This paper has appeared in ACM Sigsoft Software Engineering Notes, Vol. 21, No. 4, 1996, pp. 81-92.
Figures 2(a) and 2(b) are at the end of the paper in this reconstructed edition. Addresses are not valid any more.

Estimating Understandability of Software Documents

Kari Laitinen

VTT Electronics / Embedded Software
P.O. Box 1100, 90571 Oulu, Finland
Email: Kari.Laitinen@vtt.fi

Abstract

Software developers and maintainers need to read and understand source programs and other kinds of software documents in their work. Understandability of software documents is thus important. This paper introduces a method for estimating the understandability of software documents. The method is based on a language theory according to which every software document is considered to contain a language of its own, which is a set of symbols. The understandability of documents written according to different documentation practices can be compared using the rules of the language theory. The method and the language theory are presented by using source programs with different naming styles as example documents. The method can, at least theoretically, be applied to any kind of document. It can also be used to explain the benefits of some well-known software design methods.

1 Introduction

Software development is a process during which software developers have to communicate with each other as well as with various interest groups, such as sales people, customers, and end users. Curtis et al. (1988) have noted that software development should, at least partly, be treated as a communication and learning process. Communication can take place through spoken words, but formal communication in software development is most often carried out through various types of software documents. The quality and understandability of software documentation thereby affect the success of software development.

Because documents are used in communication and learning, software development can be treated as a documentation process during which more and more elaborate documents emerge as the development proceeds. This approach has been discussed by Laitinen (1992) and Welsh and Han (1994). In a typical software development project documents such as requirements definitions are developed first, and they are followed by various design documents which give a more detailed description of the system being developed. The final documents of software development are source programs which describe every possible detail of the system, and from which the executable system can be generated using compilers and other tools. Nowadays there are development systems which generate source programs from higher-level design descriptions (e.g. ReaGeniX, 1994). In these cases, the design descriptions which are fed into the program generator are the final documents of software development.

It is important to try to estimate the effectiveness of software development practices (Fenton, 1993). When we treat software development as a documentation process, we should have means for estimating the understandability of software documents in order to compare different documentation practices. For this reason, we shall introduce a method with which we can compare the understandability of differently written software documents. The method is based on identifying distinct symbols in software documents. A documentation practice which results in avoiding unnecessary symbols is considered more appropriate than others.

The method is particularly suitable for estimating the understandability of source programs. Therefore, we will largely concentrate on source programs in this paper. The understandability of source programs is important because they are usually the most complex software documents. Moreover, they are often the only reliable documents that describe a system in maintenance situations (Bennett et al. 1991). Although we use source programs as example documents, we will also explain that the method can be used with other software documents as well.

```
is_string_a_palindrome(  char  given_string[],
                         int   *string_inspection_result )

{
   int  string_start_index,  string_end_index   ;
   int  string_length  ;

   string_length  =  strlen( given_string ) ;

   string_start_index         =  0 ;
   string_end_index           =  string_length - 1 ;
   *string_inspection_result  =  STRING_INSPECTION_NOT_READY ;

   while ( *string_inspection_result == STRING_INSPECTION_NOT_READY )
   {
      if ( given_string[ string_start_index ] ==
                                 given_string[ string_end_index ] )
      {
         if ( string_start_index  ==  string_length - 1 )
         {
            *string_inspection_result  =  STRING_IS_A_PALINDROME ;
         }
         else
         {
            string_start_index  ++  ;
            string_end_index    --  ;
         }
      }
      else
      {
         *string_inspection_result  =  STRING_IS_NOT_A_PALINDROME ;
      }
   }
}
```

VERSION (a)

```
ispdrome( char str[], int *presult )

{
    int  i,j,len ;

    len = strlen( str ) ;

    i = 0 ;
    j = len - 1 ;
    *presult = NREADY ;

    while ( *presult == NREADY )
    {
       if ( str[ i ] == str[ j ] )
       {
          if ( i == len - 1 )
          {
             *presult = YES ;
          }
          else
          {
             i ++ ;
             j -- ;
          }
       }
       else
       {
          *presult = NO ;
       }
    }
}
```

VERSION (b)

Figure 1. The same program with natural names (a) and with abbreviated names (b).

Names for variables, constants, tables, functions, etc. are important symbols in source programs. As we can see in Figure 1, different naming styles can significantly affect the visual appearance of programs. Therefore, we have reasons to believe that naming affects their understandability as well. Psychological tests with small programs have been carried out to test how naming affects understandability (Weissman, 1974; Sheppard et al., 1979; Shneiderman, 1980; Teasley, 1994). These tests suggest that the use of mnemonic names enhances the understandability of source programs, but the results have not always been statistically convincing. For this reason, and also because real software systems with hundreds of source program files cannot be examined in short classroom tests, we need other methods for estimating understandability.

