenting information. To exploit the naming principles in practical

1. To inform non-Finnish readers, we would like to point out that it is common in Finland, and in some other northern European countries, to construct a doctoral thesis of papers which have been accepted in conference proceedings and journals. This thesis is of this type.

work, one should use a compiler that can distinguish rather long names, preferably at least 30 characters in length.

Section 2 of Paper I presents the naming principles. As the principles are based on a natural language, some characteristics of natural languages are discussed as an introduction. The names needed in programs are classified into two fundamentally different categories: names that represent information and names that represent action in programs. The most important names in the latter category are the names of functions and procedures. The names of macros and labels also represent action, but they are given little attention in Paper I. The names that represent information include constants and variable data (variables, tables, structures, etc.). The essential difference between these two types of names that represent information is that information expressed by the names of constants is fixed during the process of programming, whereas the information expressed by the names of variable data can change during the process of program execution.

Separate name tables for constants, variable data, and functions are presented in Section 2. In fact, each table provides a low-level classification of program elements in its main category. For instance, variable data is broken down into indexing variables, statuses, counters, etc. The name tables provide keywords as well as example names for each low-level class of program elements. Keywords are recommended to be used in name formation and their usage is guided by a rule that a keyword should be either the first or the last word in a name. Keywords for function names are verbs which should be used as the first words of the names.

Programs often include variables that contain the same information but are used in different contexts. They may also include functions that are performing similar actions but differ in some aspects. For these reasons, there has to be means for constructing related names in programs so that their relatedness is evident and their difference as clear as possible. For this purpose, Paper I presents attributes which can be characterized as name refining words. Attributes are classified according to which properties they express or in what kinds of situations they are used.

Section 2 includes a program example in which the natural naming principles are followed. A program to evaluate and print prime numbers is used for this purpose.

The third section of Paper I is devoted to discussion of the historical background of naming, the benefits of natural naming, and the difficulties in the use of natural naming. It is evident that the use of natural naming affects the entire program module documentation. Obviously, it makes the documentation as a whole simpler by reducing the need to use other forms of documentation, such as commenting and pseudo-coding. The need to have in-line comments is reduced, since natural names usually provide the same information that traditionally has been shown by comments (e.g. a variable definition does not need to be accompanied with a comment in the same line to explain what the variable is used for). Pseudo-code for a program can be characterized as structured natural language. Its under-standability is between that of a programming language and a natural language. As the use of natural naming brings source programs closer to a natural language, the need to have an intermediate pseudo-code is reduced.

Being able to simplify program module documentation is one of the most important advantages of natural naming. Other advantages, such as easier pronunciation and ease in remembering the names, are mentioned in Section 3, but they are less obvious. The difficulties which may be involved in the use of natural naming include the facts that some software development tools still limit the lengths of names and the syntax of a typical programming language is designed for short names. Another difficulty is that software developers have to learn to invent names, to spend time in choosing appropriate names, and to write with a style that allows long names to be fitted conveniently on screen and on paper.

Paper I ends with the following paragraph which concisely summarizes the message of the paper:

> The basic purpose of informative naming is that the documentary value of programs would increase. This has the practical benefit of raising the quality and understandability. Programs are information. Success in our current information society depends much on how effectively information will be transferred, understood, and learned.

5.2 PAPER II: A METHOD FOR INITIAL NAME CREATION

When software systems are developed in a disciplined manner, the implementation phase is usually preceded by other development phases which also involve writing documents that describe the system under development. These high-level documents can be, for example, written requirement descriptions or graphical-textual models such as data flow diagrams and state transition diagrams. Obviously, all types of documents that describe the same system must be written using the same concepts related to the system under development. For example, a requirement description may define that customers must be identified by a "customer number", and a source program must have a corresponding variable in which the customer number is stored. When the same concepts are used in different types of documents, they should have the same names so that the entire documentation of a system would be understandable. Thus, a concept which is "customer number" in a requirement description document should be called with the same word combination in other documents including source programs. It should not, for example, be called "c_number", "cnbr", or "customer code".

In order to ensure that all software documents have the same names for the same concepts, we should be able to create the names before any software documents are written. Paper II provides a solution to this problem by introducing a method called *Disciplined Natural Naming (DNN).* The DNN method should be used prior to the actual software development to create a repository of names that are acceptable in various types of software documents.

The DNN method uses *DNN tools* in a systematic name creation process. Section 2 of Paper II introduces the DNN tools, which are *naming principles, name creation tables, name creation templates,* and *reference name tables*. The essential naming principle is the idea of natural naming. Some of the principles presented in Paper I are briefly discussed. Additionally, a new alternative naming approach for functions, "object-oriented naming", is explained. A simple name shortening rule is given.

Name creation tables contain applicable final words to be added in the end of existing names in order to derive new names. There exist name creation tables to create attribute names, event names, and state value names. For each name creation table an example of its usage is given. For instance, if one needs to create event names that can be associated to the object "customer", one uses the relevant name creation table with appropriate final words yielding the event names

"customer_introduction",
"customer_removal",
"customer_entering", and
"customer_leaving".

Name creation templates are similar to name creation tables, except that the templates also provide name refining words to be added in the beginning of existing names. There exist name creation templates for event-related and constraining names. A typical constraining name is one which limits the properties of a certain object or attribute, for example,

"maximum_customer_number_length"

describes the limits of "customer_number". A typical event-related name is

"customer_entering_time"

which can be used to describe temporal properties related to the event "customer_entering".

Reference name tables provide final words to implement data structures and single data items, verbs to implement functions, and some general constants to be used to describe state information. Reference name tables are not used in equally systematic way as the other DNN tools. They may be referred to whenever needs to find new names emerge.

The third section of Paper II introduces the *DNN name creation process* in which the DNN tools -- name creation tables and name templates -- are used in certain order. Name creation is a stepwise process, involving twelve steps, in which the name creator applies a name creation tool, analyzes the derived candidate names, and makes decisions whether the candidate names represent essential concepts of the application domain. According to the name creator's decisions, the candidate names are accepted, rejected, or modified. Those candidate names which are accepted as such, or after modification, become potential names, which means that they might be needed in the documentation of the system to be developed. The stepwise name creation process is demonstrated using an example. The created names are then used in some example software documents.

The fourth section of Paper II is devoted to discussion about the benefits and weaknesses of the DNN method. The essential benefit of the method is the same as that of the principle of natural naming in general: the understandability of source programs is improved. As software development is partly a communication and learning process, fixing the names early should help software developers communicate efficiently with each other as well as with different interest groups for the system being developed. Software developers need to communicate with such interest groups as end users, marketing people, and service personnel who are usually not familiar with software development. A benefit is that it is likely that the communication activities are efficient when software developers can communicate with the same natural words and names which they use in the entire software documentation. Additional plausible benefits of the DNN method include simplified software documentation and easier search activities for finding specific locations in source programs and other documents.

