Extracting Concepts from File Names; a New File Clustering Criterion*

Nicolas Anquetil **Timothy Lethbridge**

School of Information Technology and Engineering
150 Louis Pasteur, University of Ottawa
Ottawa, Canada, K1N 6N5

(1) (613) 562-5800 x6688 (1) (613) 562-5800 x6685
anquetil@csi.uottawa.ca tcl@site.uottawa.ca

ABSTRACT

Decomposing complex software systems into conceptually independent subsystems is a significant software engineering activity which received considerable research attention. Most of the research in this domain considers the body of the source code; trying to cluster together files which are conceptually related.

This paper discusses techniques for extracting concepts (we call them "abbreviations") from a more informal source of information: file names. The task is difficult because nothing indicates where to split the file names into substrings. In general, finding abbreviations would require domain knowledge to identify the concepts that are referred to in a name and intuition to recognize such concepts in abbreviated forms. We show by experiment that the techniques we propose allow about 90% of the abbreviations to be found automatically.

KEYWORDS

Reverse Engineering, Design Recovery, Artificial Intelligence, Program-Understanding

1 INTRODUCTION

Maintaining legacy software systems is a problem which many companies face. To help software engineers in this task, researchers are trying to provide tools to help extract the design structure of the software system using whatever source of information is available. Clustering files into subsystems is considered an important part of this activity. It allows software engineers to concentrate on the parts of the system they are interested in; it provides a high level view of the system, and it allows them to relate the code to application domain concepts.

Much of the research in the file clustering domain deals with the source code; trying to extract design concept from it. However, we studied a legacy software system, and noticed that software engineers were organizing files according to some file naming convention. In this paper we propose to use this more informal source of information to extract subsystems.

Our overall project aims at building a conceptual browser for legacy software systems. A key part of this is a design recovery facility where files are clustered according to the concepts they refer to. These clusters or concepts may be called subsystems.

Merlo [7] already proposed to cluster files according to the concepts referred to in their comments and the function names they contain. However an experiment we conducted [1] shows that in our case, file names are a much better clustering criterion than function names or comments.

But file names are more difficult to deal with than function names: they rarely contain word markers (e.g. capital letters, hyphens and underscores) which help differentiate the various concepts they refer to. For example, in our corpus, only 11 file names out of more than 1800 have such word markers.

In this paper we propose and study various methods of decomposing a file name into a list of "concepts".

We will first advocate the need for new clustering criteria in design recovery. We then propose some solutions to extract "concepts" from file names. this is done in three steps:

- In section §3 we discuss the overall strategy we used.
- In section §4 we propose different methods for the first stage of our strategy: generating candidate concept names (we call them "abbreviations").
- In section §5 we describe the second stage of our strategy: how to decompose file names using the generated abbreviations.

*This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER).

We designed and conducted an experiment to compare the various methods we propose. This experiment will be presented and its results discussed.

2 CRITERIA FOR FILE CLUSTERING

Our goal is to build a conceptual browser for a legacy software system. This activity implies clustering semantically related files. Each cluster (i.e. subsystem) obtained should correspond to a particular concept.

Traditionally (see for example [9]), there are two possible approaches to subsystem discovery:

The top-down approach consists of analyzing the domain to discover its concepts and then trying to match parts of the code with these concepts.

The bottom-up approach consists of analyzing the code and trying to cluster the parts that are most closely related according to certain criteria.

It is generally admitted that a successful solution should use both approaches. However the difficulties associated with the top-down approach make it unpopular. It is costly because it implies extracting knowledge from experts during long interviews. It is also domain specific which limits its potential for reuse.

The bottom-up approach has received more attention. The preferred way to perform it is to look at the very material software engineers deal with every day: the source code. The difficulty is then to lift the very low level information available to a suitable level of abstraction.

In last International Workshop on Program Comprehension [4] only one paper [8] on program comprehension (out of nine papers) was based on criteria other than source code: it used the comments.

In [7], Merlo proposed to use comments and function names to perform an activity that is basically bottom-up but would help achieve the top-down approach. Using comments and function names, he extracted application domain concepts and clustered files according to these concepts.

Sayyad-Shirabad [8], also proposes to use comments to simplify the process of building an application domain knowledge base (top-down approach).

For us, an important achievement of Merlo's research was the introduction of new file clustering criteria. As Merlo states: "many sources of information may need to be used during the [design] recovery process".

