Study of the Knowledge Contained in Software Components' Identifiers*

Nicolas Anquetil
Visiting Professor
COPPE – Universidade Federal do Rio de Janeiro
C.P. 68511, Cidade Universitaria
RJ, 21945-970, Brazil
(55) (21) 590-2552 x.334
nicolas@cos.ufrj.br

Abstract

Maintaining old software, designed with obsolete methods, and poorly structured, is a knowledge intensive and difficult task. To help in this task, Reverse Engineering seeks to offer tools and analysis techniques that will help creating an abstract representation (a model) of a software system. We believe that one should take advantage of any available source of information to help in this task and know precisely what each source has to offer.

In this paper we study the knowledge contained in the name of software components. Such study is needed to ascertain if this knowledge could be of use for Reverse Engineering, and to perform what activity? Our conclusions are that there is certainly interesting information to be extracted from identifiers, but that alone they may not be sufficient, and other sources, such as comments, would provide a welcome help.

Keywords: Reverse Engineering, vocabulary analysis, program comprehension, concept extraction

1 Introduction

As valuable software gets older, it becomes increasingly difficult to maintain and evolve. Meanwhile the fact that such software still exists is a proof that it is successful and important to the organizations that keep it. The purpose of Reverse Engineering is to help software engineers understand, reorganize and evolve such software.

Reverse Engineering is defined as "the process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships; and, (2) to create representations of the system in another form or at a higher level of

*This work is sponsored by a grant from the Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ)

abstraction" [22]. It is a knowledge intensive task that requires knowledge about the application domain (e.g. what function does the system perform? For who?), software engineering domain (e.g. knowledge of the programming language used), computer science (e.g. speed optimization using register), common sense (e.g. a comment inside the body of a routine describes a particular operation and not the routine as a whole), etc.

We have long advocated that the issue of what information one uses to do reverse engineering is a fundamental one. The majority of the works consider the code as the sole source of information, but this is not enough and a new trend in Reverse Engineering seems to combine information coming from other sources, including documentation, comments or identifiers. We believe that, under certain conditions, these new sources allow to extract more abstract concepts, easier to understand for the software engineers maintaining the code.

But there are still some issues pending:

- To what extent do these sources of information, not related to the code, represent the actual state of a system?
- How can this information help us doing reverse engineering?
- What kind of information can we expect to find there?
- Is this information any better from what we find in the source code?

We will not answer all these questions here, some of them have already been treated in other works and we will come back to these in section 5 (related works). We will essentially discuss the third question and briefly comment on the last one. In order to do this, we have conducted an experiment on the Mosaic system, where all its identifiers were decomposed into words and the resulting vocabulary analyzed. In this paper, we will discuss some conclusions that we draw from this study.

The organization of the paper is the following: First, we will discuss the choice of source(s) of information for reverse engineering. Then, in section 3 we describe the experiment we conducted on Mosaic. Section 4 discusses some aspects of the results obtained. Finally, we present the related works and our conclusions.

2 Sources of Information for Reverse Engineering

One of the traditional hypothesis of Reverse Engineering is that the documentation of legacy software systems is either absent or obsolete. In this context, the sole source of information has often been the code [3], for example, considering interaction between routines, or uses of global variables and user defined types. However, this source of information is intrinsically at a very low abstraction level because it is directed toward automatic tools (e.g. compilers). The code contains much noise and details that are irrelevant to the general understanding of the system. One of the major difficulty in Reverse Engineering is to bridge the gap between the code and significantly abstract concepts from the application domain. This is known as the "concept assignment problem" [5]. An example of this problem, proposed by Biggerstaff, is to consider the fundamental gap between the concept of "reserving an airline seat" and the code "if (seat=request(flight)) && available(seat) then reserve(seat,customer)".

To try to solve this difficult problem, a new trend in Reverse Engineering combines information coming from various sources, including the "documentation" (comments, identifiers, or external documentation). We classify a source of information as *formal*[1] if it consists of information that has direct impact on, or is a direct consequence of, the software system's behavior. For example, using information on routine calls is formal because if we change a routine call in the code, we should expect a change in the system's behavior. Reciprocally, we define as *non-formal*, a source of information that has no direct influence on the system's behavior. A typical example would be the naming of routines, since changing the name of a routine has no impact on the behavior of a system. Non-formal sources of information have been used recently by various researchers [3, 4, 7].