Use of the DNN method results in a repository of names which should be used in all software documents of a system. Having such a name repository means, in fact, regulating how a natural language should be used in software development work. Because natural languages are complex and natural words can be used in so many ways, it is difficult to control the usage of natural languages. This is seen to be the most essential weakness of the DNN method. Another weakness is that the practical use and maintenance of a name repository may turn out to be difficult.

The last part of Section 4 surveys related research. Although the problems of naming in programming have been scarcely addressed by the research community, several research papers have been published in which naming or the use of natural languages has been discussed from different points of view (e.g. in the context of requirement specification and reverse engineering).

Paper II concludes by stressing the importance of naming to achieving appropriate readability and understandability of software documents. Although the DNN method needs more testing in practical software development situations, the paper has demonstrated that pre-development name creation is possible.

5.3 PAPER III: A PROGRAMMING LANGUAGE TO SUPPORT NATURAL NAMING

Because natural naming has not been widely used in programming, we presume that none of the existing programming languages has been designed to be especially suitable for programming with natural names. Natural names themselves contain much information. When conventional programming languages are used, software developers have to double-specify many things in source programs if they use natural names. For example, the naming guidelines in Paper I recommend that counting variables should be ended with the word "counter". However, counting variables are normally always integers. When we define a counting variable such as

```
int  character_counter ;
```

we actually double-specify that it is an integer variable. Both the reserved word "int" and the word "counter" indicate that here is an integer variable in question. A compiler could be made to infer solely from the word "counter" that the type of the variable is integer.

The variable definition above is one example which indicates that programming languages could be different when natural names are used. Because programming languages are fundamental tools in software development, we decided to investigate how existing procedural programming languages could be improved to make them more suitable for programming with natural names. The method in this study was that we first examined existing naturally named programs, and made decisions on how the constructs of the programming language could be improved to:

- decrease redundant information in naturally named source programs,
- make the usually-long natural names more easily fit in the programs,
- make source programs look more like documents, and
- encourage software developers to produce better software documentation.

We tested the ideas by implementing them in an experimental programming language called *Pacific*. Section 2 of Paper III describes the characteristics of this programming language.

Because we consider that names are among the most important means for making source programs understandable, the syntax of the Pacific language was designed to highlight names in source programs. To achieve this, we tried to minimize the use of special characters (e.g. ;, (,), [,], {, and }) as syntactic elements in the language. The assumption was that if programs contained many special characters, that would disturb the reader from focusing attention on the names. More about Pacific's general characteristics can be read in Subsection 2.1 of Paper III.

An essential feature in Pacific is to use the natural words in names to infer what type of information a name represents. The Pacific type system is thus unique. The compiler uses a few natural words to detect some frequently used information constructs. For instance, names ending with the word "index" are considered to be integers, and names ending with "record" are considered to be records. All defined names in a Pacific program can also serve as data type specifications for other names. A name can inherit the type of another name when there is enough similarity in the wording of the two names. For example, the name "new_customer_number" can inherit the type of the previously defined name "customer_number", because both names are ended with the same word combination. The Pacific type system is described in more detail in Subsection 2.2 of Paper III.

The Pacific compiler checks all natural words used in the names. Only those words are accepted which are found in the lexicon of the compiler. If special terms from the application domain are needed, they must be inserted into the lexicon. Software developers must thus document their programs by using a controlled vocabulary. Having a lexicon embedded into a compiler allows the introduction of a new variable type in the programming language. In Pacific, these variables are called text variables and they make it easier to incorporate naturally written phrases into source programs. Text variables are explained in Subsection 2.3 of Paper III.

The Pacific language includes some special control structures. Its procedures are special in the sense that they can always return textual status information when called. These features are discussed in Subsections 2.5 and 2.6 of Paper III.

To summarize the features of Pacific, we point out that the following features most directly support the use of natural naming:

- The Pacific lexicon, when maintained appropriately, provides a standard vocabulary for software documentation.
- The type system encourages the use of natural naming as a set of nat-

ural words are used to indicate the types.

- The text variables make it easy to present state information as textual phrases.

Some of the features in Pacific support natural naming only indirectly. For example, the special control structures reduce the need to repeat names in source code, and the syntax in which the use of special characters is minimized aims at highlighting names in source code. Pacific's mechanism for handling global data cannot be considered to be specifically related to the use of natural naming, but it is a feature which should be useful when a software system consists of many source program files.

Section 3 of Paper III provides a qualitative evaluation of the Pacific language. The main conclusions are the following:

- Writing naturally named programs with Pacific usually results in shorter programs than with other programming languages.
- Because Pacific programs resemble pseudo-code in many ways, their readability should be better than other programming languages.
- Using a lexicon for officially acceptable words should bring more discipline into software documentation, although we have no practical experience with using a lexicon in multi-person projects.
- Pacific is slightly slower than other procedural languages. It may not be suitable for time-critical programming.

The main objective in implementing an experimental compiler for the Pacific programming language was to demonstrate the ideas for making source programs more understandable. Pacific could still be improved. As explained in Subsection 3.2, natural language understanding techniques could be further studied for programming language design.

5.4 PAPER IV: DISABBREVIATION OF TECHNICAL TEXT

We use the word "disabbreviation" to mean the process of replacing abbreviations with more natural expressions. The word is also used to denote a replacement for an abbreviation. Paper IV, in which the author of this thesis is the second writer, describes an experimental tool designed to look for abbreviations in technical documents, suggest replacements for abbreviations, and ask the user of the tool to choose the best replacement for each abbreviation. The tool is called a "disabbreviator".

As it is assumed in this thesis that abbreviations are harmful for understandability, it is important to find ways for getting rid of existing abbreviations in documents which need to be used for a long time. Because replacing abbreviations with natural substitutions is quite a mechanical process, it can be carried out with a computer tool. The disabbreviator tool discussed in Paper IV can be used to disabbreviate various kinds of technical documents, including source programs.

In order to examine electronic documents and decide which character patterns are acceptable, the disabbreviator tool has a stored dictionary containing acceptable English words. Subsection 2.1 of Paper IV discusses various data structures needed in the disabbreviation process. In addition to a general dictionary, the tool needs lists of reserved words of programming languages and operating systems, and pairs of commonly used abbreviations and their natural counterparts. The disabbreviator tool can also use domain-specific dictionaries and lists of known disabbreviations which have been created when the tool has been used.

The disabbreviation process has three phases. First, the tool checks the text of the entire document and stores all unknown words in a database. In the second phase, the tool interacts with the user, who it asks to give a replacement for each unknown word. Whenever possible, the tool suggests replacements for the unknown words. In the third phase of the disabbreviation process, the selected unknown words are replaced with the expression chosen by the user.

The disabbreviator tool is intelligent as it can suggest replacements for unknown words. While inventing replacements, the tool uses special disabbreviation methods which are based on commonly used abbreviations and certain disabbreviation rules. The tool uses also knowledge about previous disabbreviations when it deduces new disabbreviations. i.e., when it tries to find possible replacements for an unknown word. The tool is thus able to become more clever as it is being used. The disabbreviation methods and process are explained in Section 2 of Paper IV.