Source code analyses offer very appealing solutions, but they require lengthy computations that are incompatible with the size of large legacy software systems (millions of LOC). More informal analyses could be used to provide a first broad decomposition of large systems.

We believe a successful design recovery tool will need more information of a higher abstraction level such as that provided by comments or naming conventions. This could allow associating the modules generated to application domain concepts.

In [1], we explain how we were led to consider file names as a file clustering criterion: we work on a project whose primary goal is to bridge the gap between academic research and industrial needs. The software system we study is big (1.5 million LOC, 3500 files) and old (over 15 years). Nobody in the company fully understands it anymore, yet it is the subject of much ongoing enhancement and is very important to the company.

We decided to provide a conceptual browser to the software engineers maintaining the software system we study. To this purpose, we asked them to give us examples of subsystems they were familiar with.

Studying each subsystem, it was obvious that its members displayed a strong similarity among their names. For each subsystem, concept names or concept abbreviations like "q2000" (the name of a particular subsystem), "list" (in a list manager subsystem) or "cp" (for "call processing") could be found in all the file names.

We were led to conclude that the company is using some kind of file naming convention to mark the subsystems.

Because we aim at building a browser, we may afford to use this informal file clustering criterion. The precision of the generated subsystems is not as critical as for other domains like re-engineering where a key requirement is precise preservation of semantics.

We do not think that the system we are working on is an exception. We already mentioned Merlo's work with function names. In another paper, Tzerpos [10] also notes that "it is very common for large software products to follow such a naming convention (e.g. the first two letters of a file name might identify the subsystem to which this file should belong)".

It seems unlikely that companies can successfully maintain huge software systems for years with constantly renewed maintenance teams without relying on some kind of structuring technique. The experiment we conducted [1] shows that for our software system, file names is the best clustering criterion. We do not pretend it is the sole solution to file clustering, but it is one of many possibilities. Hierarchical directories is another commonly used approach (e.g. in the Linux project [6]).

In the following sections, we will propose different methods to decompose file names into a list of constituent concepts. We call the component of a file name *abbrevi-*

ations because they often consist in abbreviated forms of words.

3 STRATEGY TO FILE NAMES DECOMPOSITION

Our goal is to find all the "abbreviations" composing a file name. We call *abbreviations* any substring of the names that denotes a concept. It may be an application domain term (e.g. "q2000"), an English word (e.g. "list") or an abbreviated form of a word (e.g. "svr" for server).

File names are much more difficult to deal with that function names because they rarely contain "words markers" (underscore characters, hyphens, or capital letters) which help in decomposing a name. The abbreviations they use are also much shorter, some are only one or two characters long.

To stress the difficulty of the task, we wish to point out that for some file names, manually finding the right decomposition involves looking at the file itself, at its comments and functions, or at other files which seemed to have a related name. Overall we spent more than 5 hours decomposing a little more than 10% of the file names in the whole software system. We later used this manual work as a benchmark against which to test our automatic techniques.

There can be different approaches to decompose all file names of a software system. The task implies two activities:

- we need to find which character strings form a valid abbreviation, and
- we need to find which abbreviations compose a name.

The two activities may be conducted concurrently or one after the other.

For example, while doing it manually, we first quickly identified some valid abbreviations: English words ("list", "call") or specific substrings common to many files (e.g. "q2000"). Using these abbreviations, we could decompose some simple file names and obtain further abbreviations from the remaining substrings of the names. The process would them be repeated iteratively until no more abbreviations are forthcoming.

However the process has rapidly diminishing returns, and we had to study the documentation (mainly comments) carefully to find out what some names could "mean".

From this little study, it was quite evident that the task is a knowledge intensive one. It requires application domain knowledge as well as a good deal of intuition to identify some abbreviations.

The iterative process we described above does seem a natural way to proceed, but it has several drawbacks:

- It is a very slow process, which would require frequent backtracking were it to be automated.
- It requires a function to tell when we have found the right decomposition for a name (what is the right decomposition for "fsblock": "fsb-lock" or "fs-block" ?). This function would be mainly based on intuition and seems difficult to implement.

We will rather use a sequential strategy: first we will try to find all the abbreviations used in the software system. Second, given this set of abbreviations, we will try to find the proper decomposition for each file name.

This sequential strategy also has drawbacks. The main one being that all the errors in the first step are fed into the second one, thus rendering it more difficult.

To try to avoid this as much as possible, it is important that the first step has two contradictory properties:

Precision: It should only generate actual abbreviations.