As an information intended for software engineers, such non-formal sources should allow to extract more abstract concepts, easier to understand to the maintainers of the code. Ultimately, the use of these sources of information for reverse engineering lies in three hypotheses:

- Documentation exists in some form.
- Documentation contains information related to the semantics of the software described.
- Documentation refers to abstract concepts that can help in solving the concept assignment problem.

These hypotheses are not true for all systems and we will now discuss their probability in real cases.

2.1 *Documentation Exists*

There are legacy software systems with absolutely no documentation whatsoever. We already said that traditionally external documentation is considered absent or irrelevant in Reverse Engineering. There are also cases where not even source code is available.

However there is a limit to what automatic tools can do and no significantly abstract information can be extracted from such systems. Most works in Reverse Engineering consider more favorable conditions, where the systems exhibit some minimal amount of organization (e.g. structured programming, object oriented code) or documentation (e.g. presence of comments, use of significant identifiers). We will only consider cases where there is some sort of "documentation". We refer here to comments and identifiers (names of types, variables, functions, etc.). This hypothesis is reasonable in that it actually corresponds to many real world legacy software systems as witnessed by various Reverse Engineering researchers [4, 6, 7, 8, 16].

2.2 *Documentation Relates to Semantics*

The second hypothesis is the most important one: We can only hope to use informal sources of information if they have some link with what the system is actually doing. For example, Sneed reports extreme cases where identifiers were the names of people

[1] Note that in [12], Gannod and Cheng give another definition of formality and informality based on the use or not of Formal Methods and Formal Languages (e.g. [11, 13]).

important to the software developers such as their girlfriends or favorite sportsmen [21]. Obviously in such cases, non-formal sources of information would be of no use.

Anquetil [3] had a better experience with a 15 year old legacy telecommunication software and, for example, found that file naming convention played a significant role in its organization. Our position is that, it seems unlikely that organizations can successfully maintain huge software systems for years with constantly renewed maintenance teams without relying on some kind of structuring technique. Reliable comments and identifiers is one such possible structuring technique. By reliable, we mean that the concepts used in these sources of information are related to the software components' semantics they describe and can help understand the purpose of these software components. Other studies agree to some degree with our position [4, 7, 8].

This issue is also dealt more scientifically with in [1]. It presents an experiment to test the *reliability of identifiers* of structured types and fields with regard to their definition. The experiment showed that, for the legacy system studied, the structured types' identifiers significantly relate to the types' definitions. This property is only true "locally" and not over the entire system ($\simeq$ 2 MLOC). There are different forms of "locality", but one can think of it as sub-systems: Inside a sub-system, the identifiers of fields and structured types are related to their definitions. This property has been used by others [14] to extract sub-systems using similarities between identifiers contained into source files.

We suppose that if fields and structured types identifiers are significant, other software components like variables or functions will share the same property. It seems unlikely that software engineers would pay particular attention to types naming and use completely incoherent function or variable names. The same experiment has been conducted on the system we will study here and the results are similar.

2.3 *Documentation Contains Important Abstract Concepts*

The last issue is whether the concepts referred to in the "documentation" can help in the reverse engineering process. This is the focus of this paper. This issue is not exactly the same as the previous one. We just described how we verified that the software components identifiers were related to their definitions, but this does not mean that the identifiers refer to abstract concepts of interest.

However, precisely establishing the *level of abstraction* of a concept would be a difficult if not impossible task. We will use an *ad hoc* scale with two levels of abstraction: application domain level and computer science level. This is clearly not enough and further studies should be devoted to this issue.

In our scale, application domain concepts are the most abstract, they could refer to plane, reservation, seat, passenger, or account, bank, client, stock exchange, etc. We expect computer science domain concepts to be of a lower abstraction level because they would be more related to the implementation. These could be concepts like linked-list, array, pointer, file, memory, etc.

We have few evidences to back up our scale. However, we considered that approaches such as cliché recognition [17, 18, 23, 25] are able to discover programming clichés (handling of a counter, insertion of an element in a list), but, to our knowledge, no tool considers application domain cliché. It is not even clear whether such application domain clichés could be defined. Therefore, it seems reasonable to consider that these

programming concepts are at a lower level of abstraction than the application domain ones.