Section 3 of the paper describes experiments done with the disabbreviator tool. Both source programs and other kinds of technical texts were used as test material. It seems that disabbreviating the names in compiled source programs is somewhat easier than disabbreviating other kinds of technical documents, because the names in source programs are already found "correctly spelled" in compilation. The results of the experiments with various texts indicate that the performance of the tool is appropriate in most cases. For instance, for more than half of the unknown names in source programs the tool could propose acceptable substitutions.

One of the experiments described in Paper IV is an understandability test in which students had to respond to questions about examples of source programs. Half of the students were studying source programs containing abbreviated names. The other half of the students studied the same source programs which had natural names created with the disabbreviator tool. Both groups had to respond to the same questions. The performance of the student group which studied naturally named programs was significantly better when compared to the other group which studied the same programs containing abbreviated names.

5.5 PAPER V: DISABBREVIATION OF SOURCE PROGRAMS

Paper V is continuation of the work discussed in Paper IV. Because source programs are rather special kinds of technical texts, their proper disabbreviation requires a tool which is tailored for disabbreviating source programs written with a certain programming language. For this reason, another experimental disabbreviation tool was built. This tool, which is partly based on the general disabbreviation tool introduced in Paper IV, is discussed in Paper V. The reasons for having a special tool to disabbreviate names in existing source programs can be summarized as follows:

- When a tool disabbreviates names in source programs, it must be able to treat each name as a unique whole. Therefore, it must be able to first decompose a name into separate words according to commonly used naming conventions, then disabbreviate the distinct words, and finally put the words together to make a replacement for the original unknown name.

- The terminology used in source programs is unique. For this reason, the dictionary and other data structures of a disabbreviation tool should contain words which are typically used in programming.

- Source programs are disabbreviated to make them more maintainable. A disabbreviation tool for source programs should, therefore, be able to work in harmony with other software maintenance tools.

- Source programs usually contain natural words in comments. The information in comments should be exploited in the disabbreviation methods.

Section 3 of Paper V introduces a disabbreviation tool called *InName* to disabbreviate names in C source programs. C is a rather popular programming language these days and there exist many applications written in C which are currently being maintained. Although the InName tool is specifically tailored to process C programs, the disabbreviation methods are largely independent of the programming language in use because the names are first extracted from the source programs and then they are disabbreviated in a separate phase. The following features of the InName tool are discussed in Section 3 of Paper V:

- The tool uses a special grammar to decompose names found in source programs into separate words. The grammar finds separate words when underscores or capital letters are used as word separators. For example, the names "prev_pos" and "DispBuff" are decomposed into words "prev", "pos", "Disp", and "Buff", which are then compared to the words in the stored dictionaries.

- The general dictionary of the InName tool is rather small, containing only about 1300 words commonly found in names in source programs. By analyzing the names in existing source programs, we have discovered that the number of words needed in names of a software system is usually much less than one thousand words. A disabbreviation tool is faster and produces less silly disabbreviations when its general dictionary is small.

- Disabbreviation methods include using lists of commonly used abbreviations and their natural counterparts, deducing possible disabbreviations from user-given name substitutions, and testing whether word combinations found in comments can be used as replacements for unknown names. For example, the tool is able to make the following suggestions:

Unknown names:	Suggested replacements:
`tmpnamelen`	`temporary_name_length`
`currwinheight`	`current_window_height`

  when its internal abbreviation lists include the information that the strings "tmp", "len", "curr", and "win" are commonly used abbreviations for "temporary", "length", "current", and "window", respectively. When a source program contains a variable declaration such as:

```
int  nbyte ;  /* number of bytes in buffer */
```

the tool will suggest that the name "nbytes" should be replaced with the name "number_of_bytes_in_buffer".

The third section of Paper V also discusses the phases of the disabbreviation process and the graphical user interface of the InName tool. The three phases of the disabbreviation process could be called the name inspection phase, the interactive disabbreviation phase, and the source program modification phase. The graphical user interface is used in the interactive disabbreviation phase, when the tool displays the source program to the user together with unknown names and possible substitutions.

The InName tool has been evaluated by using it to disabbreviate the source programs of several existing applications. One of the test-case applications is being developed further after being disabbreviated. In the case of each application the tool was used by a different person. The results of the tests with different applications are presented in Section 4 of Paper V. About 40% of the name substitutions suggested by the tool were acceptable in the tests. Learning to use the tool does not require much effort, and one application can be disabbreviated within a few days.

5.6 PAPER VI: AN EMPIRICAL STUDY OF THE USE OF NATURAL NAMING

The natural naming principles introduced in Paper I have been taught in several courses. A naming course has usually been combined with a general programming style course in which such issues as appropriate program module layouts, indentation practices, and formulation of different types of program statements have been taught. Also, a naming handbook has been compiled on the basis of the principles explained in Paper I. The naming handbook contains several detailed examples and gives advice on how to arrange long names in programs.

During a five-year period, about fifty people have been given a naming course and at least as many have been given a naming handbook. The software developers taught to use the natural naming principles come from different organizations and have various backgrounds. Courses have been given and handbooks delivered to university students, several groups of software developers in industrial organizations, and software developers in a research institute.

To assess the usefulness of the natural naming approach and to explore the viewpoints of software practitioners, several empirical investigations have been carried out. Some of the software developer groups who have been given a naming course or who have studied the naming handbook have been questioned by presenting them a form on which they have had to respond to questions. Paper VI presents the results from one such empirical study.

The second section of Paper VI discusses related work on naming and the difficulties in measuring the effects of different naming styles. Section 3 briefly explains what is included in the naming handbook, what has been taught in the naming courses, and how the questioning of the groups was arranged in practice. Altogether 52 software developers responded to the inquiries. One group of respondents was classified as less experienced software developers from industry, two groups represented experienced developers from industry, and one group was software developers from a research institute.

The inquiry forms presented to the respondents contain 25 statements, such as "The time required to write long names slows down software development". The subjects had to judge the relevance of each statement by answering "completely disagree", "partially disagree", "no opinion", "partially agree", or "completely agree". All the statements have been listed in a table, together with statistical data of the responses. Some of the statements deal with practical matters, some with communication and learning during the process of software development, some with understandability of programs, some with problem solving through judging suitable names, and the final two statements deal with typing.

The fourth section of the paper contains an analysis of the responses given to the inquiries. The responses are analyzed both by combining all responses together and by identifying different groups of respondents. In addition to the natural organizational division of groups, specific groups are formed by combining those respondents who reacted to a certain question in the same way.

The responses are analyzed from several points of view and many observations are made, including observations related to program understanding and communication, observations on the thinking process during programming, observations on practical matters, and observations related to writing of programs. In addition, different respondent groups are compared to each other. The following are the most interesting results of the analysis:

- The natural naming approach can be considered useful in software development. We could find nothing that would prevent us from recommending the use of the natural naming in practical work.