Completion: It should not miss any actual abbreviation.

To promote *precision*, we want to be conservative when generating abbreviations, only releasing those which have a very good chance of being valid. We must, however, consider more closely this issue: the only wrong abbreviations we need to avoid are those that we could reasonably expect to find in a file name. Accepting "stylostixis"[1] as a candidate abbreviation would probably not alter the final results.

On the other hand, to promote *completion*, we want to generate as many abbreviations as possible.

In general, we will promote precision over completion. If needed, we hope to achieve completion by combining different sources of abbreviations. Hopefully, different sources should provide a better coverage of the set of all abbreviations.

4 CANDIDATE ABBREVIATIONS

The first step of our method may be the most difficult one. It requires a lot of knowledge. For example, it seems unlikely that someone who is not expert in the

[1] Note: This means acupuncture. We, on the other hand, are working with a telecommunication software system

software system could guess what abbreviations compose a name like “actmnutsg”[2].

Intuitively there are several ways to generate our abbreviations:

- taking actual English (short) words,
- taking the prefix of English words (“aud” for audit),
- taking the consonants of a word (“dbg” for debug),
- taking the initials of composed words, i.e. acronym (“cp” for call processing).

But we wanted our method to be precise. Using these rules would generate a lot of useless candidate abbreviations. In the best case, these rules could be used to confirm that a candidate abbreviation (generated by some other means) is valid.

Our main research path will be to look for valid abbreviations used in some *sources*. For example, software engineers may use abbreviations in their comments (English words and application domain terms). Therefore, comments will be considered a possible source of abbreviations.

In the remainder of this section, we present several sources and methods of extracting abbreviations. We shall discuss latter how we use the abbreviations to decompose file names.

4.1 Source: File Names; Method: Iterative

The first source of abbreviations we will consider is file names themselves. We already outlined how we manually extracted some abbreviations from file names by starting with “obvious” ones and progressively deducing new ones from those already asserted. We will describe here an attempt to automate this process.

Some of the names are very short (2, 3 or 4-character long). As a heuristic, we assume that these are single abbreviations, and therefore will become a base set of “obvious” abbreviations. Study of the names, as well as experiments, show that 5-character long names (and longer) are usually combinations of shorter abbreviations; although names of 6 characters or more may still occasionally be a single abbreviation (e.g. “kermit”). On the other hand, 4-character long names may be a combination of two abbreviations (e.g. “swid” stands for “software-id”). Overall however, this heuristic seems to offer a good precision while remaining as complete as possible.

[2] Answer: ACTivity MoNitor UTilities SG (a product name).

Using this base set, we may try to extract further abbreviations by decomposing some simple names. We only look for names prefixed by one of the known abbreviations (for example “damsvr” is prefixed by “dam”).

After removing the prefix abbreviation from a name, if the remainder is less than or equal to 4 characters long, it will be considered an abbreviation (“damsvr” gives “svr” as remainder). Otherwise, we try to find another known abbreviation which prefix this remainder.

This method does extract wrong candidate abbreviations (candidates that are not actual abbreviations), its precision is not perfect. The problem is that some abbreviations are prefixes of other. For example “activ” and “activity” are two valid abbreviations. If we already asserted that “activ” is an abbreviation, the algorithm will wrongly deduce that “ity” is another one. This kind of wrong abbreviation is dangerous because it may actually be found in other file names. We see no simple way to avoid this problem in general (in the example given here, we could use an English dictionary to recognize that “activity” is a single word and should not be decomposed).

The algorithm sketched here does appear to follow the iterative strategy we dismissed in section §3, but there are key differences:

- The decomposition here is very straightforward, there is no backtracking. If we find several possible decompositions for a file, all resulting abbreviations will be accepted.
- We do not try to decompose all names, if some file is not prefixed by a known abbreviation, or if the remainder of this name after removing all prefixing abbreviation is longer than 4 characters, we simply do not decompose it.

The purpose here is not to decompose file names, but to try to find as many abbreviations as possible. In fact few file names were decomposed by this method, less than 400 out of 1800.

4.2 Source: File Names; Method: Statistical

There is another way we can use file names to get abbreviations: by proposing as a candidate abbreviation any substring common to several file names.

The primary reason for decomposing file names into their constituent abbreviations is that we want to build a browser on the file names. This implies we will want to gather all files referring to the same concepts. Therefore, we are primarily interested in abbreviations that appear in several file names.