The notion of *importance* of the concepts is a topic different from their level of abstraction. It should normally depend on the task at hand. For some tasks, important concepts may not be the most abstract ones. For example, if we realize that the input/outputs of a software system are very slow, the important concepts to find out why, would be related to the computer science domain. Nevertheless, for this first study of what kind of concepts we can find in our informal sources of information, we will consider that the important concepts are the most abstract ones.

The next section will describe the experiment we conducted to evaluate to what degree, concepts found in the "documentation" are abstract concepts.

3 The experiment

We will be working with the Mosaic system [15] for which we already have the needed data. 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 ($\simeq$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 first 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 four man/days work.

The informal source of information we are working with is the set of identifiers found in the source code: identifiers of variables, functions, types and macros (C code). These identifiers can usually be easily decomposed into words following some simple rules such as decomposing on "word markers" (any non alphabetic character such as the underscore sign "_"), or usage of upper and lower cases. We assumed that each word inside an identifier denotes a potential abstract concept.

The experiment consists in evaluating if this hypothesis is true and, when it is, with what kind of concept we are dealing. We will first present the domain of knowledge we considered and then the steps we followed to extract the concepts and classify them into the various domains.

3.1 *The Domains of Knowledge*

We will partition the set of concepts in three domains: the two domains of our abstraction scale ((A)pplication domain, and (C)omputer science domain) and a (G)eneral domain which will contain all the concepts we could not classify in the two previous ones. We already cited some possible examples of concepts belonging to the first two domains, examples for the last one could be mathematical notions like "square root" and "absolute value", or references to actions commonly performed in computer science like read, write, get and set, etc.

The application domain is divided in a number of independent sub-domains. The number of these application sub-domains is larger than we expected. We limited ourselves to five sub-domains, but actually identified more than that. The choice of sub-domains

was based on the number of concepts they contained (we kept the most populated). The five application sub-domains are:

(I)nterface: Everything dealing with the GUI (e.g.: window, buttons, slide bars), displaying images in the browser (e.g.: jpeg, pixels), formating 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, addresses, data-gram, 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 URLs, anchors, 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 browser is an ancestor of the current web browsers and it included the ability to deal 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 todays Yahoo, AltaVista, InfoSeek, ...). We found enough concepts to justify the creation of a separate application sub-domain. This sub-domain 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, biology) .

The interface and telecommunication sub-domains were a bit of a problem, since we would normally consider the GUI and management of internet connections as part of the computer science domain rather than the application domain. However, in the case of a web browser, they become parts of the application domain.

For the sake of brevity, we will refer to all the domains by their first letter: capitalized for the three main domains (A, C and G) and lower case for the five application sub-domains (i, t, h, u, and w). We will also sometimes 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.

3.2 *Extraction and Classification of Concepts*

The procedure for extracting and classifying the concepts is the following:

1. Extract all identifiers from the system.
2. Decompose extracted identifiers into words.
3. Assign each word to a concept.
4. Assign each concept to a domain.

In the first step, we *extracted all the global identifiers* from the Mosaic system, that is to say the variables, types, functions, and macros. 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 the vocabulary).

As described at the beginning of this section, the identifiers are automatically *decomposed in words*. We then manually corrected this decomposition. This step raised some problems for acronyms that became part of our vocabulary. Things like "printf" (for C programmers), "IO" (also known as: I/O) raise the problem of deciding if they should be included "as is" or be decomposed.

As non expert of the application domain(s), we also had difficulties with many of the acronyms. It was often difficult to discover the meaning of such things as "cci", "apdu" (which appears to be "a" + "pdu") 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.

We terminate this step by applying a standard stop list to remove some words as "the", "a", "you", etc. The stop list comes from the Information Retrieval system Smart [20]. 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" which is meaningful in computer science.

Cimitile [9] reports problems similar to ours, particularly concerning the decision of keeping or decomposing acronyms. He proposes a small "algorithm" (actually seven heuristics) that he manually applied to try to formalize his method. We applied a similar approach in all the different steps of the concept extraction process. Concerning the decision of decomposing acronyms the heuristics were:

- 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 were 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 and also kept it "as is", such that "colormap" gave "color", "map" and "colormap". This allows us to recognize the concept: "map" and its sub-concepts: "colormap" or "keymap". Likewise, the concept: "key" has several sub-concepts such as "keymap" or "keyboard".
- The majority of the acronyms were decomposed. This may actually be an error and we now think that the list of special strings (first heuristic) could have been larger. We will come back to this point later in the discussion of the results.