- Experienced software developers in industrial organizations were more enthusiastic about natural naming than less experienced developers or the software developers at the research institute.

- Compared to using abbreviations, the respondents believe that using natural names facilitates their thinking process. Trying to invent descriptive names is obviously an important means for problem analysis in software development.

- The understandability of programs is hard to assess, since the respondents did not give clear opinions whether natural naming facilitates communication or had improved the understandability of source programs.

The concluding section of Paper VI contains discussion about the implications of the results of the study. As the natural naming approach seems to be useful in practice, both software development organizations and the research community should focus more attention on the subject.

6 CONCLUDING DISCUSSION

6.1 RESEARCH SUMMARY AND EVALUATION

This thesis describes constructive research which involves some empiricism. The purpose has been to investigate how we could facilitate the use of natural naming in software development and maintenance. The research has resulted in guidelines, methods, and experimental tools. Now, the question is how well we have succeeded in this work.

The usual problem of software-engineering research also concerns this work, as we can only provide a soft evaluation of what has been done. It is usually difficult to provide hard and unambiguous evidence about the usefulness of software engineering methods and tools. Fenton (1993), Fenton et al. (1994), and Glass (1994) have noted that we have practically no hard evidence about how good commonly-used software engineering methods and tools are. We believe that the reason for this "software-research crisis" is not solely that researchers were too lazy and reluctant to provide practical evidence, but a partial reason is that it is hard to make quantitative measurements when software development and maintenance work is proceeding. If some measurements can be done in certain cases, it is difficult to be sure that a certain method or tool has caused the mentioned effect. Because of these reasons we have not tried to unambiguously show that natural naming is always useful in software development. The appropriateness of the natural naming approach is merely a hypothesis in this thesis, though we have produced some positive empirical evidence.

Naming in programming and the use of natural languages in software documentation are research subjects which have not attracted many researchers. To our knowledge, this thesis is the most extensive publication which deals with problems of naming in software documentation. Because these problems have not been widely addressed by others, we wanted to study them with a wide perspective, both in software development and maintenance. The breadth of the research area has been an advantage for this work. Because it is often difficult to explicitly separate software development from maintenance, it is better to study them both in the same research. We noted earlier that the reasons for using abbreviations are partly historical. Therefore, part of this work could have been performed during the time when the first high-level programming languages were invented, when there was only one wide computer research area. This also is a good reason for having a wide research area in this thesis.

The research reported in this thesis has been evaluated in the included papers. Below, we briefly discuss how well we have succeeded in each of the research activities from (a1) to (a5) which were defined in Section 3.1:

(a1) The naming principles (Paper I) which were developed based on the idea of natural naming have been received with much interest by the software developers who have participated in the naming courses, and who have been equipped with a naming handbook. Therefore, we consider to have succeeded well in this research activity. In one company, for example, the name tables presented in Paper I have been hung on the walls close to the desks of the software developers.

(a2) Paper II introduces a method to create a name repository prior to the actual software development process, and thereby to assure that all software documents would contain the same names for the same concepts. The most significant contribution of Paper II is that it explicitly shows that pre-development name creation is possible, and the created names can be used in software documents. Name creation is also an analytical process which provokes the name creator to consider the problems of the application domain. The name creation process can thus be perceived as a domain analysis task.

The DNN method presented in Paper II has not been tested extensively. The method is introduced in the context of one application domain, that of library automation. We have tried to use the DNN method in several other application domains, but it has not always been successful. Fyson (1995) reports similar experiences while working with the method. To make the method more applicable over a wider range of application domains, it should be refined in some way. It might be possible to enlarge the name tables which are used during the first steps of the name creation process. However, because applying the DNN method is not time consuming, it can always be tried in cases of new application domains. If it does not seem to produce good results, other ways for terminology control should be considered.

(a3) With the programming language presented in Paper III, we have shown that there are possibilities of improving existing programming languages by making them friendlier towards natural naming. The advantages and weaknesses of the new language are discussed in Paper III. However, it is hard to prove that one programming language is better than another language. Sammet (1972) notes that the reasons why some programming languages become popular are not always technical or scientific. The history of programming languages has examples of this. Algol is considered a scientifically excellent language, but it never became widely used in practice. COBOL has not been favored by many scientists (Shneiderman 1985), but it is one of the most widely used languages in the world. For these reasons, it is hard to say whether the Pacific programming language would become popular if it were developed as a commercial product.

Nevertheless, Pacific has features not found in other languages. The most unique feature in the Pacific language is its type system according to which type information is encoded in names. In most other programming languages, names are supposed to contain no information that can be exploited in compilation. Having type information encoded in names decreases redundant information in source programs.

(a4) Papers IV and V describe tools to convert abbreviated names into natural ones in existing source programs. The InName tool discussed in Paper V is specifically tailored for disabbreviating source programs. We consider to have succeeded well in finding ways to disabbreviate existing programs. The InName tool has been successfully used to process programs from various application domains. In a subjective evaluation, the usefulness of the tool can be considered obvious, although we cannot measure its performance explicitly. Using InName can also be seen as a way for learning a previously unfamiliar application that needs to be maintained.

(a5) This research activity aimed at finding more evidence of the usefulness of natural naming. We formally questioned software developers who had been given a handbook of natural naming, or who had attended a course on natural naming, and described the results in Paper VI.

We did achieve results in support of the use of natural naming. One of the interesting findings was that experienced software developers in industrial organizations were more enthusiastic about natural naming than younger software developers or people working in a research institute.

Considering the methodological approach, it is hard to consider that such an issue as naming could be profoundly studied without trying to construct something. When a researcher is trying to construct something (e.g. a programming language or a tool) which should support something else (e.g. natural naming), he or she learns more about that "something else". Doing constructive research allowed us to get more personal experience in the use of natural naming in practical software development work. While developing the Pacific compiler and the InName tool, we used natural naming in programming. We had only positive experiences in using it.

Because natural naming means using a natural language in software documentation, our research is related to other fields studying natural languages. These fields include linguistics, semiotics, and some areas of philosophy. As discussed in Chapter 4, we could not find anything in these fields which would cast doubt on the ideas presented in this thesis. In the field of technical communication we found support for the idea of avoiding abbreviations (Logsdon and Logsdon 1986, Ibrahim 1989). The fact that this work can be related to some philosophical works indicates the complexity of our research subject. Doing research related to natural languages is difficult, because we cannot study languages very objectively. To do that, we should perhaps step outside the domain of natural languages, or stop using natural languages which is humanly impossible. Even this thesis is written using a natural language. As we are interested in natural languages in this thesis, we could even say that we have described this research by using our research subject as a description tool.