We will use naturally named source programs in our case studies. Natural naming means that all names in source programs should be constructed using, preferably several, natural words of a natural language, while respecting the grammatical rules of the natural language. The natural names must also describe the functionality of the program. Guidelines for using natural naming have been published by Keller (1990), Laitinen and Seppänen (1990), and Laitinen and Mukari (1992). As shown in Figure 1, natural naming means that abbreviations are avoided. The use of abbreviations has been criticized by Logsdon and Logsdon (1986) and Ibrahim (1989). Intuitively, it is obvious that programs with natural names are more understandable than those with abbreviated names, but we need more proofs for this matter.

There are many methods for estimating complexity[1] of source programs. Some methods are surveyed by Weiser and Shneiderman (1987). None of these methods, however, is very close to our estimation method, which is a rather general one. There are methods which require that some people participate as test subjects when the understandability of documents is assessed (Hall and Zweben, 1986). Our method can, at least theoretically, be used with any software document without doing any modifications to the document.

[1]Complexity is inversely proportional to understandability, i.e., when we can increase the understandability of a document, its complexity decreases.

Table 1. Examples of symbols in software documents.

SYMBOL CATEGORY	EXAMPLES
TECHNICAL SYMBOLS (Graphical symbols)	The original paper has bubbles and arrows in this table cell. They were drawn by hand to the original paper.
TECHNICAL SYMBOLS (Character-based symbols)	= == >> << < > != * _ - /* */ , . %s %d ! & && #define .GT. .EQ. GO TO CONTINUE if else while int char struct
TEXTUAL SYMBOLS	1 4 5 99 256 p2 ptr2 s1 i j k ptr chk numb buff len file counter result record message buffer number of bytes this program is used to

The core idea in our method is that every software document is seen to contain a unique language, which consists of the technical and textual symbols of the document. To study documents in this way, we present a special language theory in the second section of the paper. In the third section we estimate the understandability of some example programs, and in the fourth section we discuss our studies concerning the source programs of real software systems. In the last section we discuss some linguistic issues and explain how software engineering methods can be evaluated using our theory.

2 Redefining the word "language"

Many languages are used in software development: natural languages (e.g. English), programming languages (e.g. C, C++, COBOL, and Ada), and various modeling languages which belong to software development methods (e.g. Yourdon, 1989; Coad and Yourdon, 1990). Programming and modeling languages use special symbols, but they also include some elements of natural languages, such as single words used as reserved words and various word combinations used as symbolic names. We can divide the symbols of software documentation into technical and textual symbols. Technical symbols can be further divided into character-based and graphical symbols. Examples of these kinds of symbols are given in Table 1.

Graphical technical symbols include the ones which are used in the context of modeling methods. Character-based technical symbols are single characters or combinations of characters which have a special meaning in the context of a modeling or programming language. In the case of a programming language, technical symbols have a clearly defined meaning for the compiler of the language. Textual symbols are natural words, combinations of letters, numbers, or combinations of letters and numbers. Examples of textual symbols are names for variables in source programs and names for data flows in data-flow diagrams. In this paragraph nearly all symbols are textual symbols.

Dividing the symbols of software documentation into the mentioned categories may not always be indisputable, but this division is sufficient for studying the textual complexity of software documents. It is namely the category "textual symbols" which greatly affects the understandability of documents, and we

consider that we can distinguish this category sufficiently well from the other symbol categories. Textual symbols are usually chosen or invented by software developers, whereas technical symbols are dictated by the used software development methods and tools.

The words of natural languages are essential "components" in software documents. However, we cannot say that software documents would be written using natural languages because they also contain technical symbols which do not belong to such languages. On the other hand, if we study such software documents as source programs from the viewpoint of understandability, we cannot say that they would be written entirely with a programming language, because they contain textual symbols which cannot be considered to belong to programming languages. To clarify this matter, let us study the simple C function named "ispdrome" in Figure 1(b). We cannot say that the symbol "ispdrome" belongs to the C programming language because, if it belonged to it, it would be explained in every textbook of C. Neither can we say that the symbol "ispdrome" belongs to some natural language, because we cannot find that symbol being explained in any dictionary. Therefore, we have difficulty in identifying into which language the name "ispdrome" belongs. We have the same difficulty with all other names in Figure 1(b) and with many symbols in many other source programs and other software documents.