The next step of the process consists in *assigning each word to a concept*. We also took care of abbreviations in this step. After this step, nouns, adjectives, verbs and abbreviations are converted to standardized concepts, for example, the four words: "alloc", "allocate", "allocator" and "allocation" were all converted to the same concept.

Finally we assigned each concept to a domain. Clearly our lack of understanding of some acronyms raised problems here. Another difficulty was to decide for a clear border between the different domains: border between G and C, between C and A, and between some application sub-domains. We tried to structure our work along these lines:

- Assign a concept to the most specific domain.
- Only consider the semantics of the concept as used in the identifiers. We, sometime, assigned a word to two concepts when it was used with two different semantics, for example, the word "address" can be the 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 (or pdu) 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.

In the next section we will propose some examples of concepts extracted in this experiment and their classification by domain (see Table 1). We also present and discuss other results.

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:

- for the entire system,
- decomposed by "sub-systems" (directories in the source code), and,
- decomposed by software components' classes (i.e. variables, types, functions or macros).

4.1 *Concepts Extracted*

We found close to 6200 global identifiers from which we extracted about 2900 words. After normalization of the words into concepts, filtering with the stop list and classification in the domains, we ended up with 1038 different concepts. The first remark we wish to make concerns this small number of concepts. Considering that there are close to 6200 identifiers in Mosaic and that each identifier 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 is a good point if one was to cluster software components based on the concepts they have in common. This high repetition factor is not significantly impacted by the fact that we "duplicated" some acronyms by keeping them "as is" and decomposing them as well. There are to few such cases to have a significant impact.

We see the high repetition factor as a good point. It means identifiers are well "focused" on a few important concepts, a conclusion that was not obvious a priori.

We present, in Table 1, examples of concepts extracted and their classification. Concepts "xm" and "xmx" relate to the X-Window system, they are i-concepts. Concept

Domains	Concepts (with frequencies)
Application	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
Computer science	data=259; file=210; type=174; record=163; tag=135; string=122; class=120; function=108; table=101; id=100
General	set=243; header=208; text=176; size=172; list=170; document=163; read=145; write=139; make=119; initial=117

Table 1: **The 10 most frequent concepts in each top level domain with their respective frequency (number of software components where they appear). For A-concepts, the sub-domain is indicated in parenthesis.**

"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. Conversely, "list" could have been a C-concept. We opted for the general domain because 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, which would have been more clearly related to computer science. These two examples give an idea of the difficulties one can face when classifying concepts.

As a minor point, one should note that although the identifiers are globally significant in Mosaic, we did found a few strange examples such as: "been_here_before", "mo_here_we_are_son", "mo_been_here_before_huh_dad", "dont_nuke_after_me", etc.

4.2 *Domain distribution for Mosaic*

We will now start the first of three discussions on the distribution of the concepts in the domains. In this sub-section, we will discuss the overall distribution, we then study more closely the distributions per classes of software components and per sub-systems.

The distribution of the concepts for the three domains is: A$\simeq$ 23%, C$\simeq$ 20% and G$\simeq$ 57%. We were deceived by this result. According to our abstraction scale, about half the concepts are of an interesting abstraction level (A and C domains), and about a quarter are A-concepts which are the ones we judge the most abstract.

The high percentage of G-concepts 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. It is clear that many times, the classification of an acronym is different from the classification of its composing words, and therefore that the decision of decomposing or not as an impact on the distribution of the concepts. This impact should be quantified more precisely.

At this time, it is not clear whether the low percentage of A-concepts that can be found in the identifiers is sufficient to help in solving the "concept assignment problem" (presented in section 2). A significant effort would have to be made to extract as many concepts as possible from the identifiers. A possible research path could be to look for information coming from other sources such as comments or external documentation. We will come back to this issue in the section on related works.

We already mentioned that, in general, the concepts have a high repetition factor.

A good point is that the A-concepts have an even higher one: G $\simeq$ 12 software components per concepts, C $\simeq$ 19 and A $\simeq$ 23. As a consequence, although only 23% of the concepts are from the application domain, 59% of the software components contain an application domain concept. A-concepts are less numerous (than G-concepts for example), but each one is repeated more frequently. This higher repetition factor could mean that clustering the software components based on the concepts they have in common (e.g. see [2, 24]) would create groups (modules) representing principally application domain concepts. Such clusters would be easier to understand for the software engineers.

