File Clustering Using Naming Conventions for Legacy Systems*

Nicolas Anquetil and Timothy Lethbridge

School of Information Technology and Engineering (SITE)
150 Louis Pasteur, University of Ottawa
Canada
{anquetil,tcl}@csi.uottawa.ca

Abstract

Decomposing complex software systems into conceptually independent subsystems represents a significant software engineering activity that receives considerable research attention. Most of the research in this domain deals with the source code; trying to cluster together files which are conceptually related. In this paper we propose using a more informal source of information: file names.

We present an experiment which shows that file naming convention is the best file clustering criteria for the software system we are studying.

Based on the experiment results, we also sketch a method to build a conceptual browser on a software system.

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 the software engineers to concentrate on the part of the system they are interested in, it provides a high level view of the system, it allows to relate the code to application domain concepts.

Our research is part of a project that aims to develop tools to help software engineers more effectively maintain software. An important requirement of this project is that we try to work in close relationship with the people actually maintaining software in order to uncover ideas that will be readily applicable in the industry.

Much of the research in the file clustering domain deals with the source code; trying to extract design concepts 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.

To set the context of this research, we will first give an overview of the project we are engaged in.

In a second stage, we will review some clustering criterion we have identified in the literature. We will compare them to the one we chose: file naming convention.

Part of the comparison is based on experiments we conducted with our particular legacy system. We will present the experiments and discuss their results.

Finally we will sketch a method to build a conceptual browser that uses file naming convention as a file clustering criterion.

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

1 The project

"Does the method scale up?" is a recurring question about software reverse-engineering research. This is also true in the file clustering domain. A key aspect of our project is to deal with an actual software system in a real world company, to ensure that scaling up is not a problem.

Our goal is to help software engineers to maintain a legacy telecommunication system. As is the rule in this field, we face many difficulties:

- It is a very large system ($\simeq$ 3500 files, 1.5 million lines of code).
- It is an old system (over 15 years old).
- The code has undergone many modifications and still evolves.
- The software is a main revenue source for the company and therefore is of the utmost importance.
- Nobody has a global understanding of the system (mainly because of its size).
- The software engineers are very busy and cannot be distracted from their maintenance task for extended periods of time.

A project goal is to design an environment that will help software engineers to browse the code, find closely related files or understand specific parts of the system.

The experiment reported in this paper is mainly concerned with file clustering, an activity which consists of breaking a large set of files (the entire software system) down into coherent subsets. These subsets are called subsystems. Identifying subsystems is an important part of the software design process. In reverse-engineering, being able to identify subsystems is a necessary step towards design recovery.

File clustering is a popular research domain; however the conditions associated with this project create a set of particular constraints:

- Because the system is large, we cannot engage in lengthy analyses of source code.
- Because the software engineers are busy people, we cannot afford to disturb them too often.
- Because we are dealing with a real world system, we must cope with the usual difficulties associated to real world problems (e.g. size, noise in the data).

On the other hand, because we aim at building a browser, 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.

2 Criteria for file clustering

When dealing with legacy system, one usually makes strong assumptions about the existing constraints:

- external documentation is nonexistent,
- comments are obsolete and misguiding,
- software design information is not available.

Given this, there are two possible approaches to subsystem discovery (see for example [**?**]):

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

The bottom-up approach consists of analyzing the code and trying to cluster the parts that pertain to the same concepts.

It is generally admitted that a successful solution should use both approaches. However the top-down one is very difficult. It is costly because it implies "extracting" domain knowledge from experts during long interviews. It is also domain specific which limits the potential for reuse.

We propose a bottom-up approach that could help in building a model of the application domain (usual outcome of the top-down approach).

Considering the constraints listed above, the natural way to perform the bottom-up approach is to look at the very material software engineers deal with every day: the source code.

However, we propose to use another source of information: file names. We shall explain why we made this choice and discuss its potential advantages and drawbacks.

We will also compare it to other file clustering criteria, including source code.

2.1 Relevance of the file name criterion

One of the primary goals of our project is to bridge the gap between academic research and industrial needs. As such, before trying to extract subsystems, we wanted to get a better idea what the software engineers considered a subsystem. We asked them to give us examples of subsystems they were familiar with. Four software engineers provided us with 10 subsystems covering 140 files. The experience of the software engineers with the system ranges from a few months to several years.