6.2 POSSIBILITIES FOR FURTHER RESEARCH

More attention should be focused on the use of natural languages in software documentation. The software developers who were questioned in the inquiry described in Paper VI have the opinion that naming is a too much neglected issue in software documentation. A natural language is an essential component in most types of software documents (e.g. written documents, data flow diagrams, and object diagrams). When studying natural languages in software documentation, we should keep in mind that natural languages are not static, but dynamic. They are changing all the time (Fromkin and Rodman 1988). Technical development, including development and introduction of new software systems, is one reason why languages change. The most obvious way in which languages are changed by technical development is the emergence of new words (see Figure 14). Because technical development is accelerating rather than slowing down, we have to take care that our languages are developing in the right direction. Natural naming means taking care that the languages of software documentation do not contain too many abbreviations.

Setting technical and language development processes in parallel as in Figure 14 raises interesting philosophical questions. Did a microprocessor exist before or only after it was given the name "microprocessor"? Was the microprocessor invented at the moment when somebody called it for the first time by that name? How much of the development is actually language development? Although questions like these are difficult ones, they need to be considered, at least in the case of software development, which results in products that are merely conceptual than physical. A possible step towards this kind of research would be to include the language change into the theories about mental mechanisms of software development (Curtis 1985, Hoc et al. 1990, Detienne and Soloway 1990).

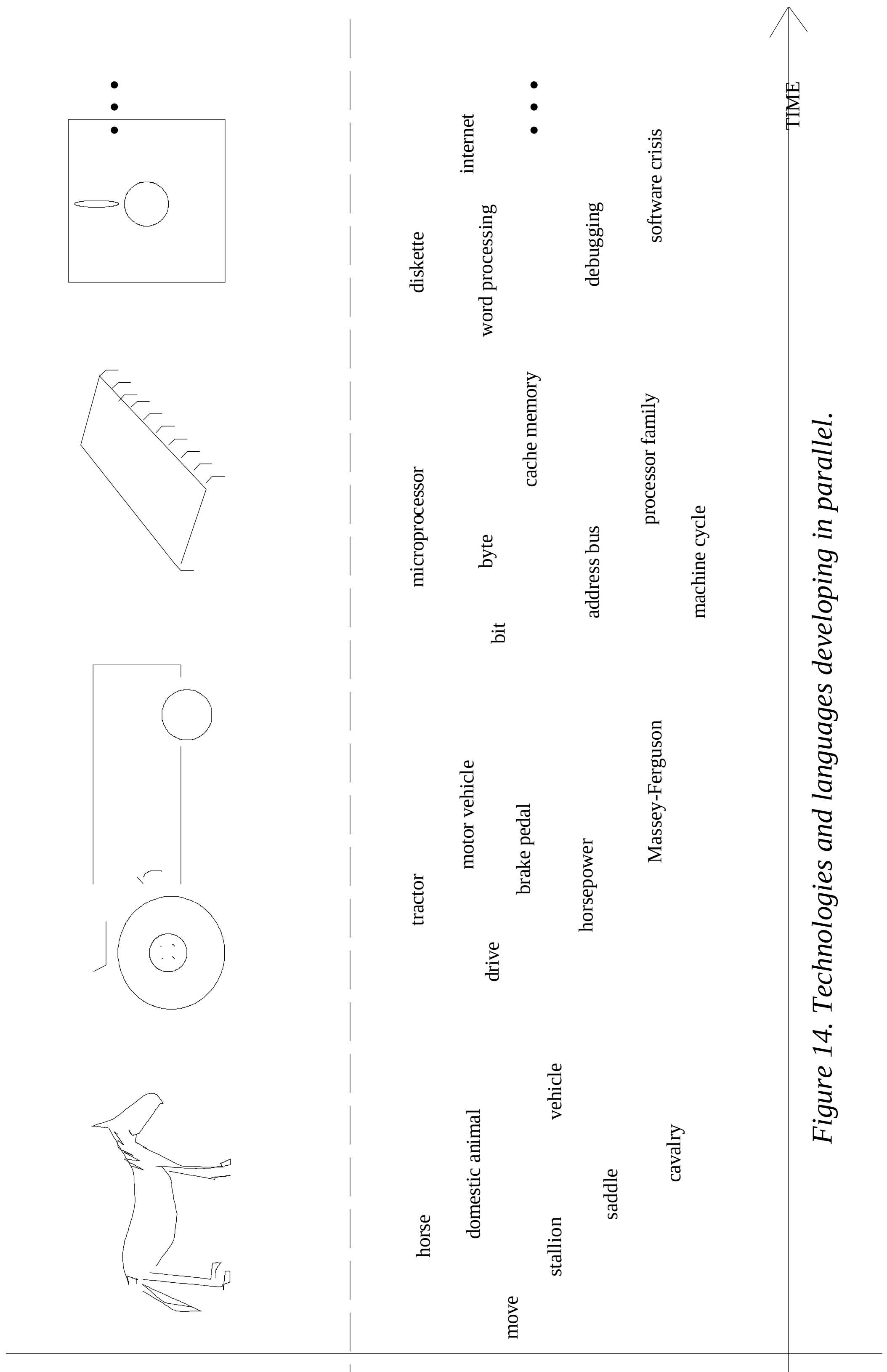


Figure 14. Technologies and languages developing in parallel.

TECHNICAL DEVELOPMENT:

New technical innovations emerging into the world

LANGUAGE DEVELOPMENT:

New words and word combinations emerging into natural languages

Because this thesis has focused mostly on conventional procedural programming, it could be fruitful to study natural naming with other styles of software. As object-orientation is gaining popularity these days, natural naming could be studied in object-oriented programming which may need some specific naming principles. We believe, however, that it is not a problem to use this naming approach in the context of other programming paradigms, because we applied natural naming in logic programming with Prolog while developing the InName tool.

One way to continue research related to natural naming is to use naturally named programs in teaching. This is what the author of this thesis is going to do. Although it is hard to measure how effectively people learn, it is important to try to investigate the reactions of students when they are exposed to programs which contain more natural words than conventional examples of computer programs. Software development is also a learning and communication process (Curtis et al. 1988). The process is already active when software developers are learning the subjects of their profession while studying in schools, colleges, and universities.

REFERENCES

Abbott, R. J. 1983. Program Design by Informal English Descriptions. Communications of the ACM, Vol. 26, No. 11, pp. 882 - 894.

Ackerman, F. A., Buchwald, L. S. and Lewski, F. H. 1989. Software Inspections: An Effective Verification Process. IEEE Software, Vol. 6, No. 3, pp. 31 - 36.

Aho, A.V., Sethi, R., and Ullman, J. D. 1986. Compilers: Principles, Techniques, and Tools. Reading, Massachusetts: Addison-Wesley. 796 p.

Allen, J. 1987. Natural Language Understanding. Menlo Park, California: The Benjamin Cummings Publishing Company. 574 p.

AMES. 1993. ESPRIT III Project no. 8156: Application Management Environments and Support. Technical Annex. Grenoble, France: Cap Gemini Innovation. 122 p.

Anand, N. 1988. Clarify Function! ACM SIGPLAN Notices, Vol. 23, No. 6, pp. 69 - 79.