As a final point, we will suggest that, given its larger size, the general domain should be decomposed in sub-domains too. This could exhibit interesting subsets of this domain with important abstract concepts. This would be an extension of our abstraction scale.

4.3 *Domain distribution per Directories*

To give an idea how automatically extracted clusters of software components could be mapped to application domain concepts, we studied the directories of Mosaic. 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. Some of these (e.g. "libjpeg" and "libdtm") seem to be third party directories that were included inside Mosaic.

In Table 2, we present the concept distribution for each directories of the Mosaic system. The purpose of this study is to show how the directories can be mapped to domains of knowledge, which is a first step toward mapping cluster of software components to concepts.

Directories	G	C	A	i	t	h	u	w
Mosaic	*592*	*207*	*239*	*144*	*31*	*21*	*20*	*23*
WAIS/ir	269	107	51	10	13	1	6	21
WAIS/lib	11	3	1	0	0	0	0	1
libXmx	51	16	29	26	0	1	2	0
libdtm	109	61	29	12	13	4	0	0
libhtmlw	181	61	74	64	3	5	2	0
libjpeg	157	53	43	41	0	1	1	0
libnet	60	40	25	14	7	4	0	0
libwww2	169	91	54	12	18	11	11	2
src	222	89	94	61	13	7	12	1

Table 2: **Number of different concepts found in the identifiers of the various directories of Mosaic.**

One can observe some basic facts that give an idea of the interest that such a domain decomposition could have for reverse engineering. We will not consider here the distribution following the three main domains which offer few interesting points. We will only briefly note that these distributions are quite different from the system wide distribution. Our interest is going mainly on the application sub-domains. One can observe:

- WAIS concepts are found mainly in the "WAIS/ir" directory. There are almost none in the seven "Mosaic directories".
- Reciprocally, there is only one HTML concept in the WAIS directories.
- The "libjpeg" and "libXmx" directories 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" directory has also a strong interface composition. It seems to be the part of the code responsible for parsing the web pages and displaying them. It also handles the browser's window (displaying, resize, scroll bars, etc.)
- The two directories "libwww2" and "src" seem polyvalent as they contain concepts from many application sub-domains (including a few 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 (worst percentage of A-concept with only 7%) and few C-concepts (third worst percentage of C-concepts with 20%). The small size of this directory could explain this outliers behavior, but it could also mark a property of this directory. We did not have time to study this issue more in depth.

The distribution of the concepts for each directory does not specify explicitly its function, however the mapping to application sub-domain does provide a first idea of what functionality some of the directories implement. One must consider that directories are very high level structures, usually automatically created clusters of software components are much smaller. This would simplify their mapping to sub-domains or even to concepts.

4.4 *Domain distribution per "Class" of Software Components*

We also studied the repartition of concepts in the domains for each independent "class" of software component, where there are four "classes" of software components: variables, functions, types and macros.

	Application	Computer	General
Mosaic	*23%*	*20%*	*57%*
variable	29%	25%	47%
type	27%	27%	47%
function	24%	20%	56%
macro	23%	21%	56%

Table 3: **Concept distribution by classes of software components.**

The conclusion (see Table 3) is that the repartition is similar to the system-wide distribution for functions and macros. For variables and types, there are more A-concepts

($\simeq$ 28% of all concepts, approx. +5% from the system-wide distribution) and C-concepts ($\simeq$ 26%, approx. +6%) and less G-concepts ($\simeq$ 47%, approx. −10%).

This distribution gives the idea that variables and types behave similarly. It seems logical since variables and types are data-related. However, this does not concur with our past experiences (see for example [2]). Again more study is needed here. Note that the similarity of functions and macros with the entire system distribution could be caused by their large proportion among the software components (2500 functions and 2400 macros for a total of 6200 software components).

It is interesting to note that data-oriented software components (variables and types) seem to be slightly more related to the application domain and computer science domain. Of course, it is difficult to derive significant conclusions from only one system.

5 Related Works

The closest work is the ZEROIN experiment [10] which analyzes the knowledge contained into a very short program. Their are three categories of knowledge defined: domain knowledge, language (FORTRAN) knowledge and programming knowledge. These categories do not map exactly with our top level domains. Their Domain knowledge is our application domain, their language and programming knowledge fit in our computer science domain, but they don't explicitly mention a general domain. However, they identified five "knowledge atoms" that are not related to any of their three knowledge types. For lack of more information, we will suppose that these correspond to our general domain.