Extracting all common substrings from a set of names may be very time and space consuming. A naive way to do it would consist of extracting all the substrings (of any length) of each name and then trying to find similar substrings in different names. However, in practice this can prove impossible because there can be many thousands of names.

We propose instead to extract from each name, all the substrings of a given length it contains. These strings are called n-grams [5]; for a length of 3, one speaks of 3-grams. For example, "listdbg" contains the following 3-grams: "lis", "ist", "std", "tdb" and "dbg". The number of n-grams in a name is linear in the length of this name, whereas the total number of substrings would be quadratic in the length of the name.

After extracting all n-grams from the names, we build a Galois Lattice (see for example [3, 11]). This structure allows us to cluster file names which share n-grams. The Galois lattice has an important property: it will find all clusters of files sharing at least one n-gram. That way we are sure we will not miss any substring common to any set of files.

The length of n-grams is important, too long and they will not allow short substrings to be detected, too short they can produce too many clusters to be manageable.

This is a case where dealing with an actual software system may be a problem, because of the size of the system we are studying, we have not yet been able to generate the Galois Lattice for 2-grams, the program crashes for lack of memory. We are working on a new implementation that will avoid having to keep the data structure entirely im memory.

As a consequence, the experiments reported in this paper (see section §6) were made using 3-grams. This means we miss the 2-character substrings. Note that candidate abbreviations of length greater than n (3) are easily found using the Galois Lattice.

It may happen that two very close names differ only by one letter (e.g. "activmon" and "activmonr"). We do not want the full name to become a candidate abbreviation. To avoid this, we decided to keep only those candidate abbreviations shorter than a given length (5 characters, inclusive).

Then again, some files contain words longer than our threshold (e.g. "serial"). We modified the above rule to accept abbreviations longer than 5 characters if they were English words. We recognize English words using an English dictionary (see §4.6).

For the sake of clarity, this algorithm is sumarized in figure 1.

```
foreach file name
   Extract the 3-grams it contains
Build the Galois Lattice for all file names
foreach cluster in the Galois Lattice
   sstr = substring shared by all names in the cluster
   if string_length(sstr) > 5 then
      if not is_english_word(sstr) then
         reject sstr
      else
         accept sstr as a candidate abbreviation
   else
      accept sstr as a candidate abbreviation
```

Figure 1: Algorithm for statistical extraction of abbreviations from file names

4.3 Source: Comments

The algorithm presented in the preceding section may extract wrong candidate abbreviations. For example "dial" and "diag" (for diagnostic) are actual abbreviations, but all files having them, also share the string "dia" which is not a valid abbreviation. These wrong abbreviations are dangerous because they are found in file names.

We will now present an attempt to filter out the wrong candidates.

Some abbreviations used in file names are words, either English words or application domain terms. For these reasons, many "abbreviations" appear in the comments. Our filter will simply consist of taking each candidate abbreviation generated by the previous method and keeping only those one we find in the comments.

In the software system we are studying, many files ($\simeq$ 70%) have a summary comment which describes the main purpose of the file (as opposed to that of functions). We restricted ourselves to these comments. The primary purpose of this restriction is to limit the size of data to deal with.

In a first experiment, we looked for a candidate abbreviation only in the summary comment of the files that had this particular candidate abbreviation in their names. In a second experiment, we used abbreviations found in the summary comments of all files; this gave better results. We believe that because comments are only used as a filter for pre-generated substrings, the more comments we have, the better it would be.

As a logical conclusion, it seems that although the summary comments may appear more "focused", we should get better results using all of them.

4.4 Method: Computing Abbreviated Forms

The comments may be a good filter for those abbreviations which are actual words, but some abbreviated forms of words like "dbg" (for debug) will probably not be used as such in the comments. Instead only the full word will appear.

We present here an attempt to recognize these abbreviated forms. This method is intended as an extension of the previous one.

In section §4, we gave some rules to build abbreviated forms from words:

1. taking the prefix of words ("aud" for audit),
2. taking the consonants of words ("dbg" for debug),
3. taking the initials of composed words, i.e. acronym ("cp" for call processing).

We tried to implement the first two rules. For each candidate abbreviation (substring common to some names), we searched for a word for which it would be a prefix. We also computed an abbreviated form of the words by taking their first letter and all of their consonants (e.g. "abbreviate" would give "abbrvt"). If a candidate abbreviation matches a computed abbreviated form we keep it.