Studying each subsystem, it was obvious that their 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.

This conclusion goes against the common assumption that source code is the only relevant source of information. Because several software engineers contributed to this small sampling, it seems very likely that this property applies to the whole system.

We thus decided to experiment with file names as our file clustering criterion.

One can make several objections to this choice:

1. If file names are actually marking subsystems in the software we are studying, is there any chance to generalize the results to other systems?

2. If the subsystems are marked using file names, they must be well known. Therefore, why should we need a tool to extract them?

3. If a file name is representative of its original subsystem (when it was created), does this necessarily mean its contents still logically belongs in that subsystem after extensive maintenance has been applied?

First, we do not think that the system we are working on is such an exception. In a recent paper, Tzerpos [?] also noted 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)".

Merlo et al. [?] also noticed that routine names could be related to application domain concepts. This work will be mentioned in the next section.

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. We do not pretend file naming convention is the sole possibility, but it is one of them. Hierarchical directories is another commonly used one (e.g. in the Linux project [?]).

To the second objection, we may answer that one of the reasons the company is sponsoring our research is that nobody in the company has a complete understanding of the system. However, not being aware of all the existing subsystems, software engineers are still able to generate file names that respect some informal naming convention. For this, they only need *local* understanding of the system structure. A file name is derived from already existing files that are close to the new one. For example, when a file "activmon" was split in two, the second part (new file) was named "activmonr". Other similar examples may be found.

Finally maintenance can cause a file to drift away from its original purpose. But this can only be a very slow process and it will mainly be perceptible with regard to fine details. The high level abstract view of the file should remain relevant.

Because file names are short ($\leq$ 9 characters in our case), they may express very little information. If this information describes the purpose of the file, it may only do it at a very

high level of abstraction. Therefore file names may express only very generic concepts, which barely change over the time.

Using file names as file clustering criterion also brings an advantage that source code does not have: The clusters they allow one to create are readily understandable by the software engineers, one can name clusters after those substrings of file names that cluster members have in common: "q2000", "2ks" or "mnms" refer to application domain concepts that they can understand.

2.2 Other file clustering criteria

In last International Workshop on Program Comprehension [?] only one paper on program comprehension (out of nine papers) was based on other source than code analysis: it used the comments. The paper was concerned with building an application domain knowledge base [?].

The reason for this predominance of source code over other sources of information is that one usually assumes that it is the only reliable one. Documentation, either internal (comments) or external is usually considered out-of date or lacking.

But source code provides information of low abstraction level. This makes it mandatory to involve a human expert in the process; for example to filter the candidate clusters or to associate these clusters with domain concepts.

Very few researchers have considered or proposed using other sources of information:

- Merlo [?] states that "Many sources of information may need to be used during the [design] recovery process". He used comments and the mnemonics of identifiers names to extract application domain concepts.

- Tzerpos [?] identifies file naming convention as a possible criterion for file clustering. However he does not propose any solution to do so.

- Sayyad-Shirabad [?] uses comments to extract application domain concepts to build a knowledge base.

To our knowledge, documentation (other than comments) has never been used, but it could be considered as well.

The advantages of these informal sources of information over source code would be:

- They refer to application domain concepts in a more direct way because they contain words or abbreviations that are readily understandable by domain experts.

- They are of a higher abstraction level.

Before experimenting with the above mentioned criteria, we will briefly discuss their relative strengths and weaknesses as compared with file names:

Identifier names (used by Merlo [?]) are useful to extract application domain concepts, but our experiments show poor results for file clustering.

It must be noted that identifier names are more easy to deal with than file names because they may be longer and also allow the use of "words markers" ("_" or capital letters). We shall come back to this problem in section §4.1.

Hierarchical directories are also easier to deal with because concepts correspond to individual levels in the hierarchy. A limitation is that directories form a tree, whereas file names may refer to several independent concepts, possibly forming a general graph of concepts.

Comments (used by Merlo [?] and Sayyad-Shirabad [?]) may pertain to different parts of the code. A first difficulty consists in identifying comments pertaining to a whole file, or to a particular routine, type definition, etc. Comments also have the same problem as any free style (in natural language) documents: nouns have synonyms, verbs are conjugated, etc.