Experiments	Application	Computer	General	Total
ZEROIN	80% (57)	85% (60)	7% (5)	71
Mosaic	59% (3614)	47% (2883)	75% (4589)	6128

Table 4: **Number and percentage of "knowledge atoms" for our three domains in the Mosaic and ZEROIN experiments.**

Table 4 summarize our results and those of the ZEROIN experiment. We don't know exactly what are the "knowledge atoms" considered in the ZEROIN program, in our case they are global identifiers. Note that the sum of percentages in the table is greater than 100% because many knowledge atoms cover more than one concept in more than one domain.

One can see that the distributions are very different in these two cases. We believe that the small size of both the program (102 lines of code) and the problem for ZEROIN make it non-typical. However, the fact that one (ZEROIN) is based on code and the other (Mosaic) on words inside identifiers could also have an impact. This is an issue which would be very interesting to resolve.

There are other, more remotely connected, works. We will mention them and try to see how they could help in solving some of the problems identified in our discussion but left opened:

1. To what extent do non-formal sources of information represent the actual state of a system?

2. How can this information help us doing reverse engineering?
3. Is the low percentage of A-concepts found in identifiers sufficient to help solving the "concept assignment problem"?
4. How can we better decompose identifiers to automatically extracts concepts?

As explained in section 2.2, we do not address the first question in this paper. Many other works have used informal sources of information to do reverse engineering, for example [4, 7, 8, 14], showing, a posteriori, that such a source of information has its interest and actually relates to the semantics of the system. The issue was also specifically dealt with in [1].

All these works also provide examples of how documentation can help us doing reverse engineering (question 2): It ranges from building a modularization of a system [14], to extracting classes from procedural code [8], to relating portions of code with sections of external documentation [4]. In this article, we did show (as a side issue) how automatically extracted groups of software components could be mapped to knowledge domains or concepts, thus easing their interpretation.

Question 3 was raised during this study. Works like [4] seem to answer positively to the question because the authors actually used words found in identifiers to relate classes to sections of the documentation describing them. This is certainly not a definitive answer, because it only deals with one part of the concept assignment problem, which is how to relate the code with already known abstract concepts (i.e. sections of the documentation in this example). The second part, which is to extract abstract concepts from the code, is still unanswered. We did perform this extraction, but it was mainly a manual work which is not practical for large legacy software. A similar extraction method was used by Sayyad in [19]. He used words found in comments to help building (manually) an ontology of the application domain. This work could nullify question 3 by already proposing a solution if the answer were "no". Comments could be used jointly with identifiers to provide more A-concepts.

Finally, [3] proposes a technique, using various sources of information, to decompose identifiers without word markers. This technique could be adapted to improve our current decomposition algorithm and thus solve point 4. Note that a possible useful source of information in this case would again be the comments.

6 Conclusion and Future Work

In this paper, we propose to use words contained in identifiers to help solve 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. We explain that, in order to do this, one must first answer several questions, one of which is: What kind of knowledge can we find in this source of information, and can it actually help in solving our problem? This is what we tried to answer here, with an experiment where we study to what domain of knowledge the words contained inside identifiers pertain.

A first result of the study was to highlight the fact that global identifiers, in the system studied, are highly "focused". There are relatively few concepts (1038) considering the number of identifiers (close to 6200) and the average number of concept per identifier (2.67). We see it as a proof that identifiers contain few noise.

Another result is that, for the system studied, about 50% of the concepts contained in the identifiers are of interest (following our definition) and about 25% are from the application domain. This result appears intuitively insufficient to solve the concept assignment problem, but it must be mitigated:

- It could be a consequence of the experimental conditions and particularly of our choice to decompose most of the acronyms found, thus generating many G-concepts.
- We do not know what amount of concept would be needed to solve the concept assignment problem. Various other works seem to show that there is hope to solve the problem, therefore 25% of A-concepts may be enough.
- The higher repetition factor of the A-concepts is also a good point. It means that clustering methods based on the common concept found in identifiers would extract application domain related cluster with higher probability. We briefly showed how the clusters extracted could be mapped to knowledge domain or concepts.

An important point of this research is to propose a possible direction for future studies which would be to try to combine information from the identifiers and from the comments. A new study, similar to this one but focused on the comments would clarify this point.

