Design Quality of Subsystems Extracted from File Names[*]

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

April 19, 1998

Abstract

There is a demand in software engineering for automatic design recovery tools. An important part of this activity consists of clustering files into subsystems representative of the "lost" design. Most of the research in this domain considers the body of the code; trying to cluster together files which are conceptually related.

We, on the other hand, are trying to automatically cluster files using their names. This source of information seems to better match the organization of the legacy system we are studying. We show that this informal source of information can give results comparable with what source code can provide.

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 recover the "lost" design structure of the software system using whatever source of information is available. An important part of this activity consists of clustering files into subsystems. This allows software engineers to concentrate on the parts of the system in which they are most interested, and provides a high level view of the system.

In general, existing research focuses on the source code as the place from which to extract design concepts. However, various researchers [15, 16] have raised an important question: Is it possible to actually lift the very low abstraction information available in the source code to the level of design?

We do not actually answer this question, rather, we take the opposite approach and try to show that one can extract design information from a more abstract source: file names. We evaluate the design quality of subsystems extracted from a legacy software system using file names; we show show that the quality is comparable to what an analysis of source code can provide.

We first give an overview of our project.

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

We explain how we were led to consider file names as a criterion for subsystem extraction. We also summarize and compare various methods of automatically clustering files based on their name.

In the second part of the paper, we measure the design quality of the subsystems extracted using two well accepted metrics: cohesion and coupling. We show that the design quality of the subsystems (in terms of cohesion and coupling) is dependent on the quality of the file name decomposition.

2 File clustering

We will now set the context in which our current experiments took place: How we were led to consider file names as a criterion for file clustering, how we perform the clustering and what are some earlier results.

2.1 Choice of file name source

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 large (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.

Our goal is to build a conceptual browser for this legacy software system. This activity implies clustering semantically related files. Files can be clustered along many dimensions, one of which is the design structure. One can try to recover the design structure using a top-down or a bottom-up approach ([13]). The top-down approach consists of analyzing the domain to discover its concepts and trying to match parts of the code with them. It is less popular, because it implies extracting knowledge from experts during long interviews and it is also domain specific which limits its potential for reuse.

The bottom-up approach consists of clustering closely related parts of the code and assuming they correspond to design concepts. The preferred way to perform this is to look at the code.

For example, in [5], Lakhotia lists 22 criteria for file clustering, among which all but one is based on code. Also, the 1995 Working Conference on Reverse Engineering had a special track entitled "Analysis of Non-Code Sources" (three papers); there has been no subsequent similar exercise.

However the assumption that code provides the only useful source of information for file clustering does not match our own experience. Before trying to cluster files, we asked the software engineers to give us examples of subsystems they were familiar with. Four software engineers provided us with 10 subsystems covering 68 files.

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 showed by experiment that the use of file names is actually the best criterion to recover the example subsystems [1].

In the context of a conceptual browser, files names offer many advantages over code:

- they are more concise and therefore allow more in-depth analysis,
- they are not limited to code files or can easily deal with different implementation languages,
- they are not limited to a particular design point of view of the system,

- they refer to application domain concepts in a more direct way; hence the clusters are readily understandable by the software engineers.

2.2 File name decomposition methods

Having decided to use file names to cluster files, we defined a method to automatically extract the concepts "hidden" in the file names. This is done in two steps which we will only summarize here. A more in depth description of this part may be found in [2].

We define an *abbreviation* to be any substring of a name 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).

As a rule, it is more difficult to find the abbreviations composing a file name than composing an identifier. File names rarely contain "word markers" (underscore characters, hyphens, or capital letters). Also the abbreviations they use are much shorter and more cryptic, some being only one or two characters long.

To stress the difficulty of the task, we wish to point out that for those who are not experts in the particular software system (such as ourselves), manually finding the right decomposition of certain file names implied looking at the contents of the files (their comments and identifier names), at the external documentation or at other related files.

We studied several sources and approaches for extracting abbreviations [2]. By taking comment words that are also substrings of file names, as well as all substrings common to several file names, we were able to find on average 88% of the abbreviations composing a file name.

This metric indicates how good we are in finding *all* the abbreviations composing a name. One would say that we have a good *recall* (on precision and recall, see for example [11]).