Andersen, P. B. 1990. A Theory of Computer Semiotics. Cambridge, United Kingdom: Cambridge University Press. 416 p.

Baecker, R. M. and Marcus, A. 1990. Human Factors and Typography for More Readable Programs. Reading, Massachusetts: Addison-Wesley. 348 p.

Balzer, R. 1985. A 15 Year Perspective on Automatic Programming, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp. 1257 - 1268.

Barnett, M. P. 1969. Computer Programming in English. New York: Harcourt, Brace & World, Inc. 260 p.

Bennett, K., Cornelius, B., Munro, M., and Robson, D. 1991. Software Maintenance. In: McDermid, J. A. (ed.) Software Engineer's Reference Book. Oxford, United Kingdom: Butterworth-Heinemann. Chapter 20. 18 p.

Biggerstaff, T. J. 1989. Design Recovery for Maintenance and Reuse. Computer, Vol. 22, No. 7, 36 - 49.

Boehm, B. W. 1988. A Spiral Model of Software Development and Enhancement. Computer, Vol. 21, No. 5, pp. 61 - 72.

Boldyreff, C., Elzer, P., Hall, P., Kaaber, U., Keilmann, J., and Witt, J. 1990. PRACTITIONER: Pragmatic Support for the Reuse of Concepts in Existing Software. Proceedings of Software Engineering '90 (SE90). Cambridge, United Kingdom: Cambridge University Press. Pp. 574 - 591.

Boldyreff, C., Burd, E. L., and Hather, R. M. 1994. An Evaluation of the State of the Art for Application Management. Proceedings of the International Conference on Software Maintenance. Los Alamitos, California: IEEE Computer Society Press. Pp. 161 - 169.

Booch, G. 1991. Object-Oriented Design with Applications. Menlo Park, California: The Benjamin Cummings Publishing Company. 565 p.

Bourne, C. P. and Ford, D. F. 1961. A Study of Methods for Systematically Abbreviating English Words and Names. Journal of the ACM, Vol. 8, pp. 538 - 552.

Brooks, F. P. 1987. No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, Vol. 20, No. 4, pp. 10 - 19.

Brooks, R. 1978. Using a Behavioral Theory of Program Comprehension in Software Engineering. Proceedings of 3th International Conference on Software Engineering. Los Alamitos, California: IEEE Computer Society Press. Pp. 196 - 201. Also in (Parikh and Zvegintzov 1983. Pp. 109 - 114).

Brooks, R. 1983. Towards a Theory of the Comprehension of Computer Programs. International Journal of Man-Machine Studies, Vol. 18, No. 6, pp. 543 - 554.

Caine, S. H. and Gordon, E. K. 1975. PDL -- A Tool for Software Design. In: Freeman, P. and Wasserman, A. I. (eds.) 1983. Tutorial on Software Design Techniques. 4th ed. Los Alamitos, California: IEEE Computer Society Press. Pp. 485 - 490.

Carroll, J. B. (ed.) 1956. Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf. Cambridge, Massachusetts: The M.I.T. Press. 278 p.

Carter, B. 1982. On Choosing Identifiers. ACM SIGPLAN Notices, Vol. 17, No. 5, pp. 54 - 59.

Chikofsky, E. J. and Cross, J. H. 1990. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, Vol. 7, No. 1, pp. 13 - 17.

Ching, E. 1983. Problems Caused by the Neologism in Teaching Chinese. Annual Meeting of the American Council on the Teaching of Foreign Language. San Francisco, California, November 24 - 26, 1983. 12 p.

Coad, P. and Yourdon, E. 1990. Object Oriented Analysis. Englewood Cliffs, New Jersey: Prentice-Hall. 223 p.

Cordes, D. and Brown, M. 1991. The Literate-Programming Paradigm. Computer, Vol. 24, No. 6, pp. 52 - 61.

CSTB (Computer Science Technology Board). 1990. Scaling Up: A Research Agenda for Software Engineering. Communications of the ACM, Vol. 33, No. 3, pp. 281 - 293.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., and Love, T. 1979. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, pp. 96 - 104.

Curtis, B. 1984. Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science. Proceedings of 7th International Conference on Software Engineering. Los Alamitos, California: IEEE Computer Society Press. Pp. 97 - 106.

Curtis, B. (ed.) 1985. Tutorial: Human Factors in Software Development. 2nd ed. Los Alamitos, California: IEEE Computer Society Press. 730 p.

Curtis, B., Krasner, H., and Iscoe, N. 1988. A Field Study of Software Design Process for Large Systems. Communications of the ACM, Vol. 31, No. 11, pp. 1268 - 1287.

Cusumano, M. A. 1989. The Software Factory: A Historical Interpretation. IEEE Software, Vol. 6, No. 3, pp. 23 - 30.

Detienne, F. and Soloway, E. 1990. An Empirically-Derived Control Structure for the Process of Program Understanding. International Journal of Man-Machine Studies, Vol. 33, No. 3, pp. 323 - 342.

Eco, U. 1984. Semiotics and the Philosophy of Language. London: Macmillan Press. 241 p.

Eco, U. 1990. The Limits of Interpretation. Bloomington: Indiana University Press. 296 p.

ESA. 1991. PSS-05-0 Issue 2. ESA Software Engineering Standards, Issue 2. Noordwijk, The Netherlands: European Space Agency. 130 p.

Fenton, N. 1993. How Effective Are Software Engineering Methods? Journal of Systems and Software, Vol. 22, No. 2, pp. 141 - 146.

Fenton, N., Pfleeger, S. L., and Glass, R. L. 1994. Science and Substance: A Challenge to Software Engineers. IEEE Software, Vol. 11, No. 4, pp. 86 - 95.

Feyerabend, P. 1975. Against Method: Outline of an Anarchistic Theory of Knowledge. London: New Line Books. 339 p.

Fromkin, V. and Rodman, R. 1988. An Introduction to Language. 4th ed. New York: Holt, Rinehart and Winston, Inc. 460 p.

Fyson, J. 1995. An Investigation into Methods of Improving Program Comprehension through Better Documentation. Project Report. Durham, United Kingdom: University of Durham, Department of Computer Science. 78 p.

Gellenbeck, E. M. and Cook, C. R. 1991. Does Signaling Help Professional Programmers Read and Understand Computer Programs? Technical Report 91-60-3. Corvallis, Oregon: Oregon State University, Computer Science Department. 20 p.

Glass, R. L. 1994. The Software-Research Crisis. IEEE Software, Vol. 11, No. 6, pp. 42 - 47.

Green, T. R. G. 1990. The Nature of Programming. In: Hoc, J. M., Green, T. R. G., Samurcay, R., and Gilmore, D. J. (eds.) Psychology of Programming. London: Academic Press. Pp. 21 - 44.

Grogono, P. 1989. Comments, Assertions, and Pragmas. ACM SIGPLAN Notices, Vol. 24, No. 3, pp. 79 - 84.