Here again, the words were taken from the comments. But because the calculations performed on each word are more complex than in the preceding section, we limited ourselves to the summary of those files which had a candidate abbreviation in their name. We do believe considering all summary comments or all comments of all files would produce better results.

We know this method is not complete. The second rule (taking all consonants of a word) should be loosened a bit. For example, a common abbreviation for server is "svr", where the first "r" is dropped. But it seems difficult to implement this efficiently, and without loosing all precision. As future research, it might be possible to use a dictionary that contains "soundex" information about words; many abbreviations (e.g. "svr" sound simular to the full word.

4.5 Source: Identifiers

Identifiers are another possible source for abbreviations. In general, identifiers would include the names of functions, variables, structured types, etc. However in our experiments, the only identifiers we considered are function names. We made this choice both for practical reasons and because we believed functions have a better chance of addressing the same concepts as the files.

Merlo [7] already recognized functions as a useful file clustering criterion. Here however, we do not want to cluster files based on the identifier names they contain (this has already proved unsuccessful in our case; see [1]). Instead we will use identifiers as an easy way to get some abbreviations the software engineers may use.

Identifier names are much easier to decompose than file names. As a rule, identifier names contain word markers: the underscore character ("_") and capital letters. Because of these word markers, breaking down an identifier's name into the list of its abbreviations is a straightforward task.

But identifier names may be longer than file names. As a consequence, some abbreviations are not the same (mainly the ones that are abbreviated forms of words). This should not be a problem as those longer abbreviated forms have few chances of appearing in file names. For example, assume the word "abbreviation" may be abbreviated by "abrv" in file names and "abbrev" in identifier names. For the latter to appear only fortuitously in file names, it would require two contiguous shorter abbreviations like "abb" and "rev". This seems highly improbable.

Another difficulty —more serious— is that some functions have the name of the file that defines them in their own name. We do not want these file names to be unconditionally proposed as abbreviations. To avoid this, we use the same length filter as already discussed (§4.2) and keep only those shorter than 5 characters (unless they are English words).

4.6 Source: English Dictionary

In several of the above presented methods, we noted the need for an English dictionary: some abbreviations (e.g. "list") or file names ("e.g. "activity") are actual words.

This gave us the idea of using an English dictionary to produce candidate abbreviations. This will obviously give poor results by itself, because many abbreviations are not English words. But we will see that it may be a good auxiliary technique when combined with other sources.

The English dictionary we used is the one usually found in `/usr/dict/words` on unix systems. On our system (Solaris 2.5), it contains a little more than 25000 words.

The method here simply consists of assuming any of these words is an abbreviation and then trying to decompose the file names using these candidate abbreviations. How we decompose the names will be detailed in the next section.

The dictionary has some precision problem. It contains words like "in", "the", or even "AC" that could easily be found in file names and may not be abbreviations in our system. We tried to deal with this by using a standard stop word list (from the Information Retrieval system Smart [2]), but it proved harmful. Some more specialized stop list should be set-up.

Using this dictionary also presents completion problem. It lacks some common words we need (e.g. "display"). It also lacks the ability to inflect words (conjugate verbs, put an "s" at the end of nouns). A more intelligent tool like "ispell" could give better results.

5 DECOMPOSING FILE NAMES

Having extracted a set of abbreviations, we now want to decompose each file name into a list of abbreviations. We call each decomposition of a name a *split*.

For each name, we find all candidate abbreviations it could contain. We then generate all combinations of abbreviations that could compose the name. For more flexibility, we allow free characters in a split (a character not belonging to any abbreviation). For example assuming we have two candidate abbreviations "list" and "db", the possible splits for "listdbg" would be:

- list – db – g
- list – d – b – g
- l – i – s – t – db – g
- l – i – s – t – d – b – g

As there are relatively few possible splits for each name, we are able to generate all of them. But some are obviously wrong (e.g. the last one in the above example). We use a rating function to assess the correctness of the splits. We only keep the split(s) with the best rate (there may be ties).

The current rating function simply gives the higher rating to the split with fewer members (candidate abbreviations and free characters). The rationale is to discard those splits with a lot of free characters.

We considered using more complex rating functions, with multiple criteria, such as:

- Length of the abbreviations: from informal study, it seems that 3-character abbreviations are more numerous than other sizes. We could take this into account and give more weight to these abbreviations.
- Number of sources proposing each abbreviation: when using several sources together, we could give more weight to an abbreviation proposed by different sources.
- Weighted source for abbreviations: we do not have the same confidence in all sources, some could have a higher weight than other, therefore favoring the abbreviations they generated.