A simple way to achieve good recall consists of extracting all the possible substrings composing a name. But the result would be useless because we would not know which of the substrings are the right ones. Therefore, in order to have good *precision*, we also want to extract *only* the actual abbreviations (concepts) composing a name. Improving the precision is more difficult because it implies focusing on the right abbreviations whereas we don't know which ones are right. We were able to get a precision rate of 84%.

The next section presents various name decomposition methods, along with the measured recall and precision of these methods.

2.3 Methods' precision & recall

Starting from the best result we obtained (recall 88%, precision 84%) we tried to improve it by exploring alternatively in two directions:

- We tried to improve the precision by filtering out the abbreviations that did not seem right.
- We tried to improve the recall by "reincluding" abbreviations that could belong to a name but where discarded in the initial process.

Some of our results are given by the curve in figure 1; this shows a series of specific name decomposition methods (1 to 8). The initial (and best) result is point (1). Subsequent results, by themselves, are not very good; therefore we will not explain the details of each decomposition method. However, we would like to emphasize some points which may prove useful for subsequent discussions:

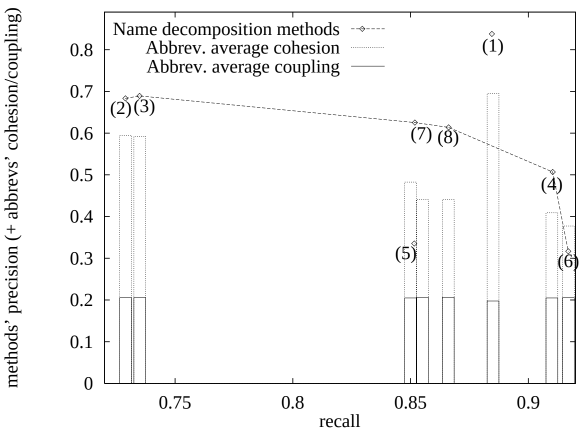

Figure 1: Precision and recall of different name decomposition methods (curve). The boxes give the respective average cohesion and coupling of the extracted abbreviations for each method. Cohesion and coupling for (5) are given by the left column, for (7) by the right column.

- (2) and (3) are derived from (1), by trying two slightly different ways of filtering out supposedly wrong abbreviations (to improve precision).
- (4) is also derived from (1) by trying to improve recall.
- (5) and (6) are derived from (3) by trying to improve recall.
- And finally, (7) and (8) are derived from (6) by filtering out "wrong" abbreviations (to improve precision).

3 Cohesion & coupling of abbreviations

Our goal was not to extract good "design" subsystems[1]. We were primarily interested in extracting meaningful "concepts" (i.e. abbreviations) and associating the right file names with them. However, it turned out that these concepts gave relatively good results with regard to cohesion and coupling.

We first restate some basic notions of the two metrics cohesion and coupling. Then we present and analyze our results. We will finish with a discussion on the correlation between the quality of the abbreviations automatically extracted as well as their average cohesion and coupling.

3.1 Cohesion & coupling

One can measure the design quality of modules by evaluating their *cohesion* and *coupling* (see for example [12]).

Cohesion measures the amount of interaction between the members of a module. A good module should exhibit high cohesion, which implies that there are a lot of interactions between its components. This ensures that the module performs only one activity and that all the components of a module are useful to it.

Coupling measures the amount of interaction between the members of a module and the outside. A good module should exhibit low coupling, which implies that there are few interactions between the components of the module and the outside (one can also say that the module has a small interface). This ensures that modifying the module will not have too much impact on the outside and vice-versa.

In our case, the modules for which we will measure cohesion and coupling are subsystems composed of a set of files. We used the cohesion and coupling metrics defined in [4, 10]:

[1]The notion of "subsystem" implicitly refers to the design, but the way we extract them introduces some confusion.

Cohesion is computed as the average *similarity* between all the files in the subsystem.

Coupling is computed as the average *similarity* between any file in the subsystem and any file outside it.

Similarity is computed as the Euclidean distance between characteristic vectors describing each file. A characteristic vector counts the references to a particular user defined type within the file it describes.

In the following experiments, each abbreviation is considered to define a subsystem. All the file names from which the abbreviation was extracted are member of the associated subsystem. Note that an abbreviation may be substring of a file name and yet not be extracted from that particular name. This can happen when the file name is broken down in such as way that the abbreviation in question is considered to be either part of a larger abbreviation, or else is split into two abbreviations.