Hakalahti, H., Lappalainen, P., and Tervonen, M. 1978. Minitietokoneet [Minicomputers]. Oulu, Finland: Sähköinsinöörikilta ry., Oulun Yliopisto. 431 p. (In Finnish.)

Hall-Quest, A. L. 1979. Abbreviations. In: Collier's Encyclopedia. Volume 1. New York: Macmillan. Pp. 13 - 14.

Harel, D. 1992. Biting the Silver Bullet, Toward a Brighter Future for System Development. IEEE Computer, Vol. 25, No. 1, pp. 8 - 20.

Hoc, J. M., Green, T. R. G., Samurcay, R. and Gilmore, D. J. (eds.) 1990. Psychology of Programming. London: Academic Press. 290 p.

Hodges, A. 1983. Alan Turing: The Enigma. New York: Simon and Schuster. 571 p.

Horowitz, E. (ed.) 1987. Programming Languages: A Grand Tour. 3rd ed. Rockville, Maryland: Computer Science Press. 583 p.

Ibrahim, A. M. 1989. Acronyms Observed. IEEE Transactions on Professional Communication, Vol. 32, No. 1, pp. 27 - 28.

ICSM. 1994. Proceedings of International Conference on Software Maintenance. Los Alamitos, California: IEEE Computer Society Press. 449 p.

Iivari, J. 1991. A Paradigmatic Analysis of Contemporary Schools of IS Development. European Journal of Information Systems, Vol. 1, No. 4, pp. 249 - 272.

Intel. 1979. MCS-80/85 Family User's Manual. Santa Clara, California: Intel Corporation.

ISO 9000-3. 1991. Quality Management and Quality Assurance Standards - Part 3: Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software. Geneva, Switzerland: International Organization for Standardization. 15 p.

Jackson, M. 1994. Problems, Methods, and Specialization. Software Engineering Journal, Vol. 9, No. 6, pp. 249 - 255. (A slightly condensed version of the paper is in: IEEE Software, Vol. 11, No. 6, 1994, pp. 57 - 62.)

Jarman, D. 1993. Wittgenstein (motion picture). London: Channel Four and British Film Institute. 75 minutes.

Johnson, W. L. 1987. Some Comments on Coding Practice. ACM SIGSOFT Software Engineering Notes, Vol. 12, No. 2, pp. 32 - 35.

Jokela, T. 1991. A Modeling Method for Early Validation of Embedded Systems. Licentiate Thesis. Oulu, Finland: University of Oulu, Department of Electrical Engineering. 80 p.

Järvinen, P. and Järvinen, A. 1993. Tutkimustyön metodeista [On Methods in Research Work]. Tampere, Finland: University of Tampere, Department of Information Processing Science. 121 p. (In Finnish with English Abstract.)

Kaelbling, M. J. 1988. Programming Languages Should NOT Have Comment Statements. ACM SIGPLAN Notices, Vol. 23, No. 10, pp. 59 - 60.

Kauranen, I., Ropponen, P., and Aaltonen, M. 1993. Tutkimusraportin kirjoittamisen opas [A Guide for Writing Research Reports]. Espoo, Finland: Helsinki University of Technology. 113 p. (In Finnish.)

Keller, D. A. 1990. Guide to Natural Naming. ACM SIGPLAN Notices, Vol. 25, No. 5, pp. 95 - 102.

Kerola, P. and Freeman, P. 1981. A Comparison of Lifecycle Models. Proceedings of the 5th International Conference on Software Engineering. Los Alamitos, California: IEEE Computer Society Press. Pp. 90 - 99.

Knuth, D. E. 1984. Literate Programming, The Computer Journal, Vol.27, No. 2, 1984, pp. 97 - 111.

Kuhn, T. S. 1962. The Structure of Scientific Revolutions. International Encyclopedia of Unified Science. Volume II. Number 2. Chicago, Illinois: The University of Chicago Press. 172 p.

Laitinen, K. 1992. Document Classification for Software Quality Systems. ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4, pp. 32 - 39.

Laitinen, K. and Taramaa, J. 1994. A Theory to Support the Use of Natural Naming in Software Documentation. Working papers series B33. Oulu, Finland: University of Oulu, Department of Information Processing Science. 27 p. ISBN 951-42-3967-9.

Laitinen, K. 1995. Estimating Understandability of Software Documents. Working Paper. Oulu, Finland: VTT Electronics. 12 p. To appear in ACM SIGSOFT Software Engineering Notes in January or April 1996.

Ledgard, H., Whiteside, J. A., Singer, A., and Seymour, W. 1980. The Natural Language of Interactive Systems. Communications of the ACM, Vol. 23, No. 10, pp. 556 - 563.

Ledgard, H. and Tauer, J. 1987. Professional Software. Volume II. Programming Practice. Reading, Massachusetts: Addison-Wesley. 220 p.

Logsdon, D. and Logsdon, T. 1986. The Curse of the Acronym. In: Proceedings of the International Professional Communications Conference. Washington D. C: IEEE. Pp. 145 - 152.

MacLennan, B. J. 1983. Principles of Programming Languages: Design, Evaluation, and Implementation. New York: Holt, Rinehart and Winston. 544 p.

Marca, D. 1981. Some Pascal Style Guidelines. ACM SIGPLAN Notices, Vol. 16, No. 4, pp. 70 - 80.

Matsumoto, Y. 1987. A Software Factory: An Overall Approach to Software Production. In: Freeman, P. (ed.) Software Reusablity. Los Alamitos, California: IEEE Computer Society Press. Pp. 155 - 178.

Nelson, R. J. 1992. Naming and Reference. London: Routledge. 297 p.

Newsted, P. R. 1979. Flowchart-Free Approach to Documentation. Journal of Systems Management, Vol. 30, No. 4, pp. 18 - 21.

Oman, P. W. and Cook, C. R. 1991. A Programming Style Taxonomy. The Journal of Systems and Software, Vol. 15, No. 3, pp. 287 - 301.

Page-Jones, M. 1988. The Practical Guide to Structured Systems Design. 2nd ed. Englewood Cliffs, New Jersey: Prentice Hall. 249 p.

Parikh, G. and Zvegintzov, N. (eds.) 1983. Tutorial on Software Maintenance. Los Angeles, California: IEEE Computer Society Press. 359 p.

Parnas, D. L. and Clements P. C. 1986. A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 251 - 257.

Plum, T. 1984. C Programming Guidelines. Englewood Cliffs, New Jersey: Prentice Hall. 146 p.

Potts, C. 1993. Software-Engineering Research Revisited. IEEE Software, Vol. 10, No. 5, pp. 19 - 28.

Prieto-Diaz, R. and Arango, G. 1991. Domain Analysis and Software System Modeling. Los Alamitos, California: IEEE Computer Society Press. 312 p.

Raghavan, S. A. and Chand, D. R. 1989. Diffusing Software-Engineering Methods. IEEE Software, Vol. 6, No. 4, pp. 81 - 90.