We can overcome this difficulty by giving a new definition for the word "language". According to our new definition for languages, there are not solely natural languages, programming languages, and other modeling languages in the world, but there are hybrid languages which contain symbols from natural languages, programming languages, and modeling languages, as well as symbols which are invented for a particular purpose. For example, the name "ispdrome" in Figure 1(b) is a symbol invented for that program, and it belongs to the language of that program.

The central issue here is that we consider a set of symbols a language. In this vein of thinking, every document, every set of documents, every person, a group of persons, etc. have their own languages. To discuss the "new" languages, we introduce the notation L[language domain] to refer to language. With the notation L[language domain] = { list of symbols } we can define languages. We define a language also by listing all its symbols inside a rectangle. Using these notations we give the following examples of languages:

- L[traffic lights] = { red yellow green } defines a language which could be used to study traffic lights.
- L[Figure 1(b)] is the language of the program in Figure 1(b). This language can be defined as follows:

L[Figure 1(b)] =

```
ispdrome ( char str [ ] , int
* presult ) { i j len ;
= strlen 0 - 1 NREADY while ==
if YES } else ++ -- NO
```

- L[This paper] is the language which contains all the symbols used in this paper. We do not list all those symbols here, but we want to emphasize that it would be possible to do it in an exact manner. L[Figure 1(b)] is a sublanguage of L[This paper].
- L[Small child] could be a language containing very few symbols like "mama" and "papa".

As we consider a defined set of symbols to be a language, we, on purpose, pay less attention to the syntax of the language. We consider that the syntax is embedded in every symbol, i.e., it is in the meaning of a symbol, how it can be combined with other symbols. Because we are interested in the understandability and complexity of software documents, we can pay less attention to syntax, since it is not the syntax but the special symbols which mostly affect the understandability of software documents. The syntax of a source program, for example, is checked through compilation. Therefore, software developers do not produce syntactically incorrect programs. Syntax is also an important part of natural languages (Fromkin and Rodman, 1988), but software developers are not, on purpose, inclined to break the syntactic rules when they use a natural language in software documentation. They may, however, include special symbols among known natural words. For these reasons, we need to focus more attention on symbols and their meanings, and less attention on the order of symbols (the syntax), when studying understandability.

According to our language theory, a given software document can be understood if one masters the language of the document in question. This means that one must figure out what the meanings of the symbols of that language are. For example, if someone would like to understand the program in Figure 1(b), he should learn to master language L[Figure 1(b)] which we introduced above. If the person were already competent in C, he would already have meanings for the C symbols (i.e. the technical symbols) of L[Figure 1(b)]. Therefore, the person's task would be to learn the following language:

```
L[Figure 1(b), textual symbols] =
```

```
ispdrome   str   presult  i  j  len

0  1  NREADY   YES   NO
```

Considering that a software development project produces a set of documents, the symbols used in the produced software documents form the language of that specific software system. The developers must learn to master that language while they are developing the system. The language of the system needs to be learned also by a new person who enters the software development team or who starts to maintain the system.

3 Examples of documents

We will use our theory of languages as sets of symbols to compare the understandability of some simple software documents. The understandability of a document is proportional to how understandable the language of the document is, and inversely proportional to how complex the language is. We define two rules to estimate the understandability of languages. To explain the rules, we use the following hypothetical simple languages:

L[1] = { A B C D }
L[2] = { K L M }
L[3] = { A B X }
L[4] = { C Y Z }

RULE I:

Smaller languages are usually easier to understand than larger languages. This means that the number of symbols of a language affects its understandability. Provided that a person would know none of the hypothetical languages above, this rule says that languages L[2], L[3], and L[4] are easier to understand than language L[1], which contains four symbols, while the other ones contain only three symbols.

RULE II:

It is easier to understand closely related languages than more distantly related languages. By referring to the hypothetical languages above, this can be explained as follows. If a person would master completely only the language L[1] above, then L[3] would be the easiest for him, because L[3] has only one different symbol when compared to L[1]. L[4] would be the next easiest to understand, because it contains one symbol which is also in L[1]. L[2] would be the most difficult to understand, because all symbols are different than those in L[1].

On the basis of these rules we are able to say which languages are more understandable and which less understandable in relation to some known language. We must point out that we cannot give any absolute measures for understandability. We always need a language to compare with, but that is not a difficult problem, since everybody knows some language.