3 Experiment with file clustering criteria

To provide a more objective comparison of the file clustering criteria, we measured their efficiency in retrieving the example subsystems already mentioned.

Before proposing an experiment design to compare the criteria, we will explain how we have been using the file name criterion.

Finally we will present and discuss the experiments' results.

3.1 Using the file name criterion

Tzerpos [?] suggested using file names as a clustering criterion, however he did not implement the idea. Merlo [?] did experiment with identifiers names, but we already explained that they are much easier to deal with because they contain "word markers". These markers are rarer in file names: in our system, only 19 files out of 3500 contained the "_" character and none contain upper case letters. Because nothing indicates that "listaud" should be decomposed in "list" and "aud" (for audit)—, we need a way to guess it or to work around the problem.

We call the part of a file name that refers to a concept ("list" or "aud") *abbreviations*, even though some of them are full words.

Tzerpos proposes to use a simple pattern matcher, but this implies we know what abbreviations to look for. This is not our case. Instead, we propose to work around the problem by relying on statistical techniques.

Our method consists in decomposing the file names into all the substrings of a given length they contain. These strings are called n-grams [?]; for a length of 3, one speaks of 3-grams. For example the file name "qlistmgr" contains the following 3-grams: "qli", "lis", "ist", "stm", "tmg" and "mgr".

N-grams are all the substrings of a given length that can be extracted from a name. Abbreviations are part of the names that relate to a specific concept. The length of abbreviations is not fixed, although with our method, we cannot extract abbreviations shorter than the n-grams we are working with (see below).

Two file names sharing some abbreviations (e.g. "q<u>list</u>mgr" and "<u>list</u>aud") will have one or more n-grams in common (here "lis" and "ist"). The more n-grams two file names have in common, the closer they are.

The length of the n-grams must be chose carefully. Excessive length will forbid short abbreviations from coming out and excessive brevity will mean the n-grams will not be significant enough.

For example for a length of 5 characters (too long), abbreviations up to 4 characters will produce no results: "qlistmgr" has the 5-grams: "qlist", "listm", "istmg", "stmgr" and "listaud" has: "lista", "istau", "staud". The two files no longer have any n-gram in common.

With 2-grams (too short) "tm" will be common to the names "listmgr" and "initmem" which is only fortuitous.

The right length would be the same as that of the most common abbreviations. In our case, abbreviations are mainly of 2, 3 and 4 characters. There appears to be a significant number of 2-characters abbreviations, although we have no precise statistics on this. Nevertheless, decomposing the file names into 3-grams gave good results (see section §3.3).

From discussions with the software engineers it also appeared that the position of an abbreviation in the file name have some significance. To try to cope with this issue, we added two virtual letters, at the beginning ("^") and the end ("$") of the names. This proved useful and improved the results. It also has the interesting outcome that some 2-characters abbreviations are taken into account if they appear at the beginning or the end of file names. For example, the abbreviation "cp" often appears at the beginning of file names, therefore, the 3-gram "^cp" is shared by all these names.

3.2 Experiments design

We will now describe the experiment we designed to compare all the file clustering criteria.

The first step was to try to define what the software engineers considered a subsystem. Four software engineers proposed 10 subsystems containing 140 files. The subsystems are small, their size varied from 5 files to 40 files.

The experiment consists of trying to generate some subsystems and compare them with the examples we were provided with. We will prefer the criterion that allows us to extract subsystem closest to the examples.

For each of the 140 files:

1. Use the selected file as the "seed".
2. Find all other files close enough to the seed (according to the tested criterion) to belong to the same subsystem.
3. Compare this set of files with the actual subsystem.

This algorithm is similar to what is done in Information Retrieval [**?**, **?**]: given a base of "documents" (the system's files), find all the documents (a subset of the files) relevant to a query (the seed file).

Although it was for a different purpose, we should note that Information Retrieval techniques have already been used in software engineering by Patel [**?**]. His purpose was not to compare criteria but to define a module cohesion measure.

One measures the efficiency of Information Retrieval techniques by comparing the results they give to known queries with known results. The efficiency is measured using two metrics: precision and recall.