The most exciting outcome of this study is actually the number of new questions and research problems it opens:

- Being established that non formal sources of information are of interest, it is now important to discover how they differ from the traditional source code and how the two could be combined to provide better results.
- Some questions remains on the experimental conditions: What was the impact of considering only global identifiers? What is the importance of the manual correction in the decomposition of identifiers into words and in the classification of words in concepts?
- Another direction of research deals with the conceptual aspect of reverse engineering: How can we classify automatically extracted concepts in the various domains of knowledge that a software encompasses? How can we formalize the level of abstraction of a concept? What percentage of the entire application domain is covered by the concepts found in identifiers? An important part of any conceptual knowledge, which is not considered in this article, is the relation between them: How can we find them? How to classify them? etc.

Thanks

The author whishes to thank the members of the Reverse Engineering at COPPE (Federal University of Rio de Janeiro) for helpful comments during the elaboration of this paper: Pr. Claudia Werner, Emerson Cordeiro Morais, Alexandre Luis Correa, José Maria David, and Mônica Rosette.

References

[1] Nicolas Anquetil and Timothy C. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In J. Howard Johnson Stephen A. MacKay, editor, *CASCON'98*, pages 213–22. IBM Centre for Advanced Studies, Dec. 1998.

[2] Nicolas Anquetil and Timothy 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] Nicolas Anquetil and Timothy 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 Andrea De 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] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. Program Understanding and the Concept Assignement Problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[6] Elizabeth Burd, Malcom Munro, and Clazien 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] Bruno Caprile and Paolo 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. De Lucia, G.A. Di Lucca, and A.R. Fasolino. Identifying Objects in Legacy Systems. In *5th International Workshop on Program Comprehension, IWPC'97*, pages 138–47. IEEE, IEEE Comp. Soc. Press, 1997.

[9] Aniello Cimitile, Anna Rita Fasolino, and Giuseppe 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.

[10] Richard Clayton, Spencer Rugaber, and Linda 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] Gerald C. Gannod and Betty H.C. Cheng. Using Informal and Formal Techniques for the Reverse Engineering of C Programs. In *International Conference on Software Maintenance, ICSM'96*, pages 265–74. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[12] Gerald C. Gannod and Betty H.C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *Working Conference on Reverse Engineering*, pages 77–88. IEEE, IEEE Comp. Soc. Press, Oct. 1999.

[13] H.P. Haughton and K. Lano. Objects Revisited. In *Conference on Software Maintenance*, pages 152–61. IEEE, IEEE Comp. Soc. Press, 1991.

[14] Etore Merlo, Ian McAdam, and Renato De Mori. Source Code Informal Information Analysis Using Connectionnist Models. In Ruzena Bajcsy, editor, *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, volume 2, pages 1339–44. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1993.

[15] NCSA Mosaic Version 2.6. Available through anonymous ftp at `ftp.ncsa.uiuc.edu`, in `/Mosaic/Unix/source`.

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

[17] Srinivas Palthepu, Jim E. Greer, and Gordon I. McCalla. Cliché recognition in legacy software: A scalable, knowledge-based approach. In *Working Conference on Reverse Engineering*, pages 94–103. IEEE, IEEE Comp. Soc. Press, Oct. 1997.

[18] Alyson A. Reeves and Judith D. Schlesinger. Jackal: A hierarchical approach to program understanding. In *Working Conference on Reverse Engineering*, pages 84–93. IEEE, IEEE Comp. Soc. Press, Oct. 1997.

[19] Jelber Sayyad-Shirabad, Timothy C. Lethbridge, and Steve Lyon. A Little Knowledge Can Go a Long Way Towards Program Understanding. In *5th International Workshop on Program Comprehension*, pages 111–117. IEEE, IEEE Comp. Soc. Press, May 1997.

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

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

[22] IEEE Technical Council on Software Engineering. `http://www.tcse.org/revengr/`.

[23] P. Tonella, R. Fiutem, G. Antoniol, and E. Merlo. Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic – A Case Study. In *Working Conference on Reverse Engineering*, pages 198–207. IEEE, IEEE Comp. Soc. Press, Nov 1996.

[24] Theo A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Working Conference on Reverse Engineering*, pages 33–43. IEEE, IEEE Comp. Soc. Press, Oct. 1997.

[25] Steven Woods and Alex Quilici. Some Experiments Toward Understanding How Program Plan Recognition Algorithms Scale. In *Working Conference on Reverse Engineering*, pages 21–30. IEEE, IEEE Comp. Soc. Press, Nov 1996.