Let us now study the two versions of the same program in Figure 1. Version (a) has long natural names and version (b) contains abbreviated names. In the case of natural names, we can say that they consist of several distinct symbols (i.e. natural words) when the words are separated with an underscore character (e.g. string_index) or when capital letters indicate where another word starts (e.g. StringIndex). When no word separation mechanisms are used, like in Figure 1(b), we can think that the person who has written the program has regarded each name as a single symbol. Using these conventions in identifying symbols, we can find the following language for the program in Figure 1(a):

```
L[Figure 1(a)] =
```

```
is  _  string  a  palindrome  (  char

given  [  ]  ,  int  *  inspection

result  )  {  start  index  end  ;

length  strlen  =  0  -  1  STRING

INSPECTION  NOT  READY  while  ==  if

IS  A  PALINDROME  }  else  ++  --  NOT
```

L[Figure 1(a)] contains 42 symbols while L[Figure 1(b)], which was defined in the previous section, contains only 31 symbols. By using these numbers to compare the understandability of the two programs in Figure 1, it seems that version (b) is easier to understand according to RULE I, although intuitively version (a) is easier to read.

When studying only the technical symbols or only the textual symbols of the programs in Figure 1, we find the following languages:

L[Figure 1(a), technical symbols] =

```
_ ( char [ ] , int * ) { ; =
strlen - while == if } else ++ --
```

L[Figure 1(b), technical symbols] =

```
( char [ ] , int * ) { ; =
strlen - while == if } else ++ --
```

L[Figure 1(a), textual symbols] =

```
is string a palindrome given
inspection result start index end
length 0 1 STRING INSPECTION NOT
READY IS A PALINDROME NOT
```

L[Figure 1(b), textual symbols] =

```
ispdrome str presult i j len 0 1
NREADY YES NO
```

The only difference between languages L[Figure 1(a), technical symbols] and L[Figure 1(b), technical symbols] is the symbol "_" which is used to separate words in natural names. By comparing languages L[Figure 1(a), textual symbols] and L[Figure 1(b), textual symbols] we can easily see that the difference between L[Figure 1(a)] and L[Figure 1(b)] is caused by the textual symbols. Therefore, we can study the understandability of differently named programs by comparing the languages which are composed by listing the textual symbols only.

The symbols of L[Figure 1(a), textual symbols] contain both uppercase and lowercase versions of some words. This is redundant. When listing textual symbols, the lowercase version of a sequence of characters (e.g. of a word) can also represent the uppercase version or the capitalized version of the same sequence of characters. It is merely a syntactic, not a semantic matter whether a word is written in uppercase (e.g. "WORD") or lowercase (e.g. "word") letters, or whether it is capitalized (e.g. "Word"). For example, it is a syntactic convention to start sentences with capitalized words, or write constants in uppercase letters in C programs. The meaning or the word is about the same regardless to how the letters are written. It is also a convention in dictionaries that words are listed using lowercase letters, although in real texts the same words can be written in uppercase letters or as capitalized words.

As it seems evident that textual symbols make the difference when we compare differently named program examples, we will study another example by concentrating only on textual symbols. We will not make a distinction between lowercase and uppercase symbols either. Because Figure 1 is a very simple program, we will study a more realistic example shown in Figure 2. Again, program version (a) in Figure 2 contains natural names and version (b) abbreviated names, and the programs in Figures 2(a) and 2(b) are functionally equivalent. Now we find the following languages:

L[Figure 2(a), textual symbols] =

```
max number of words in lexicon
1000 word length 13 record 14
successful insertion 0 is not 1
already 3 full 4 type insert
into new callers memory index to
caller return code for the
searching status move block
truncated !!! search let's make
space now we have put there
```

L[Figure 2(b), textual symbols] =

```
maxlexsiz 1000 max number of words
in the lexicon maxwrdlen 13 word
length lreclen 14 record okins 0
successful insertion nfound 1 is
not oldwrd 3 already lfull 4 full
lwrdcnt wtype type inswrd nwordin
new caller's memory ntype windex
index to caller rcode return code
i for stat searching status
nword movmem truncated !!! srchwrd
let's make space now we have
put there
```

By using a more realistic program example, we can see that the naturally-named program version results in a smaller language than the program version with abbreviations. L[Figure 2(a), textual symbols] has 50 symbols whereas L[Figure 2(b), textual symbols] has 64 symbols. On the basis of these numbers we can say that the program version in Figure 2(a) is easier to understand according to RULE I, if all the textual symbols in both programs are previously unknown to the person studying the programs.