The rating function does have an influence on the quality of the results (although it seems to be a minor one). But, because our goal is to compare various sources of abbreviations, we did not want to alter the results by using a rating function that is too complex.

6 EVALUATING FILE NAMES DECOMPOSITION

In previous sections, we have proposed several sources and methods of generating candidate abbreviations and have explained how to decompose file names using these candidate abbreviations.

Each of the abbreviation set generated produces a different result. We now need to evaluate these results to access which method is the best one.

We will first describe the experiment we designed and then present and discuss the results.

6.1 Experiment Design

To evaluate the quality of the decompositions, we need some known results against which to compare them. For this purpose, we manually decomposed some of the names. To get a representative sample, we decomposed the first 10 names (in alphabetical order) for each initial letter. As some of the letters have very few or no file name beginning with them, we decomposed only 221 names. This sample represents 12% of the whole corpus (1817 file names).

We decided to accept several correct splits for some names. There are different reasons for this:

1. some words may be abbreviated in many ways ("activ" or "act" for activity),
2. some words are singular and plural ("error" and/or "errors"),
3. some abbreviations overlap ("saservice" stands for "sas service"),
4. some words are composed ("wakeup").

In each of these cases, we did not want to arbitrarily choose one of the alternatives. We will accept all "sensible" splits. For example, for "activity", we accept "activity", "activ" and "act". In the last two cases, the remainder of the word is simply ignored. In the case of overlapping abbreviations like "saservice", we accept "sa – service" and "sas" ("ervice" cannot be an abbreviation and is ignored).

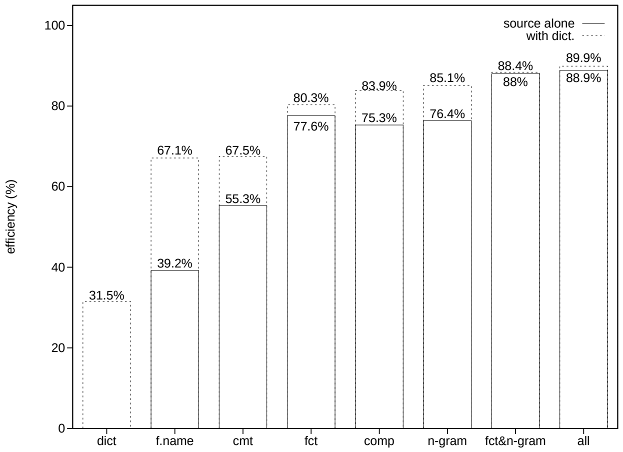

Figure 2: Efficiency of different sources of abbreviations. The methods are: (dict) abbreviations from English dictionary, (f.name) from file names, (cmt) from comments, (fct) from functions, (comp) computed abbreviated forms, (n-gram) substring common to names.

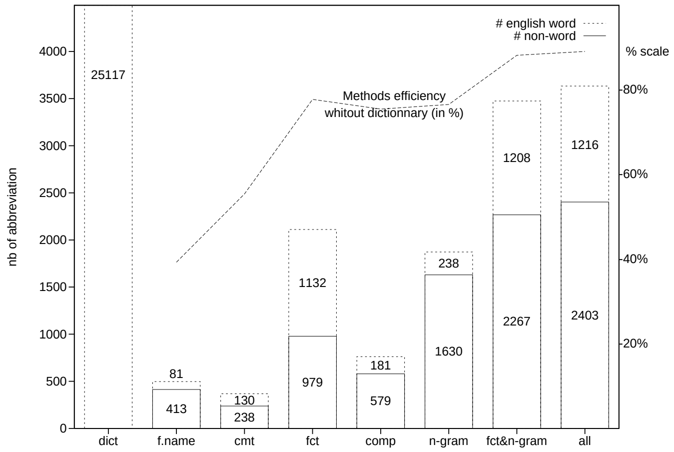

Figure 3: Influence of the number of English words and non-word on the efficiency of the methods. The methods are: (dict) abbreviations from English dictionary, (f.name) from file names, (cmt) from comments, (fct) from functions, (comp) computed abbreviated forms, (n-gram) substrings common to names.

For the 221 names, we accepted 256 splits. Only 31 names have more than one accepted split. The maximum is four accepted splits (for one name), there are two names with three accepted splits (like "activity") and 28 names have two accepted splits.