3.2 Cohesion/coupling measured

Considering again figure 1; for each name decomposition method, the boxes show average cohesion and coupling for the subsystems extracted.

One can observe that the results are reasonably good. Most name decomposition methods exhibit an average cohesion significantly higher than the average coupling.

But these results are still quite far from what may be obtained using code as a source of information. In figure 2 we give the average cohesion/coupling of subsystems in a number of different cases:

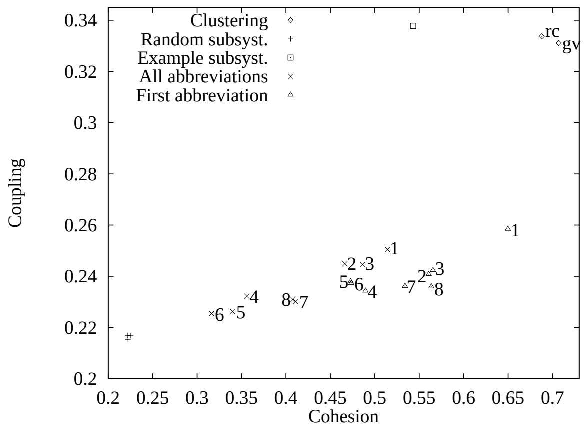

Figure 2: Comparing Cohesion/Coupling for all abbreviations of the names and for the first abbreviation (potentially marking the design subsystems). The numbers refer to the name decomposition methods in figure 1.

Code : Subsystems are obtained by clustering files which are most similar according to a code criterion. Because we did not want to give an unfair advantage to the code experiments, the similarity between files is computed using several different criteria for the cohesion/coupling metrics and the clustering. One experiment uses reference to global variables (labeled “gv”) and the other uses cross routine calls (labeled “rc”).

Example Subsystems : These are the subsystems given to us by the software engineers.

Random Subsystems : These are three sets of randomly generated subsystems. Each one contains 1000 subsystems with a size $\frac{x+100}{x+1}$ where x is a randomly generated number between 0 and 50. These figures have been chosen to approximately match the configuration of the *code* subsystems.

All abbreviations : These are the results already presented in figure 1. There is however a minor difference on the actual subsystems taken into account (see below).

It is somehow unfair to compute cohesion and coupling for all the abbreviations composing a name because not all of them mark design subsystems. For example, in the name "listdbg", the first abbreviation "list" marks the subsystem whereas the second one "dbg" (for debug) marks another concept.

This led to the last set of experiments:

First abbreviation : These are the very same name decomposition methods as the "all abbreviations" set, but considering only the first abbreviation extracted for each file name.

Note that the "first abbreviation" heuristic sometimes proves wrong. For example files "qlistmgr" and "flistaud" are also members of the "list" subsystem, but they have an extra initial letter which goes against the heuristic.

In one case, the "example subsystems", there are only 10 subsystems covering 68 files (out of more than 1800). To try to have a common base on which to compare all the results, we did not consider all subsystems in the other experiments. Rather, we only took into account those subsystems which included at least one "example file" (i.e. a file belonging to an example subsystem). The experiments are still dissimilar in that the other experiments include on average 76 subsystems covering 613 files.

Before commenting the results, we want to make a last point:

In the figure, the ideal point is the lower right corner, with high cohesion and low coupling. Please note that the scales are not the same for cohesion and coupling. If they were, the results would all appear closer to the ideal than they do here because they would all be located in the lower half of the graph.

From this figure, we see that:

- The *code* clustering methods (rc and gv) perform very well, although one could object that they yield a high degree of coupling. However, given our experiments, it seems to be intrinsic to the measures of our system that a better cohesion always goes with a slightly increased coupling.

  These sets of experiments are intended to give an "absolute upper bound" comparison base.

- The *example subsystems* also give good results. We already mentioned that these subsystems are clearly based on a file naming convention. The results are slightly worst than in the preceding case, since we have lower cohesion and higher coupling.

  One should not pay too much attention to the bad coupling for the *example subsystems*, the very small size of this experiment does not allow to draw any significant conclusion.

  This experiment is intended to give another "upper bound" comparison base, that focuses on the file naming convention.