Recall is the percentage of relevant files (subsystem's actual files) that the system did extract.

$$\text{recall} = \frac{\text{retrieved members}}{\text{actual members}} \times 100$$

Precision is the percentage of relevant files among the ones retrieved by the system.

$$\text{precision} = \frac{\text{retrieved members}}{\text{total retrieved}} \times 100$$

A recall of 100% is easily achieved by retrieving all files (thus all the subsystem's actual files will be extracted), but the precision will be very poor. Conversely, by extracting only the "seed", the precision rate will be 100% (the file belongs to its own subsystem), but the recall will be correspondingly low. One must look for the right balance between high precision and high recall. Presumably, we should favor recall because it is more important to present the software engineers all the potentially useful files (even if there is some noise) rather than presenting only those we are sure are useful files but missing some of them.

We used Smart [**?**], an Information Retrieval tool designed for such experiments. The tool *ranks* files according to their similarity to the seed. Doing so, it does not actually define subsystems, for one seed it would potentially extract all the files, only a lot of them would have a null similarity. One way to actually define a subsystem, is to decide on a threshold. Only files ranked higher than the threshold will be in the subsystem.

However, we did not want to use an arbitrary algorithm to set the boundaries of each subsystem because it could introduce noise into the measures.

We grouped all files with the same name but different extensions (e.g. ".pas", ".if" in our case, ".c" ".h" for C files) into one virtual file. For some criteria, this was mandatory, for example for the "included files" criterion (see below), only header files (".h" in C) contain the information. This also has the advantage of reducing the number of files (from 3500 to 1800).

The criteria we tested are the following:

File names: Each file name is indexed with the n-gram it contains as explained in section §3.1. We experimented with n-grams of length 2, 3, 4 and 5.

Routine names: As a mean of comparison with Merlo's work [**?**], we decomposed the names of routines declared in each file according to the word markers they contain ("_" and capital letters). We clustered together the files containing the same abbreviations thus obtained.

Included files: This is our "source code" criterion. In [**?**], Patel proposes to cluster together routines that refer to the same types. We tried to do the same thing on the file level.

All comments: Files are compared according to their comments. The smart system offers mechanisms to deal with free style text. It may discard the common words

(adverbs, articles), and try to put other words in a canonical form (for example by suppressing "s" and "ing" at the end of words).

"Summary" comment: Many files ($\simeq$ 70% of all files and $\simeq$ 84% of the known subsystems' files) have a summary comment at the beginning of the file that describes the main purpose of the file as opposed to that of routines.

References in documentation: We extracted all the file names referred to in the documentation. Each file is indexed with all the documents that refer to it.

3.3 Results

Figure 1 presents the results for all criteria experimented with. For each criterion, the curve gives different recall rates and the corresponding precision rates.

As we want to promote recall over precision (see §3.2), a good indicator of a criterion's efficiency would be the precision rate for 100% recall (right hand side of the graph).

The file name criterion with 3-grams ends up with a precision of 73.1%. It means that given a subsystem of 10 files, on average, these 10 files are ranked among the 14 most similar to any of them. We consider this result satisfactory enough.

The "Included files" criterion ends up with 15.4% precision. This means that on average, the 10 files would be ranked among the 65 most similar to any of them. This is not acceptable, although one could consider that looking at 65 files out of 3500 is not so bad a result.

It is no surprise that the file name criterion is the best one. We already noticed that the subsystems showed some similarity between their file names.

Among the different n-gram for the file name criterion, length 3 is the best one as already mentioned (final precision = 73.1%).

Length 4 also gives good results (final precision = 69.3%) although we know there are many 3 and 2-characters abbreviations in the file names. This may be a particularity of our sample which contains only two small subsystems with 2-characters abbreviations, and one small subsystem with 3-characters abbreviations. Thus more than 78% of the files experimented with have abbreviations of 4 or more characters.

As expected, 5-grams give very poor results (final precision = 3%).

Results for 2-grams came as a surprise (final precision = 64.5%) Although our subsystem sampling tends to favor long abbreviations, the results are quite good (on average, 10 files would be ranked in the 16 most similar to any of them). This would suggest that noise introduced by the inevitable fortuitous similarities when using so short n-grams is not that important.

The other criteria are by far worst than file name. One should take some precautions in interpreting these results, there may be several valid structures for the system. Our sample favors the file name criterion (because software engineers do so), this does not mean it is the only valid one.

The source code criterion (indexing on included files) is the least bad of the alternatives (final precision = 15.4%).