ReaGeniX. 1994. ReaGeniX: Real-Time Application Generator - User's Manual. Oulu, Finland: VTT Electronics. 32 p.

Rowe, N. C. 1985. Naming in Programming. Computers in Schools, Vol. 2, No. 2 - 3, pp. 241 - 253.

Rowe, N. C. and Guglielmo, E. J. 1993. Exploiting Captions in Retrieval of Multimedia Data. Information Processing and Management, Vol. 29, No. 4, pp. 453 - 461.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. Object Oriented Modeling and Design. Englewood Cliffs, New Jersey: Prentice Hall. 538 p.

Saeki, M., Horai, H., and Enomoto, H. 1989. Software Development Process from Natural Language Specification. Proceedings of the 11th International Conference on Software Engineering. Los Alamitos, California: IEEE Computer Society Press. Pp. 64 - 73.

Sammet, J. E. 1966. The Use of English as a Programming Language. Communications of the ACM, Vol. 9, No. 3, pp. 228 - 230.

Sammet, J. E. 1972. Programming Languages: History and Future. Communications of the ACM, Vol. 15, No. 7, pp. 601 - 610.

Sammet, J. E. 1981. The Early History of COBOL. In: Wexelblat, R. L. (ed.) History of Programming Languages. London: Academic Press. Pp. 199 - 243.

Seppänen, V. 1990. Acquisition and Reuse of Knowledge to Design Embedded Software. VTT Publications 66. Espoo, Finland: Technical Research Centre of Finland (VTT). 216 p. + app. 10p.

Sheppard, S. B., Curtis, B., Milliman, P., and Love, T. 1979. Modern Coding Practices and Programmer Performance. Computer, Vol. 12, No. 12, pp. 41 - 49.

Shneiderman, B. 1980. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, Massachusetts: Winthrop Publishers. 320 p.

Shneiderman. B. 1985. The Relationship between COBOL and Computer Science. In: Horowitz, E. (ed.) Programming Languages: A Grand Tour. 3rd ed. Rockville, Maryland: Computer Science Press. Pp. 417 - 421. Also in: Annals of the History of Computing, Vol. 7, No. 4. Reston, Virginia: AFIPS.

Smeaton, A. F. 1992. Progress in the Application of Natural Language Processing to Information Retrieval Tasks. The Computer Journal, Vol. 35, No. 3, pp. 268 - 278.

Smith, M. and Taffler, R. 1992. Readability and Understandability: Different Measures of the Textual Complexity of Accounting Narrative. Accounting & Accountability Journal, Vol. 5, No. 4, pp. 84 - 98.

Soloway, E. and Ehrlich, K. 1984. Empirical Studies of Programming Knowledge. IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 595 - 609. Also in (Curtis 1985).

Soloway, E. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. Communications of the ACM, Vol. 29, No. 9, pp. 850 - 858.

Sowa, J. F. 1990. Finding Structure in Knowledge Soup. In: Part II of Proceedings of InfoJapan'90 International Conference. Tokyo: Information Processing Society of Japan. Pp. 245 - 253.

Suitiala, R. 1993. Work-Oriented Development of Interactive Software Tools. Understanding the Work of Software Maintainers and Making an Interactive Tool for Them. VTT Publications 139. Espoo, Finland: Technical Research Centre of Finland (VTT). 176 p. + app. 10 p.

Swartout, W. and Balzer, R. 1982. On the Inevitable Intertwining of Specification and Implementation. Communications of the ACM, Vol. 25, No. 7, pp. 438 - 440.

Taramaa, J. and Oivo, M. 1993. Evaluation of Software Maintenance of Embedded Computer Systems. Proceedings of International Symposium on Engineered Software Systems. Singapore: World Scientific Publishing Co. Pp. 193 - 203.

Tarasti, E. 1990. Johdatusta semiotiikkaan [An Introduction to Semiotics].

Helsinki, Finland: Gaudeamus. 317 p. (In Finnish.)

Tausworthe, R. C. 1992. Information Models of Software Productivity: Limits on Productivity Growth. Journal of Systems and Software, Vol. 19, No. 2, pp. 185 - 201.

Teasley, B. E. 1994. The Effects of Naming Style and Expertise on Program Comprehension. International Journal of Human-Computer Studies, Vol. 40, No. 5, pp. 757 - 770.

Tichy, W. F., Habermann, N., and Prechelt, L. 1993. ACM SIGSOFT Software Engineering Notes, Vol. 18, No. 1, pp. 35 - 48.

Turing, A. M. 1937a. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, Ser. 2, Vol. 42, pp. 230 - 265. Reprinted in: Davis M. (ed.) 1965. The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Hewlett, New York: Raven Press. Pp. 116 - 151.

Turing, A. M. 1937b. On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction. Proceedings of the London Mathematical Society, Ser. 2, Vol. 43, pp. 544 - 546. Reprinted in: Davis M. (ed.) 1965. The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions. Hewlett, New York: Raven Press. Pp. 152 - 154.

Ward, P. T. and Mellor, S. J. 1985. Structured Development for Real-Time Systems, Vol. 1-3. New York: Yourdon Press. 509 p.

Webster's. 1989. Webster's Dictionary of the English Language. New York: Lexicon Publications. 1149 p.

Weinberg, G. M. 1971. The Psychology of Computer Programming. New York: Van Nostrand Reinhold Company. 288 p.

Weissman, L. M. 1974. A Methodology for Studying the Psychological Complexity of Computer Programs. Ph.D. Thesis. Toronto: University of Toronto, Department of Computer Science. 231 p.

Welsh, J. and Han, J. 1994. Software Documents: Concepts and Tools. Software -- Concepts and Tools, Vol. 15, No. 1, pp. 12 - 25.

Wittgenstein, L. 1921. Tractatus Logico Philosophicus. London: Routledge. 207 p.

Wittgenstein, L. 1953. Philosophical investigations. Oxford, United Kingdom: Basil Blackwell. 250 p.

WPC. 1993. Proceedings of the Second Workshop on Program Comprehension. Los Alamitos, California: IEEE Computer Society Press. 193 p.

Yngve, V. H. 1986. Linguistics as a Science. Indianapolis: Indiana University Press. 120 p.

Yonezaki, N. 1989. Natural Language Interface for Requirements Specification. In: Matsumoto, Y. and Ohno, Y. (eds.) Japanese Perspectives in Software Engineering. Singapore: Addison-Wesley. Pp. 41 - 76.

Yourdon, E. 1989. Modern Structured Analysis. Englewood Cliffs, New Jersey: Prentice-Hall. 717 p.

Zemanek, H. 1974. Formalization: Past, Present, and Future. In: Shaw, B. (ed.) Formal Aspects of Computing Science: Proceedings of Joint IBM University of Newcastle upon Tyne Seminar. Pp. 177 - 197. Also in: Lecture Notes in Computer Science 23: Programming Methodology. Berlin: Springer-Verlag, 1975.