Using this sample, we measure the efficiency of a source of abbreviations by taking the best split it can produce for each name and counting how many correct abbreviations it could find.

For example if the proposed split for "listdbg" is "list – db – g", we will give it a score of 50% because it found half of the two right abbreviations ("list" and "dbg").

When we accept several splits for a name, we take the best result.

It may happen that the splitting tool returns several splits for one name (ties). In this case, we take the best of the ties' scores.

The final result for each method is the average percentage for the 221 names.

6.2 Experiment Results

The results are presented in figure 2. Efficiency is measured as a percentage of abbreviations found.

The height of a column gives the total number of abbreviations the corresponding method proposes.

The first source is the *English dictionary* (1st column: "dict"; see §4.6). It gives poor result, but we mentioned it was only intended as an improvement over the other methods. For all other sources, the lower result in figure 2 (plain box) is the efficiency of the source alone and the upper result (dashed box) is the efficiency when combined with the English dictionary.

Also, to try to understand a bit more what is happening, we give in figure 3 the number of English words (dashed box) and non-words (plain box) each method extracted. The non-words may be application domain terms or abbreviated forms of a word.

In this second figure we also give a reminder of the efficiency of each method used alone, without the auxiliary dictionary.

In both graphics, the methods are presented in ascending efficiency order, but we will discuss them in the order they were introduced (in section §4) which is more natural.

It may appear that *file names* (2nd column: "f.name"; see §4.1) are a poor source for abbreviations. Taken alone it only scores 39.2%. When combined with an English dictionary, it gives better results (67.1%) but still less than other methods. We will see however that "n-gram" also based on file names (see below) but using a different extracting method gives better results.

Looking at the second figure, one can see this first source"f.name" gives very few abbreviations which are English words. This may be an explanation for its bad score. But we also see that extracting a lot fewer abbreviations and barely more English words, "cmt" scores a lot better. It definitely seems the method measured here has poor precision and completion.

Common substrings's result (6th column: "n-gram"; see §4.2) comes as a surprise. We did not think a simple statistical method would score so well. The result without dictionary is comparable to the best "intelligent" method. It is significantly improved by the addition of the dictionary.

However, we must consider more closely this improvement. As the abbreviations proposed by the method are all substrings common to some names, the improvement may only come from words that are used in a single name. For this reason they are less interesting because they won't help gathering related file names together. They may, nevertheless, help differentiate unrelated names.

Comments (3rd column: "cmt"; see §4.3) constitute a filter over "n-gram". One can see they do propose many fewer candidate abbreviations ($\simeq$ 20%) and still give a relatively good result. We definitely improved the precision of the candidate abbreviations set. Unfortunately it appears to have been done at the expense of completion.

We proposed a way to improve results of the comments by trying to compute some abbreviated forms (5th column: "comp"; §4.4). The method definitely proves worthy. However simple the rules were, efficiency is among the best, and is almost the same as the original method ("n-gram") with only half the number of abbreviations. This seems to support the analysis we made: comments are useful to extract full words and application domain concepts, whereas computing abbreviated forms will provide the other abbreviations. One can note that between comment and computed abbreviated forms, the number of English words is multiplied by only 1.4 whereas the number of non English words is multiplied by 2.4.

Abbreviations from *function names* (4th column: "fct"; §4.5) give very good results, either alone or with a dictionary. The effect of the dictionary becomes less noticeable. It seems natural that with better and better results, it becomes more and more difficult to get significant improvements. Another explanation is that function names already contain many words, therefore, the addition of a dictionary does not have the same importance.

Finally the last two results are a combination of several sources, first the two best single ones ("fct" and "n-gram") and then all of them. The final best results (90%) are very good and should prove difficult to improve. These two results are somehow deceiving. We hoped that combining several methods would noticeably improve the score. Here we only gain 5%, thus it may not be worth the extra work to compute all the different methods. A better solution seems to concentrate on one of the two best ("fct" or "n-gram") and try to find another way to improve the result.

7 CONCLUSION AND FUTURE WORK

Discovering subsystems in a legacy software system is an important research issue. While studying a legacy telecommunication software system, and the software engineers who maintain it, we discovered their definition of subsystems was mainly based on the files' names. This goes against the commonly accepted idea that the body of the source code is the sole reliable source of information when performing file clustering.