Documentation is the next one (final precision = 6.6%). This could be blamed on the general nature of the documentation used. Another experiment should be made with only those documents concerned with the design structure.

Abbreviations found in routine names have a final precision of 5.5%. Maybe this is to blame on the difference of abstraction level between the files we clustered and the routines these abbreviations describe.

Finally, results for the comments are similar, whether we considered them all or only the file "summary comment" (respective final precisions 3% and 2.6%).

4 A "conceptual" browser

The method we used to compare file clustering criteria did not actually extract subsystems. Rather for each seed file it was given, it proposed a (potentially very long) list of files ranked by similarity with the seed.

But our goal was to built a browser based on the subsystems extracted. This section

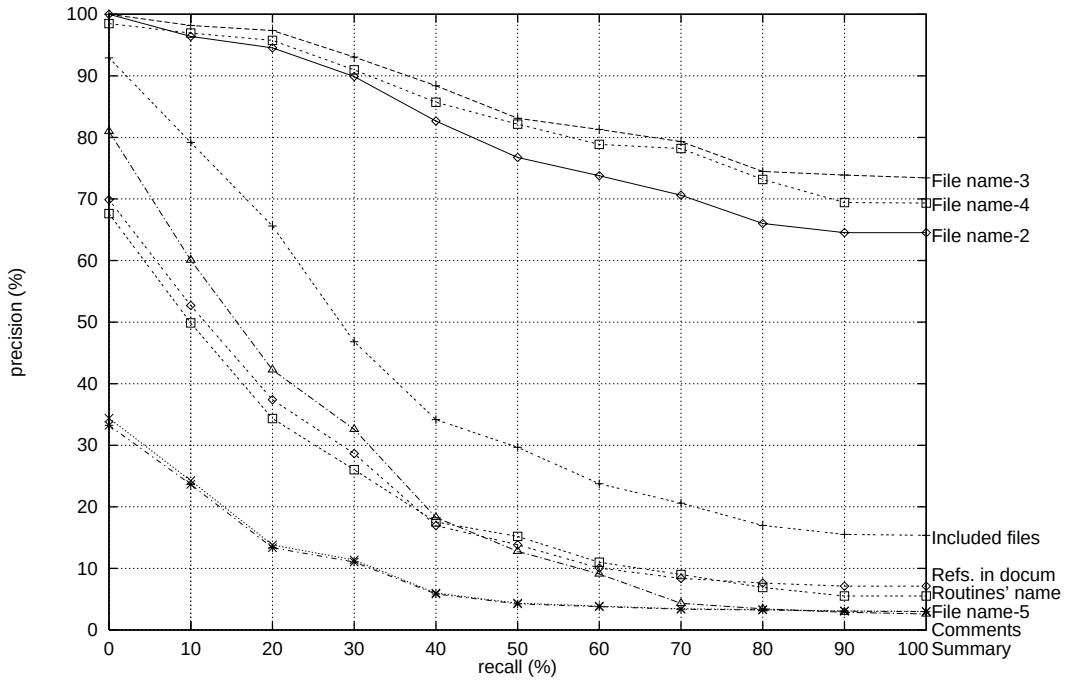

Figure 1: Recall and precision rates for various file clustering criteria

presents an early work in this direction. This work is by no means completed, and some questions remain open. However, the early results are encouraging.

A straightforward solution would have been to cluster together all files sharing some common n-gram(s). Each cluster would have define a subsystem named with the abbreviation common to all of its member files. The subsystem and the name can be considered to form a concept.

However experience shows this method gives many meaningless "concepts" because there are many fortuitous n-grams appearing in different file names. For example all the names containing "diag" (diagnosis) and "dial" share the 3-gram "dia" which means nothing for the software engineers.

Other candidate abbreviations can be misleading, for example, "clock" and "block" are two abbreviations found in file names, but their intersection, "lock" is not a concept referred to. This example is more dangerous than the first one because the candidate abbreviation is a word and therefore seems to be valid.

The first step in building the conceptual browser will be to filter the substrings.

The second step is to assign to each file all the concepts its name contains. This may be a difficult task; for example assuming "lock" was a valid concept, we would not want to assign it to a file containing the "clock" concept.

4.1 Abbreviation filtering

The first step to get a list of candidate abbreviations is to extract all the n-grams common to several file names. We will then try to filter out those candidate abbreviations which do not correspond to concepts.

There are several algorithms that could cluster the files names and exhibit the candidate abbreviations they have in common. We chose an algorithm from Godin (see for example [?, ?]) to built a Galois lattice. This algorithm clusters together all the file names which share some n-grams. This structure has the property that all possible clusters are extracted. This would not be the case for most of the clustering algorithms. Because they build trees, they

have to select only part of the possible clusters and discard the others ([?, ?]).

Using the Galois lattice, we are able to extract all groups of names possessing some similar n-gram. A first filtering process discard those clusters that do not represent a single abbreviation:

- All the n-grams may not be connected, for example: "memmonstr" and "monpastra" form a cluster with the 3-grams: "mon" and "str" but do not share a single abbreviation.

- All the n-grams may be connected differently in the file names, for example "gwaititem" and "partition" form a cluster with the 3-grams: "iti" and "tit" but it is "itit" in one file and "titi" in the other.

A second filtering process will try to discard those candidate abbreviations which are not actual concepts. Our solution is to look for each candidate abbreviation in a "dictionnary", the problem being to find a dictionary containing the application domain concepts and the particular mnemonics used in the company.

For this, we propose to use the comments. For each cluster, we look for the associated candidate abbreviation in the comments. We only keep those abbreviations that we found in the comments. To avoid going through megabytes of comments for each candidate abbreviation, we only search the comments in the members of the given cluster.

This allows us to solve such problems as the two presented earlier. No file containing the "dia" substring in its name had it in its comments, whereas "dial" and "diag" where found, same thing for "lock" and "block" or "clock".

Numerous other abbreviations specific to the company are found in the comments such as "q2000", "cp", etc. Unfortunately, others, like "svr" for "server", "mgr" for "manager" etc. do not appear in the comments, instead, the full word is used. We tried to look for them among the abbreviations used in identifiers, but the results are not satisfactory.

From 1808 file names, the Galois lattice extracted 3410 clusters. Among them, 667 were discarded during the first filtering process (no single abbreviations in a cluster), leaving 2743 candidate abbreviations.

The second filtering process (searching for the candidate abbreviations in a "dictionnary") found 501 in the comments and 327 in the identifier names, 279 being found in both. Which leaves us with 549 recognized abbreviations.

4.2 File name decomposition

The reason for extracting abbreviations was to understand the file names: "actmnutsg" means "act" (for activity), "mn" (for monitor), "ut" (for utilities) and "sg" (a product name).

We call the splitting of a file name in all the abbreviations it contains a *split*.

File names rarely contain word markers therefore, we have to guess how a name like "actmnutsg" must be split. The task is rendered more difficult by the following facts:

- one concept may be abbreviated in different ways: "activity", "activ" and "act" or "mon" and "mn" for monitor.

- a short abbreviation (e.g. "lock") may appear inside another longer one (e.g. "clock") or between two abbreviations.

The ideal solution would be to have an extensive dictionary of all abbreviations. Given this, we believe it would be possible to find a unique correct decomposition for the great majority of file names.

However, building such a dictionary would be a difficult task, comparable to building an application domain knowledge base. Maintaining this dictionary would be a hard task as well. We chose a more lightweight solution where the work is done automatically, and we accept the possibility of errors. These errors or of two kinds:

- we split the file name using a wrong abbreviation, for example in "actmnutsg", we could detect "mnu", abbreviation which means "menu" whereas it's in fact "mn" and "ut"

- we miss an abbreviation, like "mn" in the above example.

The first problem does not seem critical because while browsing the system the software engineers will be able to correct these errors easily. If the system present them with 10 files (out of 1800), it's an easy task for them to detect the few wrong ones.

The second problem seems more serious because it could cause the software engineers to miss a file (possibly precisely the one with a bug in it). To solve this we decided to propose more than one possible split of a file name, therefore increasing a bit the first problem, but we saw it is not critical.

Our algorithm for splitting the file names consists of generating all possible splits with the abbreviations. All remaining characters (not belonging to any abbreviation) are left alone (free characters). For example, assuming we recognize "act", "mnu" and/or "ut", the possible splits for "actmnutsg" will be:

- "act", "mnu", "t", "s" and "g"
- "act", "m", "n", "ut", "s" and "g"

We then rate the splits according to some function. Hopefully, the best rating corresponds to the correct split.

We first tried to generate splits only considering the recognized abbreviations (both from the comments and the identifiers names). But too few recognized abbreviations are available and the results did not satisfied us.

We opted for another solution which consists of generating the splits with all the candidate abbreviations (recognized or not), aware that a lot of them are erroneous. The splits with more recognized abbreviations (from comments or routine names) get a higher rating.

The rating function includes several criteria:

- Number of abbreviation in the split, the less, the better. The rationale behind this is to discard those splits with a lot of free characters.
- Proportion of recognized abbreviations in a split, the higher, the better.
- Source for recognized abbreviation. This is intended as an improvement upon the last criterion. One can put different weights on comments and routine names. In our case, we were also recognizing abbreviations which were file names by themselves. But we gave a smaller weight to this source because we did not want to accept splits consisting of only 1 abbreviation (the file name itself). A tendency that is already enforced by the first criterion.

To try to give the reader an objective idea of the results, we present, in table 1, splits for the first 10 files in alphabetical order . The table gives the two best rated splits for each file along with the correct one.

Usually when there is a recognized abbreviation, it appears in one of the two best rated split (generally in the first one). Errors arise because we do not have enough recognized abbreviations, which is due to the following problems:

- Because we used 3-grams, all 2 characters long abbreviations are not extracted and therefore cannot be subsequently recognized.
- A lot of correct abbreviations are still not recognized.

The solution to the first problem is to use 2-grams instead of 3-grams. This should work as we saw in section §3.3 that 2-grams give relatively good results.

A solution to the second problem could be to add a new filtering of the candidate abbreviations after the splitting. In table 1, we see that "activmon" is splitted correctly even though "activ" is not a recognized abbreviations. After the split, we could add "activ" as a new recognized abbreviation, and do the split again. Thus, by iterating over splitting and filtering, we may be able to improve our results.

Conclusion and future work

Discovering subsystems in a legacy software system is an important research issue. Most of the research tries to do so by analysing the source code because it is considered the only relevant source of information on the actual state of the software.

file name	best split	2nd best split	correct split
abrvdial	ab rv dial	ab rv di al	abrv DIAL (abbreviated-dialing)
accessdta	access dta	acc es sd ta	ACCESS dta (accesses-resource-data)
accessfwd	access fwd	acc ess fwd	ACCESS FWD (access-call-forwarding-data)
accntcode	acc nt code	ac cnt code	accnt CODE (account-code)
activity	activ ity	activ it y	activity (activity-control)
activmon	activ mon	activ m on	activ MON (activity-monitor)
activmonr	act i vmo nr	activ mon r	activ MON r (activity-monitor-redundant-system)
actmnuts	act mnu ts	ac t mnu ts	ACT mn ut s (activity-monitor-utilities-s)
actmnutsg	actm nut sg	act mnu tsg	ACT mn ut SG (activity-monitor-utilities-sg)
actmonsrv	act mon srv	act monsrv	ACT MON srv (activity–monitor-server)

Table 1: Splits for the 10 first file names. Upper case abbreviations are the recognized ones. Note: "sg" and "s" are two product names.

One of the requirements of our project is to try to work in close relationship with the people that are actually maintaining legacy software. By studying them, we hope to discover new ideas that will be readily applicable in the industry.

Thus, while studying a telecommunication legacy software, 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 source code is the sole reliable source of information when doing file clustering.

We experimented some file clustering criteria and proved that the file name criterion was the most likely to rediscover what the software engineers called a subsystem.

Although this result could be a particularity of the software we are studying, we gave some reasons to believe that it could be more widely applicable.

Based on these results, we proposed a method to build a conceptual browser on the software system. The browser is based on splitting file names retrieved as members of subsystems into their logical components. This part of the research is still in an early stage and we proposed several possible improvements.

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 Janice Singer an Jelber Sayyad-Shirabad for fruitful discussions we had with them.

About the authors

Nicolas Anquetil recently completed is Ph.D. at the Université de Montréal and is now working as a research associate and part time professor at the university of Ottawa.

Timothy C. Lethbridge is an Assistant Professor in the newly-formed School of Information Technology and Engineering (SITE) at the University of Ottawa. He leads the Knowledge-Based Reverse Engineering group, which is one of the projects sponsored by the Consortium for Software Engineering Research.

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