- The *random subsystems* have, as one would expect, the same cohesion and coupling ($\simeq 0.215$) that corresponds to the average similarity of any two files in the system.

  This is considered to be a "lower bound" comparison base.

- The results for *all abbreviations* (e.g. name decomposition methods 1, 2 and 3) are significantly better than completely random experiments.

- Despite the problem we mentioned, the *first abbreviation* heuristic, definitely proves useful and gives better cohesion/coupling.

  In the case of name decomposition method 1, one could even argue that the results are comparable to the *code* results, considering that it is closer to the "ideal" point (lower coupling).

  The cohesion results for name decomposition methods 1, 2, 3, 7 and 8 are better than with the example subsystems, but we already mention that we can not actually deduce much from this.

It seems clear from these experiments that, in our system, file naming convention is a valid criterion to extract good design subsystems.

3.3 Cohesion/Coupling versus Precision/Recall

If we compare the precision and recall rates of each name decomposition method (figure 1) and the corresponding average cohesion and coupling of the extracted abbreviations (figure 2), one can observe some interesting facts:

- The best precision (name decomposition method 1) corresponds to the best average cohesion/coupling.

- Pairs of name decomposition methods 2/3 and 7/8 have close precision/recall and close cohesion/coupling.

- The pair 5/6 has close precision (not recall) and similarly close cohesion/coupling.

We are not sure yet if these paired similarities are due to some correlation between average cohesion/coupling of the abbreviations and the precision/recall of the name decomposition methods or if they are due to the fact that each pair includes very close methods. We mentioned earlier that 2 and 3 are derived from 1, while 5 and 6 are derived from 3 and 7 and 8 are derived from 6.

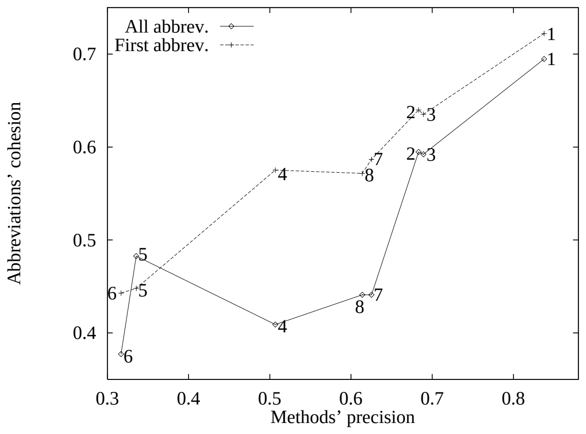

Figure 3: Precision of name decomposition methods versus cohesion of the abbreviations.

Some facts point toward a correlation between the precision of the name decomposition methods and the average cohesion of the abbreviations:

- In figure 1, the name decomposition methods in increasing order of precision are: 6, 5, 4, 8, 7, 2, 3 and 1. This is very similar to the order that may be found when looking at figure 2 (from the worst results on the left toward better ones on the right).

- The two metrics (cohesion and precision) are strongly correlated. The correlation coefficients from data in figure 3 are: 0.78 if we consider all abbreviations and 0.98 if we consider only the first abbreviation.

- The two name decomposition methods (1 and 4) which are not paired (approach different from the other), do fit in this correlation scheme.

For the reason we already mentioned, it seems natural that the "first abbreviation" results be better: all abbreviations do not refer to design subsystems.

This correlation between the cohesion of the abbreviations and the precision of the name decomposition methods extracting them re-enforce the idea that these abbreviations do mark design subsystems. Having more precise name decomposition methods implies extracting mostly good abbreviations (even if we must extract fewer of them). Assuming abbreviations do mark design subsystems, these good abbreviations in turn produce a better cohesion.

A better recall could not have this effect. This suggests that one should concentrate on having a good precision which is unfortunately more difficult.

4 Conclusion

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 code is the sole reliable source of information when performing file clustering.

This raises some issues that we will address now.

4.1 Generality of this organization scheme ?

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

We do not think that the system we are working on is such an exception. Other authors already noticed that file names can constitute good markers of design subsystems [9, 14]. However, in these two works, the file name decomposition was manual.

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 naming convention is the sole solution to file organization, but it is one possibility. Hierarchical directories is another commonly used approach (e.g. in the Linux project [6]).

Other works also use high abstraction level sources to cluster files [7, 8]. These sources (external documentation for [7], comment and identifier names for [8]) proved irrelevant in our system [1].

To our knowledge, none of these works tried to assert the design quality of the subsystem decomposition obtained.

4.2 Usefulness of the file name criterion ?

This is the major issue we wish to address: what can be the utility of a tool which only discovers subsystems that have been explicitly marked in the file names?

It is true that using file names we are only "re-discovering" subsystems that have been explicitly put there. This seems to be opposed to the usual approach which aims at recovering a lost design.

We believe that this vision is erroneous. For large systems with thousands of interactions between files, the GIGO[2] rule

[2]A computer, to print out a fact,
Will divide, multiply, and subtract.
But this output can be
No more than debris,
If the input was short of exact.
– Gigo (from the "fortune" program)

(Garbage-In, Garbage-Out) would certainly apply. That is to say if nobody explicitly and repeatedly tried to organize the files, there will be no subsystem discovery at all.

This claim is supported by the following research: in [3], Carmichael et al. applied a design-extraction tool to a recently developed system. The system was medium sized (300 KLOC, 200 files) and described as a "reasonably well designed and well constructed piece of software". Despite having all these advantages, the experiment showed that the extracted design was easily perturbed by a few minor implementation decisions.

If the design extraction can be perturbated by minor implementation flaws even in fresh, well designed software, one can doubt, automated tools would discover modules from old, huge software, unless these modules were explicitly and deliberately enforced during the maintenance.

Therefore the question seems rather to be: what can be the utility of a tool which only discovers subsystems that have been explicitly marked?

Other researchers [15, 16] have already expressed doubts about the tractability of automatic design recovery based on code source. One of their conclusions was that it does not rule out the utility of the all approach. Automatic design recovery techniques can still prove useful in:

- helping the software engineers to rediscover a design organization they don't quite remember,
- helping the software engineers to respect a design they don't fully "understand".

We believe that this goal can be better achieved by relying on many different sources of information (including code and file names).

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 and Stéphane Somé for fruitful discussions we have had with them.

The authors can be reached by email at `{anquetil,tcl}@csi.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 J. Howard Johnson, editor, *CASCON'97*, pages 184–95. IBM Centre for Advanced Studies, nov 1997.

[2] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *International conference on Software Engineering, ICSE'98*. IEEE, IEEE Comp. Soc. Press, 1998. Accepted for publication.

[3] Ian Carmichael, Vassilios Tzerpos, and R.C. Holt. Design maintenance: Unexpected architectural interactions (experience report). In *International Conference on Software Maintenance, ICSM'95*, pages 134–37. IEEE, IEEE Comp. Soc. Press, oct. 1995.

[4] Thomas Kunz and James P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Transaction on Software Engineering*, 21(6):515–527, jun 1995.

[5] Arun Lakhotia. A unified framework for expressing software subsystem classification techniques. *J. of Systems and Software*, 36:211–231, mar 1997.

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

[7] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[8] 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.

[9] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse-engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[10] Sukesh Patel, William Chu, and Rich Baxter. A Measure for Composite Module Cohesion. In *14^{th} International Conference on Software Engineering*. ACM SIGSoft/IEEE Comp. Soc. Press, 1992.

[11] G. Salton. *Automatic Text Processing*. Addison-Wesley, Reading, MA, 1973.

[12] Ian Sommerville. *Software Engineering*. International Computer Science. Addison-Wesley Publishing Comp., 5th edition, 1995.

[13] 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.

[14] Vassilios Tzerpos and Ric C. Holt. The orphan adoption problem in architecture maintenance. In Chris Verhoef Ira Baxter, Alex Quilici, editor, *Proceedings of the Working Conference on Reverse Engineering 1997*, pages 76–82. IEEE, IEEE Comp. Soc. Press, oct. 1997.

[15] Bruce W. Weide, Wayne D. Heym, and Joseph E. Hollingsworth. Reverse engineering of legacy code exposed. In *International Conference on Software Engineering, ICSE'95*, pages 327–331. IEEE, IEEE Comp. Soc. Press, 1995.

[16] Steven Woods and Qiang Yang. The program understanding problem: Analysis and a heuristic approach. In *International Conference on Software Engineering, ICSE'96*. IEEE, IEEE Comp. Soc. Press, 1996.