Following these results, we wanted to build a conceptual browser on the software system. We believe extracting concepts from names could greatly contribute to the design recovery activity by providing information of a higher abstraction level.

For this we need to extract the constituent abbreviations of file names. This is a difficult task because it requires a lot of knowledge to find the concepts file names refer to, and intuition to associate abbreviated forms to the concepts.

In this paper we described an experiment to compare several methods of extracting abbreviations.

The overall strategy we chose seems appropriate as it allows us to get good results (90% of correct decompositions). It consist of first trying to extract probable abbreviations and then trying to decompose file names according to the abbreviations extracted.

From all the methods we proposed, two definitely prove better:

- Extracting abbreviations from *function names.* The great appeal of this method is its extreme simplicity.
- Extracting *common substrings* in file names. This is the best of all, but it may be difficult to compute for large software systems.

The results these methods achieve are very good (80% to 85% of abbreviations found when used with an English dictionary). They should prove difficult to improve.

Some simple steps may be taken:

- Recompute the common substrings with 2-grams. This would give us access to the 2-character abbreviations which we don't have right now.
- Improved the ranking function in the splitting tool (second stage of our strategy).
- Assess more precisely the precision and completion of each method. Up to now we have only been

able to estimate them. Having a precise comparison scale should help improve the methods. The problem is it seems to require manually computing all files exact decomposition.

However, we think that significant improvement may only come from a quite different direction:

- It may be a good time to reconsider the overall strategy, and taking advantage of the good results we obtained, to try to iteratively improve them.
- We plan to experiment with a method we call the "file names evolution tree". New file names are often derived from already existing ones (e.g. "activmonr" derived from "activmon"). By finding from which other name a file name is derived, it could be possible to deduce more easily what abbreviations it is made of.

We are mostly able to extract the abbreviations from the names. If we actually want to help in the design recovery activity and provide information on a higher abstraction level, we should soon try to associate the abbreviations with concepts. Another important step will be to combine our methods with more traditional, code based, file clustering methods.

THANKS

The authors would like to thank the software engineers at Mitel who participated in this research by providing us with data to study and by discussing ideas with us. We are also indebted to all the members of the KBRE research group Jelber Sayyad-Shirabad, Janice Singer an Stéphane Somé for fruitful discussions we have had with them.

The authors can be reached by email at `anquetil@csi.uottawa.ca` and `tcl@site.uottawa.ca`. The URL for the project is `http://www.csi.uottawa.ca/~tcl/kbre`.

REFERENCES

[1] Nicolas Anquetil and Timothy Lethbridge. File Clustering Using Naming Conventions for Legacy Systems. In *CASCON'97*. IBM Centre for Advanced Studies, nov 1997. Accepted for publication.

[2] Maintained by Chris Buckley. Smart v11.0. available via anonymous ftp from `ftp.cs.cornell.edu`, in `pub/smart/smart.11.0.tar.Z`.

[3] Robert Godin and Hafedh Mili. Building and Maintaining Analysis-Level Class Hierarchies Using Galois Lattices. *ACM SIGplan Notices*, 28(10):394–410, 1993. OOPSLA'93 Proceedings.

[4] IEEE. *5th International Workshop on Program Comprehension*. IEEE comp. soc. press, may 1997.

[5] Roy E. Kimbrell. Searching for text? send an n-gram! *Byte*, 13(5):197, may 1988.

[6] DEBIAN Gnu/Linux web page. `http://www.debian.org`.

[7] Etore Merlo, Ian McAdam, and Renato De Mori. Source code informal information analysis using connectionnist models. In Ruzena Bajcsy, editor, *IJCAI'93, International Joint Conference on Artificial Intelligence*, volume 2, pages 1339–44. Los Altos, Calif., 1993.

[8] 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 Comp. Soc. Press, mai 1997.

[9] Scott R. Tilley, Santanu Paul, and Dennis B. Smith. Towards a Framework for Program Understanding. In *Fourth Workshp on Program Comprehension*, pages 19–28. IEEE Comp. Soc. Press, mar 1996.

[10] Vassilios Tzerpos and Ric C. Holt. The Orphan Adoption Problem in Architecture Maintenance. Submited to Working Conference on Reverse Engineering, oct 1997.

[11] R. Wille. Restructuring Lattice Theory: an Approach Based on Hierarchies of Concepts. In I. Rival, editor, *Ordered Sets*, pages 445–70. Reidel, Dordrecht-Boston, 1982.