By comparing the program versions in Figure 2 and the corresponding textual languages, we can see that the natural words of the names in Figure 2(a) are about the same as the natural words in comments in Figure 2(b). Therefore, the difference between languages L[Figure 2(a), textual symbols] and L[Figure 2(b), textual symbols] is the abbreviated names of Figure 2(b). It has been found that natural naming makes comments superfluous (Keller, 1990; Laitinen and Seppänen, 1990). This study confirms the same by showing that the use of commented abbreviated names results in a more complex language than the use of only natural names or abbreviated names.

Let us suppose that a native English speaker is trying to understand the programs in Figures 1 and 2. We can approximate a native English speaker's language as L[words of an English dictionary]. Now, the difference between languages L[Figure 2(a), textual symbols] and L[words of an English dictionary] is smaller than the difference between languages L[Figure 2(b), textual symbols] and L[words of an English dictionary], because L[Figure 2(b), textual symbols] contains abbreviations. Therefore, we can say that L[Figure 2(a), textual symbols] is closer to the person's native language, and, based on RULE II, L[Figure 2(a), textual symbols] would be easier to understand and learn than L[Figure 2(b), textual symbols]. Also, on this basis, language L[Figure 1(a), textual symbols] would be easier to understand than L[Figure 1(b), textual symbols].

To make our example more realistic, let us think that there is a specification document for the programs in Figure 2. This specification document is shown in Figure 3. We can define the language of the specification document as follows:

L[Figure 3] =

```
one service for using the
electronic dictionary , lexicon is
a routine to insert new word into
. this gets following input
parameters : - as string from
caller type of must check that
there space for it also truncate
if exceeds maximum length insertion
return code which tells whether
was successful or already in
```

L[Figure 3] is closer to L[Figure 2(a), textual symbols] than to L[Figure 2(b), textual symbols]. Based on RULE II, we can say that if one already knows language L[Figure 3], i.e., has read and understood the specification of the program, then language L[Figure 2(a), textual symbols] is easier to understand than L[Figure 2(b), textual symbols]. This example shows that it is important to use the same symbols both in specifications and in implementation documents.

One service for using the electronic dictionary, the lexicon, is a routine to insert a new word into the lexicon. This routine gets the following input parameters:

- the new word as a string from the caller
- type of the new word

The routine must check that there is space for the new word. It must also truncate the new word if it exceeds the maximum word length. The insertion routine must also return a code which tells the caller whether it was a successful insertion or whether the word is already in the lexicon.

Figure 3. A textual specification for the program shown in Figure 2.

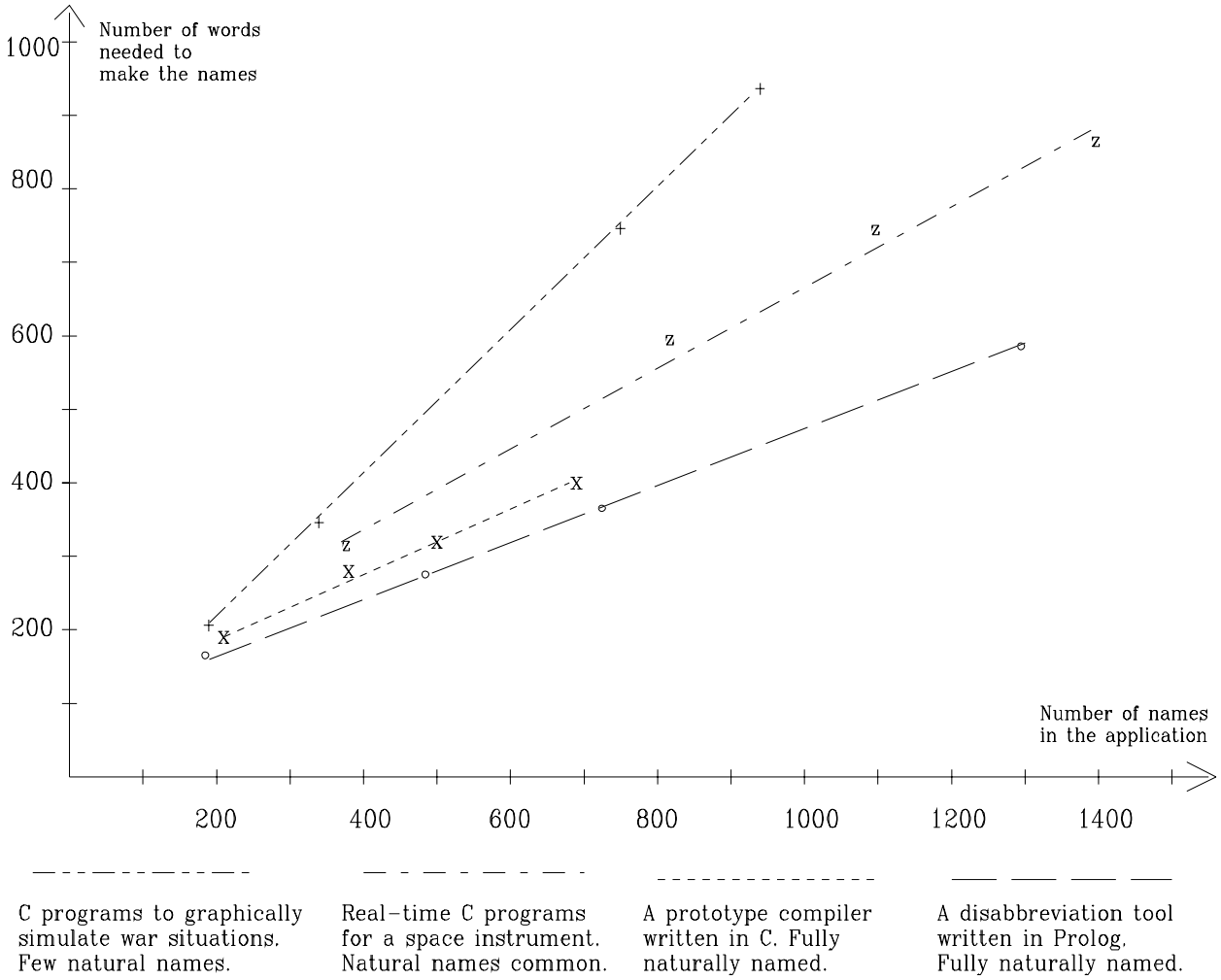

Figure 4. Name/word statistics of some existing software systems.

4 Studying the source programs of some existing software systems

In the previous section we studied small program examples in which different naming styles were used. We found that when source programs are sufficiently long, like those in Figure 2, the use of natural naming results in fewer symbols. We also saw from the example document shown in Figure 3 that the language of naturally named programs is close to documents written in normal English. Therefore, according to our language theory, naming style affects the language of the entire software documentation, and the use of natural naming in source programs can make software documentation more understandable. On this basis, we can estimate the understandability of software documentation by concentrating on the names in source programs. Therefore, to fortify our language theory, we have analyzed the names in four different real applications.

Figure 4 illustrates how many words are needed to make the names in the four applications. As we can see in the figure, we need less words (i.e. symbols) to make the names when natural names are used. According to RULE I, this means that naturally-named applications have simpler languages, and thus they should be easier to understand.

To collect the data for Figure 4, we used a Prolog-based tool which is able to extract all names from source programs. The tool splits each name into one or more words. Underscore characters and capital letters are used to split the names into words in the following way, for example:

```
nbytes              -->   nbytes
customer_number     -->   customer, number
OffsetMeasDone      -->   Offset, Meas, Done
TIME_UpdateTime     -->   TIME, Update, Time
```

The lines in Figure 4 are drawn according to the four calculation points which are also marked in Figure 4. We made four calculations for each application by first counting the words and names in 1/4 of the source program files belonging to the application, then calculating the words and names for half of the files, then in 3/4 of the files, and finally counting all names and words in every file.

Carter (1982) has estimated that the number of words needed in the names of a large application are likely to be counted in hundreds rather than in thousands. Our study confirms this estimation. When natural names were used, about 400 to 600 symbols (i.e. words) were needed, and in the case of abbreviated names, about 800 to 1000 symbols were in use. By relying on the lines in Figure 4, we can estimate that the number of words does not dramatically increase if we analyzed even larger naturally-named applications.

5 Concluding discussion

We have redefined the word "language" in this paper and used this redefinition to estimate the understandability of source programs and other software documents. Our concept of language can be considered a special kind of language theory, as languages are thought to be sets of symbols. We are not claiming that the theory would be complete, but it is a sufficient approximation about what a language is and it can be used to estimate understandability. Our concept of language is not necessarily in harmony with the traditional linguistic concept of language, as we have largely dismissed the syntax of language. We consider that the syntax is embedded in every symbol, i.e., it belongs to the meaning of a symbol how it can be combined with other symbols. To narrow the gap between our theory and traditional linguistics, we point out that traditional linguistic theories are not complete either. Basic linguistic textbooks note that natural languages are something which are not completely understood (Fromkin and Rodman, 1988). Traditional linguistics has also been criticized (Yngve, 1986). Because software documents are special descriptions filled with special symbols and elements from natural languages, we need a special kind of language theory to study the special language in these documents.

Languages are also philosophical issues. The philosopher Ludwig Wittgenstein has developed theories of languages and we consider that our language theory is in harmony with his late philosophy. According to Wittgenstein (1953), the meanings of words and other symbols depend on the "language game" within which they are used. Meanings of symbols vary according to the context of their usage, and certain symbols can be considered to belong to a language game. Our concept of language can be considered to form the symbols of an exactly one language game.

In this paper we have mostly studied source programs in which different naming styles are used. On the basis of our studies, we can recommend natural naming. We have, however, good reasons to believe that our language theory can be used to estimate other software development practices as well. For instance, based on our theory, we can explain why some software development methods can be recommended. There are books (e.g. Page-Jones, 1988) which recommend pseudo coding, i.e., writing first a description of a program with a language which is something in between a natural language and a programming language. A "pseudo coding language" has much less technical symbols than a real programming language. According to RULE I of our theory, a pseudo coding language is more understandable than a programming language. Also, some well-known system modeling and design methods (e.g. Yourdon, 1989; Coad and Yourdon, 1990) utilize graphical technical symbols (see examples in Table 1). There are far less these graphical symbols in the notational conventions of these methods than there are technical symbols in programming languages. According to our language theory, it is simpler to use these methods first than to start writing source programs from scratch.

Our language theory can also be supported by the fact that natural languages change over time (Fromkin and Rodman, 1988). During the history of mankind, natural languages have evolved tremendously. For example, new words have emerged into natural languages and old words have changed their meaning. It is relevant to presume that natural languages change, at least partly, because the world around us changes. Such words as "automobile" or "microprocessor" did not exist before these technical innovations were made. Neither did the words "marxism" and "beatlemania" exist before some people became widely known. If we think that technical innovations change our language, we can think that every software development project also changes our language. Every software system being developed is something new which did not exist before. Therefore, people who develop a new system need to change their language to communicate about the new system. Also, the users of the new system need to change their language to discuss the use of the new system. Thinking this way, we can consider software development a linguistic process which results in a new language, part of which can be found in software documents. Considering software development a linguistic process fits well in with our language theory. Then, avoiding

abbreviations or other means of refraining from the use of unnecessary symbols to enter the language can be seen as controlling the linguistic process during software development.

Acknowledgments

This work has been financed by the Technical Research Centre of Finland (VTT) and the Technology Development Centre of Finland (TEKES). The work has been mostly carried out in an ESPRIT III project called Application Management Environments and Support. This paper contains ideas that have been previously discussed in a working paper which the author wrote together with Mr. Jorma Taramaa. The author wishes to thank Prof. Veikko Seppänen and Ms. Eija Tervonen for commenting this paper.

References

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

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

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

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

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

Fromkin, V. and Rodman, R. An Introduction to Language, Fourth Edition. Holt, Rinehart and Winston, New York, 1988, 460 pages.

Hall, W.E. and Zweben, S.H. The Cloze Procedure and Software Comprehensibility Measurement. IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, 1986, pp.608-623.

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

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

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

Laitinen, K. and Mukari, T. 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, IEEE Computer Society Press, Los Alamitos, California, 1992, pp. 91-100.

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

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

Page-Jones, M. The Practical Guide to Structured Systems Desing, Second Edition, Prentice Hall, Englewood Cliffs, 1988, 249 pages.

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

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

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

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

Weiser, M. and Shneiderman, B. Human Factors of Computer Programming. In: Handbook of Human Factors, Salvendys, G. (editor), John Wiley and Sons, New York, 1987, pp. 1398-1415.

Weissman, L.M. A Methodology for Studying the Psychological Complexity of Computer Programs. PhD-Thesis. Department of Computer Science, University of Toronto, 1974, 231 pages.

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

Wittgenstein, L. Philosophical investigations, Basil Blackwell, Oxford, 1953, 250 pages.

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

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

```
#define  MAX_NUMBER_OF_WORDS_IN_LEXICON     1000
#define  MAX_WORD_LENGTH                    13
#define  LEXICON_RECORD_LENGTH              14
#define  SUCCESSFUL_WORD_INSERTION          0
#define  WORD_IS_NOT_IN_LEXICON             1
#define  WORD_IS_ALREADY_IN_LEXICON         3
#define  LEXICON_FULL                       4

extern  int  number_of_words_in_lexicon ;
extern  struct
{
    char  word[ MAX_WORD_LENGTH ] ;
    char  word_type   ;
}   lexicon [ MAX_NUMBER_OF_WORDS_IN_LEXICON ] ;


insert_word_into_lexicon ( char  new_word_in_callers_memory [],
                           char  new_word_type,
                           int  *new_word_index_to_caller,
                           int  *return_code )
{
    int   index_for_the_new_word,  word_searching_status ;
    int   new_word[ MAX_WORD_LENGTH ] ;

    move_memory_block ( MAX_WORD_LENGTH,
             new_word_in_callers_memory, new_word ) ;

    if ( ( strlen( new_word_in_callers_memory ) ) >= MAX_WORD_LENGTH )
    {
       new_word[ MAX_WORD_LENGTH - 1 ]  =  0  ;
       printf( "\nThe word %s is truncated. ", new_word ) ;
    }

    uppercase ( new_word ) ;

    if ( number_of_words_in_lexicon == MAX_NUMBER_OF_WORDS_IN_LEXICON )
    {
        *return_code  =  LEXICON_FULL  ;
         printf ( "\n Lexicon is full !!! " ) ;
         return ;
    }
    search_word_in_lexicon ( new_word,
              &index_for_the_new_word, &word_searching_status ) ;

    if  ( word_searching_status == WORD_IS_NOT_IN_LEXICON )
    {
       /* Let's make space for the new record. */
       move_memory_block (
              LEXICON_RECORD_LENGTH *
              ( number_of_words_in_lexicon - index_for_the_new_word ),
              lexicon[ index_for_the_new_word ].word,
              lexicon[ index_for_the_new_word + 1 ].word ) ;

       /* Now we have space. Let's put the new_word there. */
       move_memory_block ( MAX_WORD_LENGTH,
                           new_word,
                           lexicon[ index_for_the_new_word].word ) ;

       lexicon[ index_for_the_new_word ].word_type    = new_word_type ;
       number_of_words_in_lexicon   =  number_of_words_in_lexicon + 1 ;
       *new_word_index_to_caller    =  index_for_the_new_word ;
       *return_code                 =  SUCCESSFUL_WORD_INSERTION ;
    }
    else
    {
       *return_code   =  WORD_IS_ALREADY_IN_LEXICON ;
    }
}
```

Figure 2(a). Second example with natural names.

```
#define  MAXLEXSIZ      1000    /* max number of words in the lexicon */
#define  MAXWRDLEN      13      /* max word length  */
#define  LRECLEN        14      /* lexicon record length  */
#define  OKINS          0       /* successful word insertion  */
#define  NFOUND         1       /* word is not in lexicon  */
#define  OLDWRD         3       /* word is already in lexicon */
#define  LFULL          4       /* lexicon is full  */

extern  int  lwrdcnt ;          /* number of words in lexicon */

extern  struct
{
    char  word[ MAXWRDLEN ] ;        /*  word  */
    char  wtype   ;                  /*  word type  */
} lexicon [ MAXLEXSIZ ] ;


insword (   char  nwordin [],   /* New word in caller's memory */
            char  ntype,        /* New word type  */
            int  *windex,       /* New word index to caller */
            int  *rcode )       /* Return code  */
{
    int   i ;      /* index for the new word  */
    int   stat ;   /* word searching status   */
    int   nword[ MAXWRDLEN ] ;

    movmem ( MAXWRDLEN, nwordin, nword ) ;

    if ( ( strlen( nwordin ) ) >= MAXWRDLEN )
    {
       nword[ MAXWRDLEN - 1 ]  =  0  ;
       printf( "\nThe word %s is truncated. ", nword ) ;
    }

    uppercase ( nword ) ;

    if ( lwrdcnt == MAXLEXSIZ )
    {
        *rcode  =  LFULL  ;
         printf ( "\n Lexicon is full !!! " ) ;
         return ;
    }

    srchwrd ( nword, &i, &stat ) ;

    if  ( stat == NFOUND )
    {
       /* Let's make space for the new record. */
       movmem ( LRECLEN * ( lwrdcnt - i ),
                lexicon[ i ].word,
                lexicon[ i + 1 ].word ) ;

       /* Now we have space. Let's put the nword there. */
       movmem ( MAXWRDLEN, nword, lexicon[ i].word ) ;

       lexicon[ i ].wtype    = ntype ;
       lwrdcnt   = lwrdcnt + 1 ;
       *windex   = i ;
       *rcode    = OKINS ;
    }
    else
    {
       *rcode   =  OLDWRD ;
    }
}
```

Figure 2(b). Second example with abbreviated names.