Characterizing the Informal Knowledge Contained in Systems[*]

Nicolas Anquetil
Universidade Católica de Brasília
QS 07, lote 01, Águas Claras
Departamento de Ciencia da Computação, sala D-004
Taguatinga - DF - 72022-900, Brazil
(55) (61) 356-9000 x.9025
anquetil@ucb.br

Abstract

Program comprehension of legacy systems is a highly knowledge intensive task. One of the goal of reverse engineering is to propose automated help to relate application domain concepts to all their implementation instances. It is generally accepted that to do so would require analyzing such documentation as identifiers or comments. However, before attempting to perform this difficult analysis, it would be useful to know precisely what information the documentation contains and if it is worth trying.

We present here the results of a study of the knowledge contained in two sources of documentation for the Mosaic system. This knowledge is categorized in various domains and the relative proportion of these domains is discussed. Among other things, the results highlight the high frequency with which application domain concepts are used, which could provide the means to identify them.

1 Introduction

Reverse Engineering has set it one of its goals to help software engineers in the difficult task of understanding a program. This supposes, among other things, to be able to uncover the relationships between the code and the application domain concepts, or to show how automated operations affect the operational context of a system. This is tagged as the "concept assignment problem" in [5]. In this article, Biggerstaff opposes the *human oriented* concepts to the *computer oriented* concepts. The formers "live in a rich context of knowledge about the world", and are "designed for succinct, intentionally ambiguous communication", whereas the latters are "designed for automated treatment" using "vocabulary and grammar narrowly restricted".

The same article proposes some initial solutions (or research paths) to relate the human oriented concepts to their implementation instances. These solutions include looking at suggestive data or function names. However analyzing and understanding identifiers is a difficult task which raises problems comparable to those experienced in natural language understanding. This is probably one of the reasons why very few other work actually considered doing it.

With the hope of providing some initial information that would indicate where to starts, or even if it is possible at all, we studied the concepts referred to in two documentation sources (identifiers and comments) of a particular system. Each concept was classified in one of several pre-established knowledge domains (e.g. application domain) and we analyzed the repartition of each domain. The hope is to establish whether, and in what conditions, these sources of documentation may be used to discover what concepts are implemented. The structure of the paper is the following, first we will discuss the decomposition of knowledge in domains (section 2), then we discuss our experiment setting in some details (section 3) before presenting results (section 4). We terminate with a discussion of the issues at stake (section 5), and comparison with related work (section 6).

2 Documentation and Knowledge Domains

To try to establish whether documentation can actually provide the means to relate abstract concepts to their implementation, we studied what knowledge it contains. The basic idea is that if comments or identifiers do not refer to application domain concepts they are of no use to solve the concept assignment problem [5]. For this study we need to find out what are the concepts referred to in a source of documentation, and this will be discussed in section 3. But we

[*]This work was partly sponsored by the Universidade Católica de Brasília and partly by a grant from the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ)

also need to categorize the various domains of knowledge and establish the interest of each one. In this section we will present and discuss the knowledge domains we identified.

However before going to this topic, we will first rapidly comment on the issue of using documentation (what we call informal sources of information) for reverse engineering.

2.1 Informal Sources of Information

Using documentation as a source of information for reverse engineering is still a controversial issue for many (e.g. [13]). This is illustrated by the small quantity of work that is using it. Being informal, they are more difficult to analyze, furthermore, there is no guarantee that they do correspond to the actual state of the system (outdated documentation).

We must first state that we place ourselves in the case where some sort of documentation exists, either in the form of commented code or in the form of (apparently) meaningful software component identifiers. Although not always verified, this hypothesis is reasonable in that it corresponds to many real world legacy software systems as witnessed by various Reverse Engineering researchers [4, 6, 7, 9, 11].

Another issue is whether this documentation actually corresponds to the code's functionality or is mostly obsolete. The articles cited above are a demonstration, *a posteriori*, that documentation in many cases is meaningful and can be successfully used. This issue is also dealt with more scientifically in [1]. It presents an experiment to test the reliability of structured types' and fields' identifiers with regard to their definitions. The experiment showed that, for the legacy system studied, the structured types' identifiers significantly relate to their definitions. We conducted the same experiment on the Mosaic system, which we used in this paper and the results were similar, showing that variables' names, structured types' names and fields' names were actually related to their definitions. This experiment does not prove beyond all doubts that identifiers in general are reliable in the system studied. However we consider unlikely that software engineers would pay particular attention to structured types or variables names and use completely incoherent function names. Therefore, we will assume that if type and variable names are relevant, all identifiers are.

Based on all these works, we will accept the hypothesis that informal sources of information is useful. Our goal here will be to study what kind of information it may provide.

2.2 Main Domains of Knowledge

To characterize the knowledge contained in documentation, we felt the need to decompose this knowledge into various domains. We follow in this Clayton *et al.* [10] who studied what knowledge was required to understand a short piece of code. In this work, they identified all the "knowledge atoms" that would be necessary to understand the program under study. These atoms were classified in three knowledge types:

- domain knowledge,
- language knowledge (FORTRAN), and,
- programming knowledge.

The article also mention:

- five knowledge atoms not related to any of their three knowledge types.

Another related work is that of Biggerstaff [5] already cited. He opposes:

- the human oriented concepts, and,
- the programming oriented concepts.

The first issue to deal with is one of vocabulary. To Biggerstaff's "concept types" and Clayton's "knowledge types", we prefer the notion of *domains of knowledge* (or simply domains). Concepts are elements of one or the other domain of knowledge (note that we restrained the domains to be mutually exclusive). To avoid ambiguity, the equivalent of Clayton's domain knowledge will be referred to as the *application domain*. Examples of concepts from this domain could be: plane, reservation, seat, passenger or account, bank, client, stock exchange, etc.

Based on the two articles referenced above and our study, we identified three main domains of knowledge (see also Table 1):

- Application Domain,
- Computer Science domain, and,
- General Domain.

The first knowledge domain, *application domain*, is obvious. Clayton *et al.* recognized it, and, although Biggerstaff does not explicitly says so, all examples of human oriented concepts he gives, are application domain concepts.

Both articles identify a *programming domain* which contains such basic concepts as taught in algorithmic and data structure undergraduate courses (e.g.: searches, sorts, linked list management, etc.) Clayton *et al.* also have a *programming language domain* (FORTRAN in their case). We propose to consider these two as subdomains of a more general *computer science domain*. We expect that other subdomains could be identify as, for example, a hardware sub-domain (disk, memory, register). We have not yet completely explored this issue.

Our Knowledge Domains	Clayton *et al.* Knowledge Types	Biggerstaff's Concept Types
General Domain	"Five unrelated knowledge atoms"	-
Computer Science Domain	Language + Programming Knowledge	Programming Concepts
Application Domain	Domain Knowledge	Human Concepts

Table 1. Different Classifications of Knowledge Domains

Finally, we put the five unrelated knowledge atoms identified by Clayton under the label of a *general knowledge domain*. We see this general domain as containing "all other" concepts, for example mathematical notions (square root, absolute value), or references to actions commonly performed in the "real world" (read, write, get, set), etc. Table 1 summarizes our comparison of these three possible decompositions of knowledge.

We believe this classification in three main knowledge domains could apply to most program understanding situations, although it could be necessary to add other main domains (for example see Table 2).

But this classification is also very coarse and we felt the need to specify subdomains (as the language and programming subdomains for Computer Science). In the present study, this finer decomposition will depend on the specific application domain considered. This is a drawback as it would restrain us from making comparisons with possible similar works. Some effort should be devoted to identify more generally applicable sub-domains. Possible research paths for this are: using recognized decompositions of Computer Science activities or based on the "3-tiers" decomposition, etc. In the first case, one might consider what courses are needed to complete a "typical" computer science graduation degree: Network, Database, User Interface, Artificial Intelligence, Software Engineering, etc. In the second case, the subdomains would be: Database, User Interface and Processing.

We based our work on the assumption that the application domain should be predominant. For example, it sets the boundary of the two other ones. This is obvious for the general domain which contains "all other" concepts, but the application domain may sometimes also influence the computer science domain by appropriating some of its concepts (the domains are mutually exclusive). A good example of this would be a compiler application for which the application domain would obviously include concepts also related to the computer science domain.

We will now give a more detailed presentation of the particular application domain studied in this paper.

2.3 Subdomains of Knowledge

The study we conducted was based on the Mosaic system, the ancestor of the current Netscape web browser. This system is a well accepted workbench for reverse engineering research. It is reasonably old (code dates from 1991 to 1994), it is not a toy program (≃140 KLOC of C code, in more than 380 files), and was developed by various persons. Other, larger, *de facto* workbenches exist, for example the gcc compiler (460 KLOC) or the Linux kernel (600 KLOC). We do think that reverse engineering experiments, as a rule, should be performed on systems in the range of a million lines of code to present some significance, however for this study, which essentially consisted in manual work, we wanted to limit the work to some reasonable amount. The extraction and classification of the concepts already represented two man/weeks of work.

The number of application subdomains we found is larger than we expected. We limited ourselves to five subdomains, but actually identified more than that (ex.: a possible "gopher" subdomain, see below, or an Information Retrieval subdomain). The choice was based on the number of concepts they contained (we kept the most populated). The five application subdomains are:

(I)nterface: Everything dealing with the GUI (e.g.: window, button, slide bar), displaying images in the browser (e.g.: jpeg, pixel), formatting the text of a web page for display or printing (e.g.: postscript).

(T)elecommunication: Everything dealing with the low level aspect of the Internet (e.g.: socket, address, datagram, connect, bind, port, protocol). This also includes concepts related to "DTM" a special language used in Mosaic to transfer complex data over the net.

(H)TML: Everything dealing with the HTML language such as URL, anchor, www, http, cookie, etc.

(U)ser related features: High level concepts that concern directly the user like browser, mail, news, thread, article, telnet, etc.

(W)AIS: Everything that relates to the WAIS application. The Mosaic web browser included the ability to interact with two "concurrent" applications: WAIS and gopher. We found few things directly related to gopher and they were included in the user sub-domain. WAIS was a kind of world wide web information retrieval experiment (somehow like today's Yahoo, AltaVista, InfoSeek, ...). We found enough concepts to

(G)eneral Domain	**(C)omputer Science Domain**	**(A)pplication Domain**	Organization Domain	. . .
Mathematics	Programming Language	**(i)nterface**	Working Environment	
“Name”	Data Structure	**(t)elecommunication**	Organization Rules	
. . .	Algorithmic	**(h)tml**	. . .	
	Specific Software	**(u)ser feature**		
	. . .	**(w)ais**		

Table 2. Some Possible Main Knowledge Domains Relating to Software Reverse Engineering and Their Subdomains. Domains in bold are those that we actually used in this study. Application subdomains are specific to the system studied.

justify the creation of a separate application subdomain. This subdomain includes concepts referring to WAIS (e.g.: wais), information retrieval (e.g.: hit, libraryOfCongress, informationRetrieval) or particular types of documents that WAIS could deal with (e.g.: bibtex, medline).

The interface and telecommunication subdomains are an example of the predominance of the application domain over the computer science domain. We would normally consider them as part of the second, however, in the case of a web browser, we judged that they should be integrated in the application domain.

For the sake of consistency, we should also mention that we identified a *name subdomain* of the general domain. Concepts of the name subdomain were found only in the comments and are the names of maintainers, their email, including the name of computers they were using (ex.: watdscu . waterloo . edu), places where they lived (ex.: Los Alamitos), organizations (ex.: ANSI, Bellcore, Berkeley, CERN), etc.

The application domain differs from the two others in that we have decomposed it completely in subdomains. An application domain concept must belong to exactly one of the five subdomains presented above, whereas a computer science concept may or not belong to one the two subdomains identified previously (language or programming).

For brevity, we will refer to all the domains by their first letter, capitalized for the three main domains (G, C and A) and lower case for the application subdomains (i, t, h, u, and w). We will also refer to a concept belonging to domain X as an X-concept. For example “cookie” is an A-concept and more precisely an h-concept.

We present an overview of all the knowledge domains and subdomains identified in Table 2. Not all these domains will be actually used in this study. We marked in bold those which will. Note that, as explained in the previous section, the decomposition of the Application domain is specific to the system studied and should be revised should we wish to establish a more general framework allowing comparison with possible similar studies.

3 Extraction and Classification of Concepts

We studied two different documentation sources, global identifiers (of variables, functions, types and macros) and comments. We assumed that each word, coming from an identifier or found in a comment, denoted a potential concept. The restriction to global identifiers is for practical reasons, these are the data we already had available and it limits the size of the experiment (we spent more than 30 hours analyzing this particular source).

The experiment consisted in organizing and classifying the words found in the documentation into concepts from the knowledge domains. The general approach we followed was to (i) extract all words from the documentation, (ii) assign each word to a concept, and (iii) assign each concept to a domain. This process is summarized in Figure 1.

3.1 Extracting Words

Finding the words in comments is a simple matter. Identifiers can usually be easily decomposed into “words” following some simple rules such as decomposing on the underscore sign “_”, or usage of upper and lower cases (e.g.: [2, 3, 8]). In this case we manually corrected the automatic decomposition to facilitate the successive steps. As a minor point, one should note that although the identifiers are globally significant in Mosaic, we did found a few peculiar examples such as: “mo_here_we_are_son”, “mo_been_here_before_huh_dad”, “dont_nuke_after_me”, etc.

In the first step, the major difficulty was to deal with acronyms that became part of our vocabulary. Things like “printf” or “IO” (also “I/O”) raise the question of deciding if they should be included “as is” or be decomposed (respectively: “print” + “formatted” and “input” + “output”). As non expert of the application domain(s), another difficulty to discover the meaning of some acronyms as “apdu” or “vdata”. When we found the meaning of an acronym, we faced the same problem of deciding whether to keep the acronym or decompose it.

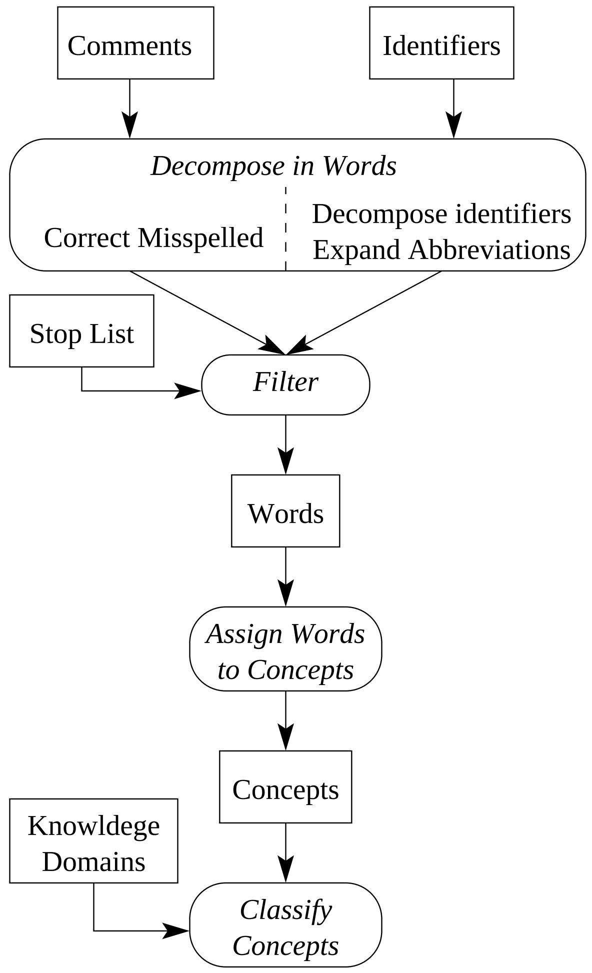

Figure 1. The Different Steps of our Experiment

Cimitile reports the same difficulties in [8]. He proposes some heuristics to help formalize the process. We used a similar approach and applied the following heuristics:

- The majority of acronyms was completely decomposed. This may actually be an error and we now think that the list of special strings (next heuristic) could have been larger.
- As proposed by Cimitile, we established a small list of "special strings" which are the acronyms we kept "as is" (not decomposed). The "jpeg" acronym is a good example of this.
- Another small set of unknown acronyms was also kept "as is" for lack of knowledge of their meaning (e.g. "apdu").
- Finally, to try to deal with the hierarchical nature of the concepts, we sometimes decomposed an acronym (or a composed word) and also kept it "as is", such that "colormap" give "color", "map" and "colormap", thus sharing one subconcept with "keymap".

The acronym problem mainly occurs with words found in identifiers. On the other hand, comments contain misspelled words which identifiers rarely do (the compiler checks for misspelled identifiers).

For both sources of documentation, we terminate the first step by applying a standard stop list to remove utility words as "the", "a", "you", etc. The stop list comes from the Information Retrieval system Smart [12]. It is a general purpose stop list for english texts, a more specialized one should probably be created. For example, this stop list eliminates all single letter words, including the word "C" (we actually re-included this particular one afterward).

3.2 Assigning Words to Concepts

In the second step, nouns, verbs, abbreviations, etc. are associated to standardized concepts. For example, the four words "alloc", "allocate", "allocator" and "allocation" are all associated to the same concept.

This step presents few difficulties, beyond the tediousness of having to consider individually several thousands of words. We tried to automate the process using a stemmer which attempts to discover the "root" of inflected words, by removing the "ing" and "ed" at the end of verbs, or the "s" at the end of nouns, etc. This did not give the results we expected, the tool did not stem many words like "allocator" and "allocation" or adverbs. Most of the work still had to be done manually. It must be noted that comments offer more difficulties here, because they do include more variations of a concept.

3.3 Assigning Concepts to Knowledge Domain

Finally in the third step, we assigned each concept to a domain. Clearly our lack of profound knowledge of the application domain complicated the classification of various concepts. We had to rely on the context of use of the concepts, this context sometimes spanning over several files (concepts from identifiers).

Another difficulty was to decide for a clear border between the different domains: border between (G)eneral and (C)omputer science domains, between (C)omputer science and (A)pplication domains, and between some application subdomains. We tried to structure our work along these lines:

- Consider the semantics of a concept as used in each particular places. We, sometime, assigned one word to two concepts when it was used with two different semantics, for example, the concept "address" can be the

	Concepts and their frequencies
(A)pplication	html(h)=479; mosaic(u)=372; xm(i)=198; ink(i)=181; jpeg(i)=172; dtm(h)=163; xmx(i)=144; cci(t)=128; wais(w)=123; anchor(h)=108
(C)omputer science	data=259; file=210; type=174; record=163; tag=135; string=122; class=120; function=108; table=101; id=100
(G)eneral	set=243; header=208; text=176; size=172; list=170; document=163; read=145; write=139; make=119; initial=117

Table 3. The 10 most frequent concepts from identifiers found in each knowledge domain with their respective frequency (number of software components where they appear). For A-concepts, the subdomain is indicated in parenthesis (see text for a description of the domains).

postal address of someone (G domain), an IP-address (t domain), an email address (u domain), or an address in memory (C domain).

- Assign unknown concepts (like abbreviations) to the most probable domain given the context of use. For example, the apdu concept is only used in files from the WAIS library or from the file HTWAIS.c in the Mosaic source. Since we don't know what it means, we classified it as a w-concept. This is clearly not the best solution and more effort should be spent on discovering the meaning of these acronyms.

- Assign irreducible concepts to the G domain. When it was not possible to assert a probable domain from the context of use, we put the concept in the G domain.

Note that the last two points actually concern a small number of concepts with no significant impact on the results.

The lack of clear boundaries between (sub-)domains, actually prevented us from using the two computer science subdomains (language and programming). We have not yet found a satisfactory definition of these subdomains that would allow to identify their concepts without doubt. We believe part of the problem may be the fact that we don't have a complete partition of the C domain as we do for the A domain.

4 Discussion of some results

As already stated, we are experiencing with the Mosaic system. We will first provide some general information about this system to try to give an idea of the context of the experiment. We will then discuss some results (i) for the entire system, and (ii) decomposed by "subsystems".

4.1 Concepts Extracted

We found close to 6200 global identifiers from which we extracted about 2900 words. After filtering out with the stop list, normalization into concepts, and classification in the domains, we ended up with 1020 different concepts.

We found 7818 different words in the comments, which gave after filtering out, normalization and classification in domains, 3000 concepts, from which, 939 were common to both documentation sources. Actually, the identifiers source is almost included in the comments one with only 14% of its concepts not found in the latter.

The first remark we wish to make concerns the small number of concepts found in the identifiers. Considering that there are close to 6200 global identifiers and that each contains on average 2.67 concepts, there was a potential for more than 16000 possible concepts. Another way to see it is that, in Mosaic, a concept is repeated, on average, in almost 16 global identifiers. This high repetition factor cannot be imputed to the fact that we duplicated some acronyms by keeping them "as is" and decomposing them as well. There are too few such cases to have a significant impact.

We see this high repetition factor as a good point. It means identifiers are well "focused" on a few important concepts. Comments are less focused which is not a surprise, there is an obligation of concision in identifiers which does not exist with comments.

We present, in Table 3, examples of the most frequent concepts extracted from identifiers and their classification. All these concepts are also found in comments and almost all are also frequent in this source of documentation. Concepts "xm" and "xmx" relate to the X-Window system, they are i-concepts. Concept "ID" is one of the concepts that exercise the definition of the border between G and C domains. We decided to classify it as a C-concept based on its use in the identifiers (mostly as a kind of primary/foreign key). Conversely, "list" could have been a C-concept, but it was mostly used in the general sense of list (e.g. "hotlist", the ancestor of the current bookmarks) rather than the more specific sense of linked-list and thus was classified as G-concept.

Frequencies of concepts common to both sources are highly correlated ($\rho = 0.7$), probably indicating that these are the most important concepts in their respective knowledge domains. However, it would be interesting to study the outliers, frequent in one source and not the other, to establish whether they represent a significant subset.

4.2 Domain distribution for Mosaic

In this subsection, we will discuss the overall distribution of the concepts. Results are summarized in Table 4.

	Comments	Identifiers
Total	2998/100%	1020/100%
G	2389/80%	604/59%
C	323/11%	190/19%
A	286/10%	226/22%

Table 4. Repartition of concepts found in comments and identifiers in Mosaic.

There is a very high proportion of G-concepts in general (80% for comments, almost 60% for identifiers). This is understandable for comments which being free text would contain more general consideration. But we were surprised to see the high proportion of G-concepts coming from identifiers. It could be a consequence of the experimental conditions and namely our decision to decompose most of the acronyms we found. An acronym like “printf” which kept as a “special string” would be a C-concept, was decomposed in “print” and “format”, a C-concept and a G-concept.

Although the metrics are different, we can compare the relative repetition factor of concepts from comments or from identifiers (see Table 5). In the comments the most repeated are the C-concepts (on average, 83 appearance of words containing a given C-concept), then A-concepts and finally the G-concepts. In the identifiers, the most repeated are the A-concepts (on average, 23 identifiers refer to a given A-concept), then the C-concepts and finally the G-concepts. This indicates that, although both sources of information contain a larger diversity of G-concepts, they refer more frequently to the A or C-concepts. Thus, although only 22% of the concepts from identifiers pertain to the application domain, 59% of the identifiers contain an A-concept.

One could hypothesize that, in Mosaic, comments are more focused on implementation (or at least computer science concepts) whereas identifiers relate more to the application domain. This would make sense, identifiers implement the A-concepts when the comments explain the implementation tricks. It would be interesting to see if this characteristic is shared by other systems.

	Comments		Identifiers	
	Avg.	StdDev	Avg.	StdDev
all	34.8	94.8	15.9	34.8
G	26.6	75.7	12.5	27.2
C	83.3	172.3	18.2	33.0
A	48.6	97.0	22.7	50.1

Table 5. Repetition factor of concepts found in comments and identifiers in Mosaic. Comments: Total number of appearance of all words referring to the concept; Identifiers: Number of identifiers referring to the concept.

4.3 Domain distribution per Directories

As a kind of validation of our classification in A-subdomains, we will now analyze the repartition of each of these five subdomains among the main directories of Mosaic. This experiment also shows how one could try to “understand” an unknown decomposition of a system using the subdomains each one refers to most frequently.

As described in [2], the Mosaic system’s source files are organized in various directories that can be considered as forming a reasonable functional decomposition, where each directory is a subsystem. There are two directories that are part of the WAIS system and seven which belong more directly to the Mosaic system (see Table 6).

We noticed the following facts:

- Results for the two sources of documentation are consistent.
- W-concepts are mainly found in the “WAIS/ir” directory.
- Reciprocally, there is only one HTML concept in the WAIS directories.
- “Libjpeg” and “libXmx” contain almost exclusively interface concepts. The first is a library to manipulate images in JPEG format, and the second an interface with the X Window System library.
- “Libhtmlw” also has a strong interface composition. It is the part of the code responsible for parsing the web pages and displaying them.

	Concepts from Comments					Concepts from Identifiers				
	i	t	h	u	w	i	t	h	u	w
WAIS/ir	25	29	1	8	19	10	13	1	6	21
WAIS/lib	1	1	0	0	3	0	0	0	0	1
libXmx	28	2	0	4	0	27	0	1	2	0
libdtm	19	24	4	3	2	14	13	4	0	0
libhtmlw	99	9	7	8	1	69	3	5	2	0
libjpeg	83	5	1	4	2	43	0	1	1	0
libnet	12	11	2	1	0	15	9	4	0	0
libwww2	36	26	14	15	3	18	18	13	11	4
src	73	22	15	19	2	71	16	7	15	1

Table 6. Number of concepts found in the comments and identifiers of Mosaic's directories.

- "Libwww2" and "src" seem polyvalent as they contain concepts from all application subdomains (including w-concepts). The first one is responsible for managing all the modules that can be inserted into Mosaic (FTP connection, telnet connection, compression utility, WAIS and Gopher clients, etc.), and the second one is the main directory of Mosaic which links everything together.

- The "WAIS/lib" directory is interesting in that it has almost no A-concepts and few C-concepts from identifiers (Identifiers: 11 G-concepts and 3 C-concepts; Comments: 50 G-concepts and 25 C-concepts). The small size of this directory could explain this outliers behavior.

5 Discussion

The objective of the study was to establish the possibility of solving the concept assignment problem using documentation by asserting whether it does contain the required knowledge. Another issue is whether it seems possible to do it automatically or not. We will now discuss these two points based on the results we presented. We will also consider other relevant issues.

We will assume here that the concept assignment problem concerns primarily application domain concepts.

The low percentage of A-concepts and C-concepts in both documentation sources apparently dismiss them as useful sources of information in this case. We think this conclusion needs to be reconsidered to the light of the following two facts:

- Both A and C concepts have a higher frequency of apparition than the G-concepts.

- The low percentage could be representative of the smaller size of these two domains in comparison with the full range of knowledge expressed.

The fact that G-concepts are globally less frequent in both documentation sources could allow to tell them apart from the other concepts (if it proves to be a property shared by other systems). For example, it seems probable that statistical clustering of Mosaic's software components, based on the common concepts their identifiers refer to, would naturally leads to application domain clusters since these concepts are more frequent in identifiers. A first interpretation of these cluster could follow a path similar to the experiment presented in section 4.3.

This low percentage could also be a consequence of the smaller number of A and C-concepts. This hypothesis would be supported by the fact that comments have almost four times more G-concepts than identifiers, but little more A-concepts.

A possible "light weight" research to assert whether A-concepts are more frequent in identifiers of other systems would be to extract the more frequent concept of identifiers of a system and see if they are A-concepts. This would be simpler than analysing all the concepts of the system as we did in this study.

The second issue is whether it seems possible to automatically solve the concept assignment problem. We must admit that we did not foresaw the extreme difficulty of analysing the concepts present in these two sources of information and the extent of knowledge from all kinds of domains it requires (see example further down). This is a very strong point against using informal sources of information because they are so difficult to analyze. Manually establishing a base of application domain concepts is a poor solution as it compares to reverse engineering a system manually. Possible ways out should integrate various approaches like establishing a base of computer science concepts, establishing a base of the most frequent general concepts, taking advantage of the higher frequency of A-concepts. All these approaches would be based on the (still unproven) fact that the results listed here for Mosaic apply as well to other systems.

An other relevant issue here is the choice of a source of information among the two considered here. This could be based on the potentials and treats each offers.

Extraction of words is relatively easy in the two cases and does not make a significant difference.

Relating words to the concepts they design is a much more difficult task. The problem of deciphering abbreviations is common to both sources of documentation although to a much lesser degree for comments. But comments usually suffer the additional problem of containing misspelled words.

Assigning the concepts to knowledge domains requires a good understanding of all the domains involved as well as a strong general culture. Comments especially have a much larger coverage. They raise problems as identifying names (see discussion on the n-subdomain in section 2), or recognizing apparently unrelated domains. For example, we found in WAIS' comments 80 biology concepts that came out of document examples that WAIS could deal with. They include such things as amoebida, acetylate, or accaagcgac (a particular DNA string). This is a very good illustration of the fact that program comprehension is a highly knowledge intensive task.

There is also the problem of code commented out. We were not sure what to do with this since it does not have a documentation function, but rather uses a property of comments to implement some kind of program editing. We decided to consider it as regular comment, which raises all the issues specifically related to identifiers (e.g.: dealing with abbreviations).

Another problem is that the domain of a concept often depends on its context of use which supposes very sophisticated analysis methods. This is common to both sources of documentation.

On the whole, we estimate it would be easier to work with identifiers than comments. With Mosaic, it would also offer the advantage of having to deal with less G-concepts and being more focused on A-concepts (higher average repetition factor).

6 Related and Future Work

We found few research relating to our. The closest work is the ZEROIN experiment [10] which analyzes the knowledge contained into a very short program. We already mentioned their three categories of knowledge (domain knowledge, language knowledge and programming knowledge) and how they map to our domains (see section 2.3).

This work is mostly oriented toward analysing what knowledge is required to understand a program, thus taking a path complementary to our. Their results show a similar number of "knowledge atoms" needed in the (C)omputer science (60 atoms) and (A)pplication (57 atoms) domains. This confirms that these are the two predominant domains. They do have 5 "knowledge atoms" not related to any of their knowledge types, which suggest that G-concepts can not be completely dismissed. We believe that the small size of both the program (102 lines of code) and the problem for ZEROIN make it non-typical. Therefore a more extensive study of what type of knowledge is mostly needed would be very useful here.

Another work by von Mayrhauser [14], is remotely related in that it could hint at what kind of knowledge software engineers need when doing maintenance. In this work, von Mayrhauser looks at strategies used by software engineers during program comprehension: in the *top-down* approach the engineer starts from his knowledge of the domain and looks for possible implementations of known application domain concepts; in the *bottom-up* approach the engineer studies the code and formulates hypotheses on what it implements. Her conclusion is that both approaches are used alternatively. In our view, the top-down approach would mainly require application domain knowledge, whereas the bottom-up approach would mainly require computer science domain knowledge. This is clearly a very imperfect match since it does not take into account the general domain and each approach cannot be solely based on one domain.

To get a better idea on the software engineers needs, we plan to start another study to identify more precisely what domains of knowledge they typically use and in what proportion.

7 Conclusion

In this paper, we propose to evaluate whether program documentation in the form of identifiers or comments could help in solving the "concept assignment problem", that is to say the problem of linking a portion of the code with abstract concepts familiar to the software engineer. Given the difficulty of analyzing these informal sources of information, it is important to know beforehand if they are actually worth it and can produce the kind of knowledge required.

We classified the concepts found in three main domains: Application, Computer science and General domain containing all other concepts. We mentioned the fact that this decomposition is very coarse and subdomains should be considered (for example: programming knowledge and language knowledge).

The experiment showed that, for Mosaic (the system studied):

- Both sources of documentation contain a large proportion of general concepts (60% for identifiers and 80% for comments). We were surprised by this fact, especially for identifiers. Comments tend to refer slightly more to computer science concepts (11% of all concepts) and identifiers more to application domain concepts (22% of all concepts).

  It is not yet clear if these small proportions would be sufficient to resolve the concept assignment problem, however, other results (see next point) are encouraging and this low percentage could also be the consequence of the small size of these two domains in comparison with the full range of knowledge expressed.

- Computer science and application domain concepts are repeated with greater frequency than general concepts. This is a very good news since it suggests a means to tell G-concepts from the other two.

  Comments and identifiers differ in this regard since comments have a higher repetition factor for C-concepts and identifiers for A-concepts. One may suppose that clustering of software components based on the concept referred to in their identifiers would results in a decomposition according to application domain concepts.

- Identifiers are "focused" on a small set of concepts. With close to 6200 identifiers and an average 2.67 concepts per identifiers, we found only 1100 concepts. This seems to be another proof that in Mosaic identifiers were globally well chosen.

We believe the use of documentation source to solve the concept assignment problem is a valid option in many system, especially if we consider identifiers. This study also pointed to a possible solution to help in the very difficult analysis task, taking advantage of the higher frequency of A-concepts. However, there is absolutely no indication whatsoever that the results presented here can be generalized to other systems. This means that other similar experiments should be conducted to get a better understanding of the use of documentation means in general.

References

[1] N. Anquetil and T. C. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In J. H. J. Stephen A. MacKay, editor, *CASCON'98*, pages 213–22. IBM Centre for Advanced Studies, Dec. 1998.

[2] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Working Conference on Reverse Engineering*, pages 235–255. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[3] N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:1–21, 1999.

[4] G. Antoniol, G. Canfora, and A. D. lucia. Recovering code to documentation links in oo systems. In *Working Conference on Reverse Engineering*, pages 136–144. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. Program Understanding and the Concept Assignement Problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[6] E. Burd, M. Munro, and C. Wezeman. Extracting Reusable Modules from Legacy Code: Considering the Issues of Module Granularity. In *Working Conference on Reverse Engineering*, pages 189–196. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[7] B. Caprile and P. Tonella. *Nomen est Omen*: Analyzing the language of function identifiers. In *Working Conference on Reverse Engineering*, pages 112–122. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[8] A. Cimitile, A. R. Fasolino, and G. Visaggio. A software model for impact analysis: A validation experiment. In *Working Conference on Reverse Engineering*, pages 212–222. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[9] A. Cimitile, A. D. Lucia, G. D. Lucca, and A. Fasolino. Identifying Objects in Legacy Systems. In *5th International Workshop on Program Comprehension, IWPC'97*, pages 138–47. IEEE, IEEE Comp. Soc. Press, 1997.

[10] R. Clayton, S. Rugaber, and L. Wills. On the knowledge required to understand a program. In *Working Conference on Reverse Engineering*, pages 69–78. IEEE, IEEE Comp. Soc. Press, Oct. 1998.

[11] P. Newcomb and G. Kotik. Reengineering Procedural Into Object-Oriented Systems. In *Working Conference on Reverse Engineering*, pages 237–49. IEEE, IEEE Comp. Soc. Press, Jul 1995.

[12] Smart v11.0. Available via anonymous ftp from `ftp.cs.cornell.edu`, in `pub/smart/smart.11.0.tar.Z`. Chris Buckley (maintainor).

[13] H. M. Sneed. Object-Oriented COBOL Recycling. In *Working Conference on Reverse Engineering*, pages 169–78. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[14] A. von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, pages 44–55, aug